



# WS-Trust 1.3

## OASIS Standard incorporating Proposed Errata 02

15 August 2008

### Specification URIs:

#### This Version:

<http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-spec-errata-cd-02.doc>  
<http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-spec-errata-cd-02.pdf>  
<http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-spec-errata-cd-02.html>

#### Previous Version:

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

#### Latest Version:

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

### 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

### Related work:

N/A

### Declared XML namespace(s):

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

### Abstract:

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

### Status:

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

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

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

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

---

## Notices

Copyright © OASIS® 1993–2008. All Rights Reserved. OASIS trademark, IPR and other policies apply.

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

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

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

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

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

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

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

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

---

# Table of Contents

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

8.3 Binary Exchanges and Negotiations.....	44
8.4 Key Exchange Tokens.....	45
8.5 Custom Exchanges.....	46
8.6 Signature Challenge Example .....	46
8.7 Custom Exchange Example .....	48
8.8 Protecting Exchanges.....	49
8.9 Authenticating Exchanges .....	50
9 Key and Token Parameter Extensions.....	52
9.1 On-Behalf-Of Parameters .....	52
9.2 Key and Encryption Requirements .....	52
9.3 Delegation and Forwarding Requirements .....	57
9.4 Policies.....	58
9.5 Authorized Token Participants.....	59
10 Key Exchange Token Binding .....	60
11 Error Handling .....	62
12 Security Considerations .....	63
A. Key Exchange .....	65
A.1 Ephemeral Encryption Keys.....	65
A.2 Requestor-Provided Keys .....	65
A.3 Issuer-Provided Keys .....	66
A.4 Composite Keys .....	66
A.5 Key Transfer and Distribution.....	67
A.5.1 Direct Key Transfer .....	67
A.5.2 Brokered Key Distribution .....	67
A.5.3 Delegated Key Transfer.....	68
A.5.4 Authenticated Request/Reply Key Transfer.....	69
A.6 Perfect Forward Secrecy.....	70
B. WSDL .....	71
C. Acknowledgements .....	73

---

# 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	<a href="http://schemas.xmlsoap.org/soap/envelope/">http://schemas.xmlsoap.org/soap/envelope/</a>	[SOAP]
S12	<a href="http://www.w3.org/2003/05/soap-envelope">http://www.w3.org/2003/05/soap-envelope</a>	[SOAP12]
wsu	<a href="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd</a>	[WS-Security]
wsse	<a href="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd</a>	[WS-Security]
wsse11	<a href="http://docs.oasis-open.org/wss/oasis-wss-wssecurity-secext-1.1.xsd">http://docs.oasis-open.org/wss/oasis-wss-wssecurity-secext-1.1.xsd</a>	[WS-Security]
wst	<a href="http://docs.oasis-open.org/ws-sx/ws-trust/200512">http://docs.oasis-open.org/ws-sx/ws-trust/200512</a>	This specification
ds	<a href="http://www.w3.org/2000/09/xmldsig#">http://www.w3.org/2000/09/xmldsig#</a>	[XML-Signature]
xenc	<a href="http://www.w3.org/2001/04/xmlenc#">http://www.w3.org/2001/04/xmlenc#</a>	[XML-Encrypt]
wsp	<a href="http://schemas.xmlsoap.org/ws/2004/09/policy">http://schemas.xmlsoap.org/ws/2004/09/policy</a>	[WS-Policy]
wsa	<a href="http://www.w3.org/2005/08/addressing">http://www.w3.org/2005/08/addressing</a>	[WS-Addressing]

xs	<a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a>	[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-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-  
173 security-1.0.pdf](http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-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-  
177 spec-os-SOAPMessageSecurity.pdf](http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-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)

