



WS-Trust 1.4

OASIS Standard

2 February 2009

Specification URIs:

This Version:

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

Previous Version:

<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cs-01.doc> (Authoritative)
<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cs-01.pdf>
<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cs-01.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® 1993–2009. 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 Standard, "WS-SecurityPolicy 1.3", February 2009
211 [http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/os/ws-](http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/os/ws-securitypolicy-1.3-spec-os.doc)
212 [securitypolicy-1.3-spec-os.doc](http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/os/ws-securitypolicy-1.3-spec-os.doc)
213 [X509] S. Santesson, et al, "Internet X.509 Public Key Infrastructure Qualified
214 Certificates Profile."
215 [http://www.itu.int/rec/recommendation.asp?type=items&lang=e&parent=T-](http://www.itu.int/rec/recommendation.asp?type=items&lang=e&parent=T-REC-X.509-200003-I)
216 [REC-X.509-200003-I](http://www.itu.int/rec/recommendation.asp?type=items&lang=e&parent=T-REC-X.509-200003-I)
217

218

2 Web Services Trust Model

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

224

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

229

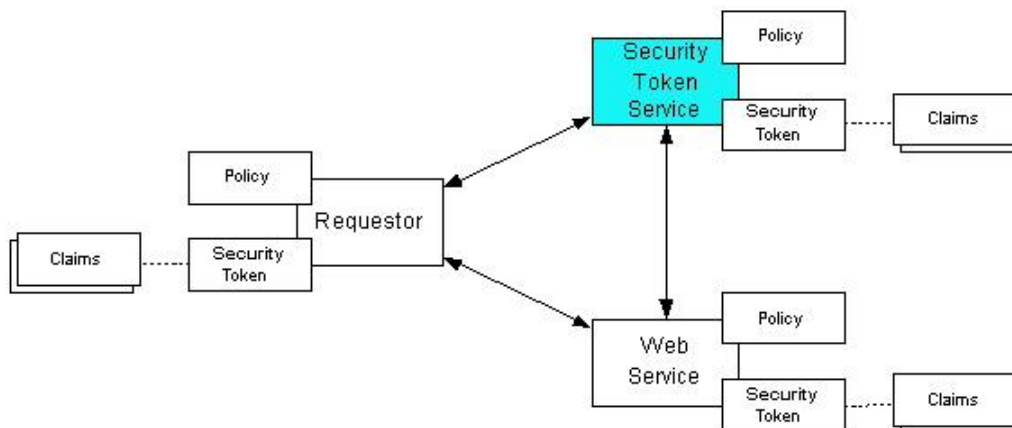
230 One way to demonstrate authorized use of a security token is to include a digital signature using the
231 associated secret key (from a proof-of-possession token). This allows a requestor to prove a required set
232 of claims by associating security tokens (e.g., PKIX, X.509 certificates) with the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

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

243

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

246



247

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

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

254

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

260

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

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

275

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

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

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

284

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

288 **2.1 Models for Trust Brokering and Assessment**

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

292 **2.2 Token Acquisition**

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

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

299

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

306 2.3 Out-of-Band Token Acquisition

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

312 2.4 Trust Bootstrap

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

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

322

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

325

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

330

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

335 **3 Security Token Service Framework**

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

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

343
344 This framework does not define specific actions; each binding defines its own actions.
345 When requesting and returning security tokens additional parameters can be included in requests, or
346 provided in responses to indicate server-determined (or used) values. If a requestor specifies a specific
347 value that isn't supported by the recipient, then the recipient MAY fault with a `wst:InvalidRequest` (or
348 a more specific fault code), or they MAY return a token with their chosen parameters that the requestor
349 MAY then choose to discard because it doesn't meet their needs.

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

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

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

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

370 **3.1 Requesting a Security Token**

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

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

376 The syntax for this element is as follows:

377
378
379
380
381
382

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

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

384 */wst:RequestSecurityToken*

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

386 */wst:RequestSecurityToken/@Context*

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

391 */wst:RequestSecurityToken/wst:TokenType*

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

396 */wst:RequestSecurityToken/wst:RequestType*

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

403 */wst:RequestSecurityToken/wst:SecondaryParameters*

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

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

411 */wst:RequestSecurityToken/{any}*

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

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

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

419 3.2 Returning a Security Token

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

424

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

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

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

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

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

447 The syntax for this element is as follows:

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

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

454 */wst:RequestSecurityTokenResponse*

455 This is the response to a security token request.

456 */wst:RequestSecurityTokenResponse/@Context*

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

462 */wst:RequestSecurityTokenResponse/wst:TokenType*

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

464 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

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

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

476 */wst:RequestSecurityTokenResponse/{any}*

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

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

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

482 3.3 Binary Secrets

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

490 *.../wst:BinarySecret*

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

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

494 This OPTIONAL attribute indicates the type of secret being encoded. The pre-defined values are
 495 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)

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

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

499 3.4 Composition

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

504 4 Issuance Binding

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

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

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

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

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

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

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

523

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

527

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

530

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

534 4.1 Requesting a Security Token

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

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

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

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

553 */wst:RequestSecurityToken/wst:TokenType*

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

562 */wst:RequestSecurityToken/wsp:AppliesTo*

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

570 */wst:RequestSecurityToken/wst:Claims*

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

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

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

578 */wst:RequestSecurityToken/wst:Entropy*

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

583 */wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret*

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

590 */wst:RequestSecurityToken/wst:Lifetime*

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

597 */wst:RequestSecurityToken/wst:Lifetime/wsua:Created*

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

604 */wst:RequestSecurityToken/wst:Lifetime/wsua:Expires*

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

610

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

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

646 4.2 Request Security Token Collection

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

651

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

656

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

660

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

662

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

697

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

699

700 */wst:RequestSecurityTokenCollection*

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

704 4.2.1 Processing Rules

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

707

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

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

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

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

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

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

722

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

725

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

728

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

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

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

736

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

739

740 4.3 Returning a Security Token Collection

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

745 The syntax for this element is as follows:

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

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

750 */wst:RequestSecurityTokenResponseCollection*

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

753 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

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

755 4.4 Returning a Security Token

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

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

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

777 */wst:RequestSecurityTokenResponse/wsp:AppliesTo*

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

782 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

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

788 */wst:RequestSecurityTokenResponse/wst:RequestedAttachedReference*

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

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

799 */wst:RequestSecurityTokenResponse/wst:RequestedUnattachedReference*

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

806 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken*

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

817 */wst:RequestSecurityTokenResponse/wst:Entropy*

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

822 */wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret*

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

825 */wst:RequestSecurityTokenResponse/wst:Lifetime*

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

829 **4.4.1 wsp:AppliesTo in RST and RSTR**

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

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

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

848

849 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.

850 4.4.2 Requested References

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

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

866 4.4.3 Keys and Entropy

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

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

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

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

878

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

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

889

890 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

891 4.4.4 Returning Computed Keys

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

```

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

904

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

906 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken/wst:ComputedKey*
 907 The value of this element is a URI describing how to compute the key. While this can be
 908 extended by defining new URIs in other bindings and profiles, the following URI pre-defines one
 909 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.

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

911 4.4.5 Sample Response with Encrypted Secret

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

```

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

930 4.4.6 Sample Response with Unencrypted Secret

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

```

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

945 4.4.7 Sample Response with Token Reference

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

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

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

969 4.4.8 Sample Response without Proof-of-Possession Token

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

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

983

984 4.4.9 Zero or One Proof-of-Possession Token Case

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

991

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

```

995         <xyz:CustomToken xmlns:xyz="...">
996             ...
997         </xyz:CustomToken>
998     </wst:RequestedSecurityToken>
999     <wst:RequestedProofToken>
1000         <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
1001             ...
1002         </xenc:EncryptedKey>
1003     </wst:RequestedProofToken>
1004 </wst:RequestSecurityTokenResponse>
1005 </wst:RequestSecurityTokenResponseCollection>

```

4.4.10 More Than One Proof-of-Possession Tokens Case

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

```

1011 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
1012     <wst:RequestSecurityTokenResponse>
1013         ...
1014     </wst:RequestSecurityTokenResponse>
1015     <wst:RequestSecurityTokenResponse>
1016         ...
1017     </wst:RequestSecurityTokenResponse>
1018     ...
1019 </wst:RequestSecurityTokenResponseCollection>

```

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

/wst:RequestSecurityTokenResponseCollection

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

/wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse

Each RequestSecurityTokenResponse element is an individual RSTR.

/wst:RequestSecurityTokenResponseCollection/{any}

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

/wst:RequestSecurityTokenResponseCollection/@{any}

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

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

```

1033 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
1034     <wst:RequestSecurityTokenResponse>
1035         <wst:RequestedSecurityToken>
1036             <xyz:CustomToken xmlns:xyz="...">
1037                 ...
1038             </xyz:CustomToken>
1039         </wst:RequestedSecurityToken>
1040         <wst:RequestedProofToken>
1041             <xenc:EncryptedKey Id="newProofA">
1042                 ...
1043             </xenc:EncryptedKey>
1044         </wst:RequestedProofToken>
1045     </wst:RequestSecurityTokenResponse>
1046     <wst:RequestSecurityTokenResponse>

```

```

1047     <wst:RequestedSecurityToken>
1048         <abc:CustomToken xmlns:abc="...">
1049             ...
1050         </abc:CustomToken>
1051     </wst:RequestedSecurityToken>
1052     <wst:RequestedProofToken>
1053         <xenc:EncryptedKey Id="newProofB xmlns:xenc="...">
1054             ...
1055         </xenc:EncryptedKey>
1056     </wst:RequestedProofToken>
1057 </wst:RequestSecurityTokenResponse>
1058 </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.

```

1066 <wst:IssuedTokens xmlns:wst="...">
1067   <wst:RequestSecurityTokenResponse>
1068     ...
1069   </wst:RequestSecurityTokenResponse>+
1070 </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.

1094 The following example illustrates a response that includes multiple `<wst:IssuedTokens>` headers.

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

5 Renewal Binding

1135

1136 Using the token request framework, this section defines bindings for requesting security tokens to be
1137 renewed:

1138 **Renew** – A previously issued token with expiration is presented (and possibly proven) and the
1139 same token is returned with new expiration semantics.

1140

1141 For this binding, the following actions are defined to enable specific processing context to be conveyed to
1142 the recipient:

1143

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

1144

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

1145

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

1146 For this binding, the `<wst:RequestType>` element uses the following URI:

1147

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

1148 For this binding the token to be renewed is identified in the `<wst:RenewTarget>` element and the
1149 OPTIONAL `<wst:Lifetime>` element MAY be specified to request a specified renewal duration.

1150

1151 Other extensions MAY be specified in the request (and the response), but the key semantics (size, type,
1152 algorithms, scope, etc.) MUST NOT be altered during renewal. Token services MAY use renewal as an
1153 opportunity to rekey, so the renewal responses MAY include a new proof-of-possession token as well as
1154 entropy and key exchange elements.

1155

1156 The request MUST prove authorized use of the token being renewed unless the recipient trusts the
1157 requestor to make third-party renewal requests. In such cases, the third-party requestor MUST prove its
1158 identity to the issuer so that appropriate authorization occurs.

1159

1160 The original proof information SHOULD be proven during renewal.

1161

1162 The renewal binding allows the use of exchanges during the renewal process. Subsequent profiles MAY
1163 define restriction around the usage of exchanges.

1164

1165 During renewal, all key bearing tokens used in the renewal request MUST have an associated signature.
1166 All non-key bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the renewal
1167 response.

1168

1169 The renewal binding also defines several extensions to the request and response elements. The syntax
1170 for these extension elements is as follows (note that the base elements described above are included
1171 here italicized for completeness):

1172

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

1173

```
  <wst:TokenType>...</wst:TokenType>
```

1174

```
  <wst:RequestType>...</wst:RequestType>
```

1175

```
  ...
```

1176

```
  <wst:RenewTarget>...</wst:RenewTarget>
```

1177

```
  <wst:AllowPostdating/>
```

1178
1179

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

1180 */wst:RequestSecurityToken/wst:RenewTarget*

1181 This REQUIRED element identifies the token being renewed. This MAY contain a
1182 `<wsse:SecurityTokenReference>` pointing at the token to be renewed or it MAY directly contain
1183 the token to be renewed.

1184 */wst:RequestSecurityToken/wst:AllowPostdating*

1185 This OPTIONAL element indicates that returned tokens SHOULD allow requests for postdated
1186 tokens. That is, this allows for tokens to be issued that are not immediately valid (e.g., a token
1187 that can be used the next day).

1188 */wst:RequestSecurityToken/wst:Renewing*

1189 This OPTIONAL element is used to specify renew semantics for types that support this operation.

1190 */wst:RequestSecurityToken/wst:Renewing/@Allow*

1191 This OPTIONAL Boolean attribute is used to request a renewable token. If not specified, the
1192 default value is *true*. A renewable token is one whose lifetime can be extended. This is done
1193 using a renewal request. The recipient MAY allow renewals without demonstration of authorized
1194 use of the token or they MAY fault.

1195 */wst:RequestSecurityToken/wst:Renewing/@OK*

1196 This OPTIONAL Boolean attribute is used to indicate that a renewable token is acceptable if the
1197 requested duration exceeds the limit of the issuance service. That is, if *true* then tokens can be
1198 renewed after their expiration. It should be noted that the token is NOT valid after expiration for
1199 any operation except renewal. The default for this attribute is *false*. It NOT RECOMMENDED to
1200 use this as it can leave you open to certain types of security attacks. Issuers MAY restrict the
1201 period after expiration during which time the token can be renewed. This window is governed by
1202 the issuer's policy.

1203 The following example illustrates a request for a custom token that can be renewed.

1204
1205
1206
1207
1208
1209
1210
1211
1212

```
<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>
```

1213

1214 The following example illustrates a subsequent renewal request and response (note that for brevity only
1215 the request and response are illustrated). Note that the response includes an indication of the lifetime of
1216 the renewed token.

1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228

```
<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>
```

```
1229 <wst:RequestSecurityTokenResponse xmlns:wst="...">
1230   <wst:TokenType>
1231     http://example.org/mySpecialToken
1232   </wst:TokenType>
1233   <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1234   <wst:Lifetime>...</wst:Lifetime>
1235   ...
1236 </wst:RequestSecurityTokenResponse>
```

6 Cancel Binding

1237

1238 Using the token request framework, this section defines bindings for requesting security tokens to be
1239 cancelled:

1240 **Cancel** – When a previously issued token is no longer needed, the Cancel binding can be used
1241 to cancel the token, terminating its use. After canceling a token at the issuer, a STS MUST not
1242 validate or renew the token. A STS MAY initiate the revocation of a token, however, revocation is
1243 out of scope of this specification and a client MUST NOT rely on it. If a client needs to ensure the
1244 validity of a token, it MUST validate the token at the issuer.

1245

1246 For this binding, the following actions are defined to enable specific processing context to be conveyed to
1247 the recipient:

```
1248 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel  
1249 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Cancel  
1250 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal
```

1251 For this binding, the `<wst:RequestType>` element uses the following URI:

```
1252 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
```

1253 Extensions MAY be specified in the request (and the response), but the semantics are not defined by this
1254 binding.

1255

1256 The request MUST prove authorized use of the token being cancelled unless the recipient trusts the
1257 requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its
1258 identity to the issuer so that appropriate authorization occurs.

1259 In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key
1260 bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the closure response.

1261

1262 A cancelled token is no longer valid for authentication and authorization usages.

1263 On success a cancel response is returned. This is an RSTR message with the
1264 `<wst:RequestedTokenCancelled>` element in the body. On failure, a Fault is raised. It should be
1265 noted that the cancel RSTR is informational. That is, the security token is cancelled once the cancel
1266 request is processed.

1267

1268 The syntax of the request is as follows:

```
1269 <wst:RequestSecurityToken xmlns:wst="...">  
1270 <wst:RequestType>...</wst:RequestType>  
1271 ...  
1272 <wst:CancelTarget>...</wst:CancelTarget>  
1273 </wst:RequestSecurityToken>
```

1274 `/wst:RequestSecurityToken/wst:CancelTarget`

1275 This REQUIRED element identifies the token being cancelled. Typically this contains a
1276 `<wsse:SecurityTokenReference>` pointing at the token, but it could also carry the token
1277 directly.

1278 The following example illustrates a request to cancel a custom token.

