

WS-Trust 1.3

Committee Specification 01, 29 November 2006

Artifact Identifier:

ws-trust-1.3-spec-cs-01

Location:

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

This Version: <http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-spec-cs-01.doc>

Previous Version: <http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-spec-cd-01.doc>

Artifact Type:

specification

Technical Committee:

OASIS Web Service Secure Exchange TC

Chair(s):

Kelvin Lawrence, IBM

Chris Kaler, Microsoft

Editor(s):

Anthony Nadalin, IBM

Marc Goodner, Microsoft

Martin Gudgin, Microsoft

Abbie Barbir, Nortel

Hans Granqvist, VeriSign

OASIS Conceptual Model topic area:

[Topic Area]

Related work:

N/A

Abstract:

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

Status:

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

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at <http://www.oasis-open.org/committees/ws-sx>.

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

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

Notices

Copyright © OASIS Open 2006. 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.

Table of Contents

1	Introduction.....	5
1.1	Goals and Non-Goals.....	5
1.2	Requirements.....	6
1.3	Namespace.....	6
1.4	Schema and WSDL Files.....	7
1.5	Terminology.....	7
1.5.1	Notational Conventions.....	8
1.6	Normative References.....	9
1.7	Non-Normative References.....	10
2	Web Services Trust Model.....	11
2.1	Models for Trust Brokering and Assessment.....	12
2.2	Token Acquisition.....	12
2.3	Out-of-Band Token Acquisition.....	13
2.4	Trust Bootstrap.....	13
3	Security Token Service Framework.....	14
3.1	Requesting a Security Token.....	14
3.2	Returning a Security Token.....	15
3.3	Binary Secrets.....	17
3.4	Composition.....	17
4	Issuance Binding.....	18
4.1	Requesting a Security Token.....	18
4.2	Request Security Token Collection.....	20
4.2.1	Processing Rules.....	22
4.3	Returning a Security Token Collection.....	22
4.4	Returning a Security Token.....	23
4.4.1	wsp:AppliesTo in RST and RSTR.....	24
4.4.2	Requested References.....	25
4.4.3	Keys and Entropy.....	25
4.4.4	Returning Computed Keys.....	26
4.4.5	Sample Response with Encrypted Secret.....	27
4.4.6	Sample Response with Unencrypted Secret.....	27
4.4.7	Sample Response with Token Reference.....	28
4.4.8	Sample Response without Proof-of-Possession Token.....	28
4.4.9	Zero or One Proof-of-Possession Token Case.....	28
4.4.10	More Than One Proof-of-Possession Tokens Case.....	29
4.5	Returning Security Tokens in Headers.....	30
5	Renewal Binding.....	32
6	Cancel Binding.....	35
6.1	STS-initiated Cancel Binding.....	36
7	Validation Binding.....	38
8	Negotiation and Challenge Extensions.....	41
8.1	Negotiation and Challenge Framework.....	42
8.2	Signature Challenges.....	42

8.3 Binary Exchanges and Negotiations.....	43
8.4 Key Exchange Tokens.....	44
8.5 Custom Exchanges.....	45
8.6 Signature Challenge Example	45
8.7 Custom Exchange Example	47
8.8 Protecting Exchanges.....	48
8.9 Authenticating Exchanges	49
9 Key and Token Parameter Extensions.....	50
9.1 On-Behalf-Of Parameters	50
9.2 Key and Encryption Requirements	50
9.3 Delegation and Forwarding Requirements	55
9.4 Policies.....	56
9.5 Authorized Token Participants.....	56
10 Key Exchange Token Binding	58
11 Error Handling	60
12 Security Considerations	61
A. Key Exchange	63
A.1 Ephemeral Encryption Keys.....	63
A.2 Requestor-Provided Keys	63
A.3 Issuer-Provided Keys	64
A.4 Composite Keys	64
A.5 Key Transfer and Distribution.....	65
A.5.1 Direct Key Transfer	65
A.5.2 Brokered Key Distribution	65
A.5.3 Delegated Key Transfer.....	66
A.5.4 Authenticated Request/Reply Key Transfer.....	67
A.6 Perfect Forward Secrecy.....	68
B. WSDL	69
C. Acknowledgements	71
D. Revision History.....	76

1 Introduction

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

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

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

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

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

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

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

Section 12 is non-normative.

1.1 Goals and Non-Goals

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

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

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

The following are explicit non-goals for this document:

- Password authentication
- Token revocation

- 41 • Management of trust policies

42

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

- 44 • Establishing a security context token
 45 • Key derivation

46 1.2 Requirements

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

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

51 1.3 Namespace

52 The [URI](#) that MUST be used by implementations of this specification is:

53

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

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

56 *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
ds	http://www.w3.org/2000/09/xmldsig#	[XML-Signature]
xenc	http://www.w3.org/2001/04/xmlenc#	[XML-Encrypt]
wsp	http://schemas.xmlsoap.org/ws/2004/09/policy	[WS-Policy]
wsa	http://www.w3.org/2005/08/addressing	[WS-Addressing]

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

57 1.4 Schema and WSDL Files

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

59 <http://docs.oasis-open.org/ws-sx/ws-trust/200512/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/200512/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
136 indicates that any attribute name can be used, from any namespace other than the namespace of
137 this specification.

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->

141 1.0.xsd). The `wsu:Id` attribute and the `wsu:Created` and `wsu:Expires` elements were added to the
142 utility schema with the intent that other specifications requiring such an ID type attribute or timestamp
143 element could reference it (as is done here).

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 Member Submission, "Web Services Policy 1.2 - Framework", 25
164 April 2006.
165 <http://www.w3.org/Submission/2006/SUBM-WS-Policy-20060425/>
- 166 [WS-PolicyAttachment] W3C Member Submission, "Web Services Policy 1.2 - Attachment", 25
167 April 2006.
168 [http://www.w3.org/Submission/2006/SUBM-WS-PolicyAttachment-
169 20060425/](http://www.w3.org/Submission/2006/SUBM-WS-PolicyAttachment-20060425/)
- 170 [WS-Security] OASIS Standard, "OASIS Web Services Security: SOAP Message Security
171 1.0 (WS-Security 2004)", March 2004.
172 [http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-
174 security-1.0.pdf](http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-
173 security-1.0.pdf)
- 174 OASIS Standard, "OASIS Web Services Security: SOAP Message Security
175 1.1 (WS-Security 2004)", February 2006.
176 [http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-
178 spec-os-SOAPMessageSecurity.pdf](http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-
177 spec-os-SOAPMessageSecurity.pdf)
- 178 [XML-C14N] W3C Recommendation, "Canonical XML Version 1.0", 15 March 2001.
179 <http://www.w3.org/TR/2001/REC-xml-c14n-20010315>
- 180 [XML-Encrypt] W3C Recommendation, "XML Encryption Syntax and Processing", 10
181 December 2002.
182 <http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/>
- 183 [XML-Schema1] W3C Recommendation, "XML Schema Part 1: Structures Second Edition",
184 28 October 2004.
185 <http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/>
- 186 [XML-Schema2] W3C Recommendation, "XML Schema Part 2: Datatypes Second Edition",
187 28 October 2004.
188 <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/>

189 [XML-Signature] W3C Recommendation, "XML-Signature Syntax and Processing", 12
190 February 2002.
191 <http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/>
192

193 **1.7 Non-Normative References**

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

2 Web Services Trust Model

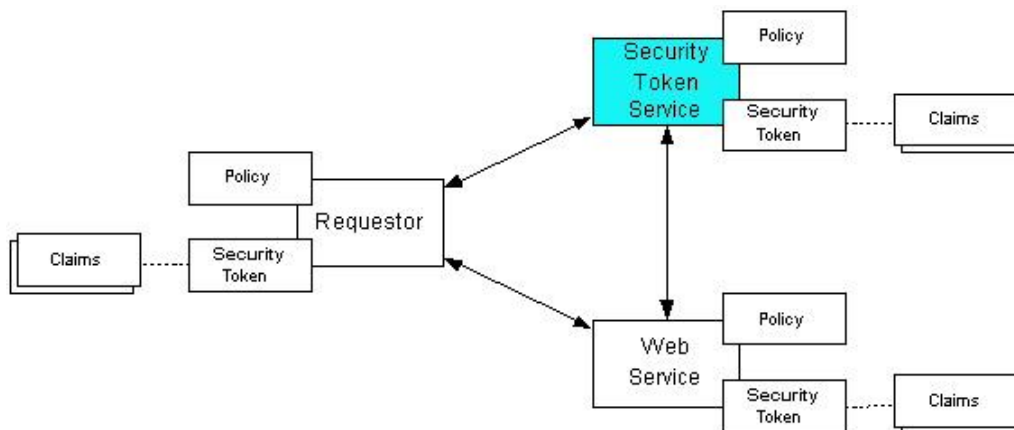
The Web service security model defined in WS-Trust is based on a process in which a Web service can require that an incoming message prove a set of claims (e.g., name, key, permission, capability, etc.). If a message arrives without having the required proof of claims, the service SHOULD ignore or reject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

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

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

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

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



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

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

241
242 In the figure above the arrows represent possible communication paths; the requestor may obtain a token
243 from the security token service, or it may have been obtained indirectly. The requestor then
244 demonstrates authorized use of the token to the Web service. The Web service either trusts the issuing
245 security token service or may request a token service to validate the token (or the Web service may
246 validate the token itself).

247
248 In summary, the Web service has a policy applied to it, receives a message from a requestor that possibly
249 includes security tokens, and may have some protection applied to it using [\[WS-Security\]](#) mechanisms.
250 The following key steps are performed by the trust engine of a Web service (note that the order of
251 processing is non-normative):

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

262
263 If these conditions are met, and the requestor is authorized to perform the operation, then the service can
264 process the service request.

265 In this specification we define how security tokens are requested and obtained from security token
266 services and how these services may broker trust and trust policies so that services can perform step 3.
267 Network and transport protection mechanisms such as IPsec or TLS/SSL [\[RFC2246\]](#) can be used in
268 conjunction with this specification to support different security requirements and scenarios. If available,
269 requestors should consider using a network or transport security mechanism to authenticate the service
270 when requesting, validating, or renewing security tokens, as an added level of security.

271
272 The [\[WS-Federation\]](#) specification builds on this specification to define mechanisms for brokering and
273 federating trust, identity, and claims. Examples are provided in [\[WS-Federation\]](#) illustrating different trust
274 scenarios and usage patterns.

275 **2.1 Models for Trust Brokering and Assessment**

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

279 **2.2 Token Acquisition**

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

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

286

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

293 **2.3 Out-of-Band Token Acquisition**

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

299 **2.4 Trust Bootstrap**

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

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

309

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

312

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

317

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

3 Security Token Service Framework

322

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

324

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

330

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

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

337

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

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

345

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

352

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

3.1 Requesting a Security Token

357

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

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

The syntax for this element is as follows:

364
365
366
367
368
369

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

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

371 */wst:RequestSecurityToken*

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

373 */wst:RequestSecurityToken/@Context*

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

378 */wst:RequestSecurityToken/wst:TokenType*

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

383 */wst:RequestSecurityToken/wst:RequestType*

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

390 */wst:RequestSecurityToken/wst:SecondaryParameters*

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

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

398 */wst:RequestSecurityToken/{any}*

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

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

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

406 3.2 Returning a Security Token

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

411

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

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

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

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

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

434 The syntax for this element is as follows:

```
435 <wst:RequestSecurityTokenResponse Context="..." xmlns:wst="...">  
436   <wst:TokenType>...</wst:TokenType>  
437   <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>  
438   ...  
439 </wst:RequestSecurityTokenResponse>
```

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

441 */wst:RequestSecurityTokenResponse*

442 This is the response to a security token request.

443 */wst:RequestSecurityTokenResponse/@Context*

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

449 */wst:RequestSecurityTokenResponse/wst:TokenType*

450 This optional element specifies the type of security token returned.

451 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

452 This optional element is used to return the requested security token. Normally the requested
453 security token is the contents of this element but a security token reference MAY be used instead.
454 For example, if the requested security token is used in securing the message, then the security
455 token is placed into the `<wsse:Security>` header (as described in [\[WS-Security\]](#)) and a
456 `<wsse:SecurityTokenReference>` element is placed inside of the
457 `<wst:RequestedSecurityToken>` element to reference the token in the `<wsse:Security>`
458 header. The response MAY contain a token reference where the token is located at a URI

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

463 */wst:RequestSecurityTokenResponse/{any}*

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

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

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

469 3.3 Binary Secrets

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

477 *.../wst:BinarySecret*

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

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

481 This optional attribute indicates the type of secret being encoded. The pre-defined values are
 482 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)

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

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

486 3.4 Composition

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

491 4 Issuance Binding

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

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

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

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

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

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

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

510

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

514

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

517

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

521 4.1 Requesting a Security Token

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

```
525 <wst:RequestSecurityToken xmlns:wst="...">  
526   <wst:TokenType>...</wst:TokenType>  
527   <wst:RequestType>...</wst:RequestType>  
528   ...  
529   <wsp:AppliesTo>...</wsp:AppliesTo>  
530   <wst:Claims Dialect="...">...</wst:Claims>  
531   <wst:Entropy>  
532     <wst:BinarySecret>...</wst:BinarySecret>  
533   </wst:Entropy>  
534   <wst:Lifetime>
```

```
535         <wsu:Created>...</wsu:Created>
536         <wsu:Expires>...</wsu:Expires>
537     </wst:Lifetime>
538 </wst:RequestSecurityToken>
```