205

## 2 Web Services Trust Model

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

211

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

216

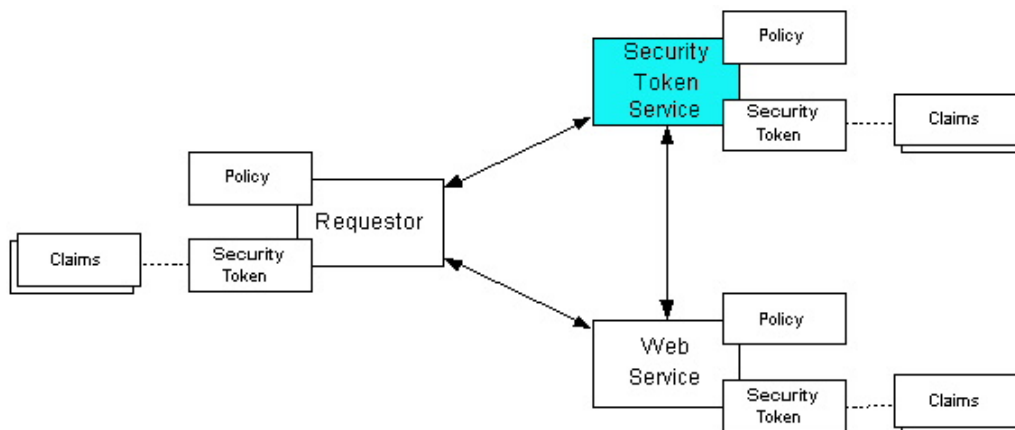
217 One way to demonstrate authorized use of a security token is to include a digital signature using the  
218 associated secret key (from a proof-of-possession token). This allows a requestor to prove a required set  
219 of claims by associating security tokens (e.g., PKIX, X.509 certificates) with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

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

230

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

233



234

235 This general security model – claims, policies, and security tokens – subsumes and supports several  
236 more specific models such as identity-based authorization, access control lists, and capabilities-based  
237 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  
243 token 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  
291 since it does trust the security token service that is engaging in the exchange for a new security token.  
292 The basis 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  
307 bootstrap trust and establish trust among a domain of services or extend this trust to other domains using  
308 any 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  
316 to 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

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

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

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

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

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

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

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

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

### 3.1 Requesting a Security Token

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

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

The syntax for this element is as follows:

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

378 */wst:RequestSecurityToken/wst:TokenType*

379 This OPTIONAL element describes the type of security token requested, specified as a URI.  
380 That is, the type of token that will be returned in the  
381 `<wst:RequestSecurityTokenResponse>` message. Token type URIs are typically defined in  
382 token profiles such as those in the OASIS 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 SHOULD NOT be required to parse the returned token. As a result, information that the requestor may  
417 desire, such as token lifetimes, SHOULD be returned in the response. Specifically, any field that the  
418 requestor includes SHOULD be returned. If an issuer doesn't want to repeat all input parameters, then, at  
419 a 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
<a href="http://docs.oasis-open.org/ws-sx/ws-trust/200512/AsymmetricKey">http://docs.oasis-open.org/ws-sx/ws-trust/200512/AsymmetricKey</a>	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
<a href="http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey">http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey</a>	A symmetric key token is returned (default)
<a href="http://docs.oasis-open.org/ws-sx/ws-trust/200512/Nonce">http://docs.oasis-open.org/ws-sx/ws-trust/200512/Nonce</a>	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  
519 the OPTIONAL use of the key extensions described in a later section. Additional profiles are needed to  
520 describe specific behaviors (and exclusions) when different combinations are used.

### 4.1 Requesting a Security Token

522 When requesting a security token to be issued, the following OPTIONAL elements MAY be included in  
523 the request and MAY be provided in the response. The syntax for these elements is as follows (note that  
524 the 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  
543 used (similar to the Kerberos target service model). This assumes that a token type can be  
544 inferred 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  
551 example, the service(s) to which this token applies. Refer to [\[WS-PolicyAttachment\]](#) for more  
552 information. Note that either this element or the `<wst:TokenType>` element SHOULD be  
553 defined in a `<wst:RequestSecurityToken>` message. In the situation where BOTH fields  
554 have values, the `<wsp:AppliesTo>` field takes precedence. This is because the issuing service  
555 is more 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  
559 REQUIRED 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  
563 this specification; it is expected that profiles and other specifications will define these URIs and  
564 the 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  
567 key. 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  
579 which 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/wsu: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 infoset 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  
589 for a postdated token. If this attribute isn't specified, then the current time is used as an initial  
590 period.

591 */wst:RequestSecurityToken/wst:Lifetime/wsu:Expires*

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

597

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

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

## 633 4.2 Request Security Token Collection

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

638

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

643

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

647

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

649

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

684

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

686

687 */wst:RequestSecurityTokenCollection*

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

## 691 4.2.1 Processing Rules

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

694

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

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

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

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

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

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

709

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

712

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

715

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

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

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

723

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

726

## 727 4.3 Returning a Security Token Collection

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

732 The syntax for this element is as follows:

```

733 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
734 <wst:RequestSecurityTokenResponse>...</wst:RequestSecurityTokenResponse> +
735 </wst:RequestSecurityTokenResponseCollection>

```

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

737 */wst:RequestSecurityTokenResponseCollection*

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

740 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

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

## 742 4.4 Returning a Security Token

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

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

```

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

```

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

764 */wst:RequestSecurityTokenResponse/wsp:AppliesTo*

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

769 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

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

775 */wst:RequestSecurityTokenResponse/wst:RequestedAttachedReference*

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



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

786 */wst:RequestSecurityTokenResponse/wst:RequestedUnattachedReference*

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

793 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken*

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

804 */wst:RequestSecurityTokenResponse/wst:Entropy*

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

809 */wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret*

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

812 */wst:RequestSecurityTokenResponse/wst:Lifetime*

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

## 816 4.4.1 wsp:AppliesTo in RST and RSTR

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

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

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

832

833 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.