```
1279 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
```

```

1280     <S11:Header>
1281       <wsse:Security>
1282         ...
1283     </wsse:Security>
1284 </S11:Header>
1285 <S11:Body>
1286   <wst:RequestSecurityToken>
1287     <wst:RequestType>
1288       http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
1289     </wst:RequestType>
1290     <wst:CancelTarget>
1291       ...
1292     </wst:CancelTarget>
1293   </wst:RequestSecurityToken>
1294 </S11:Body>
1295 </S11:Envelope>

```

1296 The following example illustrates a response to cancel a custom token.

```

1297 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
1298   <S11:Header>
1299     <wsse:Security>
1300       ...
1301     </wsse:Security>
1302   </S11:Header>
1303   <S11:Body>
1304     <wst:RequestSecurityTokenResponse>
1305       <wst:RequestedTokenCancelled/>
1306     </wst:RequestSecurityTokenResponse>
1307   </S11:Body>
1308 </S11:Envelope>

```

1309 6.1 STS-initiated Cancel Binding

1310 Using the token request framework, this section defines an OPTIONAL binding for requesting security
 1311 tokens to be cancelled by the STS:

1312 **STS-initiated Cancel** – When a previously issued token becomes invalid on the STS, the STS-
 1313 initiated Cancel binding can be used to cancel the token, terminating its use. After canceling a
 1314 token, a STS MUST not validate or renew the token. This binding can be only used when STS
 1315 can send one-way messages to the original token requestor.

1316
 1317 For this binding, the following actions are defined to enable specific processing context to be conveyed to
 1318 the recipient:

1319 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/STSCancel`

1320 For this binding, the `<wst:RequestType>` element uses the following URI:

1321 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/STSCancel`

1322 Extensions MAY be specified in the request, but the semantics are not defined by this binding.

1323
 1324 The request MUST prove authorized use of the token being cancelled unless the recipient trusts the
 1325 requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its
 1326 identity to the issuer so that appropriate authorization occurs.

1327 In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key
 1328 bearing tokens MUST be signed.

1329

1330 A cancelled token is no longer valid for authentication and authorization usages.

1331

1332 The mechanism to determine the availability of STS-initiated Cancel binding on the STS is out of scope of
1333 this specification. Similarly, how the client communicates its endpoint address to the STS so that it can
1334 send the STSCancel messages to the client is out of scope of this specification. This functionality is
1335 implementation specific and can be solved by different mechanisms that are not in scope for this
1336 specification.

1337

1338 This is a one-way operation, no response is returned from the recipient of the message.

1339

1340 The syntax of the request is as follows:

```
1341 <wst:RequestSecurityToken xmlns:wst="...">  
1342   <wst:RequestType>...</wst:RequestType>  
1343   ...  
1344   <wst:CancelTarget>...</wst:CancelTarget>  
1345 </wst:RequestSecurityToken>
```

1346 */wst:RequestSecurityToken/wst:CancelTarget*

1347 This REQUIRED element identifies the token being cancelled. Typically this contains a
1348 <wsse:SecurityTokenReference> pointing at the token, but it could also carry the token
1349 directly.

1350 The following example illustrates a request to cancel a custom token.

```
1351 <?xml version="1.0" encoding="utf-8"?>  
1352 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">  
1353   <S11:Header>  
1354     <wsse:Security>  
1355       ...  
1356     </wsse:Security>  
1357   </S11:Header>  
1358   <S11:Body>  
1359     <wst:RequestSecurityToken>  
1360       <wst:RequestType>  
1361         http://docs.oasis-open.org/ws-sx/ws-trust/200512/STSCancel  
1362       </wst:RequestType>  
1363       <wst:CancelTarget>  
1364         ...  
1365       </wst:CancelTarget>  
1366     </wst:RequestSecurityToken>  
1367   </S11:Body>  
1368 </S11:Envelope>
```

7 Validation Binding

1369

1370 Using the token request framework, this section defines bindings for requesting security tokens to be
1371 validated:

1372 **Validate** – The validity of the specified security token is evaluated and a result is returned. The
1373 result MAY be a status, a new token, or both.

1374

1375 It should be noted that for this binding, a SOAP Envelope MAY be specified as a "security token" if the
1376 requestor desires the envelope to be validated. In such cases the recipient SHOULD understand how to
1377 process a SOAP envelope and adhere to SOAP processing semantics (e.g., mustUnderstand) of the
1378 version of SOAP used in the envelope. Otherwise, the recipient SHOULD fault.

1379 For this binding, the following actions are defined to enable specific processing context to be conveyed to
1380 the recipient:

1381

```
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
```

1382

1383

1384

1385 For this binding, the `<wst:RequestType>` element contains the following URI:

1386

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

1387

1388 The request provides a token upon which the request is based and OPTIONAL tokens. As well, the
1389 OPTIONAL `<wst:TokenType>` element in the request can indicate desired type response token. This
1390 MAY be any supported token type or it MAY be the following URI indicating that only status is desired:

1391

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

1392

1393 For some use cases a status token is returned indicating the success or failure of the validation. In other
1394 cases a security token MAY be returned and used for authorization. This binding assumes that the
1395 validation requestor and provider are known to each other and that the general issuance parameters
1396 beyond requesting a token type, which is OPTIONAL, are not needed (note that other bindings and
1397 profiles could define different semantics).

1398

1399 For this binding an applicability scope (e.g., `<wsp:AppliesTo>`) need not be specified. It is assumed
1400 that the applicability of the validation response relates to the provided information (e.g. security token) as
1401 understood by the issuing service.

1402

1403 The validation binding does not allow the use of exchanges.

1404

1405 The RSTR for this binding carries the following element even if a token is returned (note that the base
1406 elements described above are included here italicized for completeness):

1407

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

1408

1409

1410

1411

1412

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

1413

1414

```

<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>

```

1415

1416

1417

1418

1419

1420

1421

1422

1423

1424 */wst:RequestSecurityToken/wst:ValidateTarget*

1425 This REQUIRED element identifies the token being validated. Typically this contains a
1426 <wsse:SecurityTokenReference> pointing at the token, but could also carry the token
1427 directly.

1428 */wst:RequestSecurityTokenResponse/wst:Status*

1429 When a validation request is made, this element MUST be in the response. The code value
1430 indicates the results of the validation in a machine-readable form. The accompanying text
1431 element allows for human textual display.

1432 */wst:RequestSecurityTokenResponse/wst:Status/wst:Code*

1433 This REQUIRED URI value provides a machine-readable status code. The following URIs are
1434 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

1435 */wst:RequestSecurityTokenResponse/wst:Status/wst:Reason*

1436 This OPTIONAL string provides human-readable text relating to the status code.

1437

1438 The following illustrates the syntax of a validation request and response. In this example no token is
1439 requested, just a status.

1440

```

<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>

```

1441

1442

1443

1444

1445

1446

1447

1448

1449

```

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

```

1450

1451

1452

1453
1454
1455
1456
1457
1458
1459

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

1460
1461

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

1462
1463
1464
1465
1466
1467
1468
1469

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

1470

1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482

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

8 Negotiation and Challenge Extensions

1483

1484 The general security token service framework defined above allows for a simple request and response for
1485 security tokens (possibly asynchronous). However, there are many scenarios where a set of exchanges
1486 between the parties is REQUIRED prior to returning (e.g., issuing) a security token. This section
1487 describes the extensions to the base WS-Trust mechanisms to enable exchanges for negotiation and
1488 challenges.

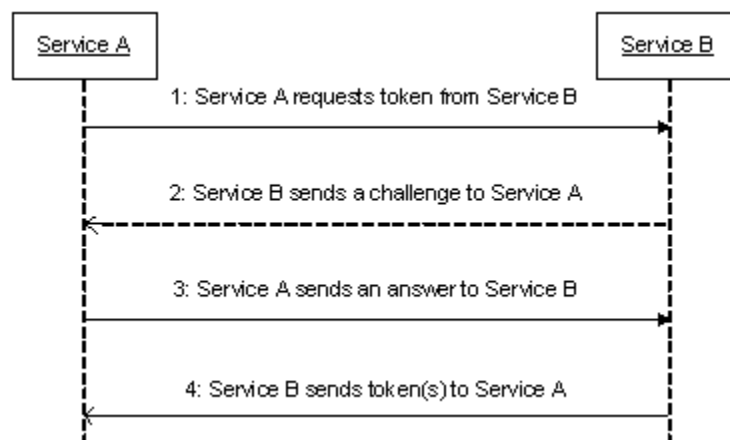
1489

1490 There are potentially different forms of exchanges, but one specific form, called "challenges", provides
1491 mechanisms in addition to those described in [WS-Security] for authentication. This section describes
1492 how general exchanges are issued and responded to within this framework. Other types of exchanges
1493 include, but are not limited to, negotiation, tunneling of hardware-based processing, and tunneling of
1494 legacy protocols.

1495

1496 The process is straightforward (illustrated here using a challenge):

1497



1498

- 1499 1. A requestor sends, for example, a `<wst:RequestSecurityToken>` message with a
1500 timestamp.
- 1501 2. The recipient does not trust the timestamp and issues a
1502 `<wst:RequestSecurityTokenResponse>` message with an embedded challenge.
- 1503 3. The requestor sends a `<wst:RequestSecurityTokenResponse>` message with an answer to
1504 the challenge.
- 1505 4. The recipient issues a `<wst:RequestSecurityTokenResponseCollection>` message with
1506 the issued security token and OPTIONAL proof-of-possession token.

1507

1508 It should be noted that the requestor might challenge the recipient in either step 1 or step 3. In which
1509 case, step 2 or step 4 contains an answer to the initiator's challenge. Similarly, it is possible that steps 2
1510 and 3 could iterate multiple times before the process completes (step 4).

1511

1512 The two services can use [WS-SecurityPolicy] to state their requirements and preferences for security
1513 tokens and encryption and signing algorithms (general policy intersection). This section defines
1514 mechanisms for legacy and more sophisticated types of negotiations.

1515 8.1 Negotiation and Challenge Framework

1516 The general mechanisms defined for requesting and returning security tokens are extensible. This
1517 section describes the general model for extending these to support negotiations and challenges.

1518

1519 The exchange model is as follows:

- 1520 1. A request is initiated with a `<wst:RequestSecurityToken>` that identifies the details of the
1521 request (and MAY contain initial negotiation/challenge information)
- 1522 2. A response is returned with a `<wst:RequestSecurityTokenResponse>` that contains
1523 additional negotiation/challenge information. Optionally, this MAY return token information in the
1524 form of a `<wst:RequestSecurityTokenResponseCollection>` (if the exchange is two legs
1525 long).
- 1526 3. If the exchange is not complete, the requestor uses a
1527 `<wst:RequestSecurityTokenResponse>` that contains additional negotiation/challenge
1528 information.
- 1529 4. The process repeats at step 2 until the negotiation/challenge is complete (a token is returned or a
1530 Fault occurs). In the case where token information is returned in the final leg, it is returned in the
1531 form of a `<wst:RequestSecurityTokenResponseCollection>`.

1532

1533 The negotiation/challenge information is passed in binding/profile-specific elements that are placed inside
1534 of the `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>` elements.

1535

1536 It is RECOMMENDED that at least the `<wsu:Timestamp>` element be included in messages (as per
1537 [\[WS-Security\]](#)) as a way to ensure freshness of the messages in the exchange. Other types of
1538 challenges MAY also be included. For example, a `<wsp:Policy>` element may be used to negotiate
1539 desired policy behaviors of both parties. Multiple challenges and responses MAY be included.

1540 8.2 Signature Challenges

1541 Exchange requests are issued by including an element that describes the exchange (e.g. challenge) and
1542 responses contain an element describing the response. For example, signature challenges are
1543 processed using the `<wst:SignChallenge>` element. The response is returned in a
1544 `<wst:SignChallengeResponse>` element. Both the challenge and the response elements are
1545 specified within the `<wst:RequestSecurityTokenResponse>` element. Some forms of negotiation
1546 MAY specify challenges along with responses to challenges from the other party. It should be noted that
1547 the requestor MAY provide exchange information (e.g. a challenge) to the recipient in the initial request.
1548 Consequently, these elements are also allowed within a `<wst:RequestSecurityToken>` element.

1549

1550 The syntax of these elements is as follows:

```
1551 <wst:SignChallenge xmlns:wst="...">  
1552   <wst:Challenge ...>...</wst:Challenge>  
1553 </wst:SignChallenge>
```

1554

```
1555 <wst:SignChallengeResponse xmlns:wst="...">  
1556   <wst:Challenge ...>...</wst:Challenge>  
1557 </wst:SignChallengeResponse>
```

1558

1559 The following describes the attributes and tags listed in the schema above:

1560 *.../wst:SignChallenge*

1561 This OPTIONAL element describes a challenge that requires the other party to sign a specified
1562 set of information.

1563 *.../wst:SignChallenge/wst:Challenge*

1564 This REQUIRED string element describes the value to be signed. In order to prevent certain
1565 types of attacks (such as man-in-the-middle), it is strongly RECOMMENDED that the challenge
1566 be bound to the negotiation. For example, the challenge SHOULD track (such as using a digest
1567 of) any relevant data exchanged such as policies, tokens, replay protection, etc. As well, if the
1568 challenge is happening over a secured channel, a reference to the channel SHOULD also be
1569 included. Furthermore, the recipient of a challenge SHOULD verify that the data tracked
1570 (digested) matches their view of the data exchanged. The exact algorithm MAY be defined in
1571 profiles or agreed to by the parties.

1572 *.../SignChallenge/{any}*

1573 This is an extensibility mechanism to allow additional negotiation types to be used.

1574 *.../wst:SignChallenge/@{any}*

1575 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added
1576 to the element.

1577 *.../wst:SignChallengeResponse*

1578 This OPTIONAL element describes a response to a challenge that requires the signing of a
1579 specified set of information.

1580 *.../wst:SignChallengeResponse/wst:Challenge*

1581 If a challenge was issued, the response MUST contain the challenge element exactly as
1582 received. As well, while the RSTR response SHOULD always be signed, if a challenge was
1583 issued, the RSTR MUST be signed (and the signature coupled with the message to prevent
1584 replay).

1585 *.../wst:SignChallengeResponse/{any}*

1586 This is an extensibility mechanism to allow additional negotiation types to be used.

1587 *.../wst:SignChallengeResponse/@{any}*

1588 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added
1589 to the element.

1590 **8.3 User Interaction Challenge**

1591 User interaction challenge requests are issued by including the <InteractiveChallenge> element. The
1592 response is returned in a <InteractiveChallengeResponse> element. Both the challenge and response
1593 elements are specified within the <wst:RequestSecurityTokenResponse> element. In some instances, the
1594 requestor may issue a challenge to the recipient or provide a response to an anticipated challenge from
1595 the recipient in the initial request. Consequently, these elements are also allowed within a
1596 <wst:RequestSecurityToken> element. The challenge/response exchange between client and server
1597 MAY be iterated over multiple legs before a final response is issued.

1598 Implementations SHOULD take into account the possibility that messages in either direction may be lost
1599 or duplicated. In the absence of a lower level protocol guaranteeing delivery of every message in order
1600 and exactly once, which retains the ordering of requests and responses traveling in opposite directions,
1601 implementations SHOULD observe the following procedures:

1602 The STS SHOULD:

1603 1. Never send a new request while an existing request is pending,

- 1604 2. Timeout requests and retransmit them.
- 1605 3. Silently discard responses when no request is pending.
- 1606
- 1607 The service consumer MAY:
- 1608 1. Respond to a repeated request with the same information
- 1609 2. Retain user input until the Challenge Iteration is complete in case it is necessary to repeat the
- 1610 response.
- 1611 Note that the xml:lang attribute may be used where allowed via attribute extensibility to specify a
- 1612 language of localized elements and attributes using the language codes specified in [RFC 3066].

1613 8.3.1 Challenge Format

1614 The syntax of the user interaction challenge element is as follows:

```

1615 <wst14:InteractiveChallenge xmlns:wst14="..." ...>
1616   <wst14:Title ...> xs:string </wst14:Title> ?
1617   <wst14:TextChallenge RefId="xs:anyURI" Label="xs:string"?
1618     MaxLen="xs:int"? HideText="xs:boolean"? ...>
1619     <wst14:Image MimeType="xs:string"> xs:base64Binary </wst14:Image> ?
1620   </wst14:TextChallenge> *
1621   <wst14:ChoiceChallenge RefId="xs:anyURI" Label="xs:string"?
1622     ExactlyOne="xs:boolean"? ...>
1623     <wst14:Choice RefId="xs:anyURI" Label="xs:string"? ...>
1624       <wst14:Image MimeType="xs:string"> xs:base64Binary </wst14:Image> ?
1625     </wst14:Choice> +
1626   </wst14:ChoiceChallenge> *
1627   < wst14:ContextData RefId="xs:anyURI"> xs:any </wst14:ContextData> *
1628   ...
1629 </wst14:InteractiveChallenge>

```

1630 The following describes the attributes and elements listed in the schema outlined above:

- 1631
- 1632 *.../wst14:InteractiveChallenge*
- 1633 A container element for a challenge that requires interactive user input.
- 1634 *.../wst14:InteractiveChallenge/wst14:Title*
- 1635 An OPTIONAL element that specifies an overall title text to be displayed to the user (e.g. a title
- 1636 describing the purpose or nature of the challenge). How the preferred language of the requestor
- 1637 is communicated to the STS is left up to implementations.
- 1638 *.../wst14:InteractiveChallenge/wst14:TextChallenge*
- 1639 An OPTIONAL element that specifies a challenge that requires textual input from the user.
- 1640 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@RefId*
- 1641 A REQUIRED attribute that specifies a reference identifier for this challenge element which is
- 1642 used to correlate the corresponding element in the response to the challenge.
- 1643 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@MaxLen*
- 1644 An OPTIONAL attribute that specifies the maximum length of the text string that is sent as the
- 1645 response to this text challenge. This value serves as a hint for the user interface software at the
- 1646 requestor which manifests the end-user experience for this challenge.
- 1647 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@HideText*
- 1648 An OPTIONAL attribute that specifies that the response to this text challenge MUST receive
- 1649 treatment as hidden text in any user interface. For example, the text entry may be displayed as a

1650 series of asterisks in the user interface. This attribute serves as a hint for the user interface
1651 software at the requestor which manifests the end-user experience for this challenge.

1652 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@Label*

1653 An OPTIONAL attribute that specifies a label for the text challenge item (e.g. a label for a text
1654 entry field) which will be shown to the user. How the preferred language of the requestor is
1655 communicated to the STS is left up to implementations.

1656 *.../wst14:InteractiveChallenge/wst14:TextChallenge/Image*

1657 An OPTIONAL element that contains a base64 encoded inline image specific to the text
1658 challenge item to be shown to the user (e.g. an image that the user must see to respond
1659 successfully to the challenge). The image presented is intended as an additional label to a
1660 challenge element which could be CAPTCHA, selection of a previously established image secret
1661 or any other means by which images can be used to challenge a user to interact in a way to
1662 satisfy a challenge.

1663 *.../wst14:InteractiveChallenge/wst14:TextChallenge/Image/@MimeType*

1664 A REQUIRED attribute that specifies a MIME type (e.g., image/gif, image/jpg) indicating the
1665 format of the image.

1666 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge*

1667 An OPTIONAL element that specifies a challenge that requires a choice among multiple items by
1668 the user.

1669 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/@RefId*

1670 A REQUIRED attribute that specifies a reference identifier for this challenge element which is
1671 used to correlate the corresponding element in the response to the challenge.

1672 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/@Label*

1673 An OPTIONAL attribute that specifies a title label for the choice challenge item (e.g., a text
1674 header describing the list of choices as a whole) which will be shown to the user. How the
1675 preferred language of the requestor is communicated to the STS is left up to implementations.

1676 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/@ExactlyOne*

1677 An OPTIONAL attribute that specifies if exactly once choice must be selected by the user from
1678 among the child element choices. The absence of this attribute implies the value "false" which
1679 means multiple choices can be selected.

1680 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice*

1681 A REQUIRED element that specifies a single choice item within the choice challenge.

1682 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/@RefId*

1683 A REQUIRED attribute that specifies a reference identifier for this specific choice item which is
1684 used to correlate the corresponding element in the response to the challenge.

1685 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/@Label*

1686 An OPTIONAL attribute that specifies a text label for the choice item (e.g., text describing the
1687 individual choice) which will be shown to the user. How the preferred language of the requestor is
1688 communicated to the STS is left up to implementations.

1689 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/wst14:Image*

1690 An OPTIONAL element that contains a base64 encoded inline image specific to the choice item
1691 to be shown to the user (e.g. an image that the user must see to respond successfully to the
1692 challenge). The image presented is intended as an additional label to a challenge element which
1693 could be CAPTCHA, selection of a previously established image secret or any other means by
1694 which images can be used to challenge a user to interact in a way to satisfy a challenge.

1695 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/wst14:Image/@MimeType*
1696 A REQUIRED attribute that specifies a MIME type (e.g., image/gif, image/jpg) indicating the
1697 format of the image.

1698 *.../wst14:InteractiveChallenge/wst14:ContextData*

1699 An OPTIONAL element that specifies a value that MUST be reflected back in the response to the
1700 challenge (e.g., cookie). The element may contain any value. The actual content is opaque to the
1701 requestor; it is not required to understand its structure or semantics. This can be used by an STS,
1702 for instance, to store information between the challenge/response exchanges that would
1703 otherwise be lost if the STS were to remain stateless.

1704 *.../wst14:InteractiveChallenge/wst14:ContextData/@RefId*

1705 A REQUIRED attribute that specifies a reference identifier for this context element which is used
1706 to correlate the corresponding element in the response to the challenge.

1707 *.../wst14:InteractiveChallenge/{any}*

1708 This is an extensibility mechanism to allow additional elements to be specified.

1709 *.../wst14:InteractiveChallenge/@{any}*

1710 This is an extensibility mechanism to allow additional attributes to be specified.

1711

1712 The syntax of the user interaction challenge response element is as follows:

```
1713 <wst14:InteractiveChallengeResponse xmlns:wst14="..." ...>  
1714 <wst14:TextChallengeResponse RefId="xs:anyURI" ...>  
1715   xs:string  
1716 </wst14:TextChallengeResponse> *  
1717 <wst14:ChoiceChallengeResponse RefId="xs:anyURI"> *  
1718   <wst14:ChoiceSelected RefId="xs:anyURI" /> *  
1719 </wst14:ChoiceChallengeResponse>  
1720 <wst14:ContextData RefId="xs:anyURI"> xs:any </wst14:ContextData> *  
1721   ...  
1722 </wst14:InteractiveChallengeResponse>
```

1723 The following describes the attributes and elements listed in the schema outlined above:

1724

1725 *.../wst14:InteractiveChallengeResponse*

1726 A container element for the response to a challenge that requires interactive user input.

1727 *.../wst14:InteractiveChallengeResponse/wst14:TextChallengeResponse*

1728 This element value contains the user input as the response to the original text challenge issued.

1729 *.../wst14:InteractiveChallengeResponse/wst14:TextChallengeResponse/@RefId*

1730 A required attribute that specifies the identifier for the text challenge element in the original
1731 challenge which can be used for correlation.

1732 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse*

1733 A container element for the response to a choice challenge.

1734 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse/@RefId*

1735 A required attribute that specifies the reference identifier for the choice challenge element in the
1736 original challenge which can be used for correlation.

1737 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse/wst14:ChoiceSelected*

1738 A required element that specifies a choice item selected by the user from the choice challenge.

1739 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse/wst14:ChoiceSelected/@RefId*

1740 A required attribute that specifies the reference identifier for the choice item in the original choice
1741 challenge which can be used for correlation.

1742 *.../wst14:InteractiveChallengeResponse/wst14:ContextData*

1743 An optional element that carries a context data item from the original challenge that is simply
1744 reflected back.

1745 *.../wst14:InteractiveChallengeResponse/wst14:ContextData/@RefId*

1746 A required attribute that specifies the reference identifier for the context data element in the
1747 original challenge which can be used for correlation.

1748 *.../wst14:InteractiveChallengeResponse/{any}*

1749 This is an extensibility mechanism to allow additional elements to be specified.

1750 *.../wst14:InteractiveChallengeResponse/@{any}*

1751 This is an extensibility mechanism to allow additional attributes to be specified.

1752 In order to prevent certain types of attacks, such as man-in-the-middle or replay of response, the
1753 challenge SHOULD be bound to the response. For example, an STS may use the <ContextData>
1754 element in the challenge to include a digest of any relevant replay protection data and verify that the
1755 same data is reflected back by the requestor.

1756 Text provided by the STS which is intended for display SHOULD NOT contain script, markup or other
1757 unprintable characters. Image data provided by the STS SHOULD NOT contain imbedded commands or
1758 other content except an image to be displayed.

1759 Service consumers MUST ignore any script, markup or other unprintable characters when displaying text
1760 sent by the STS. Service consumers MUST insure that image data does not contain imbedded
1761 commands or other content before displaying the image.

1762 **8.3.2 PIN and OTP Challenges**

1763 In some situations, some additional authentication step may be required, but the Consumer cannot
1764 determine this in advance of making the request. Two common cases that require user interaction are:

- 1765 • a challenge for a secret PIN,
- 1766 • a challenge for a one-time-password (OTP).

1767

1768 This challenge may be issued by an STS using the “text challenge” format within a user interaction
1769 challenge specified in the section above. A requestor responds to the challenge with the PIN/OTP value
1770 along with the corresponding @RefId attribute value for the text challenge which is used by the STS to
1771 correlate the response to the original challenge. This pattern of exchange requires that the requestor
1772 must receive the challenge first and thus learn the @RefId attribute value to include in the response.

1773

1774 There are cases where a requestor may know a priori that the STS challenges for a single PIN/OTP and,
1775 as an optimization, provide the response to the anticipated challenge in the initial request. The following
1776 distinguished URIs are defined for use as the value of the @RefId attribute of a
1777 <TextChallengeResponse> element to represent PIN and OTP responses using the optimization pattern.

1778

1779 <http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN>
1780 <http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/OTP>

1781

1782 An STS may choose not to support the optimization pattern above for PIN/OTP response. In some cases,
1783 an OTP challenge from the STS may include a dynamic random value that the requestor must feed into
1784 the OTP generating module before an OTP response is computed. In such cases, the optimized response
1785 pattern may not be usable.

1786 8.4 Binary Exchanges and Negotiations

1787 Exchange requests MAY also utilize existing binary formats passed within the WS-Trust framework. A
1788 generic mechanism is provided for this that includes a URI attribute to indicate the type of binary
1789 exchange.

1790

1791 The syntax of this element is as follows:

```
1792 <wst:BinaryExchange ValueType="..." EncodingType="..." xmlns:wst="...">  
1793 </wst:BinaryExchange>
```

1794 The following describes the attributes and tags listed in the schema above (note that the ellipses below
1795 indicate that this element MAY be placed in different containers. For this specification, these are limited
1796 to <wst:RequestSecurityToken> and <wst:RequestSecurityTokenResponse>):

1797 *.../wst:BinaryExchange*

1798 This OPTIONAL element is used for a security negotiation that involves exchanging binary blobs
1799 as part of an existing negotiation protocol. The contents of this element are blob-type-specific
1800 and are encoded using base64 (unless otherwise specified).

1801 *.../wst:BinaryExchange/@ValueType*

1802 This REQUIRED attribute specifies a URI to identify the type of negotiation (and the value space
1803 of the blob – the element's contents).

1804 *.../wst:BinaryExchange/@EncodingType*

1805 This REQUIRED attribute specifies a URI to identify the encoding format (if different from base64)
1806 of the negotiation blob. Refer to [WS-Security] for sample encoding format URIs.

1807 *.../wst:BinaryExchange/@{any}*

1808 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added
1809 to the element.

1810 Some binary exchanges result in a shared state/context between the involved parties. It is
1811 RECOMMENDED that at the conclusion of the exchange, a new token and proof-of-possession token be
1812 returned. A common approach is to use the negotiated key as a "secure channel" mechanism to secure
1813 the new token and proof-of-possession token.

1814 For example, an exchange might establish a shared secret S_x that can then be used to sign the final
1815 response and encrypt the proof-of-possession token.

1816 8.5 Key Exchange Tokens

1817 In some cases it MAY be necessary to provide a key exchange token so that the other party (either
1818 requestor or issuer) can provide entropy or key material as part of the exchange. Challenges MAY NOT
1819 always provide a usable key as the signature may use a signing-only certificate.

1820

1821 The section describes two OPTIONAL elements that can be included in RST and RSTR elements to
1822 indicate that a Key Exchange Token (KET) is desired, or to provide a KET.

1823 The syntax of these elements is as follows (Note that the ellipses below indicate that this element MAY be
1824 placed in different containers. For this specification, these are limited to

1825 <wst:RequestSecurityToken> and <wst:RequestSecurityTokenResponse>):