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

540 */wst:RequestSecurityToken/wst:TokenType*

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

549 */wst:RequestSecurityToken/wsp:AppliesTo*

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

557 */wst:RequestSecurityToken/wst:Claims*

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

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

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

565 */wst:RequestSecurityToken/wst:Entropy*

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

570 */wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret*

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

577 */wst:RequestSecurityToken/wst:Lifetime*

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

584 */wst:RequestSecurityToken/wst:Lifetime/ws:Created*

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

590 */wst:RequestSecurityToken/wst:Lifetime/ws:Expires*

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

596

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

```
603 <S11:Envelope xmlns:S11="..." xmlns:wsu="..." xmlns:wsse="..."
604     xmlns:xenc="..." xmlns:wst="...">
605   <S11:Header>
606     ...
607     <wsse:Security>
608       <xenc:ReferenceList>...</xenc:ReferenceList>
609       <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>
610       <ds:Signature xmlns:ds="...">
611         ...
612         <ds:KeyInfo>
613           <wsse:SecurityTokenReference>
614             <wsse:Reference URI="#myToken"/>
615           </wsse:SecurityTokenReference>
616         </ds:KeyInfo>
617       </ds:Signature>
618     </wsse:Security>
619     ...
620   </S11:Header>
621   <S11:Body wsu:Id="req">
622     <wst:RequestSecurityToken>
623       <wst:TokenType>
624         http://example.org/mySpecialToken
625       </wst:TokenType>
626       <wst:RequestType>
627         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
628       </wst:RequestType>
629     </wst:RequestSecurityToken>
630   </S11:Body>
631 </S11:Envelope>
```

632 4.2 Request Security Token Collection

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

637

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

642

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

646

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

648

```
649 <wst:RequestSecurityTokenCollection xmlns:wst="...">
650
651   <!-- identity token request -->
652   <wst:RequestSecurityToken Context="http://www.example.com/1">
653     <wst:TokenType>
654       http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
655 1.1#SAMLV2.0
656     </wst:TokenType>
657     <wst:RequestType>http://docs.oasis-open.org/ws-sx/ws-
658 trust/200512/BatchIssue</wst:RequestType>
659     <wsp:AppliesTo xmlns:wsp="..." xmlns:wsa="...">
660       <wsa:EndpointReference>
661         <wsa:Address>http://manufacturer.example.com/</wsa:Address>
662       </wsa:EndpointReference>
663     </wsp:AppliesTo>
664     <wsp:PolicyReference xmlns:wsp="..."
665 URI='http://manufacturer.example.com/IdentityPolicy' />
666   </wst:RequestSecurityToken>
667
668   <!-- certification claim token request -->
669   <wst:RequestSecurityToken Context="http://www.example.com/2">
670     <wst:TokenType>
671       http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
672 1.1#SAMLV2.0
673     </wst:TokenType>
674     <wst:RequestType>http://docs.oasis-open.org/ws-sx/ws-trust/200512
675 /BatchIssue</wst:RequestType>
676     <wst:Claims xmlns:wsp="...">
677       http://manufacturer.example.com/certification
678     </wst:Claims>
679     <wsp:PolicyReference
680 URI='http://certificationbody.example.org/certificationPolicy' />
681   </wst:RequestSecurityToken>
682 </wst:RequestSecurityTokenCollection>
```

683

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

685

686 */wst:RequestSecurityTokenCollection*

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

690 4.2.1 Processing Rules

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

693

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

697 **Note:** Each request may generate more than one RSTR sharing the same Context attribute value

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

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

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

- 703 • `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchIssue`
- 704 • `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchValidate`
- 705 • `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchRenew`
- 706 • `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchCancel`

707

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

710

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

713

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

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

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

721

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

724

725 4.3 Returning a Security Token Collection

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

730 The syntax for this element is as follows:

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

733 `</wst:RequestSecurityTokenResponseCollection>`

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

735 */wst:RequestSecurityTokenResponseCollection*

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

738 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

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

740 4.4 Returning a Security Token

741 When returning a security token, the following optional elements MAY be included in the response.

742 Security tokens can only be returned in the RSTRC on the final leg. The syntax for these elements is as

743 follows (note that the base elements described above are included here italicized for completeness):

```
744 <wst:RequestSecurityTokenResponse xmlns:wst="...">  
745   <wst:TokenType>...</wst:TokenType>  
746   <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>  
747   ...  
748   <wsp:AppliesTo xmlns:wsp="...">...</wsp:AppliesTo>  
749   <wst:RequestedAttachedReference>  
750     ...  
751   </wst:RequestedAttachedReference>  
752   <wst:RequestedUnattachedReference>  
753     ...  
754 </wst:RequestedUnattachedReference>  
755   <wst:RequestedProofToken>...</wst:RequestedProofToken>  
756   <wst:Entropy>  
757     <wst:BinarySecret>...</wst:BinarySecret>  
758   </wst:Entropy>  
759   <wst:Lifetime>...</wst:Lifetime>  
760 </wst:RequestSecurityTokenResponse>
```

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

762 */wst:RequestSecurityTokenResponse/wsp:AppliesTo*

763 This optional element specifies the scope to which this security token applies. Refer to [\[WS-PolicyAttachment\]](#) for more information. Note that if an `<wsp:AppliesTo>` was specified in the
764 request, the same scope SHOULD be returned in the response (if a `<wsp:AppliesTo>` is
765 returned).
766

767 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

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

773 */wst:RequestSecurityTokenResponse/wst:RequestedAttachedReference*

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

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

784 */wst:RequestSecurityTokenResponse/wst:RequestedUnattachedReference*

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

791 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken*

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

802 */wst:RequestSecurityTokenResponse/wst:Entropy*

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

807 */wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret*

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

810 */wst:RequestSecurityTokenResponse/wst:Lifetime*

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

814 **4.4.1 wsp:AppliesTo in RST and RSTR**

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

- 819 • If neither the requestor nor the issuer specifies a scope then the scope of the issued token is
820 implied.
- 821 • If the requestor specifies a scope and the issuer does not then the scope of the token is assumed
822 to be that specified by the requestor.
- 823 • If the requestor does not specify a scope and the issuer does specify a scope then the scope of
824 the token is as defined by the issuers scope
- 825 • If both requestor and issuer specify a scope then there are two possible outcomes:
 - 826 ○ If both the issuer and requestor specify the same scope then the issued token has that
827 scope.

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

830

831 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.

832 4.4.2 Requested References

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

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

848 4.4.3 Keys and Entropy

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

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

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

860

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

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

871

872 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

873 4.4.4 Returning Computed Keys

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

```
879 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
880   <wst:RequestSecurityTokenResponse>
881     <wst:RequestedProofToken>
882       <wst:ComputedKey>...</wst:ComputedKey>
883     </wst:RequestedProofToken>
884   </wst:RequestSecurityTokenResponse>
885 </wst:RequestSecurityTokenResponseCollection>
```

886

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

888 `/wst:RequestSecurityTokenResponse/wst:RequestedProofToken/wst:ComputedKey`

889 The value of this element is a URI describing how to compute the key. While this can be
 890 extended by defining new URIs in other bindings and profiles, the following URI pre-defines one
 891 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.

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

893 4.4.5 Sample Response with Encrypted Secret

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

```

898 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
899   <wst:RequestSecurityTokenResponse>
900     <wst:RequestedSecurityToken>
901       <xyz:CustomToken xmlns:xyz="...">
902         ...
903       </xyz:CustomToken>
904     </wst:RequestedSecurityToken>
905     <wst:RequestedProofToken>
906       <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
907         ...
908       </xenc:EncryptedKey>
909     </wst:RequestedProofToken>
910   </wst:RequestSecurityTokenResponse>
911 </wst:RequestSecurityTokenResponseCollection>
  
```

912 4.4.6 Sample Response with Unencrypted Secret

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

```

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

927 4.4.7 Sample Response with Token Reference

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

```
931 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
932   <wst:RequestSecurityTokenResponse>
933     <wst:RequestedSecurityToken>
934       <xyz:CustomToken ID="urn:fabrikam123:5445" xmlns:xyz="...">
935         ...
936       </xyz:CustomToken>
937     </wst:RequestedSecurityToken>
938     <wst:RequestedAttachedReference>
939       <wsse:SecurityTokenReference xmlns:wsse="...">
940         <wsse:Reference URI="urn:fabrikam123:5445"/>
941       </wsse:SecurityTokenReference>
942     </wst:RequestedAttachedReference>
943     ...
944   </wst:RequestSecurityTokenResponse>
945 </wst:RequestSecurityTokenResponseCollection>
```

946
947 In the example above, the recipient may place the returned custom token directly into a message and
948 include a signature using the provided proof-of-possession token. The specified reference is then placed
949 into the `<ds:KeyInfo>` of the signature and directly references the included token without requiring the
950 requestor to understand the details of the custom token format.

951 4.4.8 Sample Response without Proof-of-Possession Token

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

```
956 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
957   <wst:RequestSecurityTokenResponse>
958     <wst:RequestedSecurityToken>
959       <xyz:CustomToken xmlns:xyz="...">
960         ...
961       </xyz:CustomToken>
962     </wst:RequestedSecurityToken>
963   </wst:RequestSecurityTokenResponse>
964 </wst:RequestSecurityTokenResponseCollection>
```

965

966 4.4.9 Zero or One Proof-of-Possession Token Case

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

973

```
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:RequestedProofToken>
982         <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
983             ...
984         </xenc:EncryptedKey>
985     </wst:RequestedProofToken>
986 </wst:RequestSecurityTokenResponse>
987 </wst:RequestSecurityTokenResponseCollection>

```

988 4.4.10 More Than One Proof-of-Possession Tokens Case

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

```

993 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
994     <wst:RequestSecurityTokenResponse>
995         ...
996     </wst:RequestSecurityTokenResponse>
997     <wst:RequestSecurityTokenResponse>
998         ...
999     </wst:RequestSecurityTokenResponse>
1000     ...
1001 </wst:RequestSecurityTokenResponseCollection>

```

1002 The following describes the attributes and elements listed in the schema overview above:

1003 */wst:RequestSecurityTokenResponseCollection*

1004 This element is used to provide multiple RSTR responses, each of which has separate key
1005 information. One or more RSTR elements are returned in the collection. This MUST always be
1006 used on the final response to the RST.

1007 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

1008 Each RequestSecurityTokenResponse element is an individual RSTR.

1009 */wst:RequestSecurityTokenResponseCollection/{any}*

1010 This is an extensibility mechanism to allow additional elements, based on schemas, to be added.

1011 */wst:RequestSecurityTokenResponseCollection/@{any}*

1012 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added.

1013 The following illustrates the syntax of a response that includes multiple tokens each, in a separate RSTR,
1014 each with their own proof-of-possession token.

```

1015 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
1016     <wst:RequestSecurityTokenResponse>
1017         <wst:RequestedSecurityToken>
1018             <xyz:CustomToken xmlns:xyz="...">
1019                 ...
1020             </xyz:CustomToken>
1021         </wst:RequestedSecurityToken>
1022         <wst:RequestedProofToken>
1023             <xenc:EncryptedKey Id="newProofA">
1024                 ...
1025             </xenc:EncryptedKey>
1026         </wst:RequestedProofToken>
1027     </wst:RequestSecurityTokenResponse>
1028     <wst:RequestSecurityTokenResponse>

```

```

1029     <wst:RequestedSecurityToken>
1030         <abc:CustomToken xmlns:abc="...">
1031             ...
1032         </abc:CustomToken>
1033     </wst:RequestedSecurityToken>
1034     <wst:RequestedProofToken>
1035         <xenc:EncryptedKey Id="newProofB xmlns:xenc="...">
1036             ...
1037         </xenc:EncryptedKey>
1038     </wst:RequestedProofToken>
1039 </wst:RequestSecurityTokenResponse>
1040 </wst:RequestSecurityTokenResponseCollection>

```

1041 4.5 Returning Security Tokens in Headers

1042 In certain situations it is useful to issue one or more security tokens as part of a protocol other than
1043 RST/RSTR. This typically requires that the tokens be passed in a SOAP header. The tokens present in
1044 that element can then be referenced from elsewhere in the message. This section defines a specific
1045 header element, whose type is the same as that of the `<wst:RequestSecurityTokenCollection>`
1046 element (see Section 4.3), that can be used to carry issued tokens (and associated proof tokens,
1047 references etc.) in a message.

```

1048 <wst:IssuedTokens xmlns:wst="...">
1049   <wst:RequestSecurityTokenResponse>
1050     ...
1051   </wst:RequestSecurityTokenResponse>+
1052 </wst:IssuedTokens>

```

1053
1054 The following describes the attributes and elements listed in the schema overview above:

1055 */wst:IssuedTokens*

1056 This header element carries one or more issued security tokens. This element schema is defined
1057 using the RequestSecurityTokenResponse schema type.

1058 */wst:IssuedTokens/wst:RequestSecurityTokenResponse*

1059 This element **MUST** appear at least once. Its meaning and semantics are as defined in Section 4.2.

1060 */wst:IssuedTokens/{any}*

1061 This is an extensibility mechanism to allow additional elements, based on schemas, to be added.

1062 */wst:IssuedTokens/@{any}*

1063 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added.

1064
1065 There **MAY** be multiple instances of the `<wst:IssuedTokens>` header in a given message. Such
1066 instances **MAY** be targeted at the same actor/role. Intermediaries **MAY** add additional
1067 `<wst:IssuedTokens>` header elements to a message. Intermediaries **SHOULD NOT** modify any
1068 `<wst:IssuedTokens>` header already present in a message.

1069
1070 It is **RECOMMENDED** that the `<wst:IssuedTokens>` header be signed to protect the integrity of the
1071 issued tokens and of the issuance itself. If confidentiality protection of the `<wst:IssuedTokens>` header is
1072 required then the entire header **MUST** be encrypted using the `<wss11:EncryptedHeader>` construct.
1073 This helps facilitate re-issuance by the receiving party as that party can re-encrypt the entire header for
1074 another party rather than having to extract and re-encrypt portions of the header.

1075

1076 The following example illustrates a response that includes multiple `<wst:IssuedTokens>` headers.