## 834 4.4.2 Requested References

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

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

## 850 4.4.3 Keys and Entropy

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

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

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

862

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

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

873

874 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

#### 875 4.4.4 Returning Computed Keys

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

881  
882  
883  
884  
885  
886  
887

```

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

```

888

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

890 `/wst:RequestSecurityTokenResponse/wst:RequestedProofToken/wst:ComputedKey`

891 The value of this element is a URI describing how to compute the key. While this can be  
 892 extended by defining new URIs in other bindings and profiles, the following URI pre-defines one  
 893 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.

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

### 895 4.4.5 Sample Response with Encrypted Secret

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

```

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

### 914 4.4.6 Sample Response with Unencrypted Secret

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

```

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

#### 929 4.4.7 Sample Response with Token Reference

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

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

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

#### 953 4.4.8 Sample Response without Proof-of-Possession Token

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

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

967

#### 968 4.4.9 Zero or One Proof-of-Possession Token Case

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

975

```
976 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
977   <wst:RequestSecurityTokenResponse>  
978     <wst:RequestedSecurityToken>
```

```

979         <xyz:CustomToken xmlns:xyz="...">
980             ...
981         </xyz:CustomToken>
982     </wst:RequestedSecurityToken>
983     <wst:RequestedProofToken>
984         <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
985             ...
986         </xenc:EncryptedKey>
987     </wst:RequestedProofToken>
988 </wst:RequestSecurityTokenResponse>
989 </wst:RequestSecurityTokenResponseCollection>

```

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

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

```

995 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
996     <wst:RequestSecurityTokenResponse>
997         ...
998     </wst:RequestSecurityTokenResponse>
999     <wst:RequestSecurityTokenResponse>
1000         ...
1001     </wst:RequestSecurityTokenResponse>
1002     ...
1003 </wst:RequestSecurityTokenResponseCollection>

```

1004 The following describes the attributes and elements listed in the schema overview above:

1005 */wst:RequestSecurityTokenResponseCollection*

1006 This element is used to provide multiple RSTR responses, each of which has separate key  
1007 information. One or more RSTR elements are returned in the collection. This MUST always be  
1008 used on the final response to the RST.

1009 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

1010 Each RequestSecurityTokenResponse element is an individual RSTR.

1011 */wst:RequestSecurityTokenResponseCollection/{any}*

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

1013 */wst:RequestSecurityTokenResponseCollection/@{any}*

1014 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added.

1015 The following illustrates the syntax of a response that includes multiple tokens each, in a separate RSTR,  
1016 each with their own proof-of-possession token.

```

1017 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
1018     <wst:RequestSecurityTokenResponse>
1019         <wst:RequestedSecurityToken>
1020             <xyz:CustomToken xmlns:xyz="...">
1021                 ...
1022             </xyz:CustomToken>
1023         </wst:RequestedSecurityToken>
1024         <wst:RequestedProofToken>
1025             <xenc:EncryptedKey Id="newProofA">
1026                 ...
1027             </xenc:EncryptedKey>
1028         </wst:RequestedProofToken>
1029     </wst:RequestSecurityTokenResponse>
1030 <wst:RequestSecurityTokenResponse>

```

```

1031     <wst:RequestedSecurityToken>
1032         <abc:CustomToken xmlns:abc="...">
1033             ...
1034         </abc:CustomToken>
1035     </wst:RequestedSecurityToken>
1036     <wst:RequestedProofToken>
1037         <xenc:EncryptedKey Id="newProofB xmlns:xenc="...">
1038             ...
1039         </xenc:EncryptedKey>
1040     </wst:RequestedProofToken>
1041 </wst:RequestSecurityTokenResponse>
1042 </wst:RequestSecurityTokenResponseCollection>