```
1826 <wst:RequestKET xmlns:wst="..." />
```

1827

1828

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

1829

1830 The following describes the attributes and tags listed in the schema above:

1831 *.../wst:RequestKET*

1832 This OPTIONAL element is used to indicate that the receiving party (either the original requestor
1833 or issuer) SHOULD provide a KET to the other party on the next leg of the exchange.

1834 *.../wst:KeyExchangeToken*

1835 This OPTIONAL element is used to provide a key exchange token. The contents of this element
1836 either contain the security token to be used for key exchange or a reference to it.

1837 8.6 Custom Exchanges

1838 Using the extensibility model described in this specification, any custom XML-based exchange can be
1839 defined in a separate binding/profile document. In such cases elements are defined which are carried in
1840 the RST and RSTR elements.

1841

1842 It should be noted that it is NOT REQUIRED that exchange elements be symmetric. That is, a specific
1843 exchange mechanism MAY use multiple elements at different times, depending on the state of the
1844 exchange.

1845 8.7 Signature Challenge Example

1846 Here is an example exchange involving a signature challenge. In this example, a service requests a
1847 custom token using a X.509 certificate for authentication. The issuer uses the exchange mechanism to
1848 challenge the requestor to sign a random value (to ensure message freshness). The requestor provides
1849 a signature of the requested data and, once validated, the issuer then issues the requested token.

1850

1851 The first message illustrates the initial request that is signed with the private key associated with the
1852 requestor's X.509 certificate:

1853

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..."  
1854   xmlns:wsu="..." xmlns:wst="...">  
1855   <S11:Header>  
1856     ...  
1857     <wsse:Security>  
1858       <wsse:BinarySecurityToken  
1859         wsu:Id="reqToken"  
1860         ValueType="...X509v3">  
1861         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
1862       </wsse:BinarySecurityToken>  
1863       <ds:Signature xmlns:ds="...">  
1864         ...  
1865         <ds:KeyInfo>  
1866           <wsse:SecurityTokenReference>  
1867             <wsse:Reference URI="#reqToken"/>  
1868           </wsse:SecurityTokenReference>  
1869         </ds:KeyInfo>  
1870       </ds:Signature>  
1871     </wsse:Security>  
1872     ...  
1873   </S11:Header>  
1874   <S11:Body>  
1875     <wst:RequestSecurityToken>  
1876       <wst:TokenType>
```

```

1877         http://example.org/mySpecialToken
1878     </wst:TokenType>
1879     <wst:RequestType>
1880         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1881     </wst:RequestType>
1882 </wst:RequestSecurityToken>
1883 </S11:Body>
1884 </S11:Envelope>

```

1885

1886 The issuer (recipient) service doesn't trust the sender's timestamp (or one wasn't specified) and issues a

1887 challenge using the exchange framework defined in this specification. This message is signed using the

1888 private key associated with the issuer's X.509 certificate and contains a random challenge that the

1889 requestor must sign:

```

1890 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wst="..."
1891     xmlns:wsp="..."
1892 <S11:Header>
1893     ...
1894     <wsse:Security>
1895         <wsse:BinarySecurityToken
1896             wsu:Id="issuerToken"
1897             ValueType="...X509v3">
1898             DFJHuedsujfnrnv45JZc0...
1899         </wsse:BinarySecurityToken>
1900         <ds:Signature xmlns:ds="...">
1901             ...
1902         </ds:Signature>
1903     </wsse:Security>
1904     ...
1905 </S11:Header>
1906 <S11:Body>
1907     <wst:RequestSecurityTokenResponse>
1908         <wst:SignChallenge>
1909             <wst:Challenge>Huehf...</wst:Challenge>
1910         </wst:SignChallenge>
1911     </wst:RequestSecurityTokenResponse>
1912 </S11:Body>
1913 </S11:Envelope>

```

1914

1915 The requestor receives the issuer's challenge and issues a response that is signed using the requestor's

1916 X.509 certificate and contains the challenge. The signature only covers the non-mutable elements of the

1917 message to prevent certain types of security attacks:

```

1918 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wst="..."
1919     xmlns:wsp="..."
1920 <S11:Header>
1921     ...
1922     <wsse:Security>
1923         <wsse:BinarySecurityToken
1924             wsu:Id="reqToken"
1925             ValueType="...X509v3">
1926             MIIIEZzCCA9CgAwIBAgIQEmtJZc0...
1927         </wsse:BinarySecurityToken>
1928         <ds:Signature xmlns:ds="...">
1929             ...
1930         </ds:Signature>
1931     </wsse:Security>
1932     ...
1933 </S11:Header>
1934 <S11:Body>
1935     <wst:RequestSecurityTokenResponse>

```

```

1936         <wst:SignChallengeResponse>
1937             <wst:Challenge>Huehf...</wst:Challenge>
1938         </wst:SignChallengeResponse>
1939     </wst:RequestSecurityTokenResponse>
1940 </S11:Body>
1941 </S11:Envelope>

```

1942

1943 The issuer validates the requestor's signature responding to the challenge and issues the requested

1944 token(s) and the associated proof-of-possession token. The proof-of-possession token is encrypted for

1945 the requestor using the requestor's public key.

```

1946 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
1947     xmlns:wst="..." xmlns:xenc="...">
1948     <S11:Header>
1949         ...
1950     <wsse:Security>
1951         <wsse:BinarySecurityToken
1952             wsu:Id="issuerToken"
1953             ValueType="...X509v3">
1954             DFJHuedsujfnrnv45JZc0...
1955         </wsse:BinarySecurityToken>
1956         <ds:Signature xmlns:ds="...">
1957             ...
1958         </ds:Signature>
1959     </wsse:Security>
1960     ...
1961 </S11:Header>
1962 <S11:Body>
1963     <wst:RequestSecurityTokenResponseCollection>
1964     <wst:RequestSecurityTokenResponse>
1965         <wst:RequestedSecurityToken>
1966             <xyz:CustomToken xmlns:xyz="...">
1967                 ...
1968             </xyz:CustomToken>
1969         </wst:RequestedSecurityToken>
1970         <wst:RequestedProofToken>
1971             <xenc:EncryptedKey Id="newProof">
1972                 ...
1973             </xenc:EncryptedKey>
1974         </wst:RequestedProofToken>
1975     </wst:RequestSecurityTokenResponse>
1976 </wst:RequestSecurityTokenResponseCollection>
1977 </S11:Body>
1978 </S11:Envelope>

```

1979 8.8 Challenge Examples

1980 8.8.1 Text and choice challenge

1981 Here is an example of a user interaction challenge using both text and choice challenges. In this example,

1982 a user requests a custom token using a username/password for authentication. The STS uses the

1983 challenge mechanism to challenge the user for additional information in the form of a secret question (i.e.,

1984 Mother's maiden name) and an age group choice. The challenge additionally includes one contextual

1985 data item that needs to be reflected back in the response. The user interactively provides the requested

1986 data and, once validated, the STS issues the requested token. All messages are sent over a protected

1987 transport using SSLv3.

1988

1989 The requestor sends the initial request that includes the username/password for authentication as follows.

1990

1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008

```
<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>
```

2009

2010 The STS issues a challenge for additional information using the user interaction challenge mechanism as
2011 follows.

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
2038

```
<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>
```

2039

2040 The requestor receives the challenge, provides the necessary user experience for soliciting the required
2041 inputs, and sends a response to the challenge back to the STS as follows.

2042

2043
2044
2045
2046
2047
2048
2049

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

```

2050     <wst14:TextChallengeResponse RefId="http://.../ref#text1">
2051         Goldstein
2052     </wst14:TextChallengeResponse>
2053     <wst14:ChoiceChallengeResponse RefId="http://.../ref#choiceGroupA">
2054         <wst14:ChoiceSelected RefId="http://.../ref#choice3" />
2055     </wst14:ChoiceChallengeResponse>
2056     <wst14:ContextData RefId="http://.../ref#cookie1">
2057         ...
2058     </wst14:ContextData>
2059     </wst14:InteractiveChallengeResponse>
2060 </wst:RequestSecurityTokenResponse>
2061 </S11:Body>
2062 </S11:Envelope>

```

2063

2064 The STS validates the response containing the inputs from the user, and issues the requested token as

2065 follows.

```

2066
2067 <S11:Envelope ...>
2068   <S11:Header>
2069     ...
2070   </S11:Header>
2071   <S11:Body>
2072     <wst:RequestSecurityTokenResponseCollection>
2073       <wst:RequestSecurityTokenResponse>
2074         <wst:RequestedSecurityToken>
2075           <xyz:CustomToken xmlns:xyz="...">
2076             ...
2077           </xyz:CustomToken>
2078         </wst:RequestedSecurityToken>
2079         <wst:RequestedProofToken>
2080           ...
2081         </wst:RequestedProofToken>
2082       </wst:RequestSecurityTokenResponse>
2083     </wst:RequestSecurityTokenResponseCollection>
2084   </S11:Body>
2085 </S11:Envelope>

```

2086

2087 8.8.2 PIN challenge

2088 Here is an example of a user interaction challenge using a text challenge for a secret PIN. In this

2089 example, a user requests a custom token using a username/password for authentication. The STS uses

2090 the text challenge mechanism for an additional PIN. The user interactively provides the PIN and, once

2091 validated, the STS issues the requested token. All messages are sent over a protected transport using

2092 SSLv3.

2093

2094 The requestor sends the initial request that includes the username/password for authentication as follows.

```

2095
2096 <S11:Envelope ...>
2097   <S11:Header>
2098     ...
2099   <wsse:Security>
2100     <wsse:UsernameToken>
2101       <wsse:Username>Zoe</wsse:Username>
2102       <wsse:Password Type="http://...#PasswordText">
2103         ILoveDogs
2104     </wsse:Password>

```

```
2105     </wsse:UsernameToken>
2106     </wsse:Security>
2107 </S11:Header>
2108 <S11:Body>
2109     <wst:RequestSecurityToken>
2110         <wst:TokenType>http://example.org/customToken</wst:TokenType>
2111         <wst:RequestType>...</wst:RequestType>
2112     </wst:RequestSecurityToken>
2113 </S11:Body>
2114 </S11:Envelope>
```

2115

2116 The STS issues a challenge for a secret PIN using the text challenge mechanism as follows.

2117

```
2118 <S11:Envelope ...>
2119   <S11:Header>
2120     ...
2121   </S11:Header>
2122   <S11:Body>
2123     <wst:RequestSecurityTokenResponse>
2124       <wst14:InteractiveChallenge xmlns:wst14="..." >
2125         <wst14:TextChallenge
2126           RefId="http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN"
2127           Label="Please enter your PIN" />
2128         </wst14:TextChallenge>
2129       </wst14:InteractiveChallenge>
2130     </wst:RequestSecurityTokenResponse>
2131   </S11:Body>
2132 </S11:Envelope>
```

2133

2134 The requestor receives the challenge, provides the necessary user experience for soliciting the PIN, and
2135 sends a response to the challenge back to the STS as follows.

2136

```
2137 <S11:Envelope ...>
2138   <S11:Header>
2139     ...
2140   </S11:Header>
2141   <S11:Body>
2142     <wst:RequestSecurityTokenResponse>
2143       <wst14:InteractiveChallengeResponse xmlns:wst14="..." >
2144         <wst14:TextChallengeResponse
2145           RefId="http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN">
2146           9988
2147         </wst14:TextChallengeResponse>
2148       </wst14:InteractiveChallengeResponse>
2149     </wst:RequestSecurityTokenResponse>
2150   </S11:Body>
2151 </S11:Envelope>
```

2152

2153 The STS validates the PIN response, and issues the requested token as follows.

2154

```
2155 <S11:Envelope ...>
2156   <S11:Header>
2157     ...
2158   </S11:Header>
2159   <S11:Body>
2160     <wst:RequestSecurityTokenResponseCollection>
```

```

2161 <wst:RequestSecurityTokenResponse>
2162   <wst:RequestedSecurityToken>
2163     <xyz:CustomToken xmlns:xyz="...">
2164       ...
2165     </xyz:CustomToken>
2166   </wst:RequestedSecurityToken>
2167   <wst:RequestedProofToken>
2168     ...
2169   </wst:RequestedProofToken>
2170 </wst:RequestSecurityTokenResponse>
2171 </wst:RequestSecurityTokenResponseCollection>
2172 </S11:Body>
2173 </S11:Envelope>

```

2174

2175 8.8.3 PIN challenge with optimized response

2176 The following example illustrates using the optimized PIN response pattern for the same exact challenge
 2177 as in the previous section. This reduces the number of message exchanges to two instead of four. All
 2178 messages are sent over a protected transport using SSLv3.

2179

2180 The requestor sends the initial request that includes the username/password for authentication as well as
 2181 the response to the anticipated PIN challenge as follows.

2182

```

2183 <S11:Envelope ...>
2184   <S11:Header>
2185     ...
2186   <wsse:Security>
2187     <wsse:UsernameToken>
2188       <wsse:Username>Zoe</wsse:Username>
2189       <wsse:Password Type="http://...#PasswordText">
2190         ILoveDogs
2191       </wsse:Password>
2192     </wsse:UsernameToken>
2193   </wsse:Security>
2194 </S11:Header>
2195 <S11:Body>
2196   <wst:RequestSecurityToken>
2197     <wst:TokenType>http://example.org/customToken</wst:TokenType>
2198     <wst:RequestType>...</wst:RequestType>
2199     <wst14:InteractiveChallengeResponse xmlns:wst14="..." >
2200       <wst14:TextChallengeResponse
2201         RefId="http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN">
2202         9988
2203       </wst14:TextChallengeResponse>
2204     </wst14:InteractiveChallengeResponse>
2205   </wst:RequestSecurityToken>
2206 </S11:Body>
2207 </S11:Envelope>

```

2208

2209 The STS validates the authentication credential as well as the optimized PIN response, and issues the
 2210 requested token as follows.

2211

```

2212 <S11:Envelope ...>
2213   <S11:Header>
2214     ...
2215   </S11:Header>

```



```

2216 <S11:Body>
2217   <wst:RequestSecurityTokenResponseCollection>
2218     <wst:RequestSecurityTokenResponse>
2219       <wst:RequestedSecurityToken>
2220         <xyz:CustomToken xmlns:xyz="...">
2221           ...
2222         </xyz:CustomToken>
2223       </wst:RequestedSecurityToken>
2224       <wst:RequestedProofToken>
2225         ...
2226       </wst:RequestedProofToken>
2227     </wst:RequestSecurityTokenResponse>
2228   </wst:RequestSecurityTokenResponseCollection>
2229 </S11:Body>
2230 </S11:Envelope>

```

2231

2232 8.9 Custom Exchange Example

2233 Here is another illustrating the syntax for a token request using a custom XML exchange. For brevity,
 2234 only the RST and RSTR elements are illustrated. Note that the framework allows for an arbitrary number
 2235 of exchanges, although this example illustrates the use of four legs. The request uses a custom
 2236 exchange element and the requestor signs only the non-mutable element of the message:

```

2237   <wst:RequestSecurityToken xmlns:wst="...">
2238     <wst:TokenType>
2239       http://example.org/mySpecialToken
2240     </wst:TokenType>
2241     <wst:RequestType>
2242       http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
2243     </wst:RequestType>
2244     <xyz:CustomExchange xmlns:xyz="...">
2245       ...
2246     </xyz:CustomExchange>
2247   </wst:RequestSecurityToken>

```

2248

2249 The issuer service (recipient) responds with another leg of the custom exchange and signs the response
 2250 (non-mutable aspects) with its token:

```

2251   <wst:RequestSecurityTokenResponse xmlns:wst="...">
2252     <xyz:CustomExchange xmlns:xyz="...">
2253       ...
2254     </xyz:CustomExchange>
2255   </wst:RequestSecurityTokenResponse>

```

2256

2257 The requestor receives the issuer's exchange and issues a response that is signed using the requestor's
 2258 token and continues the custom exchange. The signature covers all non-mutable aspects of the
 2259 message to prevent certain types of security attacks:

```

2260   <wst:RequestSecurityTokenResponse xmlns:wst="...">
2261     <xyz:CustomExchange xmlns:xyz="...">
2262       ...
2263     </xyz:CustomExchange>
2264   </wst:RequestSecurityTokenResponse>

```

2265

2266 The issuer processes the exchange and determines that the exchange is complete and that a token
 2267 should be issued. Consequently it issues the requested token(s) and the associated proof-of-possession
 2268 token. The proof-of-possession token is encrypted for the requestor using the requestor's public key.

```

2269 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
2270 <wst:RequestSecurityTokenResponse>
2271 <wst:RequestedSecurityToken>
2272 <xyz:CustomToken xmlns:xyz="...">
2273 ...
2274 </xyz:CustomToken>
2275 </wst:RequestedSecurityToken>
2276 <wst:RequestedProofToken>
2277 <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
2278 ...
2279 </xenc:EncryptedKey>
2280 </wst:RequestedProofToken>
2281 <wst:RequestedProofToken>
2282 <xenc:EncryptedKey xmlns:xenc="...">...</xenc:EncryptedKey>
2283 </wst:RequestedProofToken>
2284 </wst:RequestSecurityTokenResponse>
2285 </wst:RequestSecurityTokenResponseCollection>
  
```

2286 It should be noted that other example exchanges include the issuer returning a final custom exchange
 2287 element, and another example where a token isn't returned.