```
1077 <?xml version="1.0" encoding="utf-8"?>
1078 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsp="..." xmlns:ds="..."
1079 xmlns:x="...">
1080   <S11:Header>
1081     <wst:IssuedTokens>
1082       <wst:RequestSecurityTokenResponse>
1083         <wsp:AppliesTo>
1084           <x:SomeContext1 />
1085         </wsp:AppliesTo>
1086         <wst:RequestedSecurityToken>
1087           ...
1088         </wst:RequestedSecurityToken>
1089         ...
1090       </wst:RequestSecurityTokenResponse>
1091       <wst:RequestSecurityTokenResponse>
1092         <wsp:AppliesTo>
1093           <x:SomeContext1 />
1094         </wsp:AppliesTo>
1095         <wst:RequestedSecurityToken>
1096           ...
1097         </wst:RequestedSecurityToken>
1098         ...
1099       </wst:RequestSecurityTokenResponse>
1100     </wst:IssuedTokens>
1101     <wst:IssuedTokens S11:role="http://example.org/someroles" >
1102       <wst:RequestSecurityTokenResponse>
1103         <wsp:AppliesTo>
1104           <x:SomeContext2 />
1105         </wsp:AppliesTo>
1106         <wst:RequestedSecurityToken>
1107           ...
1108         </wst:RequestedSecurityToken>
1109         ...
1110       </wst:RequestSecurityTokenResponse>
1111     </wst:IssuedTokens>
1112   </S11:Header>
1113   <S11:Body>
1114     ...
1115   </S11:Body>
1116 </S11:Envelope>
```

5 Renewal Binding

1117

1118 Using the token request framework, this section defines bindings for requesting security tokens to be
1119 renewed:

1120 **Renew** – A previously issued token with expiration is presented (and possibly proven) and the
1121 same token is returned with new expiration semantics.

1122

1123 For this binding, the following actions are defined to enable specific processing context to be conveyed to
1124 the recipient:

1125
1126
1127

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

1128 For this binding, the `<wst:RequestType>` element uses the following URI:

1129

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

1130 For this binding the token to be renewed is identified in the `<wst:RenewTarget>` element and the
1131 optional `<wst:Lifetime>` element MAY be specified to request a specified renewal duration.

1132

1133 Other extensions MAY be specified in the request (and the response), but the key semantics (size, type,
1134 algorithms, scope, etc.) MUST NOT be altered during renewal. Token services MAY use renewal as an
1135 opportunity to rekey, so the renewal responses MAY include a new proof-of-possession token as well as
1136 entropy and key exchange elements.

1137

1138 The request MUST prove authorized use of the token being renewed unless the recipient trusts the
1139 requestor to make third-party renewal requests. In such cases, the third-party requestor MUST prove its
1140 identity to the issuer so that appropriate authorization occurs.

1141

1142 The original proof information SHOULD be proven during renewal.

1143

1144 The renewal binding allows the use of exchanges during the renewal process. Subsequent profiles MAY
1145 define restriction around the usage of exchanges.

1146

1147 During renewal, all key bearing tokens used in the renewal request MUST have an associated signature.
1148 All non-key bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the renewal
1149 response.

1150

1151 The renewal binding also defines several extensions to the request and response elements. The syntax
1152 for these extension elements is as follows (note that the base elements described above are included
1153 here italicized for completeness):

1154
1155
1156
1157
1158
1159

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


1160
1161

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

1162 */wst:RequestSecurityToken/wst:RenewTarget*

1163 This required element identifies the token being renewed. This MAY contain a
1164 `<wsse:SecurityTokenReference>` pointing at the token to be renewed or it MAY directly contain
1165 the token to be renewed.

1166 */wst:RequestSecurityToken/wst:AllowPostdating*

1167 This optional element indicates that returned tokens should allow requests for postdated tokens.
1168 That is, this allows for tokens to be issued that are not immediately valid (e.g., a token that can be
1169 used the next day).

1170 */wst:RequestSecurityToken/wst:Renewing*

1171 This optional element is used to specify renew semantics for types that support this operation.

1172 */wst:RequestSecurityToken/wst:Renewing/@Allow*

1173 This optional Boolean attribute is used to request a renewable token. If not specified, the default
1174 value is *true*. A renewable token is one whose lifetime can be extended. This is done using a
1175 renewal request. The recipient MAY allow renewals without demonstration of authorized use of
1176 the token or they MAY fault.

1177 */wst:RequestSecurityToken/wst:Renewing/@OK*

1178 This optional Boolean attribute is used to indicate that a renewable token is acceptable if the
1179 requested duration exceeds the limit of the issuance service. That is, if *true* then tokens can be
1180 renewed after their expiration. It should be noted that the token is NOT valid after expiration for
1181 any operation except renewal. The default for this attribute is *false*. It NOT RECOMMENDED to
1182 use this as it can leave you open to certain types of security attacks. Issuers MAY restrict the
1183 period after expiration during which time the token can be renewed. This window is governed by
1184 the issuer's policy.

1185 The following example illustrates a request for a custom token that can be renewed.

1186
1187
1188
1189
1190
1191
1192
1193
1194

```
<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>
```

1195

1196 The following example illustrates a subsequent renewal request and response (note that for brevity only
1197 the request and response are illustrated). Note that the response includes an indication of the lifetime of
1198 the renewed token.

1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210

```
<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>
```

```
1211 <wst:RequestSecurityTokenResponse xmlns:wst="...">
1212   <wst:TokenType>
1213     http://example.org/mySpecialToken
1214   </wst:TokenType>
1215   <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1216   <wst:Lifetime>...</wst:Lifetime>
1217   ...
1218 </wst:RequestSecurityTokenResponse>
```

6 Cancel Binding

1219

1220 Using the token request framework, this section defines bindings for requesting security tokens to be
1221 cancelled:

1222 **Cancel** – When a previously issued token is no longer needed, the Cancel binding can be used
1223 to cancel the token, terminating its use. After canceling a token at the issuer, a STS MUST not
1224 validate or renew the token. A STS MAY initiate the revocation of a token, however, revocation is
1225 out of scope of this specification and a client MUST NOT rely on it. If a client needs to ensure the
1226 validity of a token, it must validate the token at the issuer.

1227

1228 For this binding, the following actions are defined to enable specific processing context to be conveyed to
1229 the recipient:

```
1230 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel  
1231 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Cancel  
1232 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal
```

1233 For this binding, the `<wst:RequestType>` element uses the following URI:

```
1234 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
```

1235 Extensions MAY be specified in the request (and the response), but the semantics are not defined by this
1236 binding.

1237

1238 The request MUST prove authorized use of the token being cancelled unless the recipient trusts the
1239 requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its
1240 identity to the issuer so that appropriate authorization occurs.

1241 In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key
1242 bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the closure response.

1243

1244 A cancelled token is no longer valid for authentication and authorization usages.

1245 On success a cancel response is returned. This is an RSTR message with the
1246 `<wst:RequestedTokenCancelled>` element in the body. On failure, a Fault is raised. It should be
1247 noted that the cancel RSTR is informational. That is, the security token is cancelled once the cancel
1248 request is processed.

1249

1250 The syntax of the request is as follows:

```
1251 <wst:RequestSecurityToken xmlns:wst="...">  
1252   <wst:RequestType>...</wst:RequestType>  
1253   ...  
1254   <wst:CancelTarget>...</wst:CancelTarget>  
1255 </wst:RequestSecurityToken>
```

1256 */wst:RequestSecurityToken/wst:CancelTarget*

1257 This required element identifies the token being cancelled. Typically this contains a
1258 `<wsse:SecurityTokenReference>` pointing at the token, but it could also carry the token
1259 directly.

1260 The following example illustrates a request to cancel a custom token.

```
1261 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
```

1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277

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

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

1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290

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

1291 6.1 STS-initiated Cancel Binding

1292 Using the token request framework, this section defines an optional binding for requesting security tokens
1293 to be cancelled by the STS:

1294 **STS-initiated Cancel** – When a previously issued token becomes invalid on the STS, the STS-
1295 initiated Cancel binding can be used to cancel the token, terminating its use. After canceling a
1296 token, a STS MUST not validate or renew the token. This binding can be only used when STS
1297 can send one-way messages to the original token requestor.

1298

1299 For this binding, the following actions are defined to enable specific processing context to be conveyed to
1300 the recipient:

1301

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

1302 For this binding, the `<wst:RequestType>` element uses the following URI:

1303

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

1304 Extensions MAY be specified in the request, but the semantics are not defined by this binding.

1305

1306 The request MUST prove authorized use of the token being cancelled unless the recipient trusts the
1307 requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its
1308 identity to the issuer so that appropriate authorization occurs.

1309 In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key
1310 bearing tokens MUST be signed.

1311

1312 A cancelled token is no longer valid for authentication and authorization usages.

1313

1314 The mechanism to determine the availability of STS-initiated Cancel binding on the STS is out of scope of
1315 this specification. Similarly, how the client communicates its endpoint address to the STS so that it can
1316 send the STSCancel messages to the client is out of scope of this specification. This functionality is
1317 implementation specific and can be solved by different mechanisms that are not in scope for this
1318 specification.

1319

1320 This is a one-way operation, no response is returned from the recipient of the message.

1321

1322 The syntax of the request is as follows:

```
1323 <wst:RequestSecurityToken xmlns:wst="...">  
1324   <wst:RequestType>...</wst:RequestType>  
1325   ...  
1326   <wst:CancelTarget>...</wst:CancelTarget>  
1327 </wst:RequestSecurityToken>
```

1328 */wst:RequestSecurityToken/wst:CancelTarget*

1329 This required element identifies the token being cancelled. Typically this contains a
1330 `<wsse:SecurityTokenReference>` pointing at the token, but it could also carry the token
1331 directly.

1332 The following example illustrates a request to cancel a custom token.

```
1333 <?xml version="1.0" encoding="utf-8"?>  
1334 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">  
1335   <S11:Header>  
1336     <wsse:Security>  
1337       ...  
1338     </wsse:Security>  
1339   </S11:Header>  
1340   <S11:Body>  
1341     <wst:RequestSecurityToken>  
1342       <wst:RequestType>  
1343         http://docs.oasis-open.org/ws-sx/ws-trust/200512/STSCancel  
1344       </wst:RequestType>  
1345       <wst:CancelTarget>  
1346         ...  
1347       </wst:CancelTarget>  
1348     </wst:RequestSecurityToken>  
1349   </S11:Body>  
1350 </S11:Envelope>
```

7 Validation Binding

1351

1352 Using the token request framework, this section defines bindings for requesting security tokens to be
1353 validated:

1354 **Validate** – The validity of the specified security token is evaluated and a result is returned. The
1355 result may be a status, a new token, or both.

1356

1357 It should be noted that for this binding, a SOAP Envelope MAY be specified as a "security token" if the
1358 requestor desires the envelope to be validated. In such cases the recipient SHOULD understand how to
1359 process a SOAP envelope and adhere to SOAP processing semantics (e.g., mustUnderstand) of the
1360 version of SOAP used in the envelope. Otherwise, the recipient SHOULD fault.

1361 For this binding, the following actions are defined to enable specific processing context to be conveyed to
1362 the recipient:

1363

```
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
```

1364

1365

1366

1367 For this binding, the `<wst:RequestType>` element contains the following URI:

1368

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

1369

1370 The request provides a token upon which the request is based and optional tokens. As well, the optional
1371 `<wst:TokenType>` element in the request can indicate desired type response token. This may be any
1372 supported token type or it may be the following URI indicating that only status is desired:

1373

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

1374

1375 For some use cases a status token is returned indicating the success or failure of the validation. In other
1376 cases a security token MAY be returned and used for authorization. This binding assumes that the
1377 validation requestor and provider are known to each other and that the general issuance parameters
1378 beyond requesting a token type, which is optional, are not needed (note that other bindings and profiles
1379 could define different semantics).

1380

1381 For this binding an applicability scope (e.g., `<wsp:AppliesTo>`) need not be specified. It is assumed
1382 that the applicability of the validation response relates to the provided information (e.g. security token) as
1383 understood by the issuing service.

1384

1385 The validation binding does not allow the use of exchanges.

1386

1387 The RSTR for this binding carries the following element even if a token is returned (note that the base
1388 elements described above are included here italicized for completeness):

1389

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

1390

1391

1392

1393

1394

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

1395

1396

```

<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>

```

1397

1398

1399

1400

1401

1402

1403

1404

1405

1406 */wst:RequestSecurityToken/wst:ValidateTarget*

1407 This required element identifies the token being validated. Typically this contains a
1408 <wsse:SecurityTokenReference> pointing at the token, but could also carry the token
1409 directly.

1410 */wst:RequestSecurityTokenResponse/wst:Status*

1411 When a validation request is made, this element MUST be in the response. The code value
1412 indicates the results of the validation in a machine-readable form. The accompanying text
1413 element allows for human textual display.

1414 */wst:RequestSecurityTokenResponse/wst:Status/wst:Code*

1415 This required URI value provides a machine-readable status code. The following URIs are
1416 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

1417 */wst:RequestSecurityTokenResponse/wst:Status/wst:Reason*

1418 This optional string provides human-readable text relating to the status code.

1419

1420 The following illustrates the syntax of a validation request and response. In this example no token is
1421 requested, just a status.

1422

```

<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>

```

1423

1424

1425

1426

1427

1428

1429

1430

1431

```

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

```

1432

1433

1434

```
1435     <wst:Status>
1436         <wst:Code>
1437             http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
1438         </wst:Code>
1439     </wst:Status>
1440     ...
1441 </wst:RequestSecurityTokenResponse>
```

1442 The following illustrates the syntax of a validation request and response. In this example a custom token
1443 is requested indicating authorized rights in addition to the status.

