



WS-Trust 1.4

OASIS Committee Draft 03

12 November 2008

Specification URIs:

This Version:

<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cd-03.doc> (Authoritative)
<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cd-03.pdf>
<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cd-03.html>

Previous Version:

<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cd-02.doc> (Authoritative)
<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cd-02.pdf>
<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cd-02.html>

Latest Version:

<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.doc>
<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.pdf>
<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html>

Technical Committee:

OASIS Web Services Secure Exchange TC

Chair(s):

Kelvin Lawrence, IBM
Chris Kaler, Microsoft

Editor(s):

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

Related work:

N/A

Declared XML namespace(s):

<http://docs.oasis-open.org/ws-sx/ws-trust/200512>
<http://docs.oasis-open.org/ws-sx/ws-trust/200802>

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

Copyright © OASIS® 1993–2008. All Rights Reserved.

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

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

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

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

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

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

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

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

Table of Contents

1	Introduction.....	6
1.1	Goals and Non-Goals.....	6
1.2	Requirements.....	7
1.3	Namespace.....	7
1.4	Schema and WSDL Files.....	8
1.5	Terminology.....	8
1.5.1	Notational Conventions.....	9
1.6	Normative References.....	10
1.7	Non-Normative References.....	11
2	Web Services Trust Model.....	12
2.1	Models for Trust Brokering and Assessment.....	13
2.2	Token Acquisition.....	13
2.3	Out-of-Band Token Acquisition.....	14
2.4	Trust Bootstrap.....	14
3	Security Token Service Framework.....	15
3.1	Requesting a Security Token.....	15
3.2	Returning a Security Token.....	16
3.3	Binary Secrets.....	18
3.4	Composition.....	18
4	Issuance Binding.....	19
4.1	Requesting a Security Token.....	19
4.2	Request Security Token Collection.....	21
4.2.1	Processing Rules.....	23
4.3	Returning a Security Token Collection.....	23
4.4	Returning a Security Token.....	24
4.4.1	wsp:AppliesTo in RST and RSTR.....	25
4.4.2	Requested References.....	26
4.4.3	Keys and Entropy.....	26
4.4.4	Returning Computed Keys.....	27
4.4.5	Sample Response with Encrypted Secret.....	28
4.4.6	Sample Response with Unencrypted Secret.....	28
4.4.7	Sample Response with Token Reference.....	29
4.4.8	Sample Response without Proof-of-Possession Token.....	29
4.4.9	Zero or One Proof-of-Possession Token Case.....	29
4.4.10	More Than One Proof-of-Possession Tokens Case.....	30
4.5	Returning Security Tokens in Headers.....	31
5	Renewal Binding.....	33
6	Cancel Binding.....	36
6.1	STS-initiated Cancel Binding.....	37
7	Validation Binding.....	39
8	Negotiation and Challenge Extensions.....	42
8.1	Negotiation and Challenge Framework.....	43
8.2	Signature Challenges.....	43

8.3	User Interaction Challenge	44
8.3.1	Challenge Format	45
8.3.2	PIN and OTP Challenges	48
8.4	Binary Exchanges and Negotiations	49
8.5	Key Exchange Tokens	49
8.6	Custom Exchanges	50
8.7	Signature Challenge Example	50
8.8	Challenge Examples	52
8.8.1	Text and choice challenge	52
8.8.2	PIN challenge	54
8.8.3	PIN challenge with optimized response	56
8.9	Custom Exchange Example	57
8.10	Protecting Exchanges	58
8.11	Authenticating Exchanges	58
9	Key and Token Parameter Extensions	60
9.1	On-Behalf-Of Parameters	60
9.2	Key and Encryption Requirements	60
9.3	Delegation and Forwarding Requirements	65
9.4	Policies	66
9.5	Authorized Token Participants	67
10	Key Exchange Token Binding	68
11	Error Handling	70
12	Security Considerations	71
13	Conformance	73
A.	Key Exchange	74
A.1	Ephemeral Encryption Keys	74
A.2	Requestor-Provided Keys	74
A.3	Issuer-Provided Keys	75
A.4	Composite Keys	75
A.5	Key Transfer and Distribution	76
A.5.1	Direct Key Transfer	76
A.5.2	Brokered Key Distribution	76
A.5.3	Delegated Key Transfer	77
A.5.4	Authenticated Request/Reply Key Transfer	78
A.6	Perfect Forward Secrecy	79
B.	WSDL	80
C.	Acknowledgements	82

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.

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

- 41 • Key derivation

42 1.2 Requirements

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

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

47 1.3 Namespace

48 Implementations of this specification MUST use the following [URI]s:

49 <http://docs.oasis-open.org/ws-sx/ws-trust/200512>
 50 <http://docs.oasis-open.org/ws-sx/ws-trust/200802>

51 When using a URI to indicate that this version of Trust is being used <http://docs.oasis-open.org/ws-sx/ws-trust/200802> MUST be used.

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

55 *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-wssecurity-secext-1.1.xsd	[WS-Security]
wst	http://docs.oasis-open.org/ws-sx/ws-trust/200512	This specification
wst14	http://docs.oasis-open.org/ws-sx/ws-trust/200802	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 or http://www.w3.org/ns/ws-policy	[WS-Policy]

wsa	http://www.w3.org/2005/08/addressing	[WS-Addressing]
xs	http://www.w3.org/2001/XMLSchema	[XML-Schema1] [XML-Schema2]

56 1.4 Schema and WSDL Files

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

58 <http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust.xsd>
59 <http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.xsd>

60

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

62 <http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.wsdl>

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

66 1.5 Terminology

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

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

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

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

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

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

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

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

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

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

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

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

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

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

103 1.5.1 Notational Conventions

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

107

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

110

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

- 112 • The syntax appears as an XML instance, but values in italics indicate data types instead of literal
113 values.
- 114 • Characters are appended to elements and attributes to indicate cardinality:
 - 115 ○ "?" (0 or 1)
 - 116 ○ "*" (0 or more)
 - 117 ○ "+" (1 or more)
- 118 • The character "|" is used to indicate a choice between alternatives.
- 119 • The characters "(" and ")" are used to indicate that contained items are to be treated as a group
120 with respect to cardinality or choice.
- 121 • The characters "[" and "]" are used to call out references and property names.
- 122 • Ellipses (i.e., "...") indicate points of extensibility. Additional children and/or attributes MAY be
123 added at the indicated extension points but MUST NOT contradict the semantics of the parent
124 and/or owner, respectively. By default, if a receiver does not recognize an extension, the receiver
125 SHOULD ignore the extension; exceptions to this processing rule, if any, are clearly indicated
126 below.
- 127 • XML namespace prefixes (see Table 1) are used to indicate the namespace of the element being
128 defined.

129

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

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

- 135
- 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.
- 136
137

138

139 In this document reference is made to the `wsu:Id` attribute and the `wsu:Created` and `wsu:Expires`
140 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
141 utility schema with the intent that other specifications requiring such an ID type attribute or timestamp
142 element could reference it (as is done here).
143
144

145 1.6 Normative References

- 146 [RFC2119] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels",
147 RFC 2119, Harvard University, March 1997.
148 <http://www.ietf.org/rfc/rfc2119.txt>
- 149 [RFC2246] IETF Standard, "The TLS Protocol", January 1999.
150 <http://www.ietf.org/rfc/rfc2246.txt>
- 151 [SOAP] W3C Note, "SOAP: Simple Object Access Protocol 1.1", 08 May 2000.
152 <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>
- 153 [SOAP12] W3C Recommendation, "SOAP 1.2 Part 1: Messaging Framework", 24
154 June 2003.
155 <http://www.w3.org/TR/2003/REC-soap12-part1-20030624/>
- 156 [URI] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers
157 (URI): Generic Syntax", RFC 3986, MIT/LCS, Day Software, Adobe
158 Systems, January 2005.
159 <http://www.ietf.org/rfc/rfc3986.txt>
- 160 [WS-Addressing] W3C Recommendation, "Web Services Addressing (WS-Addressing)", 9
161 May 2006.
162 <http://www.w3.org/TR/2006/REC-ws-addr-core-20060509>
- 163 [WS-Policy] W3C Recommendation, "Web Services Policy 1.5 - Framework", 04
164 September 2007.
165 <http://www.w3.org/TR/2007/REC-ws-policy-20070904/>
- 166 W3C Member Submission, "Web Services Policy 1.2 - Framework", 25
167 April 2006.
168 <http://www.w3.org/Submission/2006/SUBM-WS-Policy-20060425/>
- 169 [WS-PolicyAttachment] W3C Recommendation, "Web Services Policy 1.5 - Attachment", 04
170 September 2007.
171 <http://www.w3.org/TR/2007/REC-ws-policy-attach-20070904/>
- 172 W3C Member Submission, "Web Services Policy 1.2 - Attachment", 25
173 April 2006.
174 <http://www.w3.org/Submission/2006/SUBM-WS-PolicyAttachment-20060425/>
- 175
- 176 [WS-Security] OASIS Standard, "OASIS Web Services Security: SOAP Message Security
177 1.0 (WS-Security 2004)", March 2004.
178 <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>
- 179
- 180 OASIS Standard, "OASIS Web Services Security: SOAP Message Security
181 1.1 (WS-Security 2004)", February 2006.
182 <http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>
183

184 [XML-C14N] W3C Recommendation, "Canonical XML Version 1.0", 15 March 2001.
185 <http://www.w3.org/TR/2001/REC-xml-c14n-20010315>
186 W3C Recommendation, "Canonical XML Version 1.1", 2 May 2008.
187 <http://www.w3.org/TR/2008/REC-xml-c14n11-20080502/>
188 [XML-Encrypt] W3C Recommendation, "XML Encryption Syntax and Processing", 10
189 December 2002.
190 <http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/>
191 [XML-Schema1] W3C Recommendation, "XML Schema Part 1: Structures Second Edition",
192 28 October 2004.
193 <http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/>
194 [XML-Schema2] W3C Recommendation, "XML Schema Part 2: Datatypes Second Edition",
195 28 October 2004.
196 <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/>
197 [XML-Signature] W3C Recommendation, "XML-Signature Syntax and Processing", 12
198 February 2002.
199 <http://www.w3.org/TR/2002/REC-xmlsig-core-20020212/>
200 [W3C Recommendation, D. Eastlake et al. XML Signature Syntax and
201 Processing (Second Edition). 10 June 2008.
202 <http://www.w3.org/TR/2008/REC-xmlsig-core-20080610/>
203

204 1.7 Non-Normative References

205 [Kerberos] J. Kohl and C. Neuman, "The Kerberos Network 149 Authentication
206 Service (V5)," RFC 1510, September 1993.
207 <http://www.ietf.org/rfc/rfc1510.txt>
208 [WS-Federation] "Web Services Federation Language," BEA, IBM, Microsoft, RSA Security,
209 VeriSign, July 2003.
210 [WS-SecurityPolicy] OASIS Committee Draft, "WS-SecurityPolicy 1.2", September 2006
211 <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512>
212 [X509] S. Santesson, et al, "Internet X.509 Public Key Infrastructure Qualified
213 Certificates Profile."
214 [http://www.itu.int/rec/recommendation.asp?type=items&lang=e&parent=T-
215 REC-X.509-200003-I](http://www.itu.int/rec/recommendation.asp?type=items&lang=e&parent=T-REC-X.509-200003-I)
216

217

2 Web Services Trust Model

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

223

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

228

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

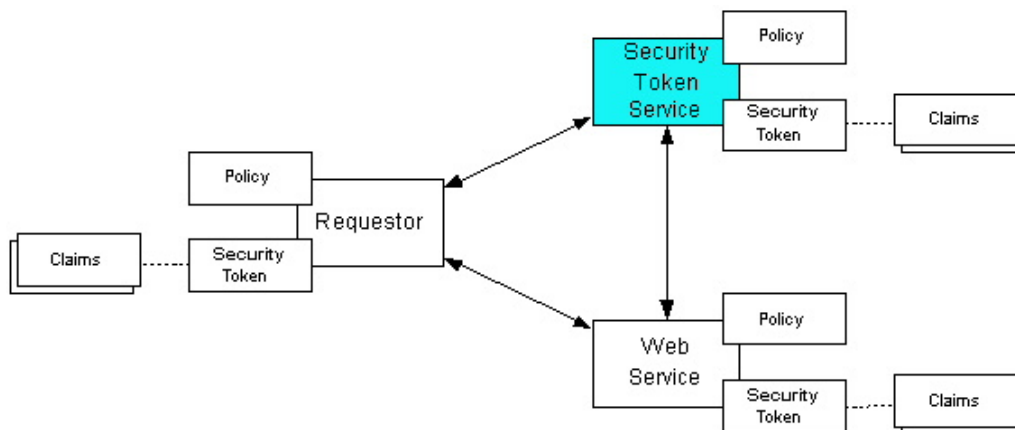
232 • If the requestor does not have the necessary token(s) to prove required claims to a service, it can
233 contact appropriate authorities (as indicated in the service's policy) and request the needed tokens
234 with the proper claims. These "authorities", which we refer to as *security token services*, may in turn
235 require their own set of claims for authenticating and authorizing the request for security tokens.
236 Security token services form the basis of trust by issuing a range of security tokens that can be used
237 to broker trust relationships between different trust domains.

238 • This specification also defines a general mechanism for multi-message exchanges during token
239 acquisition. One example use of this is a challenge-response protocol that is also defined in this
240 specification. This is used by a Web service for additional challenges to a requestor to ensure
241 message freshness and verification of authorized use of a security token.

242

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

245



246

247 This general security model – claims, policies, and security tokens – subsumes and supports several
248 more specific models such as identity-based authorization, access control lists, and capabilities-based
249 authorization. It allows use of existing technologies such as X.509 public-key certificates, XML-based

250 tokens, Kerberos shared-secret tickets, and even password digests. The general model in combination
251 with the [WS-Security] and [WS-Policy] primitives is sufficient to construct higher-level key exchange,
252 authentication, policy-based access control, auditing, and complex trust relationships.

253

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

259

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

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

274

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

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

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

283

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

287 **2.1 Models for Trust Brokering and Assessment**

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

291 **2.2 Token Acquisition**

292 As part of a message flow, a request MAY be made of a security token service to exchange a security
293 token (or some proof) of one form for another. The exchange request can be made either by a requestor

294 or by another party on the requestor's behalf. If the security token service trusts the provided security
295 token (for example, because it trusts the issuing authority of the provided security token), and the request
296 can prove possession of that security token, then the exchange is processed by the security token
297 service.

298

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

305 2.3 Out-of-Band Token Acquisition

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

311 2.4 Trust Bootstrap

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

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

321

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

324

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

329

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

334 **3 Security Token Service Framework**

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

336

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

342

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

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

349

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

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

357

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

364

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

369 **3.1 Requesting a Security Token**

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

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

375 The syntax for this element is as follows:

376
377
378
379
380
381

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

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

383 */wst:RequestSecurityToken*

384 This is a request to have a security token issued.

385 */wst:RequestSecurityToken/@Context*

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

390 */wst:RequestSecurityToken/wst:TokenType*

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

395 */wst:RequestSecurityToken/wst:RequestType*

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

402 */wst:RequestSecurityToken/wst:SecondaryParameters*

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

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

410 */wst:RequestSecurityToken/{any}*

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

415 */wst:RequestSecurityToken/@{any}*

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

418 **3.2 Returning a Security Token**

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

423

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

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

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

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

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

446 The syntax for this element is as follows:

```
447     <wst:RequestSecurityTokenResponse Context="..." xmlns:wst="...">  
448         <wst:TokenType>...</wst:TokenType>  
449         <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>  
450         ...  
451     </wst:RequestSecurityTokenResponse>
```

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

453 */wst:RequestSecurityTokenResponse*

454 This is the response to a security token request.

455 */wst:RequestSecurityTokenResponse/@Context*

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

461 */wst:RequestSecurityTokenResponse/wst:TokenType*

462 This OPTIONAL element specifies the type of security token returned.

463 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

464 This OPTIONAL element is used to return the requested security token. Normally the requested
465 security token is the contents of this element but a security token reference MAY be used instead.
466 For example, if the requested security token is used in securing the message, then the security
467 token is placed into the `<wsse:Security>` header (as described in [\[WS-Security\]](#)) and a
468 `<wsse:SecurityTokenReference>` element is placed inside of the
469 `<wst:RequestedSecurityToken>` element to reference the token in the `<wsse:Security>`
470 header. The response MAY contain a token reference where the token is located at a URI

471 outside of the message. In such cases the recipient is assumed to know how to fetch the token
 472 from the URI address or specified endpoint reference. It should be noted that when the token is
 473 not returned as part of the message it cannot be secured, so a secure communication
 474 mechanism SHOULD be used to obtain the token.

475 */wst:RequestSecurityTokenResponse/{any}*

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

478 */wst:RequestSecurityTokenResponse/@{any}*

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

481 3.3 Binary Secrets

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

489 *.../wst:BinarySecret*

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

492 *.../wst:BinarySecret/@Type*

493 This OPTIONAL attribute indicates the type of secret being encoded. The pre-defined values are
 494 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)

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

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

498 3.4 Composition

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

503 4 Issuance Binding

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

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

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

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

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

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

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

522

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

526

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

529

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

533 4.1 Requesting a Security Token

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

```
537 <wst:RequestSecurityToken xmlns:wst="...">  
538   <wst:TokenType>...</wst:TokenType>  
539   <wst:RequestType>...</wst:RequestType>  
540   ...  
541   <wsp:AppliesTo>...</wsp:AppliesTo>  
542   <wst:Claims Dialect="...">...</wst:Claims>  
543   <wst:Entropy>  
544     <wst:BinarySecret>...</wst:BinarySecret>  
545   </wst:Entropy>  
546   <wst:Lifetime>
```

```
547         <wsu:Created>...</wsu:Created>
548         <wsu:Expires>...</wsu:Expires>
549     </wst:Lifetime>
550 </wst:RequestSecurityToken>
```

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

552 */wst:RequestSecurityToken/wst:TokenType*

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

561 */wst:RequestSecurityToken/wsp:AppliesTo*

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

569 */wst:RequestSecurityToken/wst:Claims*

570 This OPTIONAL element requests a specific set of claims. Typically, this element contains
571 REQUIRED and/or OPTIONAL claim information identified in a service's policy.