```

## 1043 4.5 Returning Security Tokens in Headers

1044 In certain situations it is useful to issue one or more security tokens as part of a protocol other than  
1045 RST/RSTR. This typically requires that the tokens be passed in a SOAP header. The tokens present in  
1046 that element can then be referenced from elsewhere in the message. This section defines a specific  
1047 header element, whose type is the same as that of the `<wst:RequestSecurityTokenCollection>`  
1048 element (see Section 4.3), that can be used to carry issued tokens (and associated proof tokens,  
1049 references etc.) in a message.

```

1050 <wst:IssuedTokens xmlns:wst="...">
1051   <wst:RequestSecurityTokenResponse>
1052     ...
1053   </wst:RequestSecurityTokenResponse>+
1054 </wst:IssuedTokens>

```

1055  
1056 The following describes the attributes and elements listed in the schema overview above:

1057 */wst:IssuedTokens*

1058 This header element carries one or more issued security tokens. This element schema is defined  
1059 using the RequestSecurityTokenResponse schema type.

1060 */wst:IssuedTokens/wst:RequestSecurityTokenResponse*

1061 This element MUST appear at least once. Its meaning and semantics are as defined in Section 4.2.

1062 */wst:IssuedTokens/{any}*

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

1064 */wst:IssuedTokens/@{any}*

1065 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added.

1066  
1067 There MAY be multiple instances of the `<wst:IssuedTokens>` header in a given message. Such  
1068 instances MAY be targeted at the same actor/role. Intermediaries MAY add additional  
1069 `<wst:IssuedTokens>` header elements to a message. Intermediaries SHOULD NOT modify any  
1070 `<wst:IssuedTokens>` header already present in a message.

1071  
1072 It is RECOMMENDED that the `<wst:IssuedTokens>` header be signed to protect the integrity of the  
1073 issued tokens and of the issuance itself. If confidentiality protection of the `<wst:IssuedTokens>` header is  
1074 REQUIRED then the entire header MUST be encrypted using the `<wsse11:EncryptedHeader>` construct.  
1075 This helps facilitate re-issuance by the receiving party as that party can re-encrypt the entire header for  
1076 another party rather than having to extract and re-encrypt portions of the header.

1077

1078 The following example illustrates a response that includes multiple `<wst:IssuedTokens>` headers.

```
1079 <?xml version="1.0" encoding="utf-8"?>
1080 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsp="..." xmlns:ds="..."
1081 xmlns:x="...">
1082   <S11:Header>
1083     <wst:IssuedTokens>
1084       <wst:RequestSecurityTokenResponse>
1085         <wsp:AppliesTo>
1086           <x:SomeContext1 />
1087         </wsp:AppliesTo>
1088         <wst:RequestedSecurityToken>
1089           ...
1090         </wst:RequestedSecurityToken>
1091         ...
1092       </wst:RequestSecurityTokenResponse>
1093       <wst:RequestSecurityTokenResponse>
1094         <wsp:AppliesTo>
1095           <x:SomeContext1 />
1096         </wsp:AppliesTo>
1097         <wst:RequestedSecurityToken>
1098           ...
1099         </wst:RequestedSecurityToken>
1100         ...
1101       </wst:RequestSecurityTokenResponse>
1102     </wst:IssuedTokens>
1103     <wst:IssuedTokens S11:role="http://example.org/somerole" >
1104       <wst:RequestSecurityTokenResponse>
1105         <wsp:AppliesTo>
1106           <x:SomeContext2 />
1107         </wsp:AppliesTo>
1108         <wst:RequestedSecurityToken>
1109           ...
1110         </wst:RequestedSecurityToken>
1111         ...
1112       </wst:RequestSecurityTokenResponse>
1113     </wst:IssuedTokens>
1114   </S11:Header>
1115   <S11:Body>
1116     ...
1117   </S11:Body>
1118 </S11:Envelope>
```



---

## 5 Renewal Binding

1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161

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

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

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

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

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

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

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

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

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

The original proof information SHOULD be proven during renewal.

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

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

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

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

```
1162     <wst:Renewing Allow="..." OK="..." />
1163 </wst:RequestSecurityToken>
```

1164 */wst:RequestSecurityToken/wst:RenewTarget*

1165 This REQUIRED element identifies the token being renewed. This MAY contain a  
1166 <wsse:SecurityTokenReference> pointing at the token to be renewed or it MAY directly contain  
1167 the token to be renewed.

1168 */wst:RequestSecurityToken/wst:AllowPostdating*

1169 This OPTIONAL element indicates that returned tokens SHOULD allow requests for postdated  
1170 tokens. That is, this allows for tokens to be issued that are not immediately valid (e.g., a token  
1171 that can be used the next day).