```
1444 <wst:RequestSecurityToken xmlns:wst="...">
1445     <wst:TokenType>
1446         http://example.org/mySpecialToken
1447     </wst:TokenType>
1448     <wst:RequestType>
1449         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate
1450     </wst:RequestType>
1451 </wst:RequestSecurityToken>
```

```
1452
1453 <wst:RequestSecurityTokenResponse xmlns:wst="...">
1454     <wst:TokenType>
1455         http://example.org/mySpecialToken
1456     </wst:TokenType>
1457     <wst:Status>
1458         <wst:Code>
1459             http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
1460         </wst:Code>
1461     </wst:Status>
1462     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1463     ...
1464 </wst:RequestSecurityTokenResponse>
```

8 Negotiation and Challenge Extensions

1465

1466 The general security token service framework defined above allows for a simple request and response for
1467 security tokens (possibly asynchronous). However, there are many scenarios where a set of exchanges
1468 between the parties is required prior to returning (e.g., issuing) a security token. This section describes
1469 the extensions to the base WS-Trust mechanisms to enable exchanges for negotiation and challenges.

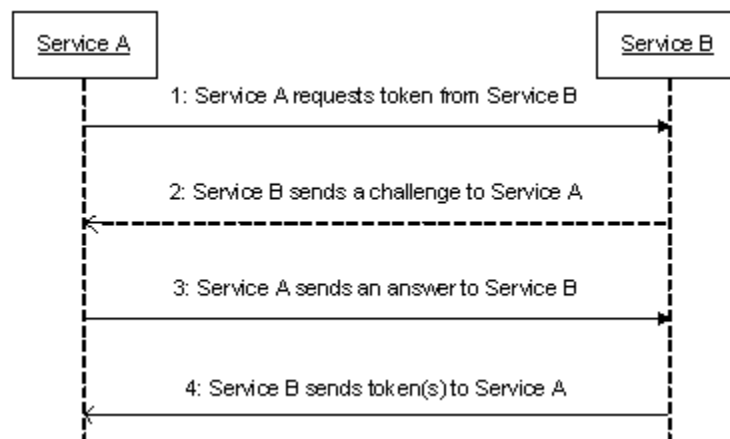
1470

1471 There are potentially different forms of exchanges, but one specific form, called "challenges", provides
1472 mechanisms in addition to those described in [WS-Security] for authentication. This section describes
1473 how general exchanges are issued and responded to within this framework. Other types of exchanges
1474 include, but are not limited to, negotiation, tunneling of hardware-based processing, and tunneling of
1475 legacy protocols.

1476

1477 The process is straightforward (illustrated here using a challenge):

1478



1479

- 1480 1. A requestor sends, for example, a `<wst:RequestSecurityToken>` message with a
1481 timestamp.
- 1482 2. The recipient does not trust the timestamp and issues a
1483 `<wst:RequestSecurityTokenResponse>` message with an embedded challenge.
- 1484 3. The requestor sends a `<wst:RequestSecurityTokenResponse>` message with an answer to
1485 the challenge.
- 1486 4. The recipient issues a `<wst:RequestSecurityTokenResponseCollection>` message with
1487 the issued security token and optional proof-of-possession token.

1488

1489 It should be noted that the requestor might challenge the recipient in either step 1 or step 3. In which
1490 case, step 2 or step 4 contains an answer to the initiator's challenge. Similarly, it is possible that steps 2
1491 and 3 could iterate multiple times before the process completes (step 4).

1492

1493 The two services can use [WS-SecurityPolicy] to state their requirements and preferences for security
1494 tokens and encryption and signing algorithms (general policy intersection). This section defines
1495 mechanisms for legacy and more sophisticated types of negotiations.

1496 8.1 Negotiation and Challenge Framework

1497 The general mechanisms defined for requesting and returning security tokens are extensible. This
1498 section describes the general model for extending these to support negotiations and challenges.

1499
1500 The exchange model is as follows:

- 1501 1. A request is initiated with a `<wst:RequestSecurityToken>` that identifies the details of the
1502 request (and may contain initial negotiation/challenge information)
- 1503 2. A response is returned with a `<wst:RequestSecurityTokenResponse>` that contains
1504 additional negotiation/challenge information. Optionally, this may return token information in the
1505 form of a `<wst:RequestSecurityTokenResponseCollection>` (if the exchange is two legs
1506 long).
- 1507 3. If the exchange is not complete, the requestor uses a
1508 `<wst:RequestSecurityTokenResponse>` that contains additional negotiation/challenge
1509 information.
- 1510 4. The process repeats at step 2 until the negotiation/challenge is complete (a token is returned or a
1511 Fault occurs). In the case where token information is returned in the final leg, it is returned in the
1512 form of a `<wst:RequestSecurityTokenResponseCollection>`.

1513
1514 The negotiation/challenge information is passed in binding/profile-specific elements that are placed inside
1515 of the `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>` elements.

1516
1517 It is RECOMMENDED that at least the `<wsu:Timestamp>` element be included in messages (as per
1518 [\[WS-Security\]](#)) as a way to ensure freshness of the messages in the exchange. Other types of
1519 challenges MAY also be included. For example, a `<wsp:Policy>` element may be used to negotiate
1520 desired policy behaviors of both parties. Multiple challenges and responses MAY be included.

1521 8.2 Signature Challenges

1522 Exchange requests are issued by including an element that describes the exchange (e.g. challenge) and
1523 responses contain an element describing the response. For example, signature challenges are
1524 processed using the `<wst:SignChallenge>` element. The response is returned in a
1525 `<wst:SignChallengeResponse>` element. Both the challenge and the response elements are
1526 specified within the `<wst:RequestSecurityTokenResponse>` element. Some forms of negotiation
1527 MAY specify challenges along with responses to challenges from the other party. It should be noted that
1528 the requestor MAY provide exchange information (e.g. a challenge) to the recipient in the initial request.
1529 Consequently, these elements are also allowed within a `<wst:RequestSecurityToken>` element.

1530
1531 The syntax of these elements is as follows:

```
1532 <wst:SignChallenge xmlns:wst="...">  
1533   <wst:Challenge ...>...</wst:Challenge>  
1534 </wst:SignChallenge>
```

```
1535  
1536 <wst:SignChallengeResponse xmlns:wst="...">  
1537   <wst:Challenge ...>...</wst:Challenge>  
1538 </wst:SignChallengeResponse>
```

1539

1540 The following describes the attributes and tags listed in the schema above:

1541 *.../wst:SignChallenge*

1542 This optional element describes a challenge that requires the other party to sign a specified set of
1543 information.

1544 *.../wst:SignChallenge/wst:Challenge*

1545 This required string element describes the value to be signed. In order to prevent certain types of
1546 attacks (such as man-in-the-middle), it is strongly RECOMMENDED that the challenge be bound
1547 to the negotiation. For example, the challenge SHOULD track (such as using a digest of) any
1548 relevant data exchanged such as policies, tokens, replay protection, etc. As well, if the challenge
1549 is happening over a secured channel, a reference to the channel SHOULD also be included.
1550 Furthermore, the recipient of a challenge SHOULD verify that the data tracked (digested)
1551 matches their view of the data exchanged. The exact algorithm MAY be defined in profiles or
1552 agreed to by the parties.

1553 *.../SignChallenge/{any}*

1554 This is an extensibility mechanism to allow additional negotiation types to be used.

1555 *.../wst:SignChallenge/@{any}*

1556 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added
1557 to the element.

1558 *.../wst:SignChallengeResponse*

1559 This optional element describes a response to a challenge that requires the signing of a specified
1560 set of information.

1561 *.../wst:SignChallengeResponse/wst:Challenge*

1562 If a challenge was issued, the response MUST contain the challenge element exactly as
1563 received. As well, while the RSTR response SHOULD always be signed, if a challenge was
1564 issued, the RSTR MUST be signed (and the signature coupled with the message to prevent
1565 replay).

1566 *.../wst:SignChallengeResponse/{any}*

1567 This is an extensibility mechanism to allow additional negotiation types to be used.

1568 *.../wst:SignChallengeResponse/@{any}*

1569 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added
1570 to the element.

1571 **8.3 Binary Exchanges and Negotiations**

1572 Exchange requests may also utilize existing binary formats passed within the WS-Trust framework. A
1573 generic mechanism is provided for this that includes a URI attribute to indicate the type of binary
1574 exchange.

1575
1576 The syntax of this element is as follows:

```
1577 <wst:BinaryExchange Value="..." EncodingType="..." xmlns:wst="...">  
1578 </wst:BinaryExchange>
```

1579 The following describes the attributes and tags listed in the schema above (note that the ellipses below
1580 indicate that this element may be placed in different containers. For this specification, these are limited to
1581 <wst:RequestSecurityToken> and <wst:RequestSecurityTokenResponse>):

1582 *.../wst:BinaryExchange*

1583 This optional element is used for a security negotiation that involves exchanging binary blobs as
1584 part of an existing negotiation protocol. The contents of this element are blob-type-specific and
1585 are encoded using base64 (unless otherwise specified).

1586 *.../wst:BinaryExchange/@ValueType*

1587 This required attribute specifies a URI to identify the type of negotiation (and the value space of
1588 the blob – the element's contents).

1589 *.../wst:BinaryExchange/@EncodingType*

1590 This required attribute specifies a URI to identify the encoding format (if different from base64) of
1591 the negotiation blob. Refer to [WS-Security] for sample encoding format URIs.

1592 *.../wst:BinaryExchange/@{any}*

1593 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added
1594 to the element.

1595 Some binary exchanges result in a shared state/context between the involved parties. It is
1596 RECOMMENDED that at the conclusion of the exchange, a new token and proof-of-possession token be
1597 returned. A common approach is to use the negotiated key as a "secure channel" mechanism to secure
1598 the new token and proof-of-possession token.

1599 For example, an exchange might establish a shared secret *Sx* that can then be used to sign the final
1600 response and encrypt the proof-of-possession token.

1601 **8.4 Key Exchange Tokens**

1602 In some cases it may be necessary to provide a key exchange token so that the other party (either
1603 requestor or issuer) can provide entropy or key material as part of the exchange. Challenges may not
1604 always provide a usable key as the signature may use a signing-only certificate.

1605

1606 The section describes two optional elements that can be included in RST and RSTR elements to indicate
1607 that a Key Exchange Token (KET) is desired, or to provide a KET.

1608 The syntax of these elements is as follows (Note that the ellipses below indicate that this element may be
1609 placed in different containers. For this specification, these are limited to

1610 `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>`):

1611 `<wst:RequestKET xmlns:wst="..." />`

1612

1613 `<wst:KeyExchangeToken xmlns:wst="...">...</wst:KeyExchangeToken>`

1614

1615 The following describes the attributes and tags listed in the schema above:

1616 *.../wst:RequestKET*

1617 This optional element is used to indicate that the receiving party (either the original requestor or
1618 issuer) should provide a KET to the other party on the next leg of the exchange.

1619 *.../wst:KeyExchangeToken*

1620 This optional element is used to provide a key exchange token. The contents of this element
1621 either contain the security token to be used for key exchange or a reference to it.

1622 8.5 Custom Exchanges

1623 Using the extensibility model described in this specification, any custom XML-based exchange can be
1624 defined in a separate binding/profile document. In such cases elements are defined which are carried in
1625 the RST and RSTR elements.

1626

1627 It should be noted that it is NOT REQUIRED that exchange elements be symmetric. That is, a specific
1628 exchange mechanism MAY use multiple elements at different times, depending on the state of the
1629 exchange.

1630 8.6 Signature Challenge Example

1631 Here is an example exchange involving a signature challenge. In this example, a service requests a
1632 custom token using a X.509 certificate for authentication. The issuer uses the exchange mechanism to
1633 challenge the requestor to sign a random value (to ensure message freshness). The requestor provides
1634 a signature of the requested data and, once validated, the issuer then issues the requested token.

1635

1636 The first message illustrates the initial request that is signed with the private key associated with the
1637 requestor's X.509 certificate:

```
1638 <S11:Envelope xmlns:S11="..." xmlns:wsse="..."
1639     xmlns:wsu="..." xmlns:wst="...">
1640   <S11:Header>
1641     ...
1642     <wsse:Security>
1643       <wsse:BinarySecurityToken
1644         wsu:Id="reqToken"
1645         ValueType="...X509v3">
1646         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...
1647       </wsse:BinarySecurityToken>
1648       <ds:Signature xmlns:ds="...">
1649         ...
1650         <ds:KeyInfo>
1651           <wsse:SecurityTokenReference>
1652             <wsse:Reference URI="#reqToken"/>
1653           </wsse:SecurityTokenReference>
1654         </ds:KeyInfo>
1655       </ds:Signature>
1656     </wsse:Security>
1657     ...
1658   </S11:Header>
1659   <S11:Body>
1660     <wst:RequestSecurityToken>
1661       <wst:TokenType>
1662         http://example.org/mySpecialToken
1663       </wst:TokenType>
1664       <wst:RequestType>
1665         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1666       </wst:RequestType>
1667     </wst:RequestSecurityToken>
1668   </S11:Body>
1669 </S11:Envelope>
```

1670

1671 The issuer (recipient) service doesn't trust the sender's timestamp (or one wasn't specified) and issues a
1672 challenge using the exchange framework defined in this specification. This message is signed using the
1673 private key associated with the issuer's X.509 certificate and contains a random challenge that the
1674 requestor must sign:

1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698

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

1699
1700
1701
1702

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

1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726

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

1727
1728
1729
1730

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

1731
1732
1733

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

```

1734     ...
1735     <wsse:Security>
1736         <wsse:BinarySecurityToken
1737             wsu:Id="issuerToken"
1738             ValueType="...X509v3">
1739             DFJHuedsujfnrnv45JZc0...
1740         </wsse:BinarySecurityToken>
1741         <ds:Signature xmlns:ds="...">
1742             ...
1743         </ds:Signature>
1744     </wsse:Security>
1745     ...
1746 </S11:Header>
1747 <S11:Body>
1748     <wst:RequestSecurityTokenResponseCollection>
1749     <wst:RequestSecurityTokenResponse>
1750     <wst:RequestedSecurityToken>
1751     <xyz:CustomToken xmlns:xyz="...">
1752     ...
1753     </xyz:CustomToken>
1754 </wst:RequestedSecurityToken>
1755 <wst:RequestedProofToken>
1756 <xenc:EncryptedKey Id="newProof">
1757     ...
1758 </xenc:EncryptedKey>
1759 </wst:RequestedProofToken>
1760 </wst:RequestSecurityTokenResponse>
1761 </wst:RequestSecurityTokenResponseCollection>
1762 </S11:Body>
1763 </S11:Envelope>

```

1764 8.7 Custom Exchange Example

1765 Here is another illustrating the syntax for a token request using a custom XML exchange. For brevity,
1766 only the RST and RSTR elements are illustrated. Note that the framework allows for an arbitrary number
1767 of exchanges, although this example illustrates the use of four legs. The request uses a custom
1768 exchange element and the requestor signs only the non-mutable element of the message:

```

1769 <wst:RequestSecurityToken xmlns:wst="...">
1770 <wst:TokenType>
1771     http://example.org/mySpecialToken
1772 </wst:TokenType>
1773 <wst:RequestType>
1774     http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1775 </wst:RequestType>
1776 <xyz:CustomExchange xmlns:xyz="...">
1777     ...
1778 </xyz:CustomExchange>
1779 </wst:RequestSecurityToken>

```

1780
1781 The issuer service (recipient) responds with another leg of the custom exchange and signs the response
1782 (non-mutable aspects) with its token:

```

1783 <wst:RequestSecurityTokenResponse xmlns:wst="...">
1784 <xyz:CustomExchange xmlns:xyz="...">
1785     ...
1786 </xyz:CustomExchange>
1787 </wst:RequestSecurityTokenResponse>

```

1789 The requestor receives the issuer's exchange and issues a response that is signed using the requestor's
 1790 token and continues the custom exchange. The signature covers all non-mutable aspects of the
 1791 message to prevent certain types of security attacks:

```
1792 <wst:RequestSecurityTokenResponse xmlns:wst="...">
1793   <xyz:CustomExchange xmlns:xyz="...">
1794     ...
1795   </xyz:CustomExchange>
1796 </wst:RequestSecurityTokenResponse>
```

1797
 1798 The issuer processes the exchange and determines that the exchange is complete and that a token
 1799 should be issued. Consequently it issues the requested token(s) and the associated proof-of-possession
 1800 token. The proof-of-possession token is encrypted for the requestor using the requestor's public key.