572 */wst:RequestSecurityToken/wst:Claims/@Dialect*

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

577 */wst:RequestSecurityToken/wst:Entropy*

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

582 */wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret*

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

589 */wst:RequestSecurityToken/wst:Lifetime*

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

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

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

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

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

609

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

```
616 <S11:Envelope xmlns:S11="..." xmlns:wsu="..." xmlns:wsse="..."  
617     xmlns:xenc="..." xmlns:wst="...">  
618   <S11:Header>  
619     ...  
620     <wsse:Security>  
621       <xenc:ReferenceList>...</xenc:ReferenceList>  
622       <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>  
623       <ds:Signature xmlns:ds="...">  
624         ...  
625         <ds:KeyInfo>  
626           <wsse:SecurityTokenReference>  
627             <wsse:Reference URI="#myToken"/>  
628           </wsse:SecurityTokenReference>  
629         </ds:KeyInfo>  
630       </ds:Signature>  
631     </wsse:Security>  
632     ...  
633   </S11:Header>  
634   <S11:Body wsu:Id="req">  
635     <wst:RequestSecurityToken>  
636       <wst:TokenType>  
637         http://example.org/mySpecialToken  
638       </wst:TokenType>  
639       <wst:RequestType>  
640         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
641       </wst:RequestType>  
642     </wst:RequestSecurityToken>  
643   </S11:Body>  
644 </S11:Envelope>
```

645 4.2 Request Security Token Collection

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

650

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

655

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

659

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

661

```
662 <wst:RequestSecurityTokenCollection xmlns:wst="...">
663
664   <!-- identity token request -->
665   <wst:RequestSecurityToken Context="http://www.example.com/1">
666     <wst:TokenType>
667       http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
668     1.1#SAMLV2.0
669     </wst:TokenType>
670     <wst:RequestType>http://docs.oasis-open.org/ws-sx/ws-
671   trust/200512/BatchIssue</wst:RequestType>
672     <wsp:AppliesTo xmlns:wsp="..." xmlns:wsa="...">
673       <wsa:EndpointReference>
674         <wsa:Address>http://manufacturer.example.com/</wsa:Address>
675       </wsa:EndpointReference>
676     </wsp:AppliesTo>
677     <wsp:PolicyReference xmlns:wsp="..."
678   URI='http://manufacturer.example.com/IdentityPolicy' />
679   </wst:RequestSecurityToken>
680
681   <!-- certification claim token request -->
682   <wst:RequestSecurityToken Context="http://www.example.com/2">
683     <wst:TokenType>
684       http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
685     1.1#SAMLV2.0
686     </wst:TokenType>
687     <wst:RequestType>http://docs.oasis-open.org/ws-sx/ws-trust/200512
688   /BatchIssue</wst:RequestType>
689     <wst:Claims xmlns:wsp="...">
690       http://manufacturer.example.com/certification
691     </wst:Claims>
692     <wsp:PolicyReference
693   URI='http://certificationbody.example.org/certificationPolicy' />
694   </wst:RequestSecurityToken>
695 </wst:RequestSecurityTokenCollection>
```

696

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

698

699 */wst:RequestSecurityTokenCollection*

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

703 4.2.1 Processing Rules

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

706

707 1. The single `RequestSecurityTokenResponseCollection` response MUST contain at least
708 one RSTR element corresponding to each RST element in the request. A RSTR element
709 corresponds to an RST element if it has the same Context attribute value as the RST element.

710 **Note:** Each request MAY generate more than one RSTR sharing the same Context attribute
711 value

712 a. Specifically there is no notion of a deferred response

713 b. If any RST request results in an error, then no RSTRs will be returned and a SOAP Fault
714 will be generated as the entire response.

715 2. Every RST in the request MUST use an action URI value in the `RequestType` element that is a
716 batch version corresponding to the non-batch version, in particular one of the following:

- 717 • `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchIssue`
- 718 • `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchValidate`
- 719 • `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchRenew`
- 720 • `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchCancel`

721

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

724

725 **Note:** that these operations require that the service can either succeed on all the RST requests or
726 MUST NOT perform any partial operation.

727

728 3. All Signatures MUST reference the entire RSTC. One or more Signatures referencing the entire
729 collection MAY be used.

730 4. No negotiation or other multi-leg authentication mechanisms are allowed in batch requests or
731 responses to batch requests; the communication with STS is limited to one RSTC request and
732 one RSTRC response.

733 5. This mechanism requires that every RST in a RSTC is to be handled by the single endpoint
734 processing the RSTC.

735

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

738

739 4.3 Returning a Security Token Collection

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

744 The syntax for this element is as follows:

```

745 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
746 <wst:RequestSecurityTokenResponse>...</wst:RequestSecurityTokenResponse> +
747 </wst:RequestSecurityTokenResponseCollection>

```

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

749 */wst:RequestSecurityTokenResponseCollection*

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

752 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

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

754 4.4 Returning a Security Token

755 When returning a security token, the following OPTIONAL elements MAY be included in the response.

756 Security tokens can only be returned in the RSTRC on the final leg. The syntax for these elements is as
757 follows (note that the base elements described above are included here italicized for completeness):

```

758 <wst:RequestSecurityTokenResponse xmlns:wst="...">
759 <wst:TokenType>...</wst:TokenType>
760 <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
761 ...
762 <wsp:AppliesTo xmlns:wsp="...">...</wsp:AppliesTo>
763 <wst:RequestedAttachedReference>
764 ...
765 </wst:RequestedAttachedReference>
766 <wst:RequestedUnattachedReference>
767 ...
768 </wst:RequestedUnattachedReference>
769 <wst:RequestedProofToken>...</wst:RequestedProofToken>
770 <wst:Entropy>
771 <wst:BinarySecret>...</wst:BinarySecret>
772 </wst:Entropy>
773 <wst:Lifetime>...</wst:Lifetime>
774 </wst:RequestSecurityTokenResponse>

```

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

776 */wst:RequestSecurityTokenResponse/wsp:AppliesTo*

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

781 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

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

787 */wst:RequestSecurityTokenResponse/wst:RequestedAttachedReference*

788 Since returned tokens are considered opaque to the requestor, this OPTIONAL element is
789 specified to indicate how to reference the returned token when that token doesn't support
790 references using URI fragments (XML ID). This element contains a
791 `<wsse:SecurityTokenReference>` element that can be used *verbatim* to reference the token
792 (when the token is placed inside a message). Typically tokens allow the use of *wsu:id* so this
793 element isn't required. Note that a token MAY support multiple reference mechanisms; this
794 indicates the issuer's preferred mechanism. When encrypted tokens are returned, this element is

795 not needed since the `<xenc:EncryptedData>` element supports an ID reference. If this
796 element is not present in the RSTR then the recipient can assume that the returned token (when
797 present in a message) supports references using URI fragments.

798 */wst:RequestSecurityTokenResponse/wst:RequestedUnattachedReference*

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

805 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken*

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

816 */wst:RequestSecurityTokenResponse/wst:Entropy*

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

821 */wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret*

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

824 */wst:RequestSecurityTokenResponse/wst:Lifetime*

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

828 **4.4.1 wsp:AppliesTo in RST and RSTR**

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

- 833 • If neither the requestor nor the issuer specifies a scope then the scope of the issued token is
834 implied.
- 835 • If the requestor specifies a scope and the issuer does not then the scope of the token is assumed
836 to be that specified by the requestor.
- 837 • If the requestor does not specify a scope and the issuer does specify a scope then the scope of
838 the token is as defined by the issuers scope
- 839 • If both requestor and issuer specify a scope then there are two possible outcomes:
 - 840 ○ If both the issuer and requestor specify the same scope then the issued token has that
841 scope.

- 842 o If the issuer specifies a wider scope than the requestor then the issued token has the
843 scope specified by the issuer.
- 844 • The requestor and issuer MUST agree on the version of [WS-Policy] used to specify the scope of
845 the issued token. The Trust13 assertion in [WS-SecurityPolicy] provides a mechanism to
846 communicate which version of [WS-Policy] is to be used.

847

848 The following table summarizes the above rules:

Requestor <code>wsp:AppliesTo</code>	Issuer <code>wsp:AppliesTo</code>	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.

849 4.4.2 Requested References

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

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

865 4.4.3 Keys and Entropy

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

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

873 element is returned inside the `<wst:RequestedProofToken>` to indicate how the key is
 874 computed.

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

877

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

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

888

889 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

890 4.4.4 Returning Computed Keys

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

```

896 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
897   <wst:RequestSecurityTokenResponse>
898     <wst:RequestedProofToken>
899       <wst:ComputedKey>...</wst:ComputedKey>
900     </wst:RequestedProofToken>
901   </wst:RequestSecurityTokenResponse>
902 </wst:RequestSecurityTokenResponseCollection>
  
```

903

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

905 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken/wst:ComputedKey*
 906 The value of this element is a URI describing how to compute the key. While this can be
 907 extended by defining new URIs in other bindings and profiles, the following URI pre-defines one
 908 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: $\text{key} = \text{P_SHA1}(\text{Ent}_{\text{REQ}}, \text{Ent}_{\text{RES}})$ It is RECOMMENDED that EntREQ be a string of length at least 128 bits.

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

910 4.4.5 Sample Response with Encrypted Secret

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

```

915 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
916   <wst:RequestSecurityTokenResponse>
917     <wst:RequestedSecurityToken>
918       <xyz:CustomToken xmlns:xyz="...">
919         ...
920       </xyz:CustomToken>
921     </wst:RequestedSecurityToken>
922     <wst:RequestedProofToken>
923       <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
924         ...
925       </xenc:EncryptedKey>
926     </wst:RequestedProofToken>
927   </wst:RequestSecurityTokenResponse>
928 </wst:RequestSecurityTokenResponseCollection>
  
```

929 4.4.6 Sample Response with Unencrypted Secret

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

```

932 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
933   <wst:RequestSecurityTokenResponse>
934     <wst:RequestedSecurityToken>
935       <xyz:CustomToken xmlns:xyz="...">
936         ...
937       </xyz:CustomToken>
938     </wst:RequestedSecurityToken>
939     <wst:RequestedProofToken>
940       <wst:BinarySecret>...</wst:BinarySecret>
941     </wst:RequestedProofToken>
942   </wst:RequestSecurityTokenResponse>
943 </wst:RequestSecurityTokenResponseCollection>
  
```

944 4.4.7 Sample Response with Token Reference

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

```
948 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
949   <wst:RequestSecurityTokenResponse>  
950     <wst:RequestedSecurityToken>  
951       <xyz:CustomToken ID="urn:fabrikam123:5445" xmlns:xyz="...">  
952         ...  
953       </xyz:CustomToken>  
954     </wst:RequestedSecurityToken>  
955     <wst:RequestedAttachedReference>  
956       <wsse:SecurityTokenReference xmlns:wsse="...">  
957         <wsse:Reference URI="urn:fabrikam123:5445"/>  
958       </wsse:SecurityTokenReference>  
959     </wst:RequestedAttachedReference>  
960     ...  
961   </wst:RequestSecurityTokenResponse>  
962 </wst:RequestSecurityTokenResponseCollection>
```

963

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

968 4.4.8 Sample Response without Proof-of-Possession Token

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

```
973 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
974   <wst:RequestSecurityTokenResponse>  
975     <wst:RequestedSecurityToken>  
976       <xyz:CustomToken xmlns:xyz="...">  
977         ...  
978       </xyz:CustomToken>  
979     </wst:RequestedSecurityToken>  
980   </wst:RequestSecurityTokenResponse>  
981 </wst:RequestSecurityTokenResponseCollection>
```

982

983 4.4.9 Zero or One Proof-of-Possession Token Case

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

990

```
991 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
992   <wst:RequestSecurityTokenResponse>  
993     <wst:RequestedSecurityToken>
```

```

994         <xyz:CustomToken xmlns:xyz="...">
995             ...
996         </xyz:CustomToken>
997     </wst:RequestedSecurityToken>
998     <wst:RequestedProofToken>
999         <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
1000             ...
1001         </xenc:EncryptedKey>
1002     </wst:RequestedProofToken>
1003 </wst:RequestSecurityTokenResponse>
1004 </wst:RequestSecurityTokenResponseCollection>

```

1005 4.4.10 More Than One Proof-of-Possession Tokens Case

1006 The second case is where multiple security tokens are returned that have separate proof-of-possession
1007 tokens. As a result, the proof-of-possession tokens, and possibly lifetime and other key parameters
1008 elements, MAY be different. To address this scenario, the body MAY be specified using the syntax
1009 illustrated below:

```

1010 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
1011     <wst:RequestSecurityTokenResponse>
1012         ...
1013     </wst:RequestSecurityTokenResponse>
1014     <wst:RequestSecurityTokenResponse>
1015         ...
1016     </wst:RequestSecurityTokenResponse>
1017     ...
1018 </wst:RequestSecurityTokenResponseCollection>

```

1019 The following describes the attributes and elements listed in the schema overview above:

1020 */wst:RequestSecurityTokenResponseCollection*

1021 This element is used to provide multiple RSTR responses, each of which has separate key
1022 information. One or more RSTR elements are returned in the collection. This MUST always be
1023 used on the final response to the RST.

1024 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

1025 Each RequestSecurityTokenResponse element is an individual RSTR.

1026 */wst:RequestSecurityTokenResponseCollection/{any}*

1027 This is an extensibility mechanism to allow additional elements, based on schemas, to be added.

1028 */wst:RequestSecurityTokenResponseCollection/@{any}*

1029 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added.

1030 The following illustrates the syntax of a response that includes multiple tokens each, in a separate RSTR,
1031 each with their own proof-of-possession token.

```

1032 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
1033     <wst:RequestSecurityTokenResponse>
1034         <wst:RequestedSecurityToken>
1035             <xyz:CustomToken xmlns:xyz="...">
1036                 ...
1037             </xyz:CustomToken>
1038         </wst:RequestedSecurityToken>
1039         <wst:RequestedProofToken>
1040             <xenc:EncryptedKey Id="newProofA">
1041                 ...
1042             </xenc:EncryptedKey>
1043         </wst:RequestedProofToken>
1044     </wst:RequestSecurityTokenResponse>
1045     <wst:RequestSecurityTokenResponse>

```

```

1046     <wst:RequestedSecurityToken>
1047         <abc:CustomToken xmlns:abc="...">
1048             ...
1049         </abc:CustomToken>
1050     </wst:RequestedSecurityToken>
1051     <wst:RequestedProofToken>
1052         <xenc:EncryptedKey Id="newProofB xmlns:xenc="...">
1053             ...
1054         </xenc:EncryptedKey>
1055     </wst:RequestedProofToken>
1056 </wst:RequestSecurityTokenResponse>
1057 </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.

```

1065 <wst:IssuedTokens xmlns:wst="...">
1066   <wst:RequestSecurityTokenResponse>
1067     ...
1068   </wst:RequestSecurityTokenResponse>+
1069 </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.

1093 The following example illustrates a response that includes multiple `<wst:IssuedTokens>` headers.

```
1094 <?xml version="1.0" encoding="utf-8"?>
1095 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsp="..." xmlns:ds="..."
1096 xmlns:x="...">
1097   <S11:Header>
1098     <wst:IssuedTokens>
1099       <wst:RequestSecurityTokenResponse>
1100         <wsp:AppliesTo>
1101           <x:SomeContext1 />
1102         </wsp:AppliesTo>
1103         <wst:RequestedSecurityToken>
1104           ...
1105         </wst:RequestedSecurityToken>
1106         ...
1107       </wst:RequestSecurityTokenResponse>
1108       <wst:RequestSecurityTokenResponse>
1109         <wsp:AppliesTo>
1110           <x:SomeContext1 />
1111         </wsp:AppliesTo>
1112         <wst:RequestedSecurityToken>
1113           ...
1114         </wst:RequestedSecurityToken>
1115         ...
1116       </wst:RequestSecurityTokenResponse>
1117     </wst:IssuedTokens>
1118     <wst:IssuedTokens S11:role="http://example.org/somerole" >
1119       <wst:RequestSecurityTokenResponse>
1120         <wsp:AppliesTo>
1121           <x:SomeContext2 />
1122         </wsp:AppliesTo>
1123         <wst:RequestedSecurityToken>
1124           ...
1125         </wst:RequestedSecurityToken>
1126         ...
1127       </wst:RequestSecurityTokenResponse>
1128     </wst:IssuedTokens>
1129   </S11:Header>
1130   <S11:Body>
1131     ...
1132   </S11:Body>
1133 </S11:Envelope>
```

5 Renewal Binding

1134

1135 Using the token request framework, this section defines bindings for requesting security tokens to be
1136 renewed:

1137 **Renew** – A previously issued token with expiration is presented (and possibly proven) and the
1138 same token is returned with new expiration semantics.

1139

1140 For this binding, the following actions are defined to enable specific processing context to be conveyed to
1141 the recipient:

```
1142 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew
1143 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Renew
1144 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/RenewFinal
```

1145 For this binding, the `<wst:RequestType>` element uses the following URI:

```
1146 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Renew
```

1147 For this binding the token to be renewed is identified in the `<wst:RenewTarget>` element and the
1148 OPTIONAL `<wst:Lifetime>` element MAY be specified to request a specified renewal duration.

1149

1150 Other extensions MAY be specified in the request (and the response), but the key semantics (size, type,
1151 algorithms, scope, etc.) MUST NOT be altered during renewal. Token services MAY use renewal as an
1152 opportunity to rekey, so the renewal responses MAY include a new proof-of-possession token as well as
1153 entropy and key exchange elements.

1154

1155 The request MUST prove authorized use of the token being renewed unless the recipient trusts the
1156 requestor to make third-party renewal requests. In such cases, the third-party requestor MUST prove its
1157 identity to the issuer so that appropriate authorization occurs.

1158

1159 The original proof information SHOULD be proven during renewal.

1160

1161 The renewal binding allows the use of exchanges during the renewal process. Subsequent profiles MAY
1162 define restriction around the usage of exchanges.