2288 8.10 Protecting Exchanges

2289 There are some attacks, such as forms of man-in-the-middle, that can be applied to token requests
 2290 involving exchanges. It is RECOMMENDED that the exchange sequence be protected. This MAY be
 2291 built into the exchange messages, but if metadata is provided in the RST or RSTR elements, then it is
 2292 subject to attack.

2293
 2294 Consequently, it is RECOMMENDED that keys derived from exchanges be linked cryptographically to the
 2295 exchange. For example, a hash can be computed by computing the SHA1 of the exclusive
 2296 canonicalization [XML-C14N] of all RST and RSTR elements in messages exchanged. This value can
 2297 then be combined with the exchanged secret(s) to create a new master secret that is bound to the data
 2298 both parties sent/received.

2299
 2300 To this end, the following computed key algorithm is defined to be OPTIONALLY used in these scenarios:

URI	Meaning
http://docs.oasis-open.org/ws-sx/ws-trust/200512/CK/HASH	The key is computed using P_SHA1 as follows: H=SHA1(ExclC14N(RST...RSTRs)) X=encrypting H using negotiated key and mechanism Key=P_SHA1(X,H+"CK-HASH") The octets for the "CK-HASH" string are the UTF-8 octets.

2301 8.11 Authenticating Exchanges

2302 After an exchange both parties have a shared knowledge of a key (or keys) that can then be used to
 2303 secure messages. However, in some cases it may be desired to have the issuer prove to the requestor

2304 that it knows the key (and that the returned metadata is valid) prior to the requestor using the data.
2305 However, until the exchange is actually completed it MAY be (and is often) inappropriate to use the
2306 computed keys. As well, using a token that hasn't been returned to secure a message may complicate
2307 processing since it crosses the boundary of the exchange and the underlying message security. This
2308 means that it MAY NOT be appropriate to sign the final leg of the exchange using the key derived from
2309 the exchange.

2310

2311 For this reason an authenticator is defined that provides a way for the issuer to verify the hash as part of
2312 the token issuance. Specifically, when an authenticator is returned, the
2313 `<wst:RequestSecurityTokenResponseCollection>` element is returned. This contains one
2314 RSTR with the token being returned as a result of the exchange and a second RSTR that contains the
2315 authenticator (this order SHOULD be used). When an authenticator is used, RSTRs MUST use the
2316 `@Context` element so that the authenticator can be correlated to the token issuance. The authenticator is
2317 separated from the RSTR because otherwise computation of the RST/RSTR hash becomes more
2318 complex. The authenticator is represented using the `<wst:Authenticator>` element as illustrated
2319 below:

2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330

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

2331

2332 The following describes the attributes and elements listed in the schema overview above (the ... notation
2333 below represents the path RSTRC/RSTR and is used for brevity):

2334 `.../wst:Authenticator`

2335 This OPTIONAL element provides verification (authentication) of a computed hash.

2336 `.../wst:Authenticator/wst:CombinedHash`

2337 This OPTIONAL element proves the hash and knowledge of the computed key. This is done by
2338 providing the base64 encoding of the first 256 bits of the P_SHA1 digest of the computed key and
2339 the concatenation of the hash determined for the computed key and the string "AUTH-HASH".
2340 Specifically, $P_SHA1(\textit{computed-key}, H + \textit{"AUTH-HASH"})_{0-255}$. The octets for the "AUTH-HASH"
2341 string are the UTF-8 octets.

2342

2343 This `<wst:CombinedHash>` element is OPTIONAL (and an open content model is used) to allow for
2344 different authenticators in the future.

2345

9 Key and Token Parameter Extensions

2346 This section outlines additional parameters that can be specified in token requests and responses.
2347 Typically they are used with issuance requests, but since all types of requests MAY issue security tokens
2348 they could apply to other bindings.

9.1 On-Behalf-Of Parameters

2350 In some scenarios the requestor is obtaining a token on behalf of another party. These parameters
2351 specify the issuer and original requestor of the token being used as the basis of the request. The syntax
2352 is as follows (note that the base elements described above are included here italicized for completeness):

```
2353 <wst:RequestSecurityToken xmlns:wst="...">  
2354   <wst:TokenType>...</wst:TokenType>  
2355   <wst:RequestType>...</wst:RequestType>  
2356   ...  
2357   <wst:OnBehalfOf>...</wst:OnBehalfOf>  
2358   <wst:Issuer>...</wst:Issuer>  
2359 </wst:RequestSecurityToken>
```

2360

2361 The following describes the attributes and elements listed in the schema overview above:

2362 */wst:RequestSecurityToken/wst:OnBehalfOf*

2363 This OPTIONAL element indicates that the requestor is making the request on behalf of another.
2364 The identity on whose behalf the request is being made is specified by placing a security token,
2365 <wsse:SecurityTokenReference> element, or <wsa:EndpointReference> element
2366 within the <wst:OnBehalfOf> element. The requestor MAY provide proof of possession of the
2367 key associated with the OnBehalfOf identity by including a signature in the RST security header
2368 generated using the OnBehalfOf token that signs the primary signature of the RST (i.e. endorsing
2369 supporting token concept from WS-SecurityPolicy). Additional signed supporting tokens
2370 describing the OnBehalfOf context MAY also be included within the RST security header.

2371 */wst:RequestSecurityToken/wst:Issuer*

2372 This OPTIONAL element specifies the issuer of the security token that is presented in the
2373 message. This element's type is an endpoint reference as defined in [\[WS-Addressing\]](#).

2374

2375 In the following illustrates the syntax for a proxy that is requesting a security token on behalf of another
2376 requestor or end-user.

```
2377 <wst:RequestSecurityToken xmlns:wst="...">  
2378   <wst:TokenType>...</wst:TokenType>  
2379   <wst:RequestType>...</wst:RequestType>  
2380   ...  
2381   <wst:OnBehalfOf>endpoint-reference</wst:OnBehalfOf>  
2382 </wst:RequestSecurityToken>
```

9.2 Key and Encryption Requirements

2384 This section defines extensions to the <wst:RequestSecurityToken> element for requesting specific
2385 types of keys or algorithms or key and algorithms as specified by a given policy in the return token(s). In
2386 some cases the service may support a variety of key types, sizes, and algorithms. These parameters
2387 allow a requestor to indicate its desired values. It should be noted that the issuer's policy indicates if input

2388 values must be adhered to and faults generated for invalid inputs, or if the issuer will provide alternative
2389 values in the response.

2390

2391 Although illustrated using the `<wst:RequestSecurityToken>` element, these options can also be
2392 returned in a `<wst:RequestSecurityTokenResponse>` element.

2393 The syntax for these OPTIONAL elements is as follows (note that the base elements described above are
2394 included here italicized for completeness):

```
2395 <wst:RequestSecurityToken xmlns:wst="...">  
2396   <wst:TokenType>...</wst:TokenType>  
2397   <wst:RequestType>...</wst:RequestType>  
2398   ...  
2399   <wst:AuthenticationType>...</wst:AuthenticationType>  
2400   <wst:KeyType>...</wst:KeyType>  
2401   <wst:KeySize>...</wst:KeySize>  
2402   <wst:SignatureAlgorithm>...</wst:SignatureAlgorithm>  
2403   <wst:EncryptionAlgorithm>...</wst:EncryptionAlgorithm>  
2404   <wst:CanonicalizationAlgorithm>...</wst:CanonicalizationAlgorithm>  
2405   <wst:ComputedKeyAlgorithm>...</wst:ComputedKeyAlgorithm>  
2406   <wst:Encryption>...</wst:Encryption>  
2407   <wst:ProofEncryption>...</wst:ProofEncryption>  
2408   <wst:KeyWrapAlgorithm>...</wst:KeyWrapAlgorithm>  
2409   <wst:UseKey Sig="..."> </wst:UseKey>  
2410   <wst:SignWith>...</wst:SignWith>  
2411   <wst:EncryptWith>...</wst:EncryptWith>  
2412 </wst:RequestSecurityToken>
```

2413

2414 The following describes the attributes and elements listed in the schema overview above:

2415 */wst:RequestSecurityToken/wst:AuthenticationType*

2416 This OPTIONAL URI element indicates the type of authentication desired, specified as a URI.
2417 This specification does not predefine classifications; these are specific to token services as is the
2418 relative strength evaluations. The relative assessment of strength is up to the recipient to
2419 determine. That is, requestors SHOULD be familiar with the recipient policies. For example, this
2420 might be used to indicate which of the four U.S. government authentication levels is REQUIRED.

2421 */wst:RequestSecurityToken/wst:KeyType*

2422 This OPTIONAL URI element indicates the type of key desired in the security token. The
2423 predefined values are identified in the table below. Note that some security token formats have
2424 fixed key types. It should be noted that new algorithms can be inserted by defining URIs in other
2425 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.

2426 */wst:RequestSecurityToken/wst:KeySize*

2427 This OPTIONAL integer element indicates the size of the key REQUIRED specified in number of
 2428 bits. This is a request, and, as such, the requested security token is not obligated to use the
 2429 requested key size. That said, the recipient SHOULD try to use a key at least as strong as the
 2430 specified value if possible. The information is provided as an indication of the desired strength of
 2431 the security.

2432 */wst:RequestSecurityToken/wst:SignatureAlgorithm*

2433 This OPTIONAL URI element indicates the desired signature algorithm used within the returned
 2434 token. This is specified as a URI indicating the algorithm (see [XML-Signature] for typical signing
 2435 algorithms).

2436 */wst:RequestSecurityToken/wst:EncryptionAlgorithm*

2437 This OPTIONAL URI element indicates the desired encryption algorithm used within the returned
 2438 token. This is specified as a URI indicating the algorithm (see [XML-Encrypt] for typical
 2439 encryption algorithms).

2440 */wst:RequestSecurityToken/wst:CanonicalizationAlgorithm*

2441 This OPTIONAL URI element indicates the desired canonicalization method used within the
 2442 returned token. This is specified as a URI indicating the method (see [XML-Signature] for typical
 2443 canonicalization methods).

2444 */wst:RequestSecurityToken/wst:ComputedKeyAlgorithm*

2445 This OPTIONAL URI element indicates the desired algorithm to use when computed keys are
 2446 used for issued tokens.

2447 */wst:RequestSecurityToken/wst:Encryption*

2448 This OPTIONAL element indicates that the requestor desires any returned secrets in issued
 2449 security tokens to be encrypted for the specified token. That is, so that the owner of the specified
 2450 token can decrypt the secret. Normally the security token is the contents of this element but a
 2451 security token reference MAY be used instead. If this element isn't specified, the token used as
 2452 the basis of the request (or specialized knowledge) is used to determine how to encrypt the key.

2453 */wst:RequestSecurityToken/wst:ProofEncryption*

2454 This OPTIONAL element indicates that the requestor desires any returned secrets in proof-of-
 2455 possession tokens to be encrypted for the specified token. That is, so that the owner of the
 2456 specified token can decrypt the secret. Normally the security token is the contents of this element
 2457 but a security token reference MAY be used instead. If this element isn't specified, the token
 2458 used as the basis of the request (or specialized knowledge) is used to determine how to encrypt
 2459 the key.

2460 */wst:RequestSecurityToken/wst:KeyWrapAlgorithm*

2461 This OPTIONAL URI element indicates the desired algorithm to use for key wrapping when STS
 2462 encrypts the issued token for the relying party using an asymmetric key.

2463 */wst:RequestSecurityToken/wst:UseKey*

2464 If the requestor wishes to use an existing key rather than create a new one, then this OPTIONAL
 2465 element can be used to reference the security token containing the desired key. This element
 2466 either contains a security token or a `<wsse:SecurityTokenReference>` element that
 2467 references the security token containing the key that SHOULD be used in the returned token. If
 2468 `<wst:KeyType>` is not defined and a key type is not implicitly known to the service, it MAY be
 2469 determined from the token (if possible). Otherwise this parameter is meaningless and is ignored.
 2470 Requestors SHOULD demonstrate authorized use of the public key provided.

2471 */wst:RequestSecurityToken/wst:UseKey/@Sig*

2472 In order to *authenticate* the key referenced, a signature MAY be used to prove the referenced
 2473 token/key. If specified, this OPTIONAL attribute indicates the ID of the corresponding signature

2474 (by URI reference). When this attribute is present, a key need not be specified inside the element
2475 since the referenced signature will indicate the corresponding token (and key).

2476 */wst:RequestSecurityToken/wst:SignWith*

2477 This OPTIONAL URI element indicates the desired signature algorithm to be used with the issued
2478 security token (typically from the policy of the target site for which the token is being requested.
2479 While any of these OPTIONAL elements MAY be included in RSTRs, this one is a likely
2480 candidate if there is some doubt (e.g., an X.509 cert that can only use DSS).

2481 */wst:RequestSecurityToken/wst:EncryptWith*

2482 This OPTIONAL URI element indicates the desired encryption algorithm to be used with the
2483 issued security token (typically from the policy of the target site for which the token is being
2484 requested.) While any of these OPTIONAL elements MAY be included in RSTRs, this one is a
2485 likely candidate if there is some doubt.