```
1801 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
1802   <wst:RequestSecurityTokenResponse>
1803     <wst:RequestedSecurityToken>
1804       <xyz:CustomToken xmlns:xyz="...">
1805         ...
1806       </xyz:CustomToken>
1807     </wst:RequestedSecurityToken>
1808     <wst:RequestedProofToken>
1809       <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
1810         ...
1811       </xenc:EncryptedKey>
1812     </wst:RequestedProofToken>
1813     <wst:RequestedProofToken>
1814       <xenc:EncryptedKey xmlns:xenc="...">...</xenc:EncryptedKey>
1815     </wst:RequestedProofToken>
1816   </wst:RequestSecurityTokenResponse>
1817 </wst:RequestSecurityTokenResponseCollection>
```

1818 It should be noted that other example exchanges include the issuer returning a final custom exchange
 1819 element, and another example where a token isn't returned.

1820 8.8 Protecting Exchanges

1821 There are some attacks, such as forms of man-in-the-middle, that can be applied to token requests
 1822 involving exchanges. It is RECOMMENDED that the exchange sequence be protected. This may be
 1823 built into the exchange messages, but if metadata is provided in the RST or RSTR elements, then it is
 1824 subject to attack.

1825
 1826 Consequently, it is RECOMMENDED that keys derived from exchanges be linked cryptographically to the
 1827 exchange. For example, a hash can be computed by computing the SHA1 of the exclusive
 1828 canonicalization [XML-C14N] of all RST and RSTR elements in messages exchanged. This value can
 1829 then be combined with the exchanged secret(s) to create a new master secret that is bound to the data
 1830 both parties sent/received.

1831
 1832 To this end, the following computed key algorithm is defined to be optionally used in these scenarios:

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

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

1833 8.9 Authenticating Exchanges

1834 After an exchange both parties have a shared knowledge of a key (or keys) that can then be used to
 1835 secure messages. However, in some cases it may be desired to have the issuer prove to the requestor
 1836 that it knows the key (and that the returned metadata is valid) prior to the requestor using the data.
 1837 However, until the exchange is actually completed it may (and is often) inappropriate to use the computed
 1838 keys. As well, using a token that hasn't been returned to secure a message may complicate processing
 1839 since it crosses the boundary of the exchange and the underlying message security. This means that it
 1840 may not be appropriate to sign the final leg of the exchange using the key derived from the exchange.

1841
 1842 For this reason an authenticator is defined that provides a way for the issuer to verify the hash as part of
 1843 the token issuance. Specifically, when an authenticator is returned, the
 1844 `<wst:RequestSecurityTokenResponseCollection>` element is returned. This contains one
 1845 RSTR with the token being returned as a result of the exchange and a second RSTR that contains the
 1846 authenticator (this order SHOULD be used). When an authenticator is used, RSTRs MUST use the
 1847 `@Context` element so that the authenticator can be correlated to the token issuance. The authenticator is
 1848 separated from the RSTR because otherwise computation of the RST/RSTR hash becomes more
 1849 complex. The authenticator is represented using the `<wst:Authenticator>` element as illustrated
 1850 below:

```

1851 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
1852   <wst:RequestSecurityTokenResponse Context="...">
1853     ...
1854   </wst:RequestSecurityTokenResponse>
1855   <wst:RequestSecurityTokenResponse Context="...">
1856     <wst:Authenticator>
1857       <wst:CombinedHash>...</wst:CombinedHash>
1858     ...
1859     </wst:Authenticator>
1860   </wst:RequestSecurityTokenResponse>
1861 </wst:RequestSecurityTokenResponseCollection>

```

1862
 1863 The following describes the attributes and elements listed in the schema overview above (the ... notation
 1864 below represents the path RSTR/RSTR and is used for brevity):

1865 `.../wst:Authenticator`

1866 This optional element provides verification (authentication) of a computed hash.

1867 `.../wst:Authenticator/wst:CombinedHash`

1868 This optional element proves the hash and knowledge of the computed key. This is done by
 1869 providing the base64 encoding of the first 256 bits of the P_SHA1 digest of the computed key and
 1870 the concatenation of the hash determined for the computed key and the string "AUTH-HASH".
 1871 Specifically, P_SHA1(*computed-key*, H + "AUTH-HASH")₀₋₂₅₅. The octets for the "AUTH-HASH"
 1872 string are the UTF-8 octets.

1873
 1874 This `<wst:CombinedHash>` element is optional (and an open content model is used) to allow for
 1875 different authenticators in the future.

9 Key and Token Parameter Extensions

1876

1877 This section outlines additional parameters that can be specified in token requests and responses.
1878 Typically they are used with issuance requests, but since all types of requests may issue security tokens
1879 they could apply to other bindings.

9.1 On-Behalf-Of Parameters

1881 In some scenarios the requestor is obtaining a token on behalf of another party. These parameters
1882 specify the issuer and original requestor of the token being used as the basis of the request. The syntax
1883 is as follows (note that the base elements described above are included here italicized for completeness):

```
1884 <wst:RequestSecurityToken xmlns:wst="...">  
1885   <wst:TokenType>...</wst:TokenType>  
1886   <wst:RequestType>...</wst:RequestType>  
1887   ...  
1888   <wst:OnBehalfOf>...</wst:OnBehalfOf>  
1889   <wst:Issuer>...</wst:Issuer>  
1890 </wst:RequestSecurityToken>
```

1891

1892 The following describes the attributes and elements listed in the schema overview above:

1893 */wst:RequestSecurityToken/wst:OnBehalfOf*

1894 This optional element indicates that the requestor is making the request on behalf of another.
1895 The identity on whose behalf the request is being made is specified by placing a security token,
1896 <wsse:SecurityTokenReference> element, or <wsa:EndpointReference> element
1897 within the <wst:OnBehalfOf> element. The requestor MAY provide proof of possession of the
1898 key associated with the OnBehalfOf identity by including a signature in the RST security header
1899 generated using the OnBehalfOf token that signs the primary signature of the RST (i.e. endorsing
1900 supporting token concept from WS-SecurityPolicy). Additional signed supporting tokens
1901 describing the OnBehalfOf context MAY also be included within the RST security header.

1902 */wst:RequestSecurityToken/wst:Issuer*

1903 This optional element specifies the issuer of the security token that is presented in the message.
1904 This element's type is an endpoint reference as defined in [\[WS-Addressing\]](#).

1905

1906 In the following illustrates the syntax for a proxy that is requesting a security token on behalf of another
1907 requestor or end-user.

```
1908 <wst:RequestSecurityToken xmlns:wst="...">  
1909   <wst:TokenType>...</wst:TokenType>  
1910   <wst:RequestType>...</wst:RequestType>  
1911   ...  
1912   <wst:OnBehalfOf>endpoint-reference</wst:OnBehalfOf>  
1913 </wst:RequestSecurityToken>
```

9.2 Key and Encryption Requirements

1915 This section defines extensions to the <wst:RequestSecurityToken> element for requesting specific
1916 types of keys or algorithms or key and algorithms as specified by a given policy in the return token(s). In
1917 some cases the service may support a variety of key types, sizes, and algorithms. These parameters
1918 allow a requestor to indicate its desired values. It should be noted that the issuer's policy indicates if input

1919 values must be adhered to and faults generated for invalid inputs, or if the issuer will provide alternative
1920 values in the response.

1921

1922 Although illustrated using the `<wst:RequestSecurityToken>` element, these options can also be
1923 returned in a `<wst:RequestSecurityTokenResponse>` element.

1924 The syntax for these optional elements is as follows (note that the base elements described above are
1925 included here italicized for completeness):

```
1926 <wst:RequestSecurityToken xmlns:wst="...">  
1927   <wst:TokenType>...</wst:TokenType>  
1928   <wst:RequestType>...</wst:RequestType>  
1929   ...  
1930   <wst:AuthenticationType>...</wst:AuthenticationType>  
1931   <wst:KeyType>...</wst:KeyType>  
1932   <wst:KeySize>...</wst:KeySize>  
1933   <wst:SignatureAlgorithm>...</wst:SignatureAlgorithm>  
1934   <wst:EncryptionAlgorithm>...</wst:EncryptionAlgorithm>  
1935   <wst:CanonicalizationAlgorithm>...</wst:CanonicalizationAlgorithm>  
1936   <wst:ComputedKeyAlgorithm>...</wst:ComputedKeyAlgorithm>  
1937   <wst:Encryption>...</wst:Encryption>  
1938   <wst:ProofEncryption>...</wst:ProofEncryption>  
1939   <wst:KeyWrapAlgorithm>...</wst:KeyWrapAlgorithm>  
1940   <wst:UseKey Sig="..."> </wst:UseKey>  
1941   <wst:SignWith>...</wst:SignWith>  
1942   <wst:EncryptWith>...</wst:EncryptWith>  
1943 </wst:RequestSecurityToken>
```

1944

1945 The following describes the attributes and elements listed in the schema overview above:

1946 */wst:RequestSecurityToken/wst:AuthenticationType*

1947 This optional URI element indicates the type of authentication desired, specified as a URI. This
1948 specification does not predefine classifications; these are specific to token services as is the
1949 relative strength evaluations. The relative assessment of strength is up to the recipient to
1950 determine. That is, requestors should be familiar with the recipient policies. For example, this
1951 might be used to indicate which of the four U.S. government authentication levels is required.

1952 */wst:RequestSecurityToken/wst:KeyType*

1953 This optional URI element indicates the type of key desired in the security token. The predefined
1954 values are identified in the table below. Note that some security token formats have fixed key
1955 types. It should be noted that new algorithms can be inserted by defining URIs in other
1956 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.

1957 */wst:RequestSecurityToken/wst:KeySize*

- 1958 This optional integer element indicates the size of the key required specified in number of bits.
 1959 This is a request, and, as such, the requested security token is not obligated to use the requested
 1960 key size. That said, the recipient SHOULD try to use a key at least as strong as the specified
 1961 value if possible. The information is provided as an indication of the desired strength of the
 1962 security.
- 1963 */wst:RequestSecurityToken/wst:SignatureAlgorithm*
- 1964 This optional URI element indicates the desired signature algorithm used within the returned
 1965 token. This is specified as a URI indicating the algorithm (see [XML-Signature](#)] for typical signing
 1966 algorithms).
- 1967 */wst:RequestSecurityToken/wst:EncryptionAlgorithm*
- 1968 This optional URI element indicates the desired encryption algorithm used within the returned
 1969 token. This is specified as a URI indicating the algorithm (see [XML-Encrypt](#)] for typical
 1970 encryption algorithms).
- 1971 */wst:RequestSecurityToken/wst:CanonicalizationAlgorithm*
- 1972 This optional URI element indicates the desired canonicalization method used within the returned
 1973 token. This is specified as a URI indicating the method (see [XML-Signature](#)] for typical
 1974 canonicalization methods).
- 1975 */wst:RequestSecurityToken/wst:ComputedKeyAlgorithm*
- 1976 This optional URI element indicates the desired algorithm to use when computed keys are used
 1977 for issued tokens.
- 1978 */wst:RequestSecurityToken/wst:Encryption*
- 1979 This optional element indicates that the requestor desires any returned secrets in issued security
 1980 tokens to be encrypted for the specified token. That is, so that the owner of the specified token
 1981 can decrypt the secret. Normally the security token is the contents of this element but a security
 1982 token reference MAY be used instead. If this element isn't specified, the token used as the basis
 1983 of the request (or specialized knowledge) is used to determine how to encrypt the key.
- 1984 */wst:RequestSecurityToken/wst:ProofEncryption*
- 1985 This optional element indicates that the requestor desires any returned secrets in proof-of-
 1986 possession tokens to be encrypted for the specified token. That is, so that the owner of the
 1987 specified token can decrypt the secret. Normally the security token is the contents of this element
 1988 but a security token reference MAY be used instead. If this element isn't specified, the token
 1989 used as the basis of the request (or specialized knowledge) is used to determine how to encrypt
 1990 the key.
- 1991 */wst:RequestSecurityToken/wst:KeyWrapAlgorithm*
- 1992 This optional URI element indicates the desired algorithm to use for key wrapping when STS
 1993 encrypts the issued token for the relying party using an asymmetric key.
- 1994 */wst:RequestSecurityToken/wst:UseKey*
- 1995 If the requestor wishes to use an existing key rather than create a new one, then this optional
 1996 element can be used to reference the security token containing the desired key. This element
 1997 either contains a security token or a `<wsse:SecurityTokenReference>` element that
 1998 references the security token containing the key that should be used in the returned token. If
 1999 `<wst:KeyType>` is not defined and a key type is not implicitly known to the service, it MAY be
 2000 determined from the token (if possible). Otherwise this parameter is meaningless and is ignored.
 2001 Requestors SHOULD demonstrate authorized use of the public key provided.
- 2002 */wst:RequestSecurityToken/wst:UseKey/@Sig*
- 2003 In order to *authenticate* the key referenced, a signature MAY be used to prove the referenced
 2004 token/key. If specified, this optional attribute indicates the ID of the corresponding signature (by

2005 URI reference). When this attribute is present, a key need not be specified inside the element
2006 since the referenced signature will indicate the corresponding token (and key).

2007 */wst:RequestSecurityToken/wst:SignWith*

2008 This optional URI element indicates the desired signature algorithm to be used with the issued
2009 security token (typically from the policy of the target site for which the token is being requested).
2010 While any of these optional elements MAY be included in RSTRs, this one is a likely candidate if
2011 there is some doubt (e.g., an X.509 cert that can only use DSS).

2012 */wst:RequestSecurityToken/wst:EncryptWith*

2013 This optional URI element indicates the desired encryption algorithm to be used with the issued
2014 security token (typically from the policy of the target site for which the token is being requested.)
2015 While any of these optional elements MAY be included in RSTRs, this one is a likely candidate if
2016 there is some doubt.

2017 The following summarizes the various algorithm parameters defined above. T is the issued token, P is the
2018 proof key.
2019

2020 **SignatureAlgorithm** - The signature algorithm to use to sign T

2021 **EncryptionAlgorithm** - The encryption algorithm to use to encrypt T

2022 **CanonicalizationAlgorithm** - The canonicalization algorithm to use when signing T

2023 **ComputedKeyAlgorithm** - The key derivation algorithm to use if using a symmetric key for P
2024 where P is computed using client, server, or combined entropy

2025 **Encryption** - The token/key to use when encrypting T

2026 **ProofEncryption** - The token/key to use when encrypting P

2027 **UseKey** - This is P. This is generally used when the client supplies a public-key that it wishes to
2028 be embedded in T as the proof key

2029 **SignWith** - The signature algorithm the client intends to employ when using P to
2030 sign

2031 The encryption algorithms further differ based on whether the issued token contains asymmetric key or
2032 symmetric key. Furthermore, they differ based on what type of key is used to protect the issued token
2033 from the STS to the relying party. The following cases can occur:

2034 T contains symmetric key/STS uses symmetric key to encrypt T for RP

2035 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP
2036 when using the proof key (e.g. AES256)

2037 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS should use to
2038 encrypt the T (e.g. AES256)

2039

2040 T contains symmetric key/STS uses asymmetric key to encrypt T for RP

2041 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP
2042 when using the proof key (e.g. AES256)

2043 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS should use to
2044 encrypt T for RP (e.g. AES256)

2045 **KeyWrapAlgorithm** – used to indicate the KeyWrap algorithm that the STS should use to wrap
2046 the generated key that is used to encrypt the T for RP

2047

2048 T contains asymmetric key/STS uses symmetric key to encrypt T for RP

2049 **EncryptWith** – used to indicate the KeyWrap algorithm that the client will use to

2050 protect the symmetric key that is used to protect messages to RP when using the proof key (e.g.
2051 RSA-OAEP-MGF1P)

2052 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS should use to
2053 encrypt T for RP (e.g. AES256)

2054

2055 T contains asymmetric key/STS uses asymmetric key to encrypt T for RP

2056 **EncryptWith** - used to indicate the KeyWrap algorithm that the client will use to
2057 protect symmetric key that is used to protect message to RP when using the proof
2058 key (e.g. RSA-OAEP-MGF1P)

2059 **EncryptionAlgorithm** - used to indicate the symmetric algorithm that the STS should use to
2060 encrypt T for RP (e.g. AES256)

2061 **KeyWrapAlgorithm** – used to indicate the KeyWrap algorithm that the STS should use to wrap
2062 the generated key that is used to encrypt the T for RP

2063

2064 The example below illustrates a request that utilizes several of these parameters. A request is made for a
2065 custom token using a username and password as the basis of the request. For security, this token is
2066 encrypted (see "encUsername") for the recipient using the recipient's public key and referenced in the
2067 encryption manifest. The message is protected by a signature using a public key from the sender and
2068 authorized by the username and password.

2069

2070 The requestor would like the custom token to contain a 1024-bit public key whose value can be found in
2071 the key provided with the "proofSignature" signature (the key identified by "requestProofToken"). The
2072 token should be signed using RSA-SHA1 and encrypted for the token identified by
2073 "requestEncryptionToken". The proof should be encrypted using the token identified by
2074 "requestProofToken".