1163

1164 During renewal, all key bearing tokens used in the renewal request MUST have an associated signature.
1165 All non-key bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the renewal
1166 response.

1167

1168 The renewal binding also defines several extensions to the request and response elements. The syntax
1169 for these extension elements is as follows (note that the base elements described above are included
1170 here italicized for completeness):

```
1171 <wst:RequestSecurityToken xmlns:wst="...">
1172   <wst:TokenType>...</wst:TokenType>
1173   <wst:RequestType>...</wst:RequestType>
1174   ...
1175   <wst:RenewTarget>...</wst:RenewTarget>
1176   <wst:AllowPostdating/>
```

1177
1178

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

1179 */wst:RequestSecurityToken/wst:RenewTarget*

1180 This REQUIRED element identifies the token being renewed. This MAY contain a
1181 `<wsse:SecurityTokenReference>` pointing at the token to be renewed or it MAY directly contain
1182 the token to be renewed.

1183 */wst:RequestSecurityToken/wst:AllowPostdating*

1184 This OPTIONAL element indicates that returned tokens SHOULD allow requests for postdated
1185 tokens. That is, this allows for tokens to be issued that are not immediately valid (e.g., a token
1186 that can be used the next day).

1187 */wst:RequestSecurityToken/wst:Renewing*

1188 This OPTIONAL element is used to specify renew semantics for types that support this operation.

1189 */wst:RequestSecurityToken/wst:Renewing/@Allow*

1190 This OPTIONAL Boolean attribute is used to request a renewable token. If not specified, the
1191 default value is *true*. A renewable token is one whose lifetime can be extended. This is done
1192 using a renewal request. The recipient MAY allow renewals without demonstration of authorized
1193 use of the token or they MAY fault.

1194 */wst:RequestSecurityToken/wst:Renewing/@OK*

1195 This OPTIONAL Boolean attribute is used to indicate that a renewable token is acceptable if the
1196 requested duration exceeds the limit of the issuance service. That is, if *true* then tokens can be
1197 renewed after their expiration. It should be noted that the token is NOT valid after expiration for
1198 any operation except renewal. The default for this attribute is *false*. It NOT RECOMMENDED to
1199 use this as it can leave you open to certain types of security attacks. Issuers MAY restrict the
1200 period after expiration during which time the token can be renewed. This window is governed by
1201 the issuer's policy.

1202 The following example illustrates a request for a custom token that can be renewed.

1203
1204
1205
1206
1207
1208
1209
1210
1211

```
<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>
```

1212

1213 The following example illustrates a subsequent renewal request and response (note that for brevity only
1214 the request and response are illustrated). Note that the response includes an indication of the lifetime of
1215 the renewed token.

1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227

```
<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>
```

```
1228 <wst:RequestSecurityTokenResponse xmlns:wst="...">
1229   <wst:TokenType>
1230     http://example.org/mySpecialToken
1231   </wst:TokenType>
1232   <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1233   <wst:Lifetime>...</wst:Lifetime>
1234   ...
1235 </wst:RequestSecurityTokenResponse>
```

6 Cancel Binding

1236

1237 Using the token request framework, this section defines bindings for requesting security tokens to be
1238 cancelled:

1239 **Cancel** – When a previously issued token is no longer needed, the Cancel binding can be used
1240 to cancel the token, terminating its use. After canceling a token at the issuer, a STS MUST not
1241 validate or renew the token. A STS MAY initiate the revocation of a token, however, revocation is
1242 out of scope of this specification and a client MUST NOT rely on it. If a client needs to ensure the
1243 validity of a token, it MUST validate the token at the issuer.

1244

1245 For this binding, the following actions are defined to enable specific processing context to be conveyed to
1246 the recipient:

```
1247 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel  
1248 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Cancel  
1249 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal
```

1250 For this binding, the `<wst:RequestType>` element uses the following URI:

```
1251 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
```

1252 Extensions MAY be specified in the request (and the response), but the semantics are not defined by this
1253 binding.

1254

1255 The request MUST prove authorized use of the token being cancelled unless the recipient trusts the
1256 requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its
1257 identity to the issuer so that appropriate authorization occurs.

1258 In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key
1259 bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the closure response.

1260

1261 A cancelled token is no longer valid for authentication and authorization usages.

1262 On success a cancel response is returned. This is an RSTR message with the
1263 `<wst:RequestedTokenCancelled>` element in the body. On failure, a Fault is raised. It should be
1264 noted that the cancel RSTR is informational. That is, the security token is cancelled once the cancel
1265 request is processed.

1266

1267 The syntax of the request is as follows:

```
1268 <wst:RequestSecurityToken xmlns:wst="...">  
1269   <wst:RequestType>...</wst:RequestType>  
1270   ...  
1271   <wst:CancelTarget>...</wst:CancelTarget>  
1272 </wst:RequestSecurityToken>
```

1273 */wst:RequestSecurityToken/wst:CancelTarget*

1274 This REQUIRED element identifies the token being cancelled. Typically this contains a
1275 `<wsse:SecurityTokenReference>` pointing at the token, but it could also carry the token
1276 directly.

1277 The following example illustrates a request to cancel a custom token.

```
1278 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
```

```

1279     <S11:Header>
1280         <wsse:Security>
1281             ...
1282         </wsse:Security>
1283     </S11:Header>
1284     <S11:Body>
1285         <wst:RequestSecurityToken>
1286             <wst:RequestType>
1287                 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
1288             </wst:RequestType>
1289             <wst:CancelTarget>
1290                 ...
1291             </wst:CancelTarget>
1292         </wst:RequestSecurityToken>
1293     </S11:Body>
1294 </S11:Envelope>

```

1295 The following example illustrates a response to cancel a custom token.

```

1296 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
1297     <S11:Header>
1298         <wsse:Security>
1299             ...
1300         </wsse:Security>
1301     </S11:Header>
1302     <S11:Body>
1303         <wst:RequestSecurityTokenResponse>
1304             <wst:RequestedTokenCancelled/>
1305         </wst:RequestSecurityTokenResponse>
1306     </S11:Body>
1307 </S11:Envelope>

```

1308 6.1 STS-initiated Cancel Binding

1309 Using the token request framework, this section defines an OPTIONAL binding for requesting security
1310 tokens to be cancelled by the STS:

1311 **STS-initiated Cancel** – When a previously issued token becomes invalid on the STS, the STS-
1312 initiated Cancel binding can be used to cancel the token, terminating its use. After canceling a
1313 token, a STS MUST not validate or renew the token. This binding can be only used when STS
1314 can send one-way messages to the original token requestor.

1315
1316 For this binding, the following actions are defined to enable specific processing context to be conveyed to
1317 the recipient:

1318 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/STSCancel`

1319 For this binding, the `<wst:RequestType>` element uses the following URI:

1320 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/STSCancel`

1321 Extensions MAY be specified in the request, but the semantics are not defined by this binding.

1322
1323 The request MUST prove authorized use of the token being cancelled unless the recipient trusts the
1324 requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its
1325 identity to the issuer so that appropriate authorization occurs.

1326 In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key
1327 bearing tokens MUST be signed.

1328

1329 A cancelled token is no longer valid for authentication and authorization usages.

1330

1331 The mechanism to determine the availability of STS-initiated Cancel binding on the STS is out of scope of
1332 this specification. Similarly, how the client communicates its endpoint address to the STS so that it can
1333 send the STSCancel messages to the client is out of scope of this specification. This functionality is
1334 implementation specific and can be solved by different mechanisms that are not in scope for this
1335 specification.

1336

1337 This is a one-way operation, no response is returned from the recipient of the message.

1338

1339 The syntax of the request is as follows:

```
1340 <wst:RequestSecurityToken xmlns:wst="...">  
1341   <wst:RequestType>...</wst:RequestType>  
1342   ...  
1343   <wst:CancelTarget>...</wst:CancelTarget>  
1344 </wst:RequestSecurityToken>
```

1345 */wst:RequestSecurityToken/wst:CancelTarget*

1346 This REQUIRED element identifies the token being cancelled. Typically this contains a
1347 <wsse:SecurityTokenReference> pointing at the token, but it could also carry the token
1348 directly.

1349 The following example illustrates a request to cancel a custom token.

```
1350 <?xml version="1.0" encoding="utf-8"?>  
1351 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">  
1352   <S11:Header>  
1353     <wsse:Security>  
1354       ...  
1355     </wsse:Security>  
1356   </S11:Header>  
1357   <S11:Body>  
1358     <wst:RequestSecurityToken>  
1359       <wst:RequestType>  
1360         http://docs.oasis-open.org/ws-sx/ws-trust/200512/STSCancel  
1361       </wst:RequestType>  
1362       <wst:CancelTarget>  
1363         ...  
1364       </wst:CancelTarget>  
1365     </wst:RequestSecurityToken>  
1366   </S11:Body>  
1367 </S11:Envelope>
```

7 Validation Binding

1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410

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>
  ...
```

1411

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

1412

1413

```

<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>

```

1414

1415

1416

1417

1418

1419

1420

1421

1422

1423 */wst:RequestSecurityToken/wst:ValidateTarget*

1424 This REQUIRED element identifies the token being validated. Typically this contains a
1425 <wsse:SecurityTokenReference> pointing at the token, but could also carry the token
1426 directly.

1427 */wst:RequestSecurityTokenResponse/wst:Status*

1428 When a validation request is made, this element MUST be in the response. The code value
1429 indicates the results of the validation in a machine-readable form. The accompanying text
1430 element allows for human textual display.

1431 */wst:RequestSecurityTokenResponse/wst:Status/wst:Code*

1432 This REQUIRED URI value provides a machine-readable status code. The following URIs are
1433 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

1434 */wst:RequestSecurityTokenResponse/wst:Status/wst:Reason*

1435 This OPTIONAL string provides human-readable text relating to the status code.

1436

1437 The following illustrates the syntax of a validation request and response. In this example no token is
1438 requested, just a status.

1439

```

<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>

```

1440

1441

1442

1443

1444

1445

1446

1447

1448

```

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

```

1449

1450

1451

```
1452     <wst:Status>
1453         <wst:Code>
1454             http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
1455         </wst:Code>
1456     </wst:Status>
1457     ...
1458 </wst:RequestSecurityTokenResponse>
```

1459 The following illustrates the syntax of a validation request and response. In this example a custom token
1460 is requested indicating authorized rights in addition to the status.

```
1461 <wst:RequestSecurityToken xmlns:wst="...">
1462     <wst:TokenType>
1463         http://example.org/mySpecialToken
1464     </wst:TokenType>
1465     <wst:RequestType>
1466         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate
1467     </wst:RequestType>
1468 </wst:RequestSecurityToken>
```

```
1469
1470 <wst:RequestSecurityTokenResponse xmlns:wst="...">
1471     <wst:TokenType>
1472         http://example.org/mySpecialToken
1473     </wst:TokenType>
1474     <wst:Status>
1475         <wst:Code>
1476             http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
1477         </wst:Code>
1478     </wst:Status>
1479     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1480     ...
1481 </wst:RequestSecurityTokenResponse>
```

8 Negotiation and Challenge Extensions

1482

1483 The general security token service framework defined above allows for a simple request and response for
1484 security tokens (possibly asynchronous). However, there are many scenarios where a set of exchanges
1485 between the parties is REQUIRED prior to returning (e.g., issuing) a security token. This section
1486 describes the extensions to the base WS-Trust mechanisms to enable exchanges for negotiation and
1487 challenges.

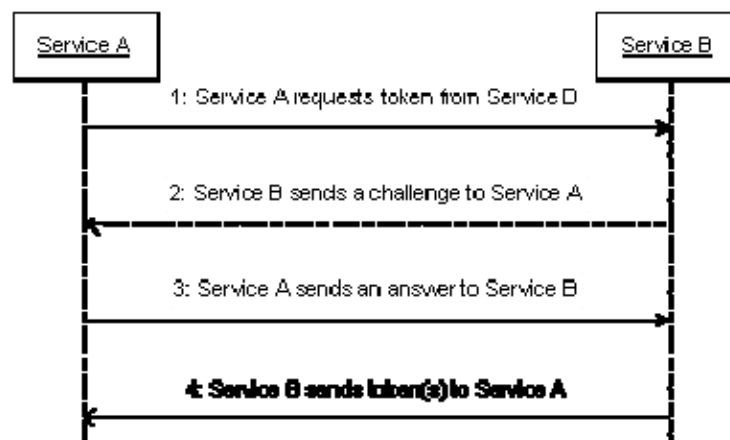
1488

1489 There are potentially different forms of exchanges, but one specific form, called "challenges", provides
1490 mechanisms in addition to those described in [WS-Security] for authentication. This section describes
1491 how general exchanges are issued and responded to within this framework. Other types of exchanges
1492 include, but are not limited to, negotiation, tunneling of hardware-based processing, and tunneling of
1493 legacy protocols.

1494

1495 The process is straightforward (illustrated here using a challenge):

1496



1497

- 1498 1. A requestor sends, for example, a `<wst:RequestSecurityToken>` message with a
1499 timestamp.
- 1500 2. The recipient does not trust the timestamp and issues a
1501 `<wst:RequestSecurityTokenResponse>` message with an embedded challenge.
- 1502 3. The requestor sends a `<wst:RequestSecurityTokenResponse>` message with an answer to
1503 the challenge.
- 1504 4. The recipient issues a `<wst:RequestSecurityTokenResponseCollection>` message with
1505 the issued security token and OPTIONAL proof-of-possession token.

1506

1507 It should be noted that the requestor might challenge the recipient in either step 1 or step 3. In which
1508 case, step 2 or step 4 contains an answer to the initiator's challenge. Similarly, it is possible that steps 2
1509 and 3 could iterate multiple times before the process completes (step 4).

1510

1511 The two services can use [WS-SecurityPolicy] to state their requirements and preferences for security
1512 tokens and encryption and signing algorithms (general policy intersection). This section defines
1513 mechanisms for legacy and more sophisticated types of negotiations.

1514 8.1 Negotiation and Challenge Framework

1515 The general mechanisms defined for requesting and returning security tokens are extensible. This
1516 section describes the general model for extending these to support negotiations and challenges.

1517

1518 The exchange model is as follows:

- 1519 1. A request is initiated with a `<wst:RequestSecurityToken>` that identifies the details of the
1520 request (and MAY contain initial negotiation/challenge information)
- 1521 2. A response is returned with a `<wst:RequestSecurityTokenResponse>` that contains
1522 additional negotiation/challenge information. Optionally, this MAY return token information in the
1523 form of a `<wst:RequestSecurityTokenResponseCollection>` (if the exchange is two legs
1524 long).
- 1525 3. If the exchange is not complete, the requestor uses a
1526 `<wst:RequestSecurityTokenResponse>` that contains additional negotiation/challenge
1527 information.
- 1528 4. The process repeats at step 2 until the negotiation/challenge is complete (a token is returned or a
1529 Fault occurs). In the case where token information is returned in the final leg, it is returned in the
1530 form of a `<wst:RequestSecurityTokenResponseCollection>`.

1531

1532 The negotiation/challenge information is passed in binding/profile-specific elements that are placed inside
1533 of the `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>` elements.

1534

1535 It is RECOMMENDED that at least the `<wsu:Timestamp>` element be included in messages (as per
1536 [WS-Security]) as a way to ensure freshness of the messages in the exchange. Other types of
1537 challenges MAY also be included. For example, a `<wsp:Policy>` element may be used to negotiate
1538 desired policy behaviors of both parties. Multiple challenges and responses MAY be included.

1539 8.2 Signature Challenges

1540 Exchange requests are issued by including an element that describes the exchange (e.g. challenge) and
1541 responses contain an element describing the response. For example, signature challenges are
1542 processed using the `<wst:SignChallenge>` element. The response is returned in a
1543 `<wst:SignChallengeResponse>` element. Both the challenge and the response elements are
1544 specified within the `<wst:RequestSecurityTokenResponse>` element. Some forms of negotiation
1545 MAY specify challenges along with responses to challenges from the other party. It should be noted that
1546 the requestor MAY provide exchange information (e.g. a challenge) to the recipient in the initial request.
1547 Consequently, these elements are also allowed within a `<wst:RequestSecurityToken>` element.

1548

1549 The syntax of these elements is as follows:

```
1550 <wst:SignChallenge xmlns:wst="...">  
1551   <wst:Challenge ...>...</wst:Challenge>  
1552 </wst:SignChallenge>
```

1553

```
1554 <wst:SignChallengeResponse xmlns:wst="...">  
1555   <wst:Challenge ...>...</wst:Challenge>  
1556 </wst:SignChallengeResponse>
```

1557

1558 The following describes the attributes and tags listed in the schema above:

1559 *.../wst:SignChallenge*

1560 This OPTIONAL element describes a challenge that requires the other party to sign a specified
1561 set of information.

1562 *.../wst:SignChallenge/wst:Challenge*

1563 This REQUIRED string element describes the value to be signed. In order to prevent certain
1564 types of attacks (such as man-in-the-middle), it is strongly RECOMMENDED that the challenge
1565 be bound to the negotiation. For example, the challenge SHOULD track (such as using a digest
1566 of) any relevant data exchanged such as policies, tokens, replay protection, etc. As well, if the
1567 challenge is happening over a secured channel, a reference to the channel SHOULD also be
1568 included. Furthermore, the recipient of a challenge SHOULD verify that the data tracked
1569 (digested) matches their view of the data exchanged. The exact algorithm MAY be defined in
1570 profiles or agreed to by the parties.

1571 *.../SignChallenge/{any}*

1572 This is an extensibility mechanism to allow additional negotiation types to be used.

1573 *.../wst:SignChallenge/@{any}*

1574 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added
1575 to the element.

1576 *.../wst:SignChallengeResponse*

1577 This OPTIONAL element describes a response to a challenge that requires the signing of a
1578 specified set of information.

1579 *.../wst:SignChallengeResponse/wst:Challenge*

1580 If a challenge was issued, the response MUST contain the challenge element exactly as
1581 received. As well, while the RSTR response SHOULD always be signed, if a challenge was
1582 issued, the RSTR MUST be signed (and the signature coupled with the message to prevent
1583 replay).

1584 *.../wst:SignChallengeResponse/{any}*

1585 This is an extensibility mechanism to allow additional negotiation types to be used.

1586 *.../wst:SignChallengeResponse/@{any}*

1587 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added
1588 to the element.

1589 **8.3 User Interaction Challenge**

1590 User interaction challenge requests are issued by including the <InteractiveChallenge> element. The
1591 response is returned in a <InteractiveChallengeResponse> element. Both the challenge and response
1592 elements are specified within the <wst:RequestSecurityTokenResponse> element. In some instances, the
1593 requestor may issue a challenge to the recipient or provide a response to an anticipated challenge from
1594 the recipient in the initial request. Consequently, these elements are also allowed within a
1595 <wst:RequestSecurityToken> element. The challenge/response exchange between client and server
1596 MAY be iterated over multiple legs before a final response is issued.

1597 Implementations SHOULD take into account the possibility that messages in either direction may be lost
1598 or duplicated. In the absence of a lower level protocol guaranteeing delivery of every message in order
1599 and exactly once, which retains the ordering of requests and responses traveling in opposite directions,
1600 implementations SHOULD observe the following procedures:

1601 The STS SHOULD:

1602 1. Never send a new request while an existing request is pending,

- 1603 2. Timeout requests and retransmit them.
- 1604 3. Silently discard responses when no request is pending.
- 1605
- 1606 The service consumer MAY:
- 1607 1. Respond to a repeated request with the same information
- 1608 2. Retain user input until the Challenge Iteration is complete in case it is necessary to repeat the
- 1609 response.
- 1610 Note that the xml:lang attribute may be used where allowed via attribute extensibility to specify a
- 1611 language of localized elements and attributes using the language codes specified in [RFC 3066].

1612 8.3.1 Challenge Format

1613 The syntax of the user interaction challenge element is as follows:

```

1614 <wst14:InteractiveChallenge xmlns:wst14="..." ...>
1615   <wst14:Title ...> xs:string </wst14:Title> ?
1616   <wst14:TextChallenge RefId="xs:anyURI" Label="xs:string"?
1617     MaxLen="xs:int"? HideText="xs:boolean"? ...>
1618     <wst14:Image MimeType="xs:string"> xs:base64Binary </wst14:Image> ?
1619   </wst14:TextChallenge> *
1620   <wst14:ChoiceChallenge RefId="xs:anyURI" Label="xs:string"?
1621     ExactlyOne="xs:boolean"? ...>
1622     <wst14:Choice RefId="xs:anyURI" Label="xs:string"? ...>
1623       <wst14:Image MimeType="xs:string"> xs:base64Binary </wst14:Image> ?
1624     </wst14:Choice> +
1625   </wst14:ChoiceChallenge> *
1626   < wst14:ContextData RefId="xs:anyURI" > xs:any </wst14:ContextData> *
1627   ...
1628 </wst14:InteractiveChallenge>

```

1629 The following describes the attributes and elements listed in the schema outlined above:

- 1630
- 1631 *.../wst14:InteractiveChallenge*
- 1632 A container element for a challenge that requires interactive user input.
- 1633 *.../wst14:InteractiveChallenge/wst14:Title*
- 1634 An OPTIONAL element that specifies an overall title text to be displayed to the user (e.g. a title
- 1635 describing the purpose or nature of the challenge). How the preferred language of the requestor
- 1636 is communicated to the STS is left up to implementations.
- 1637 *.../wst14:InteractiveChallenge/wst14:TextChallenge*
- 1638 An OPTIONAL element that specifies a challenge that requires textual input from the user.
- 1639 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@RefId*
- 1640 A REQUIRED attribute that specifies a reference identifier for this challenge element which is
- 1641 used to correlate the corresponding element in the response to the challenge.
- 1642 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@MaxLen*
- 1643 An OPTIONAL attribute that specifies the maximum length of the text string that is sent as the
- 1644 response to this text challenge. This value serves as a hint for the user interface software at the
- 1645 requestor which manifests the end-user experience for this challenge.
- 1646 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@HideText*
- 1647 An OPTIONAL attribute that specifies that the response to this text challenge MUST receive
- 1648 treatment as hidden text in any user interface. For example, the text entry may be displayed as a

1649 series of asterisks in the user interface. This attribute serves as a hint for the user interface
1650 software at the requestor which manifests the end-user experience for this challenge.

1651 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@Label*

1652 An OPTIONAL attribute that specifies a label for the text challenge item (e.g. a label for a text
1653 entry field) which will be shown to the user. How the preferred language of the requestor is
1654 communicated to the STS is left up to implementations.

1655 *.../wst14:InteractiveChallenge/wst14:TextChallenge/Image*

1656 An OPTIONAL element that contains a base64 encoded inline image specific to the text
1657 challenge item to be shown to the user (e.g. an image that the user must see to respond
1658 successfully to the challenge). The image presented is intended as an additional label to a
1659 challenge element which could be CAPTCHA, selection of a previously established image secret
1660 or any other means by which images can be used to challenge a user to interact in a way to
1661 satisfy a challenge.

1662 *.../wst14:InteractiveChallenge/wst14:TextChallenge/Image/@MimeType*

1663 A REQUIRED attribute that specifies a MIME type (e.g., image/gif, image/jpg) indicating the
1664 format of the image.

1665 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge*

1666 An OPTIONAL element that specifies a challenge that requires a choice among multiple items by
1667 the user.

1668 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/@RefId*

1669 A REQUIRED attribute that specifies a reference identifier for this challenge element which is
1670 used to correlate the corresponding element in the response to the challenge.

1671 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/@Label*

1672 An OPTIONAL attribute that specifies a title label for the choice challenge item (e.g., a text
1673 header describing the list of choices as a whole) which will be shown to the user. How the
1674 preferred language of the requestor is communicated to the STS is left up to implementations.

1675 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/@ExactlyOne*

1676 An OPTIONAL attribute that specifies if exactly once choice must be selected by the user from
1677 among the child element choices. The absence of this attribute implies the value "false" which
1678 means multiple choices can be selected.

1679 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice*

1680 A REQUIRED element that specifies a single choice item within the choice challenge.

1681 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/@RefId*

1682 A REQUIRED attribute that specifies a reference identifier for this specific choice item which is
1683 used to correlate the corresponding element in the response to the challenge.

1684 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/@Label*

1685 An OPTIONAL attribute that specifies a text label for the choice item (e.g., text describing the
1686 individual choice) which will be shown to the user. How the preferred language of the requestor is
1687 communicated to the STS is left up to implementations.

1688 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/wst14:Image*

1689 An OPTIONAL element that contains a base64 encoded inline image specific to the choice item
1690 to be shown to the user (e.g. an image that the user must see to respond successfully to the
1691 challenge). The image presented is intended as an additional label to a challenge element which
1692 could be CAPTCHA, selection of a previously established image secret or any other means by
1693 which images can be used to challenge a user to interact in a way to satisfy a challenge.

1694 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/wst14:Image/@MimeType*
1695 A REQUIRED attribute that specifies a MIME type (e.g., image/gif, image/jpg) indicating the
1696 format of the image.

1697 *.../wst14:InteractiveChallenge/wst14:ContextData*
1698 An OPTIONAL element that specifies a value that MUST be reflected back in the response to the
1699 challenge (e.g., cookie). The element may contain any value. The actual content is opaque to the
1700 requestor; it is not required to understand its structure or semantics. This can be used by an STS,
1701 for instance, to store information between the challenge/response exchanges that would
1702 otherwise be lost if the STS were to remain stateless.

1703 *.../wst14:InteractiveChallenge/wst14:ContextData/@RefId*
1704 A REQUIRED attribute that specifies a reference identifier for this context element which is used
1705 to correlate the corresponding element in the response to the challenge.

1706 *.../wst14:InteractiveChallenge/{any}*
1707 This is an extensibility mechanism to allow additional elements to be specified.

1708 *.../wst14:InteractiveChallenge/@{any}*
1709 This is an extensibility mechanism to allow additional attributes to be specified.

1710

1711 The syntax of the user interaction challenge response element is as follows:

```
1712 <wst14:InteractiveChallengeResponse xmlns:wst14="..." ...>  
1713 <wst14:TextChallengeResponse RefId="xs:anyURI" ...>  
1714   xs:string  
1715 </wst14:TextChallengeResponse> *  
1716 <wst14:ChoiceChallengeResponse RefId="xs:anyURI"> *  
1717   <wst14:ChoiceSelected RefId="xs:anyURI" /> *  
1718 </wst14:ChoiceChallengeResponse>  
1719 <wst14:ContextData RefId="xs:anyURI"> xs:any </wst14:ContextData> *  
1720   ...  
1721 </wst14:InteractiveChallengeResponse>
```

1722 The following describes the attributes and elements listed in the schema outlined above:

1723

1724 *.../wst14:InteractiveChallengeResponse*
1725 A container element for the response to a challenge that requires interactive user input.

1726 *.../wst14:InteractiveChallengeResponse/wst14:TextChallengeResponse*
1727 This element value contains the user input as the response to the original text challenge issued.

1728 *.../wst14:InteractiveChallengeResponse/wst14:TextChallengeResponse/@RefId*
1729 A required attribute that specifies the identifier for the text challenge element in the original
1730 challenge which can be used for correlation.

1731 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse*
1732 A container element for the response to a choice challenge.

1733 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse/@RefId*
1734 A required attribute that specifies the reference identifier for the choice challenge element in the
1735 original challenge which can be used for correlation.

1736 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse/wst14:ChoiceSelected*
1737 A required element that specifies a choice item selected by the user from the choice challenge.

1738 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse/wst14:ChoiceSelected/@RefId*

1739 A required attribute that specifies the reference identifier for the choice item in the original choice
1740 challenge which can be used for correlation.

1741 *.../wst14:InteractiveChallengeResponse/wst14:ContextData*

1742 An optional element that carries a context data item from the original challenge that is simply
1743 reflected back.

1744 *.../wst14:InteractiveChallengeResponse/wst14:ContextData/@RefId*

1745 A required attribute that specifies the reference identifier for the context data element in the
1746 original challenge which can be used for correlation.

1747 *.../wst14:InteractiveChallengeResponse/{any}*

1748 This is an extensibility mechanism to allow additional elements to be specified.

1749 *.../wst14:InteractiveChallengeResponse/@{any}*

1750 This is an extensibility mechanism to allow additional attributes to be specified.

1751 In order to prevent certain types of attacks, such as man-in-the-middle or replay of response, the
1752 challenge SHOULD be bound to the response. For example, an STS may use the <ContextData>
1753 element in the challenge to include a digest of any relevant replay protection data and verify that the
1754 same data is reflected back by the requestor.

1755 Text provided by the STS which is intended for display SHOULD NOT contain script, markup or other
1756 unprintable characters. Image data provided by the STS SHOULD NOT contain imbedded commands or
1757 other content except an image to be displayed.

1758 Service consumers MUST ignore any script, markup or other unprintable characters when displaying text
1759 sent by the STS. Service consumers MUST insure that image data does not contain imbedded
1760 commands or other content before displaying the image.

1761 **8.3.2 PIN and OTP Challenges**

1762 In some situations, some additional authentication step may be required, but the Consumer cannot
1763 determine this in advance of making the request. Two common cases that require user interaction are:

- 1764 • a challenge for a secret PIN,
- 1765 • a challenge for a one-time-password (OTP).

1766

1767 This challenge may be issued by an STS using the “text challenge” format within a user interaction
1768 challenge specified in the section above. A requestor responds to the challenge with the PIN/OTP value
1769 along with the corresponding @RefId attribute value for the text challenge which is used by the STS to
1770 correlate the response to the original challenge. This pattern of exchange requires that the requestor
1771 must receive the challenge first and thus learn the @RefId attribute value to include in the response.

1772

1773 There are cases where a requestor may know a priori that the STS challenges for a single PIN/OTP and,
1774 as an optimization, provide the response to the anticipated challenge in the initial request. The following
1775 distinguished URIs are defined for use as the value of the @RefId attribute of a
1776 <TextChallengeResponse> element to represent PIN and OTP responses using the optimization pattern.

1777

1778 <http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN>
1779 <http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/OTP>

1780

1781 An STS may choose not to support the optimization pattern above for PIN/OTP response. In some cases,
1782 an OTP challenge from the STS may include a dynamic random value that the requestor must feed into
1783 the OTP generating module before an OTP response is computed. In such cases, the optimized response
1784 pattern may not be usable.

1785 8.4 Binary Exchanges and Negotiations

1786 Exchange requests MAY also utilize existing binary formats passed within the WS-Trust framework. A
1787 generic mechanism is provided for this that includes a URI attribute to indicate the type of binary
1788 exchange.

1789

1790 The syntax of this element is as follows:

```
1791 <wst:BinaryExchange ValueType="..." EncodingType="..." xmlns:wst="...">  
1792 </wst:BinaryExchange>
```

1793 The following describes the attributes and tags listed in the schema above (note that the ellipses below
1794 indicate that this element MAY be placed in different containers. For this specification, these are limited
1795 to <wst:RequestSecurityToken> and <wst:RequestSecurityTokenResponse>):

1796 *.../wst:BinaryExchange*

1797 This OPTIONAL element is used for a security negotiation that involves exchanging binary blobs
1798 as part of an existing negotiation protocol. The contents of this element are blob-type-specific
1799 and are encoded using base64 (unless otherwise specified).

1800 *.../wst:BinaryExchange/@ValueType*

1801 This REQUIRED attribute specifies a URI to identify the type of negotiation (and the value space
1802 of the blob – the element's contents).

1803 *.../wst:BinaryExchange/@EncodingType*

1804 This REQUIRED attribute specifies a URI to identify the encoding format (if different from base64)
1805 of the negotiation blob. Refer to [WS-Security] for sample encoding format URIs.

1806 *.../wst:BinaryExchange/@{any}*

1807 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added
1808 to the element.

1809 Some binary exchanges result in a shared state/context between the involved parties. It is
1810 RECOMMENDED that at the conclusion of the exchange, a new token and proof-of-possession token be
1811 returned. A common approach is to use the negotiated key as a "secure channel" mechanism to secure
1812 the new token and proof-of-possession token.

1813 For example, an exchange might establish a shared secret S_x that can then be used to sign the final
1814 response and encrypt the proof-of-possession token.

1815 8.5 Key Exchange Tokens

1816 In some cases it MAY be necessary to provide a key exchange token so that the other party (either
1817 requestor or issuer) can provide entropy or key material as part of the exchange. Challenges MAY NOT
1818 always provide a usable key as the signature may use a signing-only certificate.

1819

1820 The section describes two OPTIONAL elements that can be included in RST and RSTR elements to
1821 indicate that a Key Exchange Token (KET) is desired, or to provide a KET.

1822 The syntax of these elements is as follows (Note that the ellipses below indicate that this element MAY be
1823 placed in different containers. For this specification, these are limited to
1824 <wst:RequestSecurityToken> and <wst:RequestSecurityTokenResponse>):

```
1825 <wst:RequestKET xmlns:wst="..." />
```

1826

1827 `<wst:KeyExchangeToken xmlns:wst="...">...</wst:KeyExchangeToken>`

1828

1829 The following describes the attributes and tags listed in the schema above:

1830 `.../wst:RequestKET`

1831 This OPTIONAL element is used to indicate that the receiving party (either the original requestor
1832 or issuer) SHOULD provide a KET to the other party on the next leg of the exchange.

1833 `.../wst:KeyExchangeToken`

1834 This OPTIONAL element is used to provide a key exchange token. The contents of this element
1835 either contain the security token to be used for key exchange or a reference to it.

1836 8.6 Custom Exchanges

1837 Using the extensibility model described in this specification, any custom XML-based exchange can be
1838 defined in a separate binding/profile document. In such cases elements are defined which are carried in
1839 the RST and RSTR elements.

1840

1841 It should be noted that it is NOT REQUIRED that exchange elements be symmetric. That is, a specific
1842 exchange mechanism MAY use multiple elements at different times, depending on the state of the
1843 exchange.

1844 8.7 Signature Challenge Example

1845 Here is an example exchange involving a signature challenge. In this example, a service requests a
1846 custom token using a X.509 certificate for authentication. The issuer uses the exchange mechanism to
1847 challenge the requestor to sign a random value (to ensure message freshness). The requestor provides
1848 a signature of the requested data and, once validated, the issuer then issues the requested token.

1849

1850 The first message illustrates the initial request that is signed with the private key associated with the
1851 requestor's X.509 certificate:

```
1852 <S11:Envelope xmlns:S11="..." xmlns:wsse="..."  
1853     xmlns:wsu="..." xmlns:wst="...">  
1854   <S11:Header>  
1855     ...  
1856     <wsse:Security>  
1857       <wsse:BinarySecurityToken  
1858         wsu:Id="reqToken"  
1859         ValueType="...X509v3">  
1860         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
1861       </wsse:BinarySecurityToken>  
1862       <ds:Signature xmlns:ds="...">  
1863         ...  
1864         <ds:KeyInfo>  
1865           <wsse:SecurityTokenReference>  
1866             <wsse:Reference URI="#reqToken"/>  
1867           </wsse:SecurityTokenReference>  
1868         </ds:KeyInfo>  
1869       </ds:Signature>  
1870     </wsse:Security>  
1871     ...  
1872   </S11:Header>  
1873   <S11:Body>  
1874     <wst:RequestSecurityToken>  
1875       <wst:TokenType>
```

```

1876         http://example.org/mySpecialToken
1877     </wst:TokenType>
1878     <wst:RequestType>
1879         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1880     </wst:RequestType>
1881 </wst:RequestSecurityToken>
1882 </S11:Body>
1883 </S11:Envelope>

```

1884

1885 The issuer (recipient) service doesn't trust the sender's timestamp (or one wasn't specified) and issues a
1886 challenge using the exchange framework defined in this specification. This message is signed using the
1887 private key associated with the issuer's X.509 certificate and contains a random challenge that the
1888 requestor must sign:

```

1889 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
1890     xmlns:wst="...">
1891   <S11:Header>
1892     ...
1893     <wsse:Security>
1894       <wsse:BinarySecurityToken
1895         wsu:Id="issuerToken"
1896         ValueType="...X509v3">
1897         DFJHuedsujfnrnv45JZc0...
1898       </wsse:BinarySecurityToken>
1899       <ds:Signature xmlns:ds="...">
1900         ...
1901       </ds:Signature>
1902     </wsse:Security>
1903     ...
1904   </S11:Header>
1905   <S11:Body>
1906     <wst:RequestSecurityTokenResponse>
1907       <wst:SignChallenge>
1908         <wst:Challenge>Huehf...</wst:Challenge>
1909       </wst:SignChallenge>
1910     </wst:RequestSecurityTokenResponse>
1911   </S11:Body>
1912 </S11:Envelope>

```

1913

1914 The requestor receives the issuer's challenge and issues a response that is signed using the requestor's
1915 X.509 certificate and contains the challenge. The signature only covers the non-mutable elements of the
1916 message to prevent certain types of security attacks:

```

1917 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
1918     xmlns:wst="...">
1919   <S11:Header>
1920     ...
1921     <wsse:Security>
1922       <wsse:BinarySecurityToken
1923         wsu:Id="reqToken"
1924         ValueType="...X509v3">
1925         MIEZzCCA9CgAwIBAgIQEmtJZc0...
1926       </wsse:BinarySecurityToken>
1927       <ds:Signature xmlns:ds="...">
1928         ...
1929       </ds:Signature>
1930     </wsse:Security>
1931     ...
1932   </S11:Header>
1933   <S11:Body>
1934     <wst:RequestSecurityTokenResponse>

```

```

1935         <wst:SignChallengeResponse>
1936             <wst:Challenge>Huehf...</wst:Challenge>
1937         </wst:SignChallengeResponse>
1938     </wst:RequestSecurityTokenResponse>
1939 </S11:Body>
1940 </S11:Envelope>

```

1941

1942 The issuer validates the requestor's signature responding to the challenge and issues the requested

1943 token(s) and the associated proof-of-possession token. The proof-of-possession token is encrypted for

1944 the requestor using the requestor's public key.