2486 The following summarizes the various algorithm parameters defined above. T is the issued token, P is the
2487 proof key.
2488

2489 **SignatureAlgorithm** - The signature algorithm to use to sign T

2490 **EncryptionAlgorithm** - The encryption algorithm to use to encrypt T

2491 **CanonicalizationAlgorithm** - The canonicalization algorithm to use when signing T

2492 **ComputedKeyAlgorithm** - The key derivation algorithm to use if using a symmetric key for P
2493 where P is computed using client, server, or combined entropy

2494 **Encryption** - The token/key to use when encrypting T

2495 **ProofEncryption** - The token/key to use when encrypting P

2496 **UseKey** - This is P. This is generally used when the client supplies a public-key that it wishes to
2497 be embedded in T as the proof key

2498 **SignWith** - The signature algorithm the client intends to employ when using P to
2499 sign

2500 The encryption algorithms further differ based on whether the issued token contains asymmetric key or
2501 symmetric key. Furthermore, they differ based on what type of key is used to protect the issued token
2502 from the STS to the relying party. The following cases can occur:

2503 T contains symmetric key/STS uses symmetric key to encrypt T for RP

2504 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP
2505 when using the proof key (e.g. AES256)

2506 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to
2507 encrypt the T (e.g. AES256)

2508

2509 T contains symmetric key/STS uses asymmetric key to encrypt T for RP

2510 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP
2511 when using the proof key (e.g. AES256)

2512 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to
2513 encrypt T for RP (e.g. AES256)

2514 **KeyWrapAlgorithm** – used to indicate the KeyWrap algorithm that the STS SHOULD use to
2515 wrap the generated key that is used to encrypt the T for RP

2516

2517 T contains asymmetric key/STS uses symmetric key to encrypt T for RP

2518 **EncryptWith** – used to indicate the KeyWrap algorithm that the client will use to

2519 protect the symmetric key that is used to protect messages to RP when using the proof key (e.g.
2520 RSA-OAEP-MGF1P)

2521 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to
2522 encrypt T for RP (e.g. AES256)

2523

2524 T contains asymmetric key/STS uses asymmetric key to encrypt T for RP

2525 **EncryptWith** - used to indicate the KeyWrap algorithm that the client will use to
2526 protect symmetric key that is used to protect message to RP when using the proof
2527 key (e.g. RSA-OAEP-MGF1P)

2528 **EncryptionAlgorithm** - used to indicate the symmetric algorithm that the STS SHOULD use to
2529 encrypt T for RP (e.g. AES256)

2530 **KeyWrapAlgorithm** – used to indicate the KeyWrap algorithm that the STS SHOULD use to
2531 wrap the generated key that is used to encrypt the T for RP

2532

2533 The example below illustrates a request that utilizes several of these parameters. A request is made for a
2534 custom token using a username and password as the basis of the request. For security, this token is
2535 encrypted (see "encUsername") for the recipient using the recipient's public key and referenced in the
2536 encryption manifest. The message is protected by a signature using a public key from the sender and
2537 authorized by the username and password.

2538

2539 The requestor would like the custom token to contain a 1024-bit public key whose value can be found in
2540 the key provided with the "proofSignature" signature (the key identified by "requestProofToken"). The
2541 token should be signed using RSA-SHA1 and encrypted for the token identified by
2542 "requestEncryptionToken". The proof should be encrypted using the token identified by
2543 "requestProofToken".

```
2544 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."  
2545     xmlns:wst="..." xmlns:ds="..." xmlns:xenc="...">  
2546   <S11:Header>  
2547     ...  
2548     <wsse:Security>  
2549       <xenc:ReferenceList>...</xenc:ReferenceList>  
2550       <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>  
2551       <wsse:BinarySecurityToken wsu:Id="requestEncryptionToken"  
2552         ValueType="...SomeTokenType" xmlns:x="...">  
2553         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
2554       </wsse:BinarySecurityToken>  
2555       <wsse:BinarySecurityToken wsu:Id="requestProofToken"  
2556         ValueType="...SomeTokenType" xmlns:x="...">  
2557         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
2558       </wsse:BinarySecurityToken>  
2559       <ds:Signature Id="proofSignature">  
2560         ... signature proving requested key ...  
2561         ... key info points to the "requestedProofToken" token ...  
2562       </ds:Signature>  
2563     </wsse:Security>  
2564     ...  
2565   </S11:Header>  
2566   <S11:Body wsu:Id="req">  
2567     <wst:RequestSecurityToken>  
2568       <wst:TokenType>  
2569         http://example.org/mySpecialToken  
2570       </wst:TokenType>  
2571     <wst:RequestType>
```



```

2572         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
2573     </wst:RequestType>
2574 <wst:KeyType>
2575     http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey
2576 </wst:KeyType>
2577 <wst:KeySize>1024</wst:KeySize>
2578 <wst:SignatureAlgorithm>
2579     http://www.w3.org/2000/09/xmlldsig#rsa-sha1
2580 </wst:SignatureAlgorithm>
2581 <wst:Encryption>
2582     <Reference URI="#requestEncryptionToken"/>
2583 </wst:Encryption>
2584 <wst:ProofEncryption>
2585     <wsse:Reference URI="#requestProofToken"/>
2586 </wst:ProofEncryption>
2587 <wst:UseKey Sig="#proofSignature"/>
2588 </wst:RequestSecurityToken>
2589 </S11:Body>
2590 </S11:Envelope>

```

2591 9.3 Delegation and Forwarding Requirements

2592 This section defines extensions to the `<wst:RequestSecurityToken>` element for indicating
2593 delegation and forwarding requirements on the requested security token(s).

2594 The syntax for these extension elements is as follows (note that the base elements described above are
2595 included here italicized for completeness):

```

2596 <wst:RequestSecurityToken xmlns:wst="...">
2597   <wst:TokenType>...</wst:TokenType>
2598   <wst:RequestType>...</wst:RequestType>
2599   ...
2600   <wst:DelegateTo>...</wst:DelegateTo>
2601   <wst:Forwardable>...</wst:Forwardable>
2602   <wst:Delegatable>...</wst:Delegatable>
2603   <wst:ActAs>...</wst:ActAs>
2604 </wst:RequestSecurityToken>

```

2605 */wst:RequestSecurityToken/wst:DelegateTo*

2606 This OPTIONAL element indicates that the requested or issued token be delegated to another
2607 identity. The identity receiving the delegation is specified by placing a security token or
2608 `<wsse:SecurityTokenReference>` element within the `<wst:DelegateTo>` element.

2609 */wst:RequestSecurityToken/wst:Forwardable*

2610 This OPTIONAL element, of type `xs:boolean`, specifies whether the requested security token
2611 SHOULD be marked as "Forwardable". In general, this flag is used when a token is normally
2612 bound to the requestor's machine or service. Using this flag, the returned token MAY be used
2613 from any source machine so long as the key is correctly proven. The default value of this flag is
2614 true.

2615 */wst:RequestSecurityToken/wst:Delegatable*

2616 This OPTIONAL element, of type `xs:boolean`, specifies whether the requested security token
2617 SHOULD be marked as "Delegatable". Using this flag, the returned token MAY be delegated to
2618 another party. This parameter SHOULD be used in conjunction with `<wst:DelegateTo>`. The
2619 default value of this flag is false.

2620 */wst:RequestSecurityToken/wst:ActAs*

2621 This OPTIONAL element indicates that the requested token is expected to contain information
2622 about the identity represented by the content of this element and the token requestor intends to
2623 use the returned token to act as this identity. The identity that the requestor wants to act-as is

2624 specified by placing a security token or <wsse:SecurityTokenReference> element within the
2625 <wst:ActAs> element.

2626 The following illustrates the syntax of a request for a custom token that can be delegated to the indicated
2627 recipient (specified in the binary security token) and used in the specified interval.

```
2628 <wst:RequestSecurityToken xmlns:wst="...">  
2629 <wst:TokenType>  
2630 http://example.org/mySpecialToken  
2631 </wst:TokenType>  
2632 <wst:RequestType>  
2633 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
2634 </wst:RequestType>  
2635 <wst:DelegateTo>  
2636 <wsse:BinarySecurityToken xmlns:wsse="...">  
2637 ...  
2638 </wsse:BinarySecurityToken>  
2639 </wst:DelegateTo>  
2640 <wst:Delegatable>true</wst:Delegatable>  
2641 </wst:RequestSecurityToken>
```

2642 9.4 Policies

2643 This section defines extensions to the <wst:RequestSecurityToken> element for passing policies.
2644

2645 The syntax for these extension elements is as follows (note that the base elements described above are
2646 included here italicized for completeness):

```
2647 <wst:RequestSecurityToken xmlns:wst="...">  
2648 <wst:TokenType>...</wst:TokenType>  
2649 <wst:RequestType>...</wst:RequestType>  
2650 ...  
2651 <wsp:Policy xmlns:wsp="...">...</wsp:Policy>  
2652 <wsp:PolicyReference xmlns:wsp="...">...</wsp:PolicyReference>  
2653 </wst:RequestSecurityToken>
```

2654

2655 The following describes the attributes and elements listed in the schema overview above:

2656 */wst:RequestSecurityToken/wsp:Policy*

2657 This OPTIONAL element specifies a policy (as defined in [WS-Policy]) that indicates desired
2658 settings for the requested token. The policy specifies defaults that can be overridden by the
2659 elements defined in the previous sections.

2660 */wst:RequestSecurityToken/wsp:PolicyReference*

2661 This OPTIONAL element specifies a reference to a policy (as defined in [WS-Policy]) that
2662 indicates desired settings for the requested token. The policy specifies defaults that can be
2663 overridden by the elements defined in the previous sections.

2664

2665 The following illustrates the syntax of a request for a custom token that provides a set of policy
2666 statements about the token or its usage requirements.

```
2667 <wst:RequestSecurityToken xmlns:wst="...">  
2668 <wst:TokenType>  
2669 http://example.org/mySpecialToken  
2670 </wst:TokenType>  
2671 <wst:RequestType>  
2672 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
2673 </wst:RequestType>  
2674 <wsp:Policy xmlns:wsp="...">
```

2675
2676
2677

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

2678 9.5 Authorized Token Participants

2679 This section defines extensions to the `<wst:RequestSecurityToken>` element for passing information
2680 about which parties are authorized to participate in the use of the token. This parameter is typically used
2681 when there are additional parties using the token or if the requestor needs to clarify the actual parties
2682 involved (for some profile-specific reason).

2683 It should be noted that additional participants will need to prove their identity to recipients in addition to
2684 proving their authorization to use the returned token. This typically takes the form of a second signature
2685 or use of transport security.

2686

2687 The syntax for these extension elements is as follows (note that the base elements described above are
2688 included here italicized for completeness):

```
2689 <wst:RequestSecurityToken xmlns:wst="...">  
2690 <wst:TokenType>...</wst:TokenType>  
2691 <wst:RequestType>...</wst:RequestType>  
2692 ...  
2693 <wst:Participants>  
2694 <wst:Primary>...</wst:Primary>  
2695 <wst:Participant>...</wst:Participant>  
2696 </wst:Participants>  
2697 </wst:RequestSecurityToken>
```

2698

2699 The following describes elements and attributes used in a `<wsc:SecurityContextToken>` element.

2700 */wst:RequestSecurityToken/wst:Participants/*

2701 This OPTIONAL element specifies the participants sharing the security token. Arbitrary types
2702 MAY be used to specify participants, but a typical case is a security token or an endpoint
2703 reference (see [\[WS-Addressing\]](#)).

2704 */wst:RequestSecurityToken/wst:Participants/wst:Primary*

2705 This OPTIONAL element specifies the primary user of the token (if one exists).

2706 */wst:RequestSecurityToken/wst:Participants/wst:Participant*

2707 This OPTIONAL element specifies participant (or multiple participants by repeating the element)
2708 that play a (profile-dependent) role in the use of the token or who are allowed to use the token.

2709 */wst:RequestSecurityToken/wst:Participants/{any}*

2710 This is an extensibility option to allow other types of participants and profile-specific elements to
2711 be specified.

2712 10 Key Exchange Token Binding

2713 Using the token request framework, this section defines a binding for requesting a key exchange token
2714 (KET). That is, if a requestor desires a token that can be used to encrypt key material for a recipient.

2715
2716 For this binding, the following actions are defined to enable specific processing context to be conveyed to
2717 the recipient:

```
2718 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET  
2719 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KET  
2720 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal
```

2721
2722 For this binding, the `RequestType` element contains the following URI:

```
2723 http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET
```

2724
2725 For this binding very few parameters are specified as input. **OPTIONALLY** the `<wst:TokenType>`
2726 element can be specified in the request can indicate desired type response token carrying the key for key
2727 exchange; however, this isn't commonly used.

2728 The applicability scope (e.g. `<wsp:AppliesTo>`) **MAY** be specified if the requestor desires a key
2729 exchange token for a specific scope.

2730
2731 It is **RECOMMENDED** that the response carrying the key exchange token be secured (e.g., signed by the
2732 issuer or someone who can speak on behalf of the target for which the KET applies).

2733
2734 Care should be taken when using this binding to prevent possible man-in-the-middle and substitution
2735 attacks. For example, responses to this request **SHOULD** be secured using a token that can speak for
2736 the desired endpoint.

2737
2738 The RSTR for this binding carries the `<RequestedSecurityToken>` element even if a token is returned
2739 (note that the base elements described above are included here italicized for completeness):

```
2740 <wst:RequestSecurityToken xmlns:wst="...">  
2741   <wst:TokenType>...</wst:TokenType>  
2742   <wst:RequestType>...</wst:RequestType>  
2743   ...  
2744 </wst:RequestSecurityToken>
```

```
2745  
2746 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2747   <wst:RequestSecurityTokenResponse>  
2748     <wst:TokenType>...</wst:TokenType>  
2749     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>  
2750     ...  
2751   </wst:RequestSecurityTokenResponse>  
2752 </wst:RequestSecurityTokenResponseCollection>
```

2753
2754 The following illustrates the syntax for requesting a key exchange token. In this example, the KET is
2755 returned encrypted for the requestor since it had the credentials available to do that. Alternatively the

2756 request could be made using transport security (e.g. TLS) and the key could be returned directly using
2757 <wst:BinarySecret>.

```
2758 <wst:RequestSecurityToken xmlns:wst="...">  
2759 <wst:RequestType>  
2760 http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET  
2761 </wst:RequestType>  
2762 </wst:RequestSecurityToken>
```

```
2763  
2764 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2765 <wst:RequestSecurityTokenResponse>  
2766 <wst:RequestedSecurityToken>  
2767 <xenc:EncryptedKey xmlns:xenc="...">...</xenc:EncryptedKey>  
2768 </wst:RequestedSecurityToken>  
2769 </wst:RequestSecurityTokenResponse>  
2770 </wst:RequestSecurityTokenResponseCollection>
```

2771

11 Error Handling

2772 There are many circumstances where an *error* can occur while processing security information. Errors
 2773 use the SOAP Fault mechanism. Note that the reason text provided below is RECOMMENDED, but
 2774 alternative text MAY be provided if more descriptive or preferred by the implementation. The tables
 2775 below are defined in terms of SOAP 1.1. For SOAP 1.2, the Fault/Code/Value is env:Sender (as defined
 2776 in SOAP 1.2) and the Fault/Code/Subcode/Value is the *faultcode* below and the Fault/Reason/Text is the
 2777 *faultstring* below. It should be noted that profiles MAY provide second-level detail fields, but they should
 2778 be careful not to introduce security vulnerabilities when doing so (e.g., by providing too detailed
 2779 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

2780

12 Security Considerations

2781 As stated in the Goals section of this document, this specification is meant to provide extensible
2782 framework and flexible syntax, with which one could implement various security mechanisms. This
2783 framework and syntax by itself does not provide any guarantee of security. When implementing and using
2784 this framework and syntax, one must make every effort to ensure that the result is not vulnerable to any
2785 one of a wide range of attacks.

2786

2787 It is not feasible to provide a comprehensive list of security considerations for such an extensible set of
2788 mechanisms. A complete security analysis must be conducted on specific solutions based on this
2789 specification. Below we illustrate some of the security concerns that often come up with protocols of this
2790 type, but we stress that this *is not an exhaustive list of concerns*.