```
2075 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."  
2076   xmlns:wst="..." xmlns:ds="..." xmlns:xenc="...">  
2077   <S11:Header>  
2078     ...  
2079     <wsse:Security>  
2080       <xenc:ReferenceList>...</xenc:ReferenceList>  
2081       <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>  
2082       <wsse:BinarySecurityToken wsu:Id="requestEncryptionToken"  
2083         ValueType="...SomeTokenType" xmlns:x="...">  
2084         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
2085       </wsse:BinarySecurityToken>  
2086       <wsse:BinarySecurityToken wsu:Id="requestProofToken"  
2087         ValueType="...SomeTokenType" xmlns:x="...">  
2088         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
2089       </wsse:BinarySecurityToken>  
2090       <ds:Signature Id="proofSignature">  
2091         ... signature proving requested key ...  
2092         ... key info points to the "requestedProofToken" token ...  
2093       </ds:Signature>  
2094     </wsse:Security>  
2095     ...  
2096   </S11:Header>  
2097   <S11:Body wsu:Id="req">  
2098     <wst:RequestSecurityToken>  
2099       <wst:TokenType>  
2100       http://example.org/mySpecialToken  
2101     </wst:TokenType>  
2102     <wst:RequestType>
```

2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121

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

2122 9.3 Delegation and Forwarding Requirements

2123 This section defines extensions to the `<wst:RequestSecurityToken>` element for indicating
2124 delegation and forwarding requirements on the requested security token(s).

2125 The syntax for these extension elements is as follows (note that the base elements described above are
2126 included here italicized for completeness):

2127
2128
2129
2130
2131
2132
2133
2134

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

2135 */wst:RequestSecurityToken/wst:DelegateTo*

2136 This optional element indicates that the requested or issued token be delegated to another
2137 identity. The identity receiving the delegation is specified by placing a security token or
2138 `<wsse:SecurityTokenReference>` element within the `<wst:DelegateTo>` element.

2139 */wst:RequestSecurityToken/wst:Forwardable*

2140 This optional element, of type `xs:boolean`, specifies whether the requested security token should
2141 be marked as "Forwardable". In general, this flag is used when a token is normally bound to the
2142 requestor's machine or service. Using this flag, the returned token MAY be used from any source
2143 machine so long as the key is correctly proven. The default value of this flag is true.

2144 */wst:RequestSecurityToken/wst:Delegatable*

2145 This optional element, of type `xs:boolean`, specifies whether the requested security token should
2146 be marked as "Delegatable". Using this flag, the returned token MAY be delegated to another
2147 party. This parameter SHOULD be used in conjunction with `<wst:DelegateTo>`. The default
2148 value of this flag is false.

2149

2150 The following illustrates the syntax of a request for a custom token that can be delegated to the indicated
2151 recipient (specified in the binary security token) and used in the specified interval.

2152
2153
2154

```
<wst:RequestSecurityToken xmlns:wst="...">
  <wst:TokenType>
    http://example.org/mySpecialToken
```

```

2155     </wst:TokenType>
2156     <wst:RequestType>
2157         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
2158     </wst:RequestType>
2159     <wst:DelegateTo>
2160         <wsse:BinarySecurityToken
2161 xmlns:wsse="...">...</wsse:BinarySecurityToken>
2162     </wst:DelegateTo>
2163     <wst:Delegatable>true</wst:Delegatable>
2164 </wst:RequestSecurityToken>

```

2165 9.4 Policies

2166 This section defines extensions to the `<wst:RequestSecurityToken>` element for passing policies.

2167

2168 The syntax for these extension elements is as follows (note that the base elements described above are
2169 included here italicized for completeness):

```

2170     <wst:RequestSecurityToken xmlns:wst="...">
2171         <wst:TokenType>...</wst:TokenType>
2172         <wst:RequestType>...</wst:RequestType>
2173         ...
2174         <wsp:Policy xmlns:wsp="...">...</wsp:Policy>
2175         <wsp:PolicyReference xmlns:wsp="...">...</wsp:PolicyReference>
2176     </wst:RequestSecurityToken>

```

2177

2178 The following describes the attributes and elements listed in the schema overview above:

2179 */wst:RequestSecurityToken/wsp:Policy*

2180 This optional element specifies a policy (as defined in [WS-Policy]) that indicates desired settings
2181 for the requested token. The policy specifies defaults that can be overridden by the elements
2182 defined in the previous sections.

2183 */wst:RequestSecurityToken/wsp:PolicyReference*

2184 This optional element specifies a reference to a policy (as defined in [WS-Policy]) that indicates
2185 desired settings for the requested token. The policy specifies defaults that can be overridden by
2186 the elements defined in the previous sections.

2187

2188 The following illustrates the syntax of a request for a custom token that provides a set of policy
2189 statements about the token or its usage requirements.

```

2190     <wst:RequestSecurityToken xmlns:wst="...">
2191         <wst:TokenType>
2192             http://example.org/mySpecialToken
2193         </wst:TokenType>
2194         <wst:RequestType>
2195             http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
2196         </wst:RequestType>
2197         <wsp:Policy xmlns:wsp="...">
2198             ...
2199         </wsp:Policy>
2200     </wst:RequestSecurityToken>

```

2201 9.5 Authorized Token Participants

2202 This section defines extensions to the `<wst:RequestSecurityToken>` element for passing information
2203 about which parties are authorized to participate in the use of the token. This parameter is typically used

2204 when there are additional parties using the token or if the requestor needs to clarify the actual parties
2205 involved (for some profile-specific reason).

2206 It should be noted that additional participants will need to prove their identity to recipients in addition to
2207 proving their authorization to use the returned token. This typically takes the form of a second signature
2208 or use of transport security.

2209

2210 The syntax for these extension elements is as follows (note that the base elements described above are
2211 included here italicized for completeness):

```
2212 <wst:RequestSecurityToken xmlns:wst="...">  
2213   <wst:TokenType>...</wst:TokenType>  
2214   <wst:RequestType>...</wst:RequestType>  
2215   ...  
2216   <wst:Participants>  
2217     <wst:Primary>...</wst:Primary>  
2218     <wst:Participant>...</wst:Participant>  
2219   </wst:Participants>  
2220 </wst:RequestSecurityToken>
```

2221

2222 The following describes elements and attributes used in a `<wsc:SecurityContextToken>` element.

2223 */wst:RequestSecurityToken/wst:Participants/*

2224 This optional element specifies the participants sharing the security token. Arbitrary types may be
2225 used to specify participants, but a typical case is a security token or an endpoint reference (see
2226 [\[WS-Addressing\]](#)).

2227 */wst:RequestSecurityToken/wst:Participants/wst:Primary*

2228 This optional element specifies the primary user of the token (if one exists).

2229 */wst:RequestSecurityToken/wst:Participants/wst:Participant*

2230 This optional element specifies participant (or multiple participants by repeating the element) that
2231 play a (profile-dependent) role in the use of the token or who are allowed to use the token.

2232 */wst:RequestSecurityToken/wst:Participants/{any}*

2233 This is an extensibility option to allow other types of participants and profile-specific elements to
2234 be specified.

2235 10 Key Exchange Token Binding

2236 Using the token request framework, this section defines a binding for requesting a key exchange token
2237 (KET). That is, if a requestor desires a token that can be used to encrypt key material for a recipient.

2238
2239 For this binding, the following actions are defined to enable specific processing context to be conveyed to
2240 the recipient:

```
2241 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET  
2242 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KET  
2243 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal
```

2244
2245 For this binding, the `RequestType` element contains the following URI:

```
2246 http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET
```

2247
2248 For this binding very few parameters are specified as input. Optionally the `<wst:TokenType>` element
2249 can be specified in the request can indicate desired type response token carrying the key for key
2250 exchange; however, this isn't commonly used.

2251 The applicability scope (e.g. `<wsp:AppliesTo>`) MAY be specified if the requestor desires a key
2252 exchange token for a specific scope.

2253
2254 It is RECOMMENDED that the response carrying the key exchange token be secured (e.g., signed by the
2255 issuer or someone who can speak on behalf of the target for which the KET applies).

2256
2257 Care should be taken when using this binding to prevent possible man-in-the-middle and substitution
2258 attacks. For example, responses to this request SHOULD be secured using a token that can speak for
2259 the desired endpoint.

2260
2261 The RSTR for this binding carries the `<RequestedSecurityToken>` element even if a token is returned
2262 (note that the base elements described above are included here italicized for completeness):

```
2263 <wst:RequestSecurityToken xmlns:wst="...">  
2264   <wst:TokenType>...</wst:TokenType>  
2265   <wst:RequestType>...</wst:RequestType>  
2266   ...  
2267 </wst:RequestSecurityToken>
```

```
2268  
2269 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2270   <wst:RequestSecurityTokenResponse>  
2271     <wst:TokenType>...</wst:TokenType>  
2272     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>  
2273     ...  
2274   </wst:RequestSecurityTokenResponse>  
2275 </wst:RequestSecurityTokenResponseCollection>
```

2276
2277 The following illustrates the syntax for requesting a key exchange token. In this example, the KET is
2278 returned encrypted for the requestor since it had the credentials available to do that. Alternatively the

2279 request could be made using transport security (e.g. TLS) and the key could be returned directly using
2280 <wst:BinarySecret>.