```

1945 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
1946     xmlns:wst="..." xmlns:xenc="...">
1947   <S11:Header>
1948     ...
1949     <wsse:Security>
1950       <wsse:BinarySecurityToken
1951         wsu:Id="issuerToken"
1952         ValueType="...X509v3">
1953         DFJHuedsujfnrnv45JZc0...
1954       </wsse:BinarySecurityToken>
1955       <ds:Signature xmlns:ds="...">
1956         ...
1957       </ds:Signature>
1958     </wsse:Security>
1959     ...
1960   </S11:Header>
1961   <S11:Body>
1962     <wst:RequestSecurityTokenResponseCollection>
1963       <wst:RequestSecurityTokenResponse>
1964         <wst:RequestedSecurityToken>
1965           <xyz:CustomToken xmlns:xyz="...">
1966             ...
1967           </xyz:CustomToken>
1968         </wst:RequestedSecurityToken>
1969         <wst:RequestedProofToken>
1970           <xenc:EncryptedKey Id="newProof">
1971             ...
1972           </xenc:EncryptedKey>
1973         </wst:RequestedProofToken>
1974       </wst:RequestSecurityTokenResponse>
1975     </wst:RequestSecurityTokenResponseCollection>
1976   </S11:Body>
1977 </S11:Envelope>

```

1978 8.8 Challenge Examples

1979 8.8.1 Text and choice challenge

1980 Here is an example of a user interaction challenge using both text and choice challenges. In this example,

1981 a user requests a custom token using a username/password for authentication. The STS uses the

1982 challenge mechanism to challenge the user for additional information in the form of a secret question (i.e.,

1983 Mother's maiden name) and an age group choice. The challenge additionally includes one contextual

1984 data item that needs to be reflected back in the response. The user interactively provides the requested

1985 data and, once validated, the STS issues the requested token. All messages are sent over a protected

1986 transport using SSLv3.

1987

1988 The requestor sends the initial request that includes the username/password for authentication as follows.

1989

1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007

```
<S11:Envelope ...>
  <S11:Header>
    ...
    <wsse:Security>
      <wsse:UsernameToken>
        <wsse:Username>Zoe</wsse:Username>
        <wsse:Password
          Type="http://...#PasswordText">ILoveDogs</wsse:Password>
        </wsse:UsernameToken>
      </wsse:Security>
    </S11:Header>
    <S11:Body>
      <wst:RequestSecurityToken>
        <wst:TokenType>http://example.org/customToken</wst:TokenType>
        <wst:RequestType>...</wst:RequestType>
      </wst:RequestSecurityToken>
    </S11:Body>
  </S11:Envelope>
```

2008

2009 The STS issues a challenge for additional information using the user interaction challenge mechanism as
2010 follows.

2011

2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037

```
<S11:Envelope ...>
  <S11:Header>
    ...
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityTokenResponse>
      <wst14:InteractiveChallenge xmlns:wst14="..." >
        <wst14:Title>
          Please answer the following additional questions to login.
        </wst14:Title>
        <wst14:TextChallenge RefId=http://.../ref#text1
          Label="Mother's Maiden Name" MaxLen=80 />
        <wst14:ChoiceChallenge RefId="http://.../ref#choiceGroupA"
          Label="Your Age Group:" ExactlyOne="true">
          <wst14:Choice RefId="http://.../ref#choice1" Label="18-30" />
          <wst14:Choice RefId="http://.../ref#choice2" Label="31-40" />
          <wst14:Choice RefId="http://.../ref#choice3" Label="41-50" />
          <wst14:Choice RefId="http://.../ref#choice4" Label="50+" />
        </wst14:ChoiceChallenge>
        <wst14:ContextData RefId="http://.../ref#cookie1">
          ...
        </wst14:ContextData>
      </wst14:InteractiveChallenge>
    </wst:RequestSecurityTokenResponse>
  </S11:Body>
</S11:Envelope>
```

2038

2039 The requestor receives the challenge, provides the necessary user experience for soliciting the required
2040 inputs, and sends a response to the challenge back to the STS as follows.

2041

2042
2043
2044
2045
2046
2047
2048

```
<S11:Envelope ...>
  <S11:Header>
    ...
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityTokenResponse>
      <wst14:InteractiveChallengeResponse xmlns:wst14="..." >
```

```

2049     <wst14:TextChallengeResponse RefId="http://.../ref#text1">
2050         Goldstein
2051     </wst14:TextChallengeResponse>
2052     <wst14:ChoiceChallengeResponse RefId="http://.../ref#choiceGroupA">
2053         <wst14:ChoiceSelected RefId="http://.../ref#choice3" />
2054     </wst14:ChoiceChallengeResponse>
2055     <wst14:ContextData RefId="http://.../ref#cookie1">
2056         ...
2057     </wst14:ContextData>
2058     </wst14:InteractiveChallengeResponse>
2059 </wst:RequestSecurityTokenResponse>
2060 </S11:Body>
2061 </S11:Envelope>

```

2062

2063 The STS validates the response containing the inputs from the user, and issues the requested token as follows.

2064

```

2065
2066 <S11:Envelope ...>
2067   <S11:Header>
2068     ...
2069   </S11:Header>
2070   <S11:Body>
2071     <wst:RequestSecurityTokenResponseCollection>
2072       <wst:RequestSecurityTokenResponse>
2073         <wst:RequestedSecurityToken>
2074           <xyz:CustomToken xmlns:xyz="...">
2075             ...
2076           </xyz:CustomToken>
2077         </wst:RequestedSecurityToken>
2078         <wst:RequestedProofToken>
2079           ...
2080         </wst:RequestedProofToken>
2081       </wst:RequestSecurityTokenResponse>
2082     </wst:RequestSecurityTokenResponseCollection>
2083   </S11:Body>
2084 </S11:Envelope>

```

2085

2086 8.8.2 PIN challenge

2087 Here is an example of a user interaction challenge using a text challenge for a secret PIN. In this
 2088 example, a user requests a custom token using a username/password for authentication. The STS uses
 2089 the text challenge mechanism for an additional PIN. The user interactively provides the PIN and, once
 2090 validated, the STS issues the requested token. All messages are sent over a protected transport using
 2091 SSLv3.

2092

2093 The requestor sends the initial request that includes the username/password for authentication as follows.

2094

```

2095 <S11:Envelope ...>
2096   <S11:Header>
2097     ...
2098   <wsse:Security>
2099     <wsse:UsernameToken>
2100       <wsse:Username>Zoe</wsse:Username>
2101       <wsse:Password Type="http://...#PasswordText">
2102         ILoveDogs
2103     </wsse:Password>

```

```
2104     </wsse:UsernameToken>
2105     </wsse:Security>
2106   </S11:Header>
2107   <S11:Body>
2108     <wst:RequestSecurityToken>
2109       <wst:TokenType>http://example.org/customToken</wst:TokenType>
2110       <wst:RequestType>...</wst:RequestType>
2111     </wst:RequestSecurityToken>
2112   </S11:Body>
2113 </S11:Envelope>
```

2114

2115 The STS issues a challenge for a secret PIN using the text challenge mechanism as follows.

2116

```
2117 <S11:Envelope ...>
2118   <S11:Header>
2119     ...
2120   </S11:Header>
2121   <S11:Body>
2122     <wst:RequestSecurityTokenResponse>
2123       <wst14:InteractiveChallenge xmlns:wst14="..." >
2124         <wst14:TextChallenge
2125           RefId="http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN"
2126           Label="Please enter your PIN" />
2127         </wst14:TextChallenge>
2128       </wst14:InteractiveChallenge>
2129     </wst:RequestSecurityTokenResponse>
2130   </S11:Body>
2131 </S11:Envelope>
```

2132

2133 The requestor receives the challenge, provides the necessary user experience for soliciting the PIN, and
2134 sends a response to the challenge back to the STS as follows.

2135

```
2136 <S11:Envelope ...>
2137   <S11:Header>
2138     ...
2139   </S11:Header>
2140   <S11:Body>
2141     <wst:RequestSecurityTokenResponse>
2142       <wst14:InteractiveChallengeResponse xmlns:wst14="..." >
2143         <wst14:TextChallengeResponse
2144           RefId="http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN">
2145           9988
2146         </wst14:TextChallengeResponse>
2147       </wst14:InteractiveChallengeResponse>
2148     </wst:RequestSecurityTokenResponse>
2149   </S11:Body>
2150 </S11:Envelope>
```

2151

2152 The STS validates the PIN response, and issues the requested token as follows.

2153

```
2154 <S11:Envelope ...>
2155   <S11:Header>
2156     ...
2157   </S11:Header>
2158   <S11:Body>
2159     <wst:RequestSecurityTokenResponseCollection>
```

```

2160 <wst:RequestSecurityTokenResponse>
2161   <wst:RequestedSecurityToken>
2162     <xyz:CustomToken xmlns:xyz="...">
2163       ...
2164     </xyz:CustomToken>
2165   </wst:RequestedSecurityToken>
2166   <wst:RequestedProofToken>
2167     ...
2168   </wst:RequestedProofToken>
2169 </wst:RequestSecurityTokenResponse>
2170 </wst:RequestSecurityTokenResponseCollection>
2171 </S11:Body>
2172 </S11:Envelope>

```

2173

2174 8.8.3 PIN challenge with optimized response

2175 The following example illustrates using the optimized PIN response pattern for the same exact challenge
 2176 as in the previous section. This reduces the number of message exchanges to two instead of four. All
 2177 messages are sent over a protected transport using SSLv3.

2178

2179 The requestor sends the initial request that includes the username/password for authentication as well as
 2180 the response to the anticipated PIN challenge as follows.

2181

```

2182 <S11:Envelope ...>
2183   <S11:Header>
2184     ...
2185   <wsse:Security>
2186     <wsse:UsernameToken>
2187       <wsse:Username>Zoe</wsse:Username>
2188       <wsse:Password Type="http://...#PasswordText">
2189         ILoveDogs
2190       </wsse:Password>
2191     </wsse:UsernameToken>
2192   </wsse:Security>
2193 </S11:Header>
2194 <S11:Body>
2195   <wst:RequestSecurityToken>
2196     <wst:TokenType>http://example.org/customToken</wst:TokenType>
2197     <wst:RequestType>...</wst:RequestType>
2198     <wst14:InteractiveChallengeResponse xmlns:wst14="..." >
2199       <wst14:TextChallengeResponse
2200         RefId="http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN">
2201         9988
2202       </wst14:TextChallengeResponse>
2203     </wst14:InteractiveChallengeResponse>
2204   </wst:RequestSecurityToken>
2205 </S11:Body>
2206 </S11:Envelope>

```

2207

2208 The STS validates the authentication credential as well as the optimized PIN response, and issues the
 2209 requested token as follows.

2210

```

2211 <S11:Envelope ...>
2212   <S11:Header>
2213     ...
2214   </S11:Header>

```

```

2215 <S11:Body>
2216   <wst:RequestSecurityTokenResponseCollection>
2217     <wst:RequestSecurityTokenResponse>
2218       <wst:RequestedSecurityToken>
2219         <xyz:CustomToken xmlns:xyz="...">
2220           ...
2221         </xyz:CustomToken>
2222       </wst:RequestedSecurityToken>
2223     <wst:RequestedProofToken>
2224       ...
2225     </wst:RequestedProofToken>
2226   </wst:RequestSecurityTokenResponse>
2227 </wst:RequestSecurityTokenResponseCollection>
2228 </S11:Body>
2229 </S11:Envelope>

```

2230

2231 8.9 Custom Exchange Example

2232 Here is another illustrating the syntax for a token request using a custom XML exchange. For brevity,
 2233 only the RST and RSTR elements are illustrated. Note that the framework allows for an arbitrary number
 2234 of exchanges, although this example illustrates the use of four legs. The request uses a custom
 2235 exchange element and the requestor signs only the non-mutable element of the message:

```

2236   <wst:RequestSecurityToken xmlns:wst="...">
2237     <wst:TokenType>
2238       http://example.org/mySpecialToken
2239     </wst:TokenType>
2240     <wst:RequestType>
2241       http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
2242     </wst:RequestType>
2243     <xyz:CustomExchange xmlns:xyz="...">
2244       ...
2245     </xyz:CustomExchange>
2246   </wst:RequestSecurityToken>

```

2247

2248 The issuer service (recipient) responds with another leg of the custom exchange and signs the response
 2249 (non-mutable aspects) with its token:

```

2250   <wst:RequestSecurityTokenResponse xmlns:wst="...">
2251     <xyz:CustomExchange xmlns:xyz="...">
2252       ...
2253     </xyz:CustomExchange>
2254   </wst:RequestSecurityTokenResponse>

```

2255

2256 The requestor receives the issuer's exchange and issues a response that is signed using the requestor's
 2257 token and continues the custom exchange. The signature covers all non-mutable aspects of the
 2258 message to prevent certain types of security attacks:

```

2259   <wst:RequestSecurityTokenResponse xmlns:wst="...">
2260     <xyz:CustomExchange xmlns:xyz="...">
2261       ...
2262     </xyz:CustomExchange>
2263   </wst:RequestSecurityTokenResponse>

```

2264

2265 The issuer processes the exchange and determines that the exchange is complete and that a token
 2266 should be issued. Consequently it issues the requested token(s) and the associated proof-of-possession
 2267 token. The proof-of-possession token is encrypted for the requestor using the requestor's public key.

```

2268 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
2269   <wst:RequestSecurityTokenResponse>
2270     <wst:RequestedSecurityToken>
2271       <xyz:CustomToken xmlns:xyz="...">
2272         ...
2273       </xyz:CustomToken>
2274     </wst:RequestedSecurityToken>
2275     <wst:RequestedProofToken>
2276       <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
2277         ...
2278       </xenc:EncryptedKey>
2279     </wst:RequestedProofToken>
2280     <wst:RequestedProofToken>
2281       <xenc:EncryptedKey xmlns:xenc="...">...</xenc:EncryptedKey>
2282     </wst:RequestedProofToken>
2283   </wst:RequestSecurityTokenResponse>
2284 </wst:RequestSecurityTokenResponseCollection>
  
```

2285 It should be noted that other example exchanges include the issuer returning a final custom exchange
 2286 element, and another example where a token isn't returned.

2287 8.10 Protecting Exchanges

2288 There are some attacks, such as forms of man-in-the-middle, that can be applied to token requests
 2289 involving exchanges. It is RECOMMENDED that the exchange sequence be protected. This MAY be
 2290 built into the exchange messages, but if metadata is provided in the RST or RSTR elements, then it is
 2291 subject to attack.

2292
 2293 Consequently, it is RECOMMENDED that keys derived from exchanges be linked cryptographically to the
 2294 exchange. For example, a hash can be computed by computing the SHA1 of the exclusive
 2295 canonicalization [XML-C14N] of all RST and RSTR elements in messages exchanged. This value can
 2296 then be combined with the exchanged secret(s) to create a new master secret that is bound to the data
 2297 both parties sent/received.

2298
 2299 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: $H = \text{SHA1}(\text{ExclC14N}(\text{RST} \dots \text{RSTRs}))$ $X = \text{encrypting } H \text{ using negotiated key and mechanism}$ $\text{Key} = \text{P_SHA1}(X, H + \text{"CK-HASH"})$ The octets for the "CK-HASH" string are the UTF-8 octets.

2300 8.11 Authenticating Exchanges

2301 After an exchange both parties have a shared knowledge of a key (or keys) that can then be used to
 2302 secure messages. However, in some cases it may be desired to have the issuer prove to the requestor

2303 that it knows the key (and that the returned metadata is valid) prior to the requestor using the data.
2304 However, until the exchange is actually completed it MAY be (and is often) inappropriate to use the
2305 computed keys. As well, using a token that hasn't been returned to secure a message may complicate
2306 processing since it crosses the boundary of the exchange and the underlying message security. This
2307 means that it MAY NOT be appropriate to sign the final leg of the exchange using the key derived from
2308 the exchange.

2309
2310 For this reason an authenticator is defined that provides a way for the issuer to verify the hash as part of
2311 the token issuance. Specifically, when an authenticator is returned, the
2312 `<wst:RequestSecurityTokenResponseCollection>` element is returned. This contains one
2313 RSTR with the token being returned as a result of the exchange and a second RSTR that contains the
2314 authenticator (this order SHOULD be used). When an authenticator is used, RSTRs MUST use the
2315 `@Context` element so that the authenticator can be correlated to the token issuance. The authenticator is
2316 separated from the RSTR because otherwise computation of the RST/RSTR hash becomes more
2317 complex. The authenticator is represented using the `<wst:Authenticator>` element as illustrated
2318 below:

```
2319 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2320   <wst:RequestSecurityTokenResponse Context="...">  
2321     ...  
2322   </wst:RequestSecurityTokenResponse>  
2323   <wst:RequestSecurityTokenResponse Context="...">  
2324     <wst:Authenticator>  
2325       <wst:CombinedHash>...</wst:CombinedHash>  
2326     ...  
2327   </wst:Authenticator>  
2328   </wst:RequestSecurityTokenResponse>  
2329 </wst:RequestSecurityTokenResponseCollection>
```

2330
2331 The following describes the attributes and elements listed in the schema overview above (the ... notation
2332 below represents the path RSTRC/RSTR and is used for brevity):

2333 `.../wst:Authenticator`

2334 This OPTIONAL element provides verification (authentication) of a computed hash.

2335 `.../wst:Authenticator/wst:CombinedHash`

2336 This OPTIONAL element proves the hash and knowledge of the computed key. This is done by
2337 providing the base64 encoding of the first 256 bits of the P_SHA1 digest of the computed key and
2338 the concatenation of the hash determined for the computed key and the string "AUTH-HASH".
2339 Specifically, $P_SHA1(\textit{computed-key}, H + \textit{"AUTH-HASH"})_{0-255}$. The octets for the "AUTH-HASH"
2340 string are the UTF-8 octets.

2341
2342 This `<wst:CombinedHash>` element is OPTIONAL (and an open content model is used) to allow for
2343 different authenticators in the future.

2344

9 Key and Token Parameter Extensions

2345 This section outlines additional parameters that can be specified in token requests and responses.
2346 Typically they are used with issuance requests, but since all types of requests MAY issue security tokens
2347 they could apply to other bindings.

9.1 On-Behalf-Of Parameters

2349 In some scenarios the requestor is obtaining a token on behalf of another party. These parameters
2350 specify the issuer and original requestor of the token being used as the basis of the request. The syntax
2351 is as follows (note that the base elements described above are included here italicized for completeness):

```
2352 <wst:RequestSecurityToken xmlns:wst="...">  
2353   <wst:TokenType>...</wst:TokenType>  
2354   <wst:RequestType>...</wst:RequestType>  
2355   ...  
2356   <wst:OnBehalfOf>...</wst:OnBehalfOf>  
2357   <wst:Issuer>...</wst:Issuer>  
2358 </wst:RequestSecurityToken>
```

2359

2360 The following describes the attributes and elements listed in the schema overview above:

2361 */wst:RequestSecurityToken/wst:OnBehalfOf*

2362 This OPTIONAL element indicates that the requestor is making the request on behalf of another.
2363 The identity on whose behalf the request is being made is specified by placing a security token,
2364 <wsse:SecurityTokenReference> element, or <wsa:EndpointReference> element
2365 within the <wst:OnBehalfOf> element. The requestor MAY provide proof of possession of the
2366 key associated with the OnBehalfOf identity by including a signature in the RST security header
2367 generated using the OnBehalfOf token that signs the primary signature of the RST (i.e. endorsing
2368 supporting token concept from WS-SecurityPolicy). Additional signed supporting tokens
2369 describing the OnBehalfOf context MAY also be included within the RST security header.

2370 */wst:RequestSecurityToken/wst:Issuer*

2371 This OPTIONAL element specifies the issuer of the security token that is presented in the
2372 message. This element's type is an endpoint reference as defined in [\[WS-Addressing\]](#).

2373

2374 In the following illustrates the syntax for a proxy that is requesting a security token on behalf of another
2375 requestor or end-user.

```
2376 <wst:RequestSecurityToken xmlns:wst="...">  
2377   <wst:TokenType>...</wst:TokenType>  
2378   <wst:RequestType>...</wst:RequestType>  
2379   ...  
2380   <wst:OnBehalfOf>endpoint-reference</wst:OnBehalfOf>  
2381 </wst:RequestSecurityToken>
```

9.2 Key and Encryption Requirements

2383 This section defines extensions to the <wst:RequestSecurityToken> element for requesting specific
2384 types of keys or algorithms or key and algorithms as specified by a given policy in the return token(s). In
2385 some cases the service may support a variety of key types, sizes, and algorithms. These parameters
2386 allow a requestor to indicate its desired values. It should be noted that the issuer's policy indicates if input

2387 values must be adhered to and faults generated for invalid inputs, or if the issuer will provide alternative
2388 values in the response.

2389

2390 Although illustrated using the `<wst:RequestSecurityToken>` element, these options can also be
2391 returned in a `<wst:RequestSecurityTokenResponse>` element.

2392 The syntax for these OPTIONAL elements is as follows (note that the base elements described above are
2393 included here italicized for completeness):

```
2394 <wst:RequestSecurityToken xmlns:wst="...">  
2395   <wst:TokenType>...</wst:TokenType>  
2396   <wst:RequestType>...</wst:RequestType>  
2397   ...  
2398   <wst:AuthenticationType>...</wst:AuthenticationType>  
2399   <wst:KeyType>...</wst:KeyType>  
2400   <wst:KeySize>...</wst:KeySize>  
2401   <wst:SignatureAlgorithm>...</wst:SignatureAlgorithm>  
2402   <wst:EncryptionAlgorithm>...</wst:EncryptionAlgorithm>  
2403   <wst:CanonicalizationAlgorithm>...</wst:CanonicalizationAlgorithm>  
2404   <wst:ComputedKeyAlgorithm>...</wst:ComputedKeyAlgorithm>  
2405   <wst:Encryption>...</wst:Encryption>  
2406   <wst:ProofEncryption>...</wst:ProofEncryption>  
2407   <wst:KeyWrapAlgorithm>...</wst:KeyWrapAlgorithm>  
2408   <wst:UseKey Sig="..."> </wst:UseKey>  
2409   <wst:SignWith>...</wst:SignWith>  
2410   <wst:EncryptWith>...</wst:EncryptWith>  
2411 </wst:RequestSecurityToken>
```

2412

2413 The following describes the attributes and elements listed in the schema overview above:

2414 */wst:RequestSecurityToken/wst:AuthenticationType*

2415 This OPTIONAL URI element indicates the type of authentication desired, specified as a URI.

2416 This specification does not predefine classifications; these are specific to token services as is the
2417 relative strength evaluations. The relative assessment of strength is up to the recipient to
2418 determine. That is, requestors SHOULD be familiar with the recipient policies. For example, this
2419 might be used to indicate which of the four U.S. government authentication levels is REQUIRED.

2420 */wst:RequestSecurityToken/wst:KeyType*

2421 This OPTIONAL URI element indicates the type of key desired in the security token. The
2422 predefined values are identified in the table below. Note that some security token formats have
2423 fixed key types. It should be noted that new algorithms can be inserted by defining URIs in other
2424 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.

2425 */wst:RequestSecurityToken/wst:KeySize*

2426 This OPTIONAL integer element indicates the size of the key REQUIRED specified in number of
2427 bits. This is a request, and, as such, the requested security token is not obligated to use the
2428 requested key size. That said, the recipient SHOULD try to use a key at least as strong as the
2429 specified value if possible. The information is provided as an indication of the desired strength of
2430 the security.

2431 */wst:RequestSecurityToken/wst:SignatureAlgorithm*

2432 This OPTIONAL URI element indicates the desired signature algorithm used within the returned
2433 token. This is specified as a URI indicating the algorithm (see [XML-Signature] for typical signing
2434 algorithms).

2435 */wst:RequestSecurityToken/wst:EncryptionAlgorithm*

2436 This OPTIONAL URI element indicates the desired encryption algorithm used within the returned
2437 token. This is specified as a URI indicating the algorithm (see [XML-Encrypt] for typical
2438 encryption algorithms).

2439 */wst:RequestSecurityToken/wst:CanonicalizationAlgorithm*

2440 This OPTIONAL URI element indicates the desired canonicalization method used within the
2441 returned token. This is specified as a URI indicating the method (see [XML-Signature] for typical
2442 canonicalization methods).

2443 */wst:RequestSecurityToken/wst:ComputedKeyAlgorithm*

2444 This OPTIONAL URI element indicates the desired algorithm to use when computed keys are
2445 used for issued tokens.

2446 */wst:RequestSecurityToken/wst:Encryption*

2447 This OPTIONAL element indicates that the requestor desires any returned secrets in issued
2448 security tokens to be encrypted for the specified token. That is, so that the owner of the specified
2449 token can decrypt the secret. Normally the security token is the contents of this element but a
2450 security token reference MAY be used instead. If this element isn't specified, the token used as
2451 the basis of the request (or specialized knowledge) is used to determine how to encrypt the key.

2452 */wst:RequestSecurityToken/wst:ProofEncryption*

2453 This OPTIONAL element indicates that the requestor desires any returned secrets in proof-of-
2454 possession tokens to be encrypted for the specified token. That is, so that the owner of the
2455 specified token can decrypt the secret. Normally the security token is the contents of this element
2456 but a security token reference MAY be used instead. If this element isn't specified, the token
2457 used as the basis of the request (or specialized knowledge) is used to determine how to encrypt
2458 the key.

2459 */wst:RequestSecurityToken/wst:KeyWrapAlgorithm*

2460 This OPTIONAL URI element indicates the desired algorithm to use for key wrapping when STS
2461 encrypts the issued token for the relying party using an asymmetric key.

2462 */wst:RequestSecurityToken/wst:UseKey*

2463 If the requestor wishes to use an existing key rather than create a new one, then this OPTIONAL
2464 element can be used to reference the security token containing the desired key. This element
2465 either contains a security token or a `<wsse:SecurityTokenReference>` element that
2466 references the security token containing the key that SHOULD be used in the returned token. If
2467 `<wst:KeyType>` is not defined and a key type is not implicitly known to the service, it MAY be
2468 determined from the token (if possible). Otherwise this parameter is meaningless and is ignored.
2469 Requestors SHOULD demonstrate authorized use of the public key provided.

2470 */wst:RequestSecurityToken/wst:UseKey/@Sig*

2471 In order to *authenticate* the key referenced, a signature MAY be used to prove the referenced
2472 token/key. If specified, this OPTIONAL attribute indicates the ID of the corresponding signature

2473 (by URI reference). When this attribute is present, a key need not be specified inside the element
2474 since the referenced signature will indicate the corresponding token (and key).

2475 */wst:RequestSecurityToken/wst:SignWith*

2476 This OPTIONAL URI element indicates the desired signature algorithm to be used with the issued
2477 security token (typically from the policy of the target site for which the token is being requested.
2478 While any of these OPTIONAL elements MAY be included in RSTRs, this one is a likely
2479 candidate if there is some doubt (e.g., an X.509 cert that can only use DSS).

2480 */wst:RequestSecurityToken/wst:EncryptWith*

2481 This OPTIONAL URI element indicates the desired encryption algorithm to be used with the
2482 issued security token (typically from the policy of the target site for which the token is being
2483 requested.) While any of these OPTIONAL elements MAY be included in RSTRs, this one is a
2484 likely candidate if there is some doubt.

2485 The following summarizes the various algorithm parameters defined above. T is the issued token, P is the
2486 proof key.
2487

2488 **SignatureAlgorithm** - The signature algorithm to use to sign T

2489 **EncryptionAlgorithm** - The encryption algorithm to use to encrypt T

2490 **CanonicalizationAlgorithm** - The canonicalization algorithm to use when signing T

2491 **ComputedKeyAlgorithm** - The key derivation algorithm to use if using a symmetric key for P
2492 where P is computed using client, server, or combined entropy

2493 **Encryption** - The token/key to use when encrypting T

2494 **ProofEncryption** - The token/key to use when encrypting P

2495 **UseKey** - This is P. This is generally used when the client supplies a public-key that it wishes to
2496 be embedded in T as the proof key

2497 **SignWith** - The signature algorithm the client intends to employ when using P to
2498 sign

2499 The encryption algorithms further differ based on whether the issued token contains asymmetric key or
2500 symmetric key. Furthermore, they differ based on what type of key is used to protect the issued token
2501 from the STS to the relying party. The following cases can occur:

2502 T contains symmetric key/STS uses symmetric key to encrypt T for RP

2503 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP
2504 when using the proof key (e.g. AES256)

2505 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to
2506 encrypt the T (e.g. AES256)

2507

2508 T contains symmetric key/STS uses asymmetric key to encrypt T for RP

2509 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP
2510 when using the proof key (e.g. AES256)

2511 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to
2512 encrypt T for RP (e.g. AES256)

2513 **KeyWrapAlgorithm** – used to indicate the KeyWrap algorithm that the STS SHOULD use to
2514 wrap the generated key that is used to encrypt the T for RP

2515

2516 T contains asymmetric key/STS uses symmetric key to encrypt T for RP

2517 **EncryptWith** – used to indicate the KeyWrap algorithm that the client will use to

2518 protect the symmetric key that is used to protect messages to RP when using the proof key (e.g.
2519 RSA-OAEP-MGF1P)

2520 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to
2521 encrypt T for RP (e.g. AES256)

2522

2523 T contains asymmetric key/STS uses asymmetric key to encrypt T for RP

2524 **EncryptWith** - used to indicate the KeyWrap algorithm that the client will use to
2525 protect symmetric key that is used to protect message to RP when using the proof
2526 key (e.g. RSA-OAEP-MGF1P)

2527 **EncryptionAlgorithm** - used to indicate the symmetric algorithm that the STS SHOULD use to
2528 encrypt T for RP (e.g. AES256)

2529 **KeyWrapAlgorithm** – used to indicate the KeyWrap algorithm that the STS SHOULD use to
2530 wrap the generated key that is used to encrypt the T for RP

2531

2532 The example below illustrates a request that utilizes several of these parameters. A request is made for a
2533 custom token using a username and password as the basis of the request. For security, this token is
2534 encrypted (see "encUsername") for the recipient using the recipient's public key and referenced in the
2535 encryption manifest. The message is protected by a signature using a public key from the sender and
2536 authorized by the username and password.

2537

2538 The requestor would like the custom token to contain a 1024-bit public key whose value can be found in
2539 the key provided with the "proofSignature" signature (the key identified by "requestProofToken"). The
2540 token should be signed using RSA-SHA1 and encrypted for the token identified by
2541 "requestEncryptionToken". The proof should be encrypted using the token identified by
2542 "requestProofToken".

```
2543 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."  
2544     xmlns:wst="..." xmlns:ds="..." xmlns:xenc="...">  
2545   <S11:Header>  
2546     ...  
2547     <wsse:Security>  
2548       <xenc:ReferenceList>...</xenc:ReferenceList>  
2549       <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>  
2550       <wsse:BinarySecurityToken wsu:Id="requestEncryptionToken"  
2551         ValueType="...SomeTokenType" xmlns:x="...">  
2552         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
2553       </wsse:BinarySecurityToken>  
2554       <wsse:BinarySecurityToken wsu:Id="requestProofToken"  
2555         ValueType="...SomeTokenType" xmlns:x="...">  
2556         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
2557       </wsse:BinarySecurityToken>  
2558       <ds:Signature Id="proofSignature">  
2559         ... signature proving requested key ...  
2560         ... key info points to the "requestedProofToken" token ...  
2561       </ds:Signature>  
2562     </wsse:Security>  
2563     ...  
2564   </S11:Header>  
2565   <S11:Body wsu:Id="req">  
2566     <wst:RequestSecurityToken>  
2567       <wst:TokenType>  
2568         http://example.org/mySpecialToken  
2569       </wst:TokenType>  
2570     <wst:RequestType>
```

2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589

```
        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/xmlldsig#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>
```

2590 **9.3 Delegation and Forwarding Requirements**

2591 This section defines extensions to the `<wst:RequestSecurityToken>` element for indicating
2592 delegation and forwarding requirements on the requested security token(s).

2593 The syntax for these extension elements is as follows (note that the base elements described above are
2594 included here italicized for completeness):

2595
2596
2597
2598
2599
2600
2601
2602
2603

```
<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:ActAs>...</wst:ActAs>
</wst:RequestSecurityToken>
```

2604 */wst:RequestSecurityToken/wst:DelegateTo*

2605 This OPTIONAL element indicates that the requested or issued token be delegated to another
2606 identity. The identity receiving the delegation is specified by placing a security token or
2607 `<wsse:SecurityTokenReference>` element within the `<wst:DelegateTo>` element.

2608 */wst:RequestSecurityToken/wst:Forwardable*

2609 This OPTIONAL element, of type `xs:boolean`, specifies whether the requested security token
2610 SHOULD be marked as "Forwardable". In general, this flag is used when a token is normally
2611 bound to the requestor's machine or service. Using this flag, the returned token MAY be used
2612 from any source machine so long as the key is correctly proven. The default value of this flag is
2613 true.

2614 */wst:RequestSecurityToken/wst:Delegatable*

2615 This OPTIONAL element, of type `xs:boolean`, specifies whether the requested security token
2616 SHOULD be marked as "Delegatable". Using this flag, the returned token MAY be delegated to
2617 another party. This parameter SHOULD be used in conjunction with `<wst:DelegateTo>`. The
2618 default value of this flag is false.

2619 */wst:RequestSecurityToken/wst:ActAs*

2620 This OPTIONAL element indicates that the requested token is expected to contain information
2621 about the identity represented by the content of this element and the token requestor intends to
2622 use the returned token to act as this identity. The identity that the requestor wants to act-as is

2623 specified by placing a security token or <wsse:SecurityTokenReference> element within the
2624 <wst:ActAs> element.

2625 The following illustrates the syntax of a request for a custom token that can be delegated to the indicated
2626 recipient (specified in the binary security token) and used in the specified interval.