2791

2792 The following statements about signatures and signing apply to messages sent on unsecured channels.

2793

2794 It is critical that all the security-sensitive message elements must be included in the scope of the
2795 message signature. As well, the signatures for conversation authentication must include a timestamp,
2796 nonce, or sequence number depending on the degree of replay prevention required as described in [[WS-
2797 Security](#)] and the UsernameToken Profile. Also, conversation establishment should include the policy so
2798 that supported algorithms and algorithm priorities can be validated.

2799

2800 It is required that security token issuance messages be signed to prevent tampering. If a public key is
2801 provided, the request should be signed by the corresponding private key to prove ownership. As well,
2802 additional steps should be taken to eliminate replay attacks (refer to [[WS-Security](#)] for additional
2803 information). Similarly, all token references should be signed to prevent any tampering.

2804

2805 Security token requests are susceptible to denial-of-service attacks. Care should be taken to mitigate
2806 such attacks as is warranted by the service.

2807

2808 For security, tokens containing a symmetric key or a password should only be sent to parties who have a
2809 need to know that key or password.

2810

2811 For privacy, tokens containing personal information (either in the claims, or indirectly by identifying who is
2812 currently communicating with whom) should only be sent according to the privacy policies governing
2813 these data at the respective organizations.

2814

2815 For some forms of multi-message exchanges, the exchanges are susceptible to attacks whereby
2816 signatures are altered. To address this, it is suggested that a signature confirmation mechanism be used.
2817 In such cases, each leg should include the confirmation of the previous leg. That is, leg 2 includes
2818 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
2819 correctness of the message outside of the message body.

2820

2821 There are many other security concerns that one may need to consider in security protocols. The list
2822 above should not be used as a "check list" instead of a comprehensive security analysis.

2823

2824 It should be noted that use of unsolicited RSTRs implies that the recipient is prepared to accept such
2825 issuances. Recipients should ensure that such issuances are properly authorized and recognize their
2826 use could be used in denial-of-service attacks.

2827 In addition to the consideration identified here, readers should also review the security considerations in
2828 [\[WS-Security\]](#).

2829

2830 Both token cancellation bindings defined in this specification require that the STS MUST NOT validate or
2831 renew the token after it has been successfully canceled. The STS must take care to ensure that the token
2832 is properly invalidated before confirming the cancel request or sending the cancel notification to the client.
2833 This can be more difficult if the token validation or renewal logic is physically separated from the issuance
2834 and cancellation logic. It is out of scope of this spec how the STS propagates the token cancellation to its
2835 other components. If STS cannot ensure that the token was properly invalidated it MUST NOT send the
2836 cancel notification or confirm the cancel request to the client.

2837

2838

13 Conformance

2839

2840 An implementation conforms to this specification if it satisfies all of the MUST or REQUIRED level
2841 requirements defined within this specification. A SOAP Node MUST NOT use the XML namespace
2842 identifier for this specification (listed in Section 1.3) within SOAP Envelopes unless it is compliant with this
2843 specification.

2844 This specification references a number of other specifications (see the table above). In order to comply
2845 with this specification, an implementation MUST implement the portions of referenced specifications
2846 necessary to comply with the required provisions of this specification. Additionally, the implementation of
2847 the portions of the referenced specifications that are specifically cited in this specification MUST comply
2848 with the rules for those portions as established in the referenced specification.

2849 Additionally normative text within this specification takes precedence over normative outlines (as
2850 described in section 1.5.1), which in turn take precedence over the XML Schema [XML Schema Part 1,
2851 Part 2] and WSDL [WSDL 1.1] descriptions. That is, the normative text in this specification further
2852 constrains the schemas and/or WSDL that are part of this specification; and this specification contains
2853 further constraints on the elements defined in referenced schemas.

2854 This specification defines a number of extensions; compliant services are NOT REQUIRED to implement
2855 OPTIONAL features defined in this specification. However, if a service implements an aspect of the
2856 specification, it MUST comply with the requirements specified (e.g. related "MUST" statements). If an
2857 OPTIONAL message is not supported, then the implementation SHOULD Fault just as it would for any
2858 other unrecognized/unsupported message. If an OPTIONAL message is supported, then the
2859 implementation MUST satisfy all of the MUST and REQUIRED sections of the message.

2860

A. Key Exchange

2861 Key exchange is an integral part of token acquisition. There are several mechanisms by which keys are
2862 exchanged using [WS-Security] and WS-Trust. This section highlights and summarizes these
2863 mechanisms. Other specifications and profiles MAY provide additional details on key exchange.

2864

2865 Care must be taken when employing a key exchange to ensure that the mechanism does not provide an
2866 attacker with a means of discovering information that could only be discovered through use of secret
2867 information (such as a private key).

2868

2869 It is therefore important that a shared secret should only be considered as trustworthy as its source. A
2870 shared secret communicated by means of the direct encryption scheme described in section I.1 is
2871 acceptable if the encryption key is provided by a completely trustworthy key distribution center (this is the
2872 case in the Kerberos model). Such a key would not be acceptable for the purposes of decrypting
2873 information from the source that provided it since an attacker might replay information from a prior
2874 transaction in the hope of learning information about it.

2875

2876 In most cases the other party in a transaction is only imperfectly trustworthy. In these cases both parties
2877 SHOULD contribute entropy to the key exchange by means of the <wst:entropy> element.

A.1 Ephemeral Encryption Keys

2879 The simplest form of key exchange can be found in [WS-Security] for encrypting message data. As
2880 described in [WS-Security] and [XML-Encrypt], when data is encrypted, a temporary key can be used to
2881 perform the encryption which is, itself, then encrypted using the <xenc:EncryptedKey> element.

2882

2883 The illustrates the syntax for encrypting a temporary key using the public key in an issuer name and serial
2884 number:

2885

2886

2887

2888

2889

2890

2891

2892

2893

2894

2895

2896

2897

2898

```
<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

2900 When a request sends a message to an issuer to request a token, the client can provide proposed key
2901 material using the <wst:Entropy> element. If the issuer doesn't contribute any key material, this is
2902 used as the secret (key). This information is encrypted for the issuer either using
2903 <xenc:EncryptedKey> or by using a transport security. If the requestor provides key material that the

2904 recipient doesn't accept, then the issuer SHOULD reject the request. Note that the issuer need not return
2905 the key provided by the requestor.

2906

2907 The following illustrates the syntax of a request for a custom security token and includes a secret that is
2908 to be used for the key. In this example the entropy is encrypted for the issuer (if transport security was
2909 used for confidentiality then the <wst:Entropy> element would contain a <wst:BinarySecret>
2910 element):

```
2911 <wst:RequestSecurityToken xmlns:wst="...">  
2912 <wst:TokenType>  
2913   http://example.org/mySpecialToken  
2914 </wst:TokenType>  
2915 <wst:RequestType>  
2916   http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
2917 </wst:RequestType>  
2918 <wst:Entropy>  
2919   <xenc:EncryptedData xmlns:xenc="...">...</xenc:EncryptedData>  
2920 </wst:Entropy>  
2921 </wst:RequestSecurityToken>
```

2922 **A.3 Issuer-Provided Keys**

2923 If a requestor fails to provide key material, then issued proof-of-possession tokens contain an issuer-
2924 provided secret that is encrypted for the requestor (either using <xenc:EncryptedKey> or by using a
2925 transport security).

2926

2927 The following illustrates the syntax of a token being returned with an associated proof-of-possession
2928 token that is encrypted using the requestor's public key.

```
2929 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2930 <wst:RequestSecurityTokenResponse>  
2931 <wst:RequestedSecurityToken>  
2932 <xyz:CustomToken xmlns:xyz="...">  
2933   ...  
2934 </xyz:CustomToken>  
2935 </wst:RequestedSecurityToken>  
2936 <wst:RequestedProofToken>  
2937 <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">  
2938   ...  
2939 </xenc:EncryptedKey>  
2940 </wst:RequestedProofToken>  
2941 </wst:RequestSecurityTokenResponse>  
2942 </wst:RequestSecurityTokenResponseCollection>
```

2943 **A.4 Composite Keys**

2944 The safest form of key exchange/generation is when both the requestor and the issuer contribute to the
2945 key material. In this case, the request sends encrypted key material. The issuer then returns additional
2946 encrypted key material. The actual secret (key) is computed using a function of the two pieces of data.
2947 Ideally this secret is never used and, instead, keys derived are used for message protection.

2948

2949 The following example illustrates a server, having received a request with requestor entropy returning its
2950 own entropy, which is used in conjunction with the requestor's to generate a key. In this example the
2951 entropy is not encrypted because the transport is providing confidentiality (otherwise the
2952 <wst:Entropy> element would have an <xenc:EncryptedData> element).

```

2953 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
2954   <wst:RequestSecurityTokenResponse>
2955     <wst:RequestedSecurityToken>
2956       <xyz:CustomToken xmlns:xyz="...">
2957         ...
2958       </xyz:CustomToken>
2959     </wst:RequestedSecurityToken>
2960     <wst:Entropy>
2961       <wst:BinarySecret>UIH...</wst:BinarySecret>
2962     </wst:Entropy>
2963   </wst:RequestSecurityTokenResponse>
2964 </wst:RequestSecurityTokenResponseCollection>

```

2965 **A.5 Key Transfer and Distribution**

2966 There are also a few mechanisms where existing keys are transferred to other parties.

2967 **A.5.1 Direct Key Transfer**

2968 If one party has a token and key and wishes to share this with another party, the key can be directly
2969 transferred. This is accomplished by sending an RSTR (either in the body or header) to the other party.
2970 The RSTR contains the token and a proof-of-possession token that contains the key encrypted for the
2971 recipient.

2972

2973 In the following example a custom token and its associated proof-of-possession token are known to party
2974 A who wishes to share them with party B. In this example, A is a member in a secure on-line chat
2975 session and is inviting B to join the conversation. After authenticating B, A sends B an RSTR. The RSTR
2976 contains the token and the key is communicated as a proof-of-possession token that is encrypted for B:

```

2977 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
2978   <wst:RequestSecurityTokenResponse>
2979     <wst:RequestedSecurityToken>
2980       <xyz:CustomToken xmlns:xyz="...">
2981         ...
2982       </xyz:CustomToken>
2983     </wst:RequestedSecurityToken>
2984     <wst:RequestedProofToken>
2985       <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">
2986         ...
2987       </xenc:EncryptedKey>
2988     </wst:RequestedProofToken>
2989   </wst:RequestSecurityTokenResponse>
2990 </wst:RequestSecurityTokenResponseCollection>

```

2991 **A.5.2 Brokered Key Distribution**

2992 A third party MAY also act as a broker to transfer keys. For example, a requestor may obtain a token and
2993 proof-of-possession token from a third-party STS. The token contains a key encrypted for the target
2994 service (either using the service's public key or a key known to the STS and target service). The proof-of-
2995 possession token contains the same key encrypted for the requestor (similarly this can use public or
2996 symmetric keys).

2997

2998 In the following example a custom token and its associated proof-of-possession token are returned from a
2999 broker B to a requestor R for access to service S. The key for the session is contained within the custom
3000 token encrypted for S using either a secret known by B and S or using S's public key. The same secret is
3001 encrypted for R and returned as the proof-of-possession token:

```

3002 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
3003   <wst:RequestSecurityTokenResponse>
3004     <wst:RequestedSecurityToken>
3005       <xyz:CustomToken xmlns:xyz="...">
3006         ...
3007         <xenc:EncryptedKey xmlns:xenc="...">
3008           ...
3009         </xenc:EncryptedKey>
3010         ...
3011       </xyz:CustomToken>
3012     </wst:RequestedSecurityToken>
3013     <wst:RequestedProofToken>
3014       <xenc:EncryptedKey Id="newProof">
3015         ...
3016       </xenc:EncryptedKey>
3017     </wst:RequestedProofToken>
3018   </wst:RequestSecurityTokenResponse>
3019 </wst:RequestSecurityTokenResponseCollection>

```

3020 **A.5.3 Delegated Key Transfer**

3021 Key transfer can also take the form of delegation. That is, one party transfers the right to use a key
3022 without actually transferring the key. In such cases, a delegation token, e.g. XrML, is created that
3023 identifies a set of rights and a delegation target and is secured by the delegating party. That is, one key
3024 indicates that another key can use a subset (or all) of its rights. The delegate can provide this token and
3025 prove itself (using its own key – the delegation target) to a service. The service, assuming the trust
3026 relationships have been established and that the delegator has the right to delegate, can then authorize
3027 requests sent subject to delegation rules and trust policies.

3028
3029 In this example a custom token is issued from party A to party B. The token indicates that B (specifically
3030 B's key) has the right to submit purchase orders. The token is signed using a secret key known to the
3031 target service T and party A (the key used to ultimately authorize the requests that B makes to T), and a
3032 new session key that is encrypted for T. A proof-of-possession token is included that contains the
3033 session key encrypted for B. As a result, B is *effectively* using A's key, but doesn't actually know the key.

```

3034 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
3035   <wst:RequestSecurityTokenResponse>
3036     <wst:RequestedSecurityToken>
3037       <xyz:CustomToken xmlns:xyz="...">
3038         ...
3039         <xyz:DelegateTo>B</xyz:DelegateTo>
3040         <xyz:DelegateRights>
3041           SubmitPurchaseOrder
3042         </xyz:DelegateRights>
3043         <xenc:EncryptedKey xmlns:xenc="...">
3044           ...
3045         </xenc:EncryptedKey>
3046         <ds:Signature xmlns:ds="...">...</ds:Signature>
3047         ...
3048       </xyz:CustomToken>
3049     </wst:RequestedSecurityToken>
3050     <wst:RequestedProofToken>
3051       <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">
3052         ...
3053       </xenc:EncryptedKey>
3054     </wst:RequestedProofToken>
3055   </wst:RequestSecurityTokenResponse>
3056 </wst:RequestSecurityTokenResponseCollection>

```

3057 **A.5.4 Authenticated Request/Reply Key Transfer**

3058 In some cases the RST/RSTR mechanism is not used to transfer keys because it is part of a simple
3059 request/reply. However, there may be a desire to ensure mutual authentication as part of the key
3060 transfer. The mechanisms of [WS-Security] can be used to implement this scenario.

3061

3062 Specifically, the sender wishes the following:

- 3063 • Transfer a key to a recipient that they can use to secure a reply
- 3064 • Ensure that only the recipient can see the key
- 3065 • Provide proof that the sender issued the key

3066

3067 This scenario could be supported by encrypting and then signing. This would result in roughly the
3068 following steps:

- 3069 1. Encrypt the message using a generated key
- 3070 2. Encrypt the key for the recipient
- 3071 3. Sign the encrypted form, any other relevant keys, and the encrypted key

3072

3073 However, if there is a desire to sign prior to encryption then the following general process is used:

- 3074 1. Sign the appropriate message parts using a random key (or ideally a key derived from a random
3075 key)
- 3076 2. Encrypt the appropriate message parts using the random key (or ideally another key derived from
3077 the random key)
- 3078 3. Encrypt the random key for the recipient
- 3079 4. Sign just the encrypted key

3080

3081 This would result in a <wsse:Security> header that looks roughly like the following:

```
3082 <wsse:Security xmlns:wsse="..." xmlns:wssu="..."  
3083     xmlns:ds="..." xmlns:xenc="...">  
3084   <wsse:BinarySecurityToken wssu:Id="myToken">  
3085     ...  
3086   </wsse:BinarySecurityToken>  
3087   <ds:Signature>  
3088     ...signature over #secret using token #myToken...  
3089   </ds:Signature>  
3090   <xenc:EncryptedKey Id="secret">  
3091     ...  
3092   </xenc:EncryptedKey>  
3093   <xenc:ReferenceList>  
3094     ...manifest of encrypted parts using token #secret...  
3095   </xenc:ReferenceList>  
3096   <ds:Signature>  
3097     ...signature over key message parts using token #secret...  
3098   </ds:Signature>  
3099 </wsse:Security>
```

3100

3101 As well, instead of an <xenc:EncryptedKey> element, the actual token could be passed using
3102 <xenc:EncryptedData>. The result might look like the following:

```
3103 <wsse:Security xmlns:wsse="..." xmlns:wssu="..."  
3104     xmlns:ds="..." xmlns:xenc="...">
```

```
3105 <wsse:BinarySecurityToken wsu:Id="myToken">
3106     ...
3107 </wsse:BinarySecurityToken>
3108 <ds:Signature>
3109     ...signature over #secret or #Esecret using token #myToken...
3110 </ds:Signature>
3111 <xenc:EncryptedData Id="Esecret">
3112     ...Encrypted version of a token with Id="secret"...
3113 </xenc:EncryptedData>
3114 <xenc:ReferenceList>
3115     ...manifest of encrypted parts using token #secret...
3116 </xenc:ReferenceList>
3117 <ds:Signature>
3118     ...signature over key message parts using token #secret...
3119 </ds:Signature>
3120 </wsse:Security>
```

3121 **A.6 Perfect Forward Secrecy**

3122 In some situations it is desirable for a key exchange to have the property of perfect forward secrecy. This
3123 means that it is impossible to reconstruct the shared secret even if the private keys of the parties are
3124 disclosed.

3125
3126 The most straightforward way to attain perfect forward secrecy when using asymmetric key exchange is
3127 to dispose of one's key exchange key pair periodically (or even after every key exchange), replacing it
3128 with a fresh one. Of course, a freshly generated public key must still be authenticated (using any of the
3129 methods normally available to prove the identity of a public key's owner).

3130
3131 The perfect forward secrecy property MAY be achieved by specifying a `<wst:entropy>` element that
3132 contains an `<xenc:EncryptedKey>` that is encrypted under a public key pair created for use in a single
3133 key agreement. The public key does not require authentication since it is only used to provide additional
3134 entropy. If the public key is modified, the key agreement will fail. Care should be taken, when using this
3135 method, to ensure that the now-secret entropy exchanged via the `<wst:entropy>` element is not
3136 revealed elsewhere in the protocol (since such entropy is often assumed to be publicly revealed plaintext,
3137 and treated accordingly).

3138
3139 Although any public key scheme might be used to achieve perfect forward secrecy (in either of the above
3140 methods) it is generally desirable to use an algorithm that allows keys to be generated quickly. The Diffie-
3141 Hellman key exchange is often used for this purpose since generation of a key only requires the
3142 generation of a random integer and calculation of a single modular exponent.

3143

B. WSDL

3144 The WSDL below does not fully capture all the possible message exchange patterns, but captures the
3145 typical message exchange pattern as described in this document.

```

3146 <?xml version="1.0"?>
3147 <wsdl:definitions
3148     targetNamespace="http://docs.oasis-open.org/ws-sx/ws-
3149 trust/200512/wsdl"
3150     xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/wsdl"
3151     xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
3152     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
3153     xmlns:xs="http://www.w3.org/2001/XMLSchema"
3154     xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
3155 >
3156 <!-- this is the WS-I BP-compliant way to import a schema -->
3157 <wsdl:types>
3158     <xs:schema
3159         <xs:import
3160             namespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
3161             schemaLocation="http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-
3162 trust.xsd"/>
3163     </xs:schema>
3164 </wsdl:types>
3165
3166 <!-- WS-Trust defines the following GEDs -->
3167 <wsdl:message name="RequestSecurityTokenMsg">
3168     <wsdl:part name="request" element="wst:RequestSecurityToken" />
3169 </wsdl:message>
3170 <wsdl:message name="RequestSecurityTokenResponseMsg">
3171     <wsdl:part name="response"
3172         element="wst:RequestSecurityTokenResponse" />
3173 </wsdl:message>
3174 <wsdl:message name="RequestSecurityTokenCollectionMsg">
3175     <wsdl:part name="requestCollection"
3176         element="wst:RequestSecurityTokenCollection"/>
3177 </wsdl:message>
3178 <wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
3179     <wsdl:part name="responseCollection"
3180         element="wst:RequestSecurityTokenResponseCollection"/>
3181 </wsdl:message>
3182
3183 <!-- This portType an example of a Requestor (or other) endpoint that
3184     Accepts SOAP-based challenges from a Security Token Service -->
3185 <wsdl:portType name="WSSecurityRequestor">
3186     <wsdl:operation name="Challenge">
3187         <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
3188         <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
3189     </wsdl:operation>
3190 </wsdl:portType>
3191
3192 <!-- This portType is an example of an STS supporting full protocol -->
3193 <wsdl:portType name="SecurityTokenService">
3194     <wsdl:operation name="Cancel">
3195         <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3196 trust/200512/RST/Cancel" message="tns:RequestSecurityTokenMsg"/>
3197         <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3198 trust/200512/RSTR/CancelFinal" message="tns:RequestSecurityTokenResponseMsg"/>
3199     </wsdl:operation>
3200     <wsdl:operation name="Issue">

```



```

3201     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3202 trust/200512/RST/Issue" message="tns:RequestSecurityTokenMsg"/>
3203     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3204 trust/200512/RSTRC/IssueFinal"
3205 message="tns:RequestSecurityTokenResponseCollectionMsg"/>
3206     </wsdl:operation>
3207     <wsdl:operation name="Renew">
3208     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3209 trust/200512/RST/Renew" message="tns:RequestSecurityTokenMsg"/>
3210     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3211 trust/200512/RSTR/RenewFinal" message="tns:RequestSecurityTokenResponseMsg"/>
3212     </wsdl:operation>
3213     <wsdl:operation name="Validate">
3214     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3215 trust/200512/RST/Validate" message="tns:RequestSecurityTokenMsg"/>
3216     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3217 trust/200512/RSTR/ValidateFinal
3218 message="tns:RequestSecurityTokenResponseMsg"/>
3219     </wsdl:operation>
3220     <wsdl:operation name="KeyExchangeToken">
3221     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3222 trust/200512/RST/KET" message="tns:RequestSecurityTokenMsg"/>
3223     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3224 trust/200512/RSTR/KETFinal" message="tns:RequestSecurityTokenResponseMsg"/>
3225     </wsdl:operation>
3226     <wsdl:operation name="RequestCollection">
3227     <wsdl:input message="tns:RequestSecurityTokenCollectionMsg"/>
3228     <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg"/>
3229     </wsdl:operation>
3230 </wsdl:portType>
3231
3232 <!-- This portType is an example of an endpoint that accepts
3233 Unsolicited RequestSecurityTokenResponse messages -->
3234 <wsdl:portType name="SecurityTokenResponseService">
3235     <wsdl:operation name="RequestSecurityTokenResponse">
3236     <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
3237     </wsdl:operation>
3238 </wsdl:portType>
3239
3240 </wsdl:definitions>
3241

```

3242

C. Acknowledgements

3243 The following individuals have participated in the creation of this specification and are gratefully
3244 acknowledged:

3245 **Original Authors of the initial contribution:**

3246 Steve Anderson, OpenNetwork
3247 Jeff Bohren, OpenNetwork
3248 Toufic Boubez, Layer 7
3249 Marc Chanliau, Computer Associates
3250 Giovanni Della-Libera, Microsoft
3251 Brendan Dixon, Microsoft
3252 Praerit Garg, Microsoft
3253 Martin Gudgin (Editor), Microsoft
3254 Phillip Hallam-Baker, VeriSign
3255 Maryann Hondo, IBM
3256 Chris Kaler, Microsoft
3257 Hal Lockhart, Oracle Corporation
3258 Robin Martherus, Oblix
3259 Hiroshi Maruyama, IBM
3260 Anthony Nadalin (Editor), IBM
3261 Nataraj Nagaratnam, IBM
3262 Andrew Nash, Reactivity
3263 Rob Philpott, RSA Security
3264 Darren Platt, Ping Identity
3265 Hemma Prafullchandra, VeriSign
3266 Maneesh Sahu, Actional
3267 John Shewchuk, Microsoft
3268 Dan Simon, Microsoft
3269 Davanum Srinivas, Computer Associates
3270 Elliot Waingold, Microsoft
3271 David Waite, Ping Identity
3272 Doug Walter, Microsoft
3273 Riaz Zolfonoon, RSA Security

3274

3275 **Original Acknowledgments of the initial contribution:**

3276 Paula Austel, IBM
3277 Keith Ballinger, Microsoft
3278 Bob Blakley, IBM
3279 John Brezak, Microsoft
3280 Tony Cowan, IBM
3281 Cédric Fournet, Microsoft
3282 Vijay Gajjala, Microsoft
3283 HongMei Ge, Microsoft
3284 Satoshi Hada, IBM
3285 Heather Hinton, IBM
3286 Slava Kavsan, RSA Security
3287 Scott Konersmann, Microsoft
3288 Leo Laferriere, Computer Associates

- 3289 Paul Leach, Microsoft
- 3290 Richard Levinson, Computer Associates
- 3291 John Linn, RSA Security
- 3292 Michael McIntosh, IBM
- 3293 Steve Millet, Microsoft
- 3294 Birgit Pfitzmann, IBM
- 3295 Fumiko Satoh, IBM
- 3296 Keith Stobie, Microsoft
- 3297 T.R. Vishwanath, Microsoft
- 3298 Richard Ward, Microsoft
- 3299 Hervey Wilson, Microsoft
- 3300
- 3301 **TC Members during the development of this specification:**
- 3302 Don Adams, Tibco Software Inc.
- 3303 Jan Alexander, Microsoft Corporation
- 3304 Steve Anderson, BMC Software
- 3305 Donal Arundel, IONA Technologies
- 3306 Howard Bae, Oracle Corporation
- 3307 Abbie Barbir, Nortel Networks Limited
- 3308 Charlton Barreto, Adobe Systems
- 3309 Mighael Botha, Software AG, Inc.
- 3310 Toufic Boubez, Layer 7 Technologies Inc.
- 3311 Norman Brickman, Mitre Corporation
- 3312 Melissa Brumfield, Booz Allen Hamilton
- 3313 Lloyd Burch, Novell
- 3314 Geoff Bullen, Microsoft Corporation
- 3315 Scott Cantor, Internet2
- 3316 Greg Carpenter, Microsoft Corporation
- 3317 Steve Carter, Novell
- 3318 Ching-Yun (C.Y.) Chao, IBM
- 3319 Martin Chapman, Oracle Corporation
- 3320 Kate Cherry, Lockheed Martin
- 3321 Henry (Hyenvui) Chung, IBM
- 3322 Luc Clement, Systinet Corp.
- 3323 Paul Cotton, Microsoft Corporation
- 3324 Glen Daniels, Sonic Software Corp.
- 3325 Peter Davis, Neustar, Inc.
- 3326 Martijn de Boer, SAP AG
- 3327 Duane DeCouteau, Veterans Health Administration
- 3328 Werner Dittmann, Siemens AG
- 3329 Abdeslem DJAOUI, CCLRC-Rutherford Appleton Laboratory
- 3330 Fred Dushin, IONA Technologies
- 3331 Petr Dvorak, Systinet Corp.
- 3332 Colleen Evans, Microsoft Corporation

3333 Ruchith Fernando, WSO2
3334 Mark Fussell, Microsoft Corporation
3335 Vijay Gajjala, Microsoft Corporation
3336 Marc Goodner, Microsoft Corporation
3337 Hans Granqvist, VeriSign
3338 Martin Gudgin, Microsoft Corporation
3339 Tony Gullotta, SOA Software Inc.
3340 Jiandong Guo, Sun Microsystems
3341 Phillip Hallam-Baker, VeriSign
3342 Patrick Harding, Ping Identity Corporation
3343 Heather Hinton, IBM
3344 Frederick Hirsch, Nokia Corporation
3345 Jeff Hodges, Neustar, Inc.
3346 Will Hopkins, Oracle Corporation
3347 Alex Hristov, Otecia Incorporated
3348 John Hughes, PA Consulting
3349 Diane Jordan, IBM
3350 Venugopal K, Sun Microsystems
3351 Chris Kaler, Microsoft Corporation
3352 Dana Kaufman, Forum Systems, Inc.
3353 Paul Knight, Nortel Networks Limited
3354 Ramanathan Krishnamurthy, IONA Technologies
3355 Christopher Kurt, Microsoft Corporation
3356 Kelvin Lawrence, IBM
3357 Hubert Le Van Gong, Sun Microsystems
3358 Jong Lee, Oracle Corporation
3359 Rich Levinson, Oracle Corporation
3360 Tommy Lindberg, Dajeil Ltd.
3361 Mark Little, JBoss Inc.
3362 Hal Lockhart, Oracle Corporation
3363 Mike Lyons, Layer 7 Technologies Inc.
3364 Eve Maler, Sun Microsystems
3365 Ashok Malhotra, Oracle Corporation
3366 Anand Mani, CrimsonLogic Pte Ltd
3367 Jonathan Marsh, Microsoft Corporation
3368 Robin Martherus, Oracle Corporation
3369 Miko Matsumura, Infravio, Inc.
3370 Gary McAfee, IBM
3371 Michael McIntosh, IBM
3372 John Merrells, Sxip Networks SRL
3373 Jeff Mischkinsky, Oracle Corporation
3374 Prateek Mishra, Oracle Corporation

3375 Bob Morgan, Internet2
3376 Vamsi Motukuru, Oracle Corporation
3377 Raajmohan Na, EDS
3378 Anthony Nadalin, IBM
3379 Andrew Nash, Reactivity, Inc.
3380 Eric Newcomer, IONA Technologies
3381 Duane Nickull, Adobe Systems
3382 Toshihiro Nishimura, Fujitsu Limited
3383 Rob Philpott, RSA Security
3384 Denis Pilipchuk, Oracle Corporation
3385 Darren Platt, Ping Identity Corporation
3386 Martin Raepple, SAP AG
3387 Nick Ragouzis, Enosis Group LLC
3388 Prakash Reddy, CA
3389 Alain Regnier, Ricoh Company, Ltd.
3390 Irving Reid, Hewlett-Packard
3391 Bruce Rich, IBM
3392 Tom Rutt, Fujitsu Limited
3393 Maneesh Sahu, Actional Corporation
3394 Frank Siebenlist, Argonne National Laboratory
3395 Joe Smith, Apani Networks
3396 Davanum Srinivas, WSO2
3397 David Staggs, Veterans Health Administration
3398 Yakov Sverdlov, CA
3399 Gene Thurston, AmberPoint
3400 Victor Valle, IBM
3401 Asir Vedamuthu, Microsoft Corporation
3402 Greg Whitehead, Hewlett-Packard
3403 Ron Williams, IBM
3404 Corinna Witt, Oracle Corporation
3405 Kyle Young, Microsoft Corporation
3406