```
2281 <wst:RequestSecurityToken xmlns:wst="...">  
2282 <wst:RequestType>  
2283 http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET  
2284 </wst:RequestType>  
2285 </wst:RequestSecurityToken>
```

```
2286  
2287 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2288 <wst:RequestSecurityTokenResponse>  
2289 <wst:RequestedSecurityToken>  
2290 <xenc:EncryptedKey xmlns:xenc="...">...</xenc:EncryptedKey>  
2291 </wst:RequestedSecurityToken>  
2292 </wst:RequestSecurityTokenResponse>  
2293 </wst:RequestSecurityTokenResponseCollection>
```

2294

11 Error Handling

2295
2296
2297
2298
2299
2300
2301
2302

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

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

12 Security Considerations

2303

2304 As stated in the Goals section of this document, this specification is meant to provide extensible
2305 framework and flexible syntax, with which one could implement various security mechanisms. This
2306 framework and syntax by itself does not provide any guarantee of security. When implementing and using
2307 this framework and syntax, one must make every effort to ensure that the result is not vulnerable to any
2308 one of a wide range of attacks.

2309

2310 It is not feasible to provide a comprehensive list of security considerations for such an extensible set of
2311 mechanisms. A complete security analysis must be conducted on specific solutions based on this
2312 specification. Below we illustrate some of the security concerns that often come up with protocols of this
2313 type, but we stress that this *is not an exhaustive list of concerns*.

2314

2315 The following statements about signatures and signing apply to messages sent on unsecured channels.

2316

2317 It is critical that all the security-sensitive message elements must be included in the scope of the
2318 message signature. As well, the signatures for conversation authentication must include a timestamp,
2319 nonce, or sequence number depending on the degree of replay prevention required as described in [[WS-
2320 Security](#)] and the UsernameToken Profile. Also, conversation establishment should include the policy so
2321 that supported algorithms and algorithm priorities can be validated.

2322

2323 It is required that security token issuance messages be signed to prevent tampering. If a public key is
2324 provided, the request should be signed by the corresponding private key to prove ownership. As well,
2325 additional steps should be taken to eliminate replay attacks (refer to [[WS-Security](#)] for additional
2326 information). Similarly, all token references should be signed to prevent any tampering.

2327

2328 Security token requests are susceptible to denial-of-service attacks. Care should be taken to mitigate
2329 such attacks as is warranted by the service.

2330

2331 For security, tokens containing a symmetric key or a password should only be sent to parties who have a
2332 need to know that key or password.

2333

2334 For privacy, tokens containing personal information (either in the claims, or indirectly by identifying who is
2335 currently communicating with whom) should only be sent according to the privacy policies governing
2336 these data at the respective organizations.

2337

2338 For some forms of multi-message exchanges, the exchanges are susceptible to attacks whereby
2339 signatures are altered. To address this, it is suggested that a signature confirmation mechanism be used.
2340 In such cases, each leg should include the confirmation of the previous leg. That is, leg 2 includes
2341 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
2342 correctness of the message outside of the message body.

2343

2344 There are many other security concerns that one may need to consider in security protocols. The list
2345 above should not be used as a "check list" instead of a comprehensive security analysis.

2346

2347 It should be noted that use of unsolicited RSTRs implies that the recipient is prepared to accept such
2348 issuances. Recipients should ensure that such issuances are properly authorized and recognize their
2349 use could be used in denial-of-service attacks.

2350 In addition to the consideration identified here, readers should also review the security considerations in
2351 [\[WS-Security\]](#).

2352

2353 Both token cancellation bindings defined in this specification require that the STS MUST NOT validate or
2354 renew the token after it has been successfully canceled. The STS must take care to ensure that the token
2355 is properly invalidated before confirming the cancel request or sending the cancel notification to the client.
2356 This can be more difficult if the token validation or renewal logic is physically separated from the issuance
2357 and cancellation logic. It is out of scope of this spec how the STS propagates the token cancellation to its
2358 other components. If STS cannot ensure that the token was properly invalidated it MUST NOT send the
2359 cancel notification or confirm the cancel request to the client.

2360

A. Key Exchange

2361 Key exchange is an integral part of token acquisition. There are several mechanisms by which keys are
2362 exchanged using [WS-Security] and WS-Trust. This section highlights and summarizes these
2363 mechanisms. Other specifications and profiles may provide additional details on key exchange.

2364

2365 Care must be taken when employing a key exchange to ensure that the mechanism does not provide an
2366 attacker with a means of discovering information that could only be discovered through use of secret
2367 information (such as a private key).

2368

2369 It is therefore important that a shared secret should only be considered as trustworthy as its source. A
2370 shared secret communicated by means of the direct encryption scheme described in section I.1 is
2371 acceptable if the encryption key is provided by a completely trustworthy key distribution center (this is the
2372 case in the Kerberos model). Such a key would not be acceptable for the purposes of decrypting
2373 information from the source that provided it since an attacker might replay information from a prior
2374 transaction in the hope of learning information about it.

2375

2376 In most cases the other party in a transaction is only imperfectly trustworthy. In these cases both parties
2377 should contribute entropy to the key exchange by means of the <wst:entropy> element.

2378

A.1 Ephemeral Encryption Keys

2379 The simplest form of key exchange can be found in [WS-Security] for encrypting message data. As
2380 described in [WS-Security] and [XML-Encrypt], when data is encrypted, a temporary key can be used to
2381 perform the encryption which is, itself, then encrypted using the <xenc:EncryptedKey> element.

2382

2383 The illustrates the syntax for encrypting a temporary key using the public key in an issuer name and serial
2384 number:

2385

```
<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>
```

2386

2387

2388

2389

2390

2391

2392

2393

2394

2395

2396

2397

2398

2399

A.2 Requestor-Provided Keys

2400 When a request sends a message to an issuer to request a token, the client can provide proposed key
2401 material using the <wst:Entropy> element. If the issuer doesn't contribute any key material, this is
2402 used as the secret (key). This information is encrypted for the issuer either using
2403 <xenc:EncryptedKey> or by using a transport security. If the requestor provides key material that the

2404 recipient doesn't accept, then the issuer should reject the request. Note that the issuer need not return
2405 the key provided by the requestor.

2406

2407 The following illustrates the syntax of a request for a custom security token and includes a secret that is
2408 to be used for the key. In this example the entropy is encrypted for the issuer (if transport security was
2409 used for confidentiality then the <wst:Entropy> element would contain a <wst:BinarySecret>
2410 element):

```
2411 <wst:RequestSecurityToken xmlns:wst="...">  
2412 <wst:TokenType>  
2413   http://example.org/mySpecialToken  
2414 </wst:TokenType>  
2415 <wst:RequestType>  
2416   http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
2417 </wst:RequestType>  
2418 <wst:Entropy>  
2419   <xenc:EncryptedData xmlns:xenc="...">...</xenc:EncryptedData>  
2420 </wst:Entropy>  
2421 </wst:RequestSecurityToken>
```

2422 **A.3 Issuer-Provided Keys**

2423 If a requestor fails to provide key material, then issued proof-of-possession tokens contain an issuer-
2424 provided secret that is encrypted for the requestor (either using <xenc:EncryptedKey> or by using a
2425 transport security).

2426

2427 The following illustrates the syntax of a token being returned with an associated proof-of-possession
2428 token that is encrypted using the requestor's public key.

```
2429 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2430 <wst:RequestSecurityTokenResponse>  
2431 <wst:RequestedSecurityToken>  
2432 <xyz:CustomToken xmlns:xyz="...">  
2433   ...  
2434 </xyz:CustomToken>  
2435 </wst:RequestedSecurityToken>  
2436 <wst:RequestedProofToken>  
2437 <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">  
2438   ...  
2439 </xenc:EncryptedKey>  
2440 </wst:RequestedProofToken>  
2441 </wst:RequestSecurityTokenResponse>  
2442 </wst:RequestSecurityTokenResponseCollection>
```

2443 **A.4 Composite Keys**

2444 The safest form of key exchange/generation is when both the requestor and the issuer contribute to the
2445 key material. In this case, the request sends encrypted key material. The issuer then returns additional
2446 encrypted key material. The actual secret (key) is computed using a function of the two pieces of data.
2447 Ideally this secret is never used and, instead, keys derived are used for message protection.

2448

2449 The following example illustrates a server, having received a request with requestor entropy returning its
2450 own entropy, which is used in conjunction with the requestor's to generate a key. In this example the
2451 entropy is not encrypted because the transport is providing confidentiality (otherwise the
2452 <wst:Entropy> element would have an <xenc:EncryptedData> element).

2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464

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

2465 **A.5 Key Transfer and Distribution**

2466 There are also a few mechanisms where existing keys are transferred to other parties.

2467 **A.5.1 Direct Key Transfer**

2468 If one party has a token and key and wishes to share this with another party, the key can be directly
2469 transferred. This is accomplished by sending an RSTR (either in the body or header) to the other party.
2470 The RSTR contains the token and a proof-of-possession token that contains the key encrypted for the
2471 recipient.

2472

2473 In the following example a custom token and its associated proof-of-possession token are known to party
2474 A who wishes to share them with party B. In this example, A is a member in a secure on-line chat
2475 session and is inviting B to join the conversation. After authenticating B, A sends B an RSTR. The RSTR
2476 contains the token and the key is communicated as a proof-of-possession token that is encrypted for B:

2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490

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

2491 **A.5.2 Brokered Key Distribution**

2492 A third party may also act as a broker to transfer keys. For example, a requestor may obtain a token and
2493 proof-of-possession token from a third-party STS. The token contains a key encrypted for the target
2494 service (either using the service's public key or a key known to the STS and target service). The proof-of-
2495 possession token contains the same key encrypted for the requestor (similarly this can use public or
2496 symmetric keys).

2497

2498 In the following example a custom token and its associated proof-of-possession token are returned from a
2499 broker B to a requestor R for access to service S. The key for the session is contained within the custom
2500 token encrypted for S using either a secret known by B and S or using S's public key. The same secret is
2501 encrypted for R and returned as the proof-of-possession token:

```

2502 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
2503   <wst:RequestSecurityTokenResponse>
2504     <wst:RequestedSecurityToken>
2505       <xyz:CustomToken xmlns:xyz="...">
2506         ...
2507         <xenc:EncryptedKey xmlns:xenc="...">
2508           ...
2509         </xenc:EncryptedKey>
2510         ...
2511       </xyz:CustomToken>
2512     </wst:RequestedSecurityToken>
2513     <wst:RequestedProofToken>
2514       <xenc:EncryptedKey Id="newProof">
2515         ...
2516       </xenc:EncryptedKey>
2517     </wst:RequestedProofToken>
2518   </wst:RequestSecurityTokenResponse>
2519 </wst:RequestSecurityTokenResponseCollection>

```

2520 **A.5.3 Delegated Key Transfer**

2521 Key transfer can also take the form of delegation. That is, one party transfers the right to use a key
 2522 without actually transferring the key. In such cases, a delegation token, e.g. XrML, is created that
 2523 identifies a set of rights and a delegation target and is secured by the delegating party. That is, one key
 2524 indicates that another key can use a subset (or all) of its rights. The delegate can provide this token and
 2525 prove itself (using its own key – the delegation target) to a service. The service, assuming the trust
 2526 relationships have been established and that the delegator has the right to delegate, can then authorize
 2527 requests sent subject to delegation rules and trust policies.

2528
 2529 In this example a custom token is issued from party A to party B. The token indicates that B (specifically
 2530 B's key) has the right to submit purchase orders. The token is signed using a secret key known to the
 2531 target service T and party A (the key used to ultimately authorize the requests that B makes to T), and a
 2532 new session key that is encrypted for T. A proof-of-possession token is included that contains the
 2533 session key encrypted for B. As a result, B is *effectively* using A's key, but doesn't actually know the key.

```

2534 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
2535   <wst:RequestSecurityTokenResponse>
2536     <wst:RequestedSecurityToken>
2537       <xyz:CustomToken xmlns:xyz="...">
2538         ...
2539         <xyz:DelegateTo>B</xyz:DelegateTo>
2540         <xyz:DelegateRights>
2541           SubmitPurchaseOrder
2542         </xyz:DelegateRights>
2543         <xenc:EncryptedKey xmlns:xenc="...">
2544           ...
2545         </xenc:EncryptedKey>
2546         <ds:Signature xmlns:ds="...">...</ds:Signature>
2547         ...
2548       </xyz:CustomToken>
2549     </wst:RequestedSecurityToken>
2550     <wst:RequestedProofToken>
2551       <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">
2552         ...
2553       </xenc:EncryptedKey>
2554     </wst:RequestedProofToken>
2555   </wst:RequestSecurityTokenResponse>
2556 </wst:RequestSecurityTokenResponseCollection>

```

2557 **A.5.4 Authenticated Request/Reply Key Transfer**

2558 In some cases the RST/RSTR mechanism is not used to transfer keys because it is part of a simple
2559 request/reply. However, there may be a desire to ensure mutual authentication as part of the key
2560 transfer. The mechanisms of [WS-Security] can be used to implement this scenario.

2561
2562 Specifically, the sender wishes the following:

- 2563 • Transfer a key to a recipient that they can use to secure a reply
- 2564 • Ensure that only the recipient can see the key
- 2565 • Provide proof that the sender issued the key

2566
2567 This scenario could be supported by encrypting and then signing. This would result in roughly the
2568 following steps:

- 2569 1. Encrypt the message using a generated key
- 2570 2. Encrypt the key for the recipient
- 2571 3. Sign the encrypted form, any other relevant keys, and the encrypted key

2572
2573 However, if there is a desire to sign prior to encryption then the following general process is used:

- 2574 1. Sign the appropriate message parts using a random key (or ideally a key derived from a random
2575 key)
- 2576 2. Encrypt the appropriate message parts using the random key (or ideally another key derived from
2577 the random key)
- 2578 3. Encrypt the random key for the recipient
- 2579 4. Sign just the encrypted key

2580
2581 This would result in a <wsse:Security> header that looks roughly like the following:

```
2582 <wsse:Security xmlns:wsse="..." xmlns:wsu="..."  
2583     xmlns:ds="..." xmlns:xenc="...">  
2584   <wsse:BinarySecurityToken wsu:Id="myToken">  
2585     ...  
2586   </wsse:BinarySecurityToken>  
2587   <ds:Signature>  
2588     ...signature over #secret using token #myToken...  
2589   </ds:Signature>  
2590   <xenc:EncryptedKey Id="secret">  
2591     ...  
2592   </xenc:EncryptedKey>  
2593   <xenc:ReferenceList>  
2594     ...manifest of encrypted parts using token #secret...  
2595   </xenc:ReferenceList>  
2596   <ds:Signature>  
2597     ...signature over key message parts using token #secret...  
2598   </ds:Signature>  
2599 </wsse:Security>
```

2600
2601 As well, instead of an <xenc:EncryptedKey> element, the actual token could be passed using
2602 <xenc:EncryptedData>. The result might look like the following:

```
2603 <wsse:Security xmlns:wsse="..." xmlns:wsu="..."  
2604     xmlns:ds="..." xmlns:xenc="...">
```

```

2605 <wsse:BinarySecurityToken wsu:Id="myToken">
2606     ...
2607 </wsse:BinarySecurityToken>
2608 <ds:Signature>
2609     ...signature over #secret or #Esecret using token #myToken...
2610 </ds:Signature>
2611 <xenc:EncryptedData Id="Esecret">
2612     ...Encrypted version of a token with Id="secret"...
2613 </xenc:EncryptedData>
2614 <xenc:ReferenceList>
2615     ...manifest of encrypted parts using token #secret...
2616 </xenc:ReferenceList>
2617 <ds:Signature>
2618     ...signature over key message parts using token #secret...
2619 </ds:Signature>
2620 </wsse:Security>

```

2621 **A.6 Perfect Forward Secrecy**

2622 In some situations it is desirable for a key exchange to have the property of perfect forward secrecy. This
 2623 means that it is impossible to reconstruct the shared secret even if the private keys of the parties are
 2624 disclosed.

2625
 2626 The most straightforward way to attain perfect forward secrecy when using asymmetric key exchange is
 2627 to dispose of one's key exchange key pair periodically (or even after every key exchange), replacing it
 2628 with a fresh one. Of course, a freshly generated public key must still be authenticated (using any of the
 2629 methods normally available to prove the identity of a public key's owner).

2630
 2631 The perfect forward secrecy property may be achieved by specifying a `<wst:entropy>` element that
 2632 contains an `<xenc:EncryptedKey>` that is encrypted under a public key pair created for use in a single
 2633 key agreement. The public key does not require authentication since it is only used to provide additional
 2634 entropy. If the public key is modified, the key agreement will fail. Care should be taken, when using this
 2635 method, to ensure that the now-secret entropy exchanged via the `<wst:entropy>` element is not
 2636 revealed elsewhere in the protocol (since such entropy is often assumed to be publicly revealed plaintext,
 2637 and treated accordingly).

2638
 2639 Although any public key scheme might be used to achieve perfect forward secrecy (in either of the above
 2640 methods) it is generally desirable to use an algorithm that allows keys to be generated quickly. The Diffie-
 2641 Hellman key exchange is often used for this purpose since generation of a key only requires the
 2642 generation of a random integer and calculation of a single modular exponent.

2643

B. WSDL

2644 The WSDL below does not fully capture all the possible message exchange patterns, but captures the
2645 typical message exchange pattern as described in this document.

```
2646 <?xml version="1.0"?>
2647 <wsdl:definitions
2648     targetNamespace="http://docs.oasis-open.org/ws-sx/ws-
2649     trust/200512/wsdl"
2650     xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/wsdl"
2651     xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
2652     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
2653     xmlns:xs="http://www.w3.org/2001/XMLSchema"
2654 >
2655 <!-- this is the WS-I BP-compliant way to import a schema -->
2656     <wsdl:types>
2657         <xs:schema>
2658             <xs:import
2659                 namespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
2660                 schemaLocation="http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-
2661                 trust.xsd"/>
2662             </xs:schema>
2663         </wsdl:types>
2664
2665 <!-- WS-Trust defines the following GEDs -->
2666         <wsdl:message name="RequestSecurityTokenMsg">
2667             <wsdl:part name="request" element="wst:RequestSecurityToken" />
2668         </wsdl:message>
2669         <wsdl:message name="RequestSecurityTokenResponseMsg">
2670             <wsdl:part name="response"
2671                 element="wst:RequestSecurityTokenResponse" />
2672         </wsdl:message>
2673         <wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
2674             <wsdl:part name="responseCollection"
2675                 element="wst:RequestSecurityTokenResponseCollection"/>
2676         </wsdl:message>
2677
2678 <!-- This portType models the full request/response the Security Token
2679 Service: -->
2680
2681         <wsdl:portType name="WSecurityRequestor">
2682             <wsdl:operation name="SecurityTokenResponse">
2683                 <wsdl:input
2684                     message="tns:RequestSecurityTokenResponseMsg"/>
2685             </wsdl:operation>
2686             <wsdl:operation name="SecurityTokenResponse2">
2687                 <wsdl:input
2688                     message="tns:RequestSecurityTokenResponseCollectionMsg"/>
2689             </wsdl:operation>
2690             <wsdl:operation name="Challenge">
2691                 <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
2692                 <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
2693             </wsdl:operation>
2694             <wsdl:operation name="Challenge2">
2695                 <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
2696                 <wsdl:output
2697                     message="tns:RequestSecurityTokenResponseCollectionMsg"/>
2698             </wsdl:operation>
2699         </wsdl:portType>
2700
2701 <!-- These portTypes model the individual message exchanges -->
```

```
2702 <wsdl:portType name="SecurityTokenRequestService">
2703   <wsdl:operation name="RequestSecurityToken">
2704     <wsdl:input message="tns:RequestSecurityTokenMsg"/>
2705   </wsdl:operation>
2706 </wsdl:portType>
2707
2708 <wsdl:portType name="SecurityTokenService">
2709   <wsdl:operation name="RequestSecurityToken">
2710     <wsdl:input message="tns:RequestSecurityTokenMsg"/>
2711     <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
2712   </wsdl:operation>
2713   <wsdl:operation name="RequestSecurityToken2">
2714     <wsdl:input message="tns:RequestSecurityTokenMsg"/>
2715     <wsdl:output
2716       message="tns:RequestSecurityTokenResponseCollectionMsg"/>
2717   </wsdl:operation>
2718 </wsdl:portType>
2719 </wsdl:definitions>
2720
2721
```

2722

C. Acknowledgements

2723 The following individuals have participated in the creation of this specification and are gratefully
2724 acknowledged:

2725 **Original Authors of the initial contribution:**

2726 Steve Anderson, OpenNetwork
2727 Jeff Bohren, OpenNetwork
2728 Toufic Boubez, Layer 7
2729 Marc Chanliau, Computer Associates
2730 Giovanni Della-Libera, Microsoft
2731 Brendan Dixon, Microsoft
2732 Praerit Garg, Microsoft
2733 Martin Gudgin (Editor), Microsoft
2734 Phillip Hallam-Baker, VeriSign
2735 Maryann Hondo, IBM
2736 Chris Kaler, Microsoft
2737 Hal Lockhart, BEA
2738 Robin Martherus, Oblix
2739 Hiroshi Maruyama, IBM
2740 Anthony Nadalin (Editor), IBM
2741 Nataraj Nagaratnam, IBM
2742 Andrew Nash, Reactivity
2743 Rob Philpott, RSA Security
2744 Darren Platt, Ping Identity
2745 Hemma Prafullchandra, VeriSign
2746 Maneesh Sahu, Actional
2747 John Shewchuk, Microsoft
2748 Dan Simon, Microsoft
2749 Davanum Srinivas, Computer Associates
2750 Elliot Waingold, Microsoft
2751 David Waite, Ping Identity
2752 Doug Walter, Microsoft
2753 Riaz Zolfonoon, RSA Security

2754

2755 **Original Acknowledgments of the initial contribution:**

2756 Paula Austel, IBM
2757 Keith Ballinger, Microsoft
2758 Bob Blakley, IBM
2759 John Brezak, Microsoft
2760 Tony Cowan, IBM
2761 Cédric Fournet, Microsoft
2762 Vijay Gajjala, Microsoft
2763 HongMei Ge, Microsoft
2764 Satoshi Hada, IBM
2765 Heather Hinton, IBM
2766 Slava Kavsan, RSA Security
2767 Scott Konersmann, Microsoft
2768 Leo Laferriere, Computer Associates

2769 Paul Leach, Microsoft
2770 Richard Levinson, Computer Associates
2771 John Linn, RSA Security
2772 Michael McIntosh, IBM
2773 Steve Millet, Microsoft
2774 Birgit Pfitzmann, IBM
2775 Fumiko Satoh, IBM
2776 Keith Stobie, Microsoft
2777 T.R. Vishwanath, Microsoft
2778 Richard Ward, Microsoft
2779 Hervey Wilson, Microsoft
2780
2781 **TC Members during the development of this specification:**
2782 Don Adams, Tibco Software Inc.
2783 Jan Alexander, Microsoft Corporation
2784 Steve Anderson, BMC Software
2785 Donal Arundel, IONA Technologies
2786 Howard Bae, Oracle Corporation
2787 Abbie Barbir, Nortel Networks Limited
2788 Charlton Barreto, Adobe Systems
2789 Mighael Botha, Software AG, Inc.
2790 Toufic Boubez, Layer 7 Technologies Inc.
2791 Norman Brickman, Mitre Corporation
2792 Melissa Brumfield, Booz Allen Hamilton
2793 Lloyd Burch, Novell
2794 Scott Cantor, Internet2
2795 Greg Carpenter, Microsoft Corporation
2796 Steve Carter, Novell
2797 Ching-Yun (C.Y.) Chao, IBM
2798 Martin Chapman, Oracle Corporation
2799 Kate Cherry, Lockheed Martin
2800 Henry (Hyenvui) Chung, IBM
2801 Luc Clement, Systinet Corp.
2802 Paul Cotton, Microsoft Corporation
2803 Glen Daniels, Sonic Software Corp.
2804 Peter Davis, Neustar, Inc.
2805 Martijn de Boer, SAP AG
2806 Werner Dittmann, Siemens AG
2807 Abdeslem DJAOUI, CCLRC-Rutherford Appleton Laboratory
2808 Fred Dushin, IONA Technologies
2809 Petr Dvorak, Systinet Corp.
2810 Colleen Evans, Microsoft Corporation
2811 Ruchith Fernando, WSO2
2812 Mark Fussell, Microsoft Corporation

2813 Vijay Gajjala, Microsoft Corporation
2814 Marc Goodner, Microsoft Corporation
2815 Hans Granqvist, VeriSign
2816 Martin Gudgin, Microsoft Corporation
2817 Tony Gullotta, SOA Software Inc.
2818 Jiandong Guo, Sun Microsystems
2819 Phillip Hallam-Baker, VeriSign
2820 Patrick Harding, Ping Identity Corporation
2821 Heather Hinton, IBM
2822 Frederick Hirsch, Nokia Corporation
2823 Jeff Hodges, Neustar, Inc.
2824 Will Hopkins, BEA Systems, Inc.
2825 Alex Hristov, Otecia Incorporated
2826 John Hughes, PA Consulting
2827 Diane Jordan, IBM
2828 Venugopal K, Sun Microsystems
2829 Chris Kaler, Microsoft Corporation
2830 Dana Kaufman, Forum Systems, Inc.
2831 Paul Knight, Nortel Networks Limited
2832 Ramanathan Krishnamurthy, IONA Technologies
2833 Christopher Kurt, Microsoft Corporation
2834 Kelvin Lawrence, IBM
2835 Hubert Le Van Gong, Sun Microsystems
2836 Jong Lee, BEA Systems, Inc.
2837 Rich Levinson, Oracle Corporation
2838 Tommy Lindberg, Dajeil Ltd.
2839 Mark Little, JBoss Inc.
2840 Hal Lockhart, BEA Systems, Inc.
2841 Mike Lyons, Layer 7 Technologies Inc.
2842 Eve Maler, Sun Microsystems
2843 Ashok Malhotra, Oracle Corporation
2844 Anand Mani, CrimsonLogic Pte Ltd
2845 Jonathan Marsh, Microsoft Corporation
2846 Robin Martherus, Oracle Corporation
2847 Miko Matsumura, Infravio, Inc.
2848 Gary McAfee, IBM
2849 Michael McIntosh, IBM
2850 John Merrells, Sxip Networks SRL
2851 Jeff Mischkinisky, Oracle Corporation
2852 Prateek Mishra, Oracle Corporation
2853 Bob Morgan, Internet2
2854 Vamsi Motukuru, Oracle Corporation

2855 Raajmohan Na, EDS
2856 Anthony Nadalin, IBM
2857 Andrew Nash, Reactivity, Inc.
2858 Eric Newcomer, IONA Technologies
2859 Duane Nickull, Adobe Systems
2860 Toshihiro Nishimura, Fujitsu Limited
2861 Rob Philpott, RSA Security
2862 Denis Pilipchuk, BEA Systems, Inc.
2863 Darren Platt, Ping Identity Corporation
2864 Martin Raeppe, SAP AG
2865 Nick Ragouzis, Enosis Group LLC
2866 Prakash Reddy, CA
2867 Alain Regnier, Ricoh Company, Ltd.
2868 Irving Reid, Hewlett-Packard
2869 Bruce Rich, IBM
2870 Tom Rutt, Fujitsu Limited
2871 Maneesh Sahu, Actional Corporation
2872 Frank Siebenlist, Argonne National Laboratory
2873 Joe Smith, Apani Networks
2874 Davanum Srinivas, WSO2
2875 Yakov Sverdlov, CA
2876 Gene Thurston, AmberPoint
2877 Victor Valle, IBM
2878 Asir Vedamuthu, Microsoft Corporation
2879 Greg Whitehead, Hewlett-Packard
2880 Ron Williams, IBM
2881 Corinna Witt, BEA Systems, Inc.
2882 Kyle Young, Microsoft Corporation

2884

D. Revision History

2885

[optional; should not be included in OASIS Standards]

2886

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

2887