```
2627 <wst:RequestSecurityToken xmlns:wst="...">  
2628 <wst:TokenType>  
2629 http://example.org/mySpecialToken  
2630 </wst:TokenType>  
2631 <wst:RequestType>  
2632 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
2633 </wst:RequestType>  
2634 <wst:DelegateTo>  
2635 <wsse:BinarySecurityToken xmlns:wsse="...">  
2636 ...  
2637 </wsse:BinarySecurityToken>  
2638 </wst:DelegateTo>  
2639 <wst:Delegatable>true</wst:Delegatable>  
2640 </wst:RequestSecurityToken>
```

2641 9.4 Policies

2642 This section defines extensions to the <wst:RequestSecurityToken> element for passing policies.
2643

2644 The syntax for these extension elements is as follows (note that the base elements described above are
2645 included here italicized for completeness):

```
2646 <wst:RequestSecurityToken xmlns:wst="...">  
2647 <wst:TokenType>...</wst:TokenType>  
2648 <wst:RequestType>...</wst:RequestType>  
2649 ...  
2650 <wsp:Policy xmlns:wsp="...">...</wsp:Policy>  
2651 <wsp:PolicyReference xmlns:wsp="...">...</wsp:PolicyReference>  
2652 </wst:RequestSecurityToken>
```

2653
2654 The following describes the attributes and elements listed in the schema overview above:

2655 */wst:RequestSecurityToken/wsp:Policy*

2656 This OPTIONAL element specifies a policy (as defined in [WS-Policy]) that indicates desired
2657 settings for the requested token. The policy specifies defaults that can be overridden by the
2658 elements defined in the previous sections.

2659 */wst:RequestSecurityToken/wsp:PolicyReference*

2660 This OPTIONAL element specifies a reference to a policy (as defined in [WS-Policy]) that
2661 indicates desired settings for the requested token. The policy specifies defaults that can be
2662 overridden by the elements defined in the previous sections.

2663
2664 The following illustrates the syntax of a request for a custom token that provides a set of policy
2665 statements about the token or its usage requirements.

```
2666 <wst:RequestSecurityToken xmlns:wst="...">  
2667 <wst:TokenType>  
2668 http://example.org/mySpecialToken  
2669 </wst:TokenType>  
2670 <wst:RequestType>  
2671 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
2672 </wst:RequestType>  
2673 <wsp:Policy xmlns:wsp="...">
```

2674
2675
2676

```
...  
</wsp:Policy>  
</wst:RequestSecurityToken>
```

2677 9.5 Authorized Token Participants

2678 This section defines extensions to the `<wst:RequestSecurityToken>` element for passing information
2679 about which parties are authorized to participate in the use of the token. This parameter is typically used
2680 when there are additional parties using the token or if the requestor needs to clarify the actual parties
2681 involved (for some profile-specific reason).

2682 It should be noted that additional participants will need to prove their identity to recipients in addition to
2683 proving their authorization to use the returned token. This typically takes the form of a second signature
2684 or use of transport security.

2685

2686 The syntax for these extension elements is as follows (note that the base elements described above are
2687 included here italicized for completeness):

```
2688 <wst:RequestSecurityToken xmlns:wst="...">  
2689   <wst:TokenType>...</wst:TokenType>  
2690   <wst:RequestType>...</wst:RequestType>  
2691   ...  
2692   <wst:Participants>  
2693     <wst:Primary>...</wst:Primary>  
2694     <wst:Participant>...</wst:Participant>  
2695   </wst:Participants>  
2696 </wst:RequestSecurityToken>
```

2697

2698 The following describes elements and attributes used in a `<wsc:SecurityContextToken>` element.

2699 */wst:RequestSecurityToken/wst:Participants/*

2700 This OPTIONAL element specifies the participants sharing the security token. Arbitrary types
2701 MAY be used to specify participants, but a typical case is a security token or an endpoint
2702 reference (see [WS-Addressing]).

2703 */wst:RequestSecurityToken/wst:Participants/wst:Primary*

2704 This OPTIONAL element specifies the primary user of the token (if one exists).

2705 */wst:RequestSecurityToken/wst:Participants/wst:Participant*

2706 This OPTIONAL element specifies participant (or multiple participants by repeating the element)
2707 that play a (profile-dependent) role in the use of the token or who are allowed to use the token.

2708 */wst:RequestSecurityToken/wst:Participants/{any}*

2709 This is an extensibility option to allow other types of participants and profile-specific elements to
2710 be specified.

2711 10 Key Exchange Token Binding

2712 Using the token request framework, this section defines a binding for requesting a key exchange token
2713 (KET). That is, if a requestor desires a token that can be used to encrypt key material for a recipient.

2714
2715 For this binding, the following actions are defined to enable specific processing context to be conveyed to
2716 the recipient:

```
2717 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET  
2718 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KET  
2719 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal
```

2720
2721 For this binding, the `RequestType` element contains the following URI:

```
2722 http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET
```

2723
2724 For this binding very few parameters are specified as input. OPTIONALLY the `<wst:TokenType>`
2725 element can be specified in the request can indicate desired type response token carrying the key for key
2726 exchange; however, this isn't commonly used.

2727 The applicability scope (e.g. `<wsp:AppliesTo>`) MAY be specified if the requestor desires a key
2728 exchange token for a specific scope.

2729
2730 It is RECOMMENDED that the response carrying the key exchange token be secured (e.g., signed by the
2731 issuer or someone who can speak on behalf of the target for which the KET applies).

2732
2733 Care should be taken when using this binding to prevent possible man-in-the-middle and substitution
2734 attacks. For example, responses to this request SHOULD be secured using a token that can speak for
2735 the desired endpoint.

2736
2737 The RSTR for this binding carries the `<RequestedSecurityToken>` element even if a token is returned
2738 (note that the base elements described above are included here italicized for completeness):

```
2739 <wst:RequestSecurityToken xmlns:wst="...">  
2740   <wst:TokenType>...</wst:TokenType>  
2741   <wst:RequestType>...</wst:RequestType>  
2742   ...  
2743 </wst:RequestSecurityToken>
```

```
2744  
2745 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2746   <wst:RequestSecurityTokenResponse>  
2747     <wst:TokenType>...</wst:TokenType>  
2748     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>  
2749     ...  
2750   </wst:RequestSecurityTokenResponse>  
2751 </wst:RequestSecurityTokenResponseCollection>
```

2752
2753 The following illustrates the syntax for requesting a key exchange token. In this example, the KET is
2754 returned encrypted for the requestor since it had the credentials available to do that. Alternatively the

2755 request could be made using transport security (e.g. TLS) and the key could be returned directly using
2756 <wst:BinarySecret>.

```
2757 <wst:RequestSecurityToken xmlns:wst="...">  
2758 <wst:RequestType>  
2759 http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET  
2760 </wst:RequestType>  
2761 </wst:RequestSecurityToken>
```

2762

```
2763 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2764 <wst:RequestSecurityTokenResponse>  
2765 <wst:RequestedSecurityToken>  
2766 <xenc:EncryptedKey xmlns:xenc="...">...</xenc:EncryptedKey>  
2767 </wst:RequestedSecurityToken>  
2768 </wst:RequestSecurityTokenResponse>  
2769 </wst:RequestSecurityTokenResponseCollection>
```

2770

11 Error Handling

2771 There are many circumstances where an *error* can occur while processing security information. Errors
2772 use the SOAP Fault mechanism. Note that the reason text provided below is RECOMMENDED, but
2773 alternative text MAY be provided if more descriptive or preferred by the implementation. The tables
2774 below are defined in terms of SOAP 1.1. For SOAP 1.2, the Fault/Code/Value is env:Sender (as defined
2775 in SOAP 1.2) and the Fault/Code/Subcode/Value is the *faultcode* below and the Fault/Reason/Text is the
2776 *faultstring* below. It should be noted that profiles MAY provide second-level detail fields, but they should
2777 be careful not to introduce security vulnerabilities when doing so (e.g., by providing too detailed
2778 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

2779

12 Security Considerations

2780 As stated in the Goals section of this document, this specification is meant to provide extensible
2781 framework and flexible syntax, with which one could implement various security mechanisms. This
2782 framework and syntax by itself does not provide any guarantee of security. When implementing and using
2783 this framework and syntax, one must make every effort to ensure that the result is not vulnerable to any
2784 one of a wide range of attacks.

2785

2786 It is not feasible to provide a comprehensive list of security considerations for such an extensible set of
2787 mechanisms. A complete security analysis must be conducted on specific solutions based on this
2788 specification. Below we illustrate some of the security concerns that often come up with protocols of this
2789 type, but we stress that this *is not an exhaustive list of concerns*.

2790

2791 The following statements about signatures and signing apply to messages sent on unsecured channels.

2792

2793 It is critical that all the security-sensitive message elements must be included in the scope of the
2794 message signature. As well, the signatures for conversation authentication must include a timestamp,
2795 nonce, or sequence number depending on the degree of replay prevention required as described in [[WS-
2796 Security](#)] and the UsernameToken Profile. Also, conversation establishment should include the policy so
2797 that supported algorithms and algorithm priorities can be validated.

2798

2799 It is required that security token issuance messages be signed to prevent tampering. If a public key is
2800 provided, the request should be signed by the corresponding private key to prove ownership. As well,
2801 additional steps should be taken to eliminate replay attacks (refer to [[WS-Security](#)] for additional
2802 information). Similarly, all token references should be signed to prevent any tampering.

2803

2804 Security token requests are susceptible to denial-of-service attacks. Care should be taken to mitigate
2805 such attacks as is warranted by the service.

2806

2807 For security, tokens containing a symmetric key or a password should only be sent to parties who have a
2808 need to know that key or password.

2809

2810 For privacy, tokens containing personal information (either in the claims, or indirectly by identifying who is
2811 currently communicating with whom) should only be sent according to the privacy policies governing
2812 these data at the respective organizations.

2813

2814 For some forms of multi-message exchanges, the exchanges are susceptible to attacks whereby
2815 signatures are altered. To address this, it is suggested that a signature confirmation mechanism be used.
2816 In such cases, each leg should include the confirmation of the previous leg. That is, leg 2 includes
2817 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
2818 correctness of the message outside of the message body.

2819

2820 There are many other security concerns that one may need to consider in security protocols. The list
2821 above should not be used as a "check list" instead of a comprehensive security analysis.

2822

2823 It should be noted that use of unsolicited RSTRs implies that the recipient is prepared to accept such
2824 issuances. Recipients should ensure that such issuances are properly authorized and recognize their
2825 use could be used in denial-of-service attacks.

2826 In addition to the consideration identified here, readers should also review the security considerations in
2827 [\[WS-Security\]](#).

2828

2829 Both token cancellation bindings defined in this specification require that the STS MUST NOT validate or
2830 renew the token after it has been successfully canceled. The STS must take care to ensure that the token
2831 is properly invalidated before confirming the cancel request or sending the cancel notification to the client.
2832 This can be more difficult if the token validation or renewal logic is physically separated from the issuance
2833 and cancellation logic. It is out of scope of this spec how the STS propagates the token cancellation to its
2834 other components. If STS cannot ensure that the token was properly invalidated it MUST NOT send the
2835 cancel notification or confirm the cancel request to the client.

2836

2837

13 Conformance

2838

2839 An implementation conforms to this specification if it satisfies all of the MUST or REQUIRED level
2840 requirements defined within this specification. A SOAP Node MUST NOT use the XML namespace
2841 identifier for this specification (listed in Section 1.3) within SOAP Envelopes unless it is compliant with this
2842 specification.

2843 This specification references a number of other specifications (see the table above). In order to comply
2844 with this specification, an implementation MUST implement the portions of referenced specifications
2845 necessary to comply with the required provisions of this specification. Additionally, the implementation of
2846 the portions of the referenced specifications that are specifically cited in this specification MUST comply
2847 with the rules for those portions as established in the referenced specification.

2848 Additionally normative text within this specification takes precedence over normative outlines (as
2849 described in section 1.5.1), which in turn take precedence over the XML Schema [XML Schema Part 1,
2850 Part 2] and WSDL [WSDL 1.1] descriptions. That is, the normative text in this specification further
2851 constrains the schemas and/or WSDL that are part of this specification; and this specification contains
2852 further constraints on the elements defined in referenced schemas.

2853 This specification defines a number of extensions; compliant services are NOT REQUIRED to implement
2854 OPTIONAL features defined in this specification. However, if a service implements an aspect of the
2855 specification, it MUST comply with the requirements specified (e.g. related "MUST" statements). If an
2856 OPTIONAL message is not supported, then the implementation SHOULD Fault just as it would for any
2857 other unrecognized/unsupported message. If an OPTIONAL message is supported, then the
2858 implementation MUST satisfy all of the MUST and REQUIRED sections of the message.

2859

A. Key Exchange

2860 Key exchange is an integral part of token acquisition. There are several mechanisms by which keys are
2861 exchanged using [WS-Security] and WS-Trust. This section highlights and summarizes these
2862 mechanisms. Other specifications and profiles MAY provide additional details on key exchange.

2863

2864 Care must be taken when employing a key exchange to ensure that the mechanism does not provide an
2865 attacker with a means of discovering information that could only be discovered through use of secret
2866 information (such as a private key).

2867

2868 It is therefore important that a shared secret should only be considered as trustworthy as its source. A
2869 shared secret communicated by means of the direct encryption scheme described in section I.1 is
2870 acceptable if the encryption key is provided by a completely trustworthy key distribution center (this is the
2871 case in the Kerberos model). Such a key would not be acceptable for the purposes of decrypting
2872 information from the source that provided it since an attacker might replay information from a prior
2873 transaction in the hope of learning information about it.

2874

2875 In most cases the other party in a transaction is only imperfectly trustworthy. In these cases both parties
2876 SHOULD contribute entropy to the key exchange by means of the <wst:entropy> element.

A.1 Ephemeral Encryption Keys

2878 The simplest form of key exchange can be found in [WS-Security] for encrypting message data. As
2879 described in [WS-Security] and [XML-Encrypt], when data is encrypted, a temporary key can be used to
2880 perform the encryption which is, itself, then encrypted using the <xenc:EncryptedKey> element.

2881

2882 The illustrates the syntax for encrypting a temporary key using the public key in an issuer name and serial
2883 number:

2884

2885

2886

2887

2888

2889

2890

2891

2892

2893

2894

2895

2896

2897

```
<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

2899 When a request sends a message to an issuer to request a token, the client can provide proposed key
2900 material using the <wst:Entropy> element. If the issuer doesn't contribute any key material, this is
2901 used as the secret (key). This information is encrypted for the issuer either using
2902 <xenc:EncryptedKey> or by using a transport security. If the requestor provides key material that the

2903 recipient doesn't accept, then the issuer SHOULD reject the request. Note that the issuer need not return
2904 the key provided by the requestor.

2905

2906 The following illustrates the syntax of a request for a custom security token and includes a secret that is
2907 to be used for the key. In this example the entropy is encrypted for the issuer (if transport security was
2908 used for confidentiality then the <wst:Entropy> element would contain a <wst:BinarySecret>
2909 element):

```
2910 <wst:RequestSecurityToken xmlns:wst="...">  
2911 <wst:TokenType>  
2912   http://example.org/mySpecialToken  
2913 </wst:TokenType>  
2914 <wst:RequestType>  
2915   http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
2916 </wst:RequestType>  
2917 <wst:Entropy>  
2918   <xenc:EncryptedData xmlns:xenc="...">...</xenc:EncryptedData>  
2919 </wst:Entropy>  
2920 </wst:RequestSecurityToken>
```

2921 **A.3 Issuer-Provided Keys**

2922 If a requestor fails to provide key material, then issued proof-of-possession tokens contain an issuer-
2923 provided secret that is encrypted for the requestor (either using <xenc:EncryptedKey> or by using a
2924 transport security).

2925

2926 The following illustrates the syntax of a token being returned with an associated proof-of-possession
2927 token that is encrypted using the requestor's public key.

```
2928 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2929 <wst:RequestSecurityTokenResponse>  
2930 <wst:RequestedSecurityToken>  
2931 <xyz:CustomToken xmlns:xyz="...">  
2932   ...  
2933 </xyz:CustomToken>  
2934 </wst:RequestedSecurityToken>  
2935 <wst:RequestedProofToken>  
2936 <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">  
2937   ...  
2938 </xenc:EncryptedKey>  
2939 </wst:RequestedProofToken>  
2940 </wst:RequestSecurityTokenResponse>  
2941 </wst:RequestSecurityTokenResponseCollection>
```

2942 **A.4 Composite Keys**

2943 The safest form of key exchange/generation is when both the requestor and the issuer contribute to the
2944 key material. In this case, the request sends encrypted key material. The issuer then returns additional
2945 encrypted key material. The actual secret (key) is computed using a function of the two pieces of data.
2946 Ideally this secret is never used and, instead, keys derived are used for message protection.

2947

2948 The following example illustrates a server, having received a request with requestor entropy returning its
2949 own entropy, which is used in conjunction with the requestor's to generate a key. In this example the
2950 entropy is not encrypted because the transport is providing confidentiality (otherwise the
2951 <wst:Entropy> element would have an <xenc:EncryptedData> element).

2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963

```
<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>
```

2964 **A.5 Key Transfer and Distribution**

2965 There are also a few mechanisms where existing keys are transferred to other parties.

2966 **A.5.1 Direct Key Transfer**

2967 If one party has a token and key and wishes to share this with another party, the key can be directly
2968 transferred. This is accomplished by sending an RSTR (either in the body or header) to the other party.
2969 The RSTR contains the token and a proof-of-possession token that contains the key encrypted for the
2970 recipient.

2971

2972 In the following example a custom token and its associated proof-of-possession token are known to party
2973 A who wishes to share them with party B. In this example, A is a member in a secure on-line chat
2974 session and is inviting B to join the conversation. After authenticating B, A sends B an RSTR. The RSTR
2975 contains the token and the key is communicated as a proof-of-possession token that is encrypted for B:

2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989

```
<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>
```

2990 **A.5.2 Brokered Key Distribution**

2991 A third party MAY also act as a broker to transfer keys. For example, a requestor may obtain a token and
2992 proof-of-possession token from a third-party STS. The token contains a key encrypted for the target
2993 service (either using the service's public key or a key known to the STS and target service). The proof-of-
2994 possession token contains the same key encrypted for the requestor (similarly this can use public or
2995 symmetric keys).

2996

2997 In the following example a custom token and its associated proof-of-possession token are returned from a
2998 broker B to a requestor R for access to service S. The key for the session is contained within the custom
2999 token encrypted for S using either a secret known by B and S or using S's public key. The same secret is
3000 encrypted for R and returned as the proof-of-possession token:

3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018

```
<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>
```

3019 **A.5.3 Delegated Key Transfer**

3020 Key transfer can also take the form of delegation. That is, one party transfers the right to use a key
3021 without actually transferring the key. In such cases, a delegation token, e.g. XrML, is created that
3022 identifies a set of rights and a delegation target and is secured by the delegating party. That is, one key
3023 indicates that another key can use a subset (or all) of its rights. The delegate can provide this token and
3024 prove itself (using its own key – the delegation target) to a service. The service, assuming the trust
3025 relationships have been established and that the delegator has the right to delegate, can then authorize
3026 requests sent subject to delegation rules and trust policies.

3027
3028 In this example a custom token is issued from party A to party B. The token indicates that B (specifically
3029 B's key) has the right to submit purchase orders. The token is signed using a secret key known to the
3030 target service T and party A (the key used to ultimately authorize the requests that B makes to T), and a
3031 new session key that is encrypted for T. A proof-of-possession token is included that contains the
3032 session key encrypted for B. As a result, B is *effectively* using A's key, but doesn't actually know the key.

3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3050
3051
3052
3053
3054
3055

```
<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>
```

3056 **A.5.4 Authenticated Request/Reply Key Transfer**

3057 In some cases the RST/RSTR mechanism is not used to transfer keys because it is part of a simple
3058 request/reply. However, there may be a desire to ensure mutual authentication as part of the key
3059 transfer. The mechanisms of [WS-Security] can be used to implement this scenario.

3060

3061 Specifically, the sender wishes the following:

- 3062 • Transfer a key to a recipient that they can use to secure a reply
- 3063 • Ensure that only the recipient can see the key
- 3064 • Provide proof that the sender issued the key

3065

3066 This scenario could be supported by encrypting and then signing. This would result in roughly the
3067 following steps:

- 3068 1. Encrypt the message using a generated key
- 3069 2. Encrypt the key for the recipient
- 3070 3. Sign the encrypted form, any other relevant keys, and the encrypted key

3071

3072 However, if there is a desire to sign prior to encryption then the following general process is used:

- 3073 1. Sign the appropriate message parts using a random key (or ideally a key derived from a random
3074 key)
- 3075 2. Encrypt the appropriate message parts using the random key (or ideally another key derived from
3076 the random key)
- 3077 3. Encrypt the random key for the recipient
- 3078 4. Sign just the encrypted key

3079

3080 This would result in a <wsse:Security> header that looks roughly like the following:

```
3081 <wsse:Security xmlns:wsse="..." xmlns:wsu="..."  
3082     xmlns:ds="..." xmlns:xenc="...">  
3083   <wsse:BinarySecurityToken wsu:Id="myToken">  
3084     ...  
3085   </wsse:BinarySecurityToken>  
3086   <ds:Signature>  
3087     ...signature over #secret using token #myToken...  
3088   </ds:Signature>  
3089   <xenc:EncryptedKey Id="secret">  
3090     ...  
3091   </xenc:EncryptedKey>  
3092   <xenc:ReferenceList>  
3093     ...manifest of encrypted parts using token #secret...  
3094   </xenc:ReferenceList>  
3095   <ds:Signature>  
3096     ...signature over key message parts using token #secret...  
3097   </ds:Signature>  
3098 </wsse:Security>
```

3099

3100 As well, instead of an <xenc:EncryptedKey> element, the actual token could be passed using
3101 <xenc:EncryptedData>. The result might look like the following:

```
3102 <wsse:Security xmlns:wsse="..." xmlns:wsu="..."  
3103     xmlns:ds="..." xmlns:xenc="...">
```

```

3104 <wsse:BinarySecurityToken wsu:Id="myToken">
3105   ...
3106 </wsse:BinarySecurityToken>
3107 <ds:Signature>
3108   ...signature over #secret or #Esecret using token #myToken...
3109 </ds:Signature>
3110 <xenc:EncryptedData Id="Esecret">
3111   ...Encrypted version of a token with Id="secret"...
3112 </xenc:EncryptedData>
3113 <xenc:ReferenceList>
3114   ...manifest of encrypted parts using token #secret...
3115 </xenc:ReferenceList>
3116 <ds:Signature>
3117   ...signature over key message parts using token #secret...
3118 </ds:Signature>
3119 </wsse:Security>

```

3120 **A.6 Perfect Forward Secrecy**

3121 In some situations it is desirable for a key exchange to have the property of perfect forward secrecy. This
3122 means that it is impossible to reconstruct the shared secret even if the private keys of the parties are
3123 disclosed.

3124
3125 The most straightforward way to attain perfect forward secrecy when using asymmetric key exchange is
3126 to dispose of one's key exchange key pair periodically (or even after every key exchange), replacing it
3127 with a fresh one. Of course, a freshly generated public key must still be authenticated (using any of the
3128 methods normally available to prove the identity of a public key's owner).

3129
3130 The perfect forward secrecy property MAY be achieved by specifying a `<wst:entropy>` element that
3131 contains an `<xenc:EncryptedKey>` that is encrypted under a public key pair created for use in a single
3132 key agreement. The public key does not require authentication since it is only used to provide additional
3133 entropy. If the public key is modified, the key agreement will fail. Care should be taken, when using this
3134 method, to ensure that the now-secret entropy exchanged via the `<wst:entropy>` element is not
3135 revealed elsewhere in the protocol (since such entropy is often assumed to be publicly revealed plaintext,
3136 and treated accordingly).

3137
3138 Although any public key scheme might be used to achieve perfect forward secrecy (in either of the above
3139 methods) it is generally desirable to use an algorithm that allows keys to be generated quickly. The Diffie-
3140 Hellman key exchange is often used for this purpose since generation of a key only requires the
3141 generation of a random integer and calculation of a single modular exponent.

3142

B. WSDL

3143 The WSDL below does not fully capture all the possible message exchange patterns, but captures the
3144 typical message exchange pattern as described in this document.

```
3145 <?xml version="1.0"?>
3146 <wsdl:definitions
3147     targetNamespace="http://docs.oasis-open.org/ws-sx/ws-
3148 trust/200512/wsdl"
3149     xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/wsdl"
3150     xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
3151     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
3152     xmlns:xs="http://www.w3.org/2001/XMLSchema"
3153     xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
3154 >
3155 <!-- this is the WS-I BP-compliant way to import a schema -->
3156 <wsdl:types>
3157     <xs:schema>
3158         <xs:import
3159             namespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
3160             schemaLocation="http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-
3161 trust.xsd"/>
3162     </xs:schema>
3163 </wsdl:types>
3164
3165 <!-- WS-Trust defines the following GEDs -->
3166 <wsdl:message name="RequestSecurityTokenMsg">
3167     <wsdl:part name="request" element="wst:RequestSecurityToken" />
3168 </wsdl:message>
3169 <wsdl:message name="RequestSecurityTokenResponseMsg">
3170     <wsdl:part name="response"
3171         element="wst:RequestSecurityTokenResponse" />
3172 </wsdl:message>
3173 <wsdl:message name="RequestSecurityTokenCollectionMsg">
3174     <wsdl:part name="requestCollection"
3175         element="wst:RequestSecurityTokenCollection"/>
3176 </wsdl:message>
3177 <wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
3178     <wsdl:part name="responseCollection"
3179         element="wst:RequestSecurityTokenResponseCollection"/>
3180 </wsdl:message>
3181
3182 <!-- This portType an example of a Requestor (or other) endpoint that
3183     Accepts SOAP-based challenges from a Security Token Service -->
3184 <wsdl:portType name="WSSecurityRequestor">
3185     <wsdl:operation name="Challenge">
3186         <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
3187         <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
3188     </wsdl:operation>
3189 </wsdl:portType>
3190
3191 <!-- This portType is an example of an STS supporting full protocol -->
3192 <wsdl:portType name="SecurityTokenService">
3193     <wsdl:operation name="Cancel">
3194         <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3195 trust/200512/RST/Cancel" message="tns:RequestSecurityTokenMsg"/>
3196         <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3197 trust/200512/RSTR/CancelFinal" message="tns:RequestSecurityTokenResponseMsg"/>
3198     </wsdl:operation>
3199     <wsdl:operation name="Issue">
```

```

3200     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3201 trust/200512/RST/Issue" message="tns:RequestSecurityTokenMsg"/>
3202     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3203 trust/200512/RSTRC/IssueFinal"
3204 message="tns:RequestSecurityTokenResponseCollectionMsg"/>
3205   </wsdl:operation>
3206   <wsdl:operation name="Renew">
3207     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3208 trust/200512/RST/Renew" message="tns:RequestSecurityTokenMsg"/>
3209     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3210 trust/200512/RSTR/RenewFinal" message="tns:RequestSecurityTokenResponseMsg"/>
3211   </wsdl:operation>
3212   <wsdl:operation name="Validate">
3213     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3214 trust/200512/RST/Validate" message="tns:RequestSecurityTokenMsg"/>
3215     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3216 trust/200512/RSTR/ValidateFinal
3217 message="tns:RequestSecurityTokenResponseMsg"/>
3218   </wsdl:operation>
3219   <wsdl:operation name="KeyExchangeToken">
3220     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3221 trust/200512/RST/KET" message="tns:RequestSecurityTokenMsg"/>
3222     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3223 trust/200512/RSTR/KETFinal" message="tns:RequestSecurityTokenResponseMsg"/>
3224   </wsdl:operation>
3225   <wsdl:operation name="RequestCollection">
3226     <wsdl:input message="tns:RequestSecurityTokenCollectionMsg"/>
3227     <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg"/>
3228   </wsdl:operation>
3229 </wsdl:portType>
3230
3231 <!-- This portType is an example of an endpoint that accepts
3232 Unsolicited RequestSecurityTokenResponse messages -->
3233 <wsdl:portType name="SecurityTokenResponseService">
3234   <wsdl:operation name="RequestSecurityTokenResponse">
3235     <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
3236   </wsdl:operation>
3237 </wsdl:portType>
3238
3239 </wsdl:definitions>
3240

```

3241

C. Acknowledgements

3242 The following individuals have participated in the creation of this specification and are gratefully
3243 acknowledged:

3244 **Original Authors of the initial contribution:**

3245 Steve Anderson, OpenNetwork
3246 Jeff Bohren, OpenNetwork
3247 Toufic Boubez, Layer 7
3248 Marc Chanliau, Computer Associates
3249 Giovanni Della-Libera, Microsoft
3250 Brendan Dixon, Microsoft
3251 Praerit Garg, Microsoft
3252 Martin Gudgin (Editor), Microsoft
3253 Phillip Hallam-Baker, VeriSign
3254 Maryann Hondo, IBM
3255 Chris Kaler, Microsoft
3256 Hal Lockhart, Oracle Corporation
3257 Robin Martherus, Oblix
3258 Hiroshi Maruyama, IBM
3259 Anthony Nadalin (Editor), IBM
3260 Nataraj Nagaratnam, IBM
3261 Andrew Nash, Reactivity
3262 Rob Philpott, RSA Security
3263 Darren Platt, Ping Identity
3264 Hemma Prafullchandra, VeriSign
3265 Maneesh Sahu, Actional
3266 John Shewchuk, Microsoft
3267 Dan Simon, Microsoft
3268 Davanum Srinivas, Computer Associates
3269 Elliot Waingold, Microsoft
3270 David Waite, Ping Identity
3271 Doug Walter, Microsoft
3272 Riaz Zolfonoon, RSA Security

3273

3274 **Original Acknowledgments of the initial contribution:**

3275 Paula Austel, IBM
3276 Keith Ballinger, Microsoft
3277 Bob Blakley, IBM
3278 John Brezak, Microsoft
3279 Tony Cowan, IBM
3280 Cédric Fournet, Microsoft
3281 Vijay Gajjala, Microsoft
3282 HongMei Ge, Microsoft
3283 Satoshi Hada, IBM
3284 Heather Hinton, IBM
3285 Slava Kavsan, RSA Security
3286 Scott Konersmann, Microsoft
3287 Leo Laferriere, Computer Associates

- 3288 Paul Leach, Microsoft
- 3289 Richard Levinson, Computer Associates
- 3290 John Linn, RSA Security
- 3291 Michael McIntosh, IBM
- 3292 Steve Millet, Microsoft
- 3293 Birgit Pfitzmann, IBM
- 3294 Fumiko Satoh, IBM
- 3295 Keith Stobie, Microsoft
- 3296 T.R. Vishwanath, Microsoft
- 3297 Richard Ward, Microsoft
- 3298 Hervey Wilson, Microsoft
- 3299
- 3300 **TC Members during the development of this specification:**
- 3301 Don Adams, Tibco Software Inc.
- 3302 Jan Alexander, Microsoft Corporation
- 3303 Steve Anderson, BMC Software
- 3304 Donal Arundel, IONA Technologies
- 3305 Howard Bae, Oracle Corporation
- 3306 Abbie Barbir, Nortel Networks Limited
- 3307 Charlton Barreto, Adobe Systems
- 3308 Mighael Botha, Software AG, Inc.
- 3309 Toufic Boubez, Layer 7 Technologies Inc.
- 3310 Norman Brickman, Mitre Corporation
- 3311 Melissa Brumfield, Booz Allen Hamilton
- 3312 Lloyd Burch, Novell
- 3313 Geoff Bullen, Microsoft Corporation
- 3314 Scott Cantor, Internet2
- 3315 Greg Carpenter, Microsoft Corporation
- 3316 Steve Carter, Novell
- 3317 Ching-Yun (C.Y.) Chao, IBM
- 3318 Martin Chapman, Oracle Corporation
- 3319 Kate Cherry, Lockheed Martin
- 3320 Henry (Hyenvui) Chung, IBM
- 3321 Luc Clement, Systinet Corp.
- 3322 Paul Cotton, Microsoft Corporation
- 3323 Glen Daniels, Sonic Software Corp.
- 3324 Peter Davis, Neustar, Inc.
- 3325 Martijn de Boer, SAP AG
- 3326 Duane DeCouteau, Veterans Health Administration
- 3327 Werner Dittmann, Siemens AG
- 3328 Abdeslem DJAOUI, CCLRC-Rutherford Appleton Laboratory
- 3329 Fred Dushin, IONA Technologies
- 3330 Petr Dvorak, Systinet Corp.
- 3331 Colleen Evans, Microsoft Corporation

3332 Ruchith Fernando, WSO2
3333 Mark Fussell, Microsoft Corporation
3334 Vijay Gajjala, Microsoft Corporation
3335 Marc Goodner, Microsoft Corporation
3336 Hans Granqvist, VeriSign
3337 Martin Gudgin, Microsoft Corporation
3338 Tony Gullotta, SOA Software Inc.
3339 Jiandong Guo, Sun Microsystems
3340 Phillip Hallam-Baker, VeriSign
3341 Patrick Harding, Ping Identity Corporation
3342 Heather Hinton, IBM
3343 Frederick Hirsch, Nokia Corporation
3344 Jeff Hodges, Neustar, Inc.
3345 Will Hopkins, Oracle Corporation
3346 Alex Hristov, Otecia Incorporated
3347 John Hughes, PA Consulting
3348 Diane Jordan, IBM
3349 Venugopal K, Sun Microsystems
3350 Chris Kaler, Microsoft Corporation
3351 Dana Kaufman, Forum Systems, Inc.
3352 Paul Knight, Nortel Networks Limited
3353 Ramanathan Krishnamurthy, IONA Technologies
3354 Christopher Kurt, Microsoft Corporation
3355 Kelvin Lawrence, IBM
3356 Hubert Le Van Gong, Sun Microsystems
3357 Jong Lee, Oracle Corporation
3358 Rich Levinson, Oracle Corporation
3359 Tommy Lindberg, Dajeil Ltd.
3360 Mark Little, JBoss Inc.
3361 Hal Lockhart, Oracle Corporation
3362 Mike Lyons, Layer 7 Technologies Inc.
3363 Eve Maler, Sun Microsystems
3364 Ashok Malhotra, Oracle Corporation
3365 Anand Mani, CrimsonLogic Pte Ltd
3366 Jonathan Marsh, Microsoft Corporation
3367 Robin Martherus, Oracle Corporation
3368 Miko Matsumura, Infravio, Inc.
3369 Gary McAfee, IBM
3370 Michael McIntosh, IBM
3371 John Merrells, Sxip Networks SRL
3372 Jeff Mischkinisky, Oracle Corporation
3373 Prateek Mishra, Oracle Corporation

3374 Bob Morgan, Internet2
3375 Vamsi Motukuru, Oracle Corporation
3376 Raajmohan Na, EDS
3377 Anthony Nadalin, IBM
3378 Andrew Nash, Reactivity, Inc.
3379 Eric Newcomer, IONA Technologies
3380 Duane Nickull, Adobe Systems
3381 Toshihiro Nishimura, Fujitsu Limited
3382 Rob Philpott, RSA Security
3383 Denis Pilipchuk, Oracle Corporation
3384 Darren Platt, Ping Identity Corporation
3385 Martin Raepple, SAP AG
3386 Nick Ragouzis, Enosis Group LLC
3387 Prakash Reddy, CA
3388 Alain Regnier, Ricoh Company, Ltd.
3389 Irving Reid, Hewlett-Packard
3390 Bruce Rich, IBM
3391 Tom Rutt, Fujitsu Limited
3392 Maneesh Sahu, Actional Corporation
3393 Frank Siebenlist, Argonne National Laboratory
3394 Joe Smith, Apani Networks
3395 Davanum Srinivas, WSO2
3396 David Staggs, Veterans Health Administration
3397 Yakov Sverdlov, CA
3398 Gene Thurston, AmberPoint
3399 Victor Valle, IBM
3400 Asir Vedamuthu, Microsoft Corporation
3401 Greg Whitehead, Hewlett-Packard
3402 Ron Williams, IBM
3403 Corinna Witt, Oracle Corporation
3404 Kyle Young, Microsoft Corporation
3405