1172 */wst:RequestSecurityToken/wst:Renewing*

1173 This OPTIONAL element is used to specify renew semantics for types that support this operation.

1174 */wst:RequestSecurityToken/wst:Renewing/@Allow*

1175 This OPTIONAL Boolean attribute is used to request a renewable token. If not specified, the  
1176 default value is *true*. A renewable token is one whose lifetime can be extended. This is done  
1177 using a renewal request. The recipient MAY allow renewals without demonstration of authorized  
1178 use of the token or they MAY fault.

1179 */wst:RequestSecurityToken/wst:Renewing/@OK*

1180 This OPTIONAL Boolean attribute is used to indicate that a renewable token is acceptable if the  
1181 requested duration exceeds the limit of the issuance service. That is, if *true* then tokens can be  
1182 renewed after their expiration. It should be noted that the token is NOT valid after expiration for  
1183 any operation except renewal. The default for this attribute is *false*. It NOT RECOMMENDED to  
1184 use this as it can leave you open to certain types of security attacks. Issuers MAY restrict the  
1185 period after expiration during which time the token can be renewed. This window is governed by  
1186 the issuer's policy.

1187 The following example illustrates a request for a custom token that can be renewed.

```
1188 <wst:RequestSecurityToken xmlns:wst="...">
1189   <wst:TokenType>
1190     http://example.org/mySpecialToken
1191   </wst:TokenType>
1192   <wst:RequestType>
1193     http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1194   </wst:RequestType>
1195   <wst:Renewing/>
1196 </wst:RequestSecurityToken>
```

1197  
1198 The following example illustrates a subsequent renewal request and response (note that for brevity only  
1199 the request and response are illustrated). Note that the response includes an indication of the lifetime of  
1200 the renewed token.

```
1201 <wst:RequestSecurityToken xmlns:wst="...">
1202   <wst:TokenType>
1203     http://example.org/mySpecialToken
1204   </wst:TokenType>
1205   <wst:RequestType>
1206     http://docs.oasis-open.org/ws-sx/ws-trust/200512/Renew
1207   </wst:RequestType>
1208   <wst:RenewTarget>
1209     ... reference to previously issued token ...
1210   </wst:RenewTarget>
1211 </wst:RequestSecurityToken>
```

```
1213 <wst:RequestSecurityTokenResponse xmlns:wst="...">
1214   <wst:TokenType>
1215     http://example.org/mySpecialToken
1216   </wst:TokenType>
1217   <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1218   <wst:Lifetime>...</wst:Lifetime>
1219   ...
1220 </wst:RequestSecurityTokenResponse>
```

---

## 6 Cancel Binding

1221

1222 Using the token request framework, this section defines bindings for requesting security tokens to be  
1223 cancelled:

1224 **Cancel** – When a previously issued token is no longer needed, the Cancel binding can be used  
1225 to cancel the token, terminating its use. After canceling a token at the issuer, a STS MUST not  
1226 validate or renew the token. A STS MAY initiate the revocation of a token, however, revocation is  
1227 out of scope of this specification and a client MUST NOT rely on it. If a client needs to ensure the  
1228 validity of a token, it MUST validate the token at the issuer.

1229

1230 For this binding, the following actions are defined to enable specific processing context to be conveyed to  
1231 the recipient:

```
1232 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel  
1233 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Cancel  
1234 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal
```

1235 For this binding, the `<wst:RequestType>` element uses the following URI:

```
1236 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
```

1237 Extensions MAY be specified in the request (and the response), but the semantics are not defined by this  
1238 binding.

1239

1240 The request MUST prove authorized use of the token being cancelled unless the recipient trusts the  
1241 requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its  
1242 identity to the issuer so that appropriate authorization occurs.

1243 In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key  
1244 bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the closure response.

1245

1246 A cancelled token is no longer valid for authentication and authorization usages.

1247 On success a cancel response is returned. This is an RSTR message with the  
1248 `<wst:RequestedTokenCancelled>` element in the body. On failure, a Fault is raised. It should be  
1249 noted that the cancel RSTR is informational. That is, the security token is cancelled once the cancel  
1250 request is processed.

1251

1252 The syntax of the request is as follows:

```
1253 <wst:RequestSecurityToken xmlns:wst="...">  
1254   <wst:RequestType>...</wst:RequestType>  
1255   ...  
1256   <wst:CancelTarget>...</wst:CancelTarget>  
1257 </wst:RequestSecurityToken>
```

1258 `/wst:RequestSecurityToken/wst:CancelTarget`

1259 This REQUIRED element identifies the token being cancelled. Typically this contains a  
1260 `<wsse:SecurityTokenReference>` pointing at the token, but it could also carry the token  
1261 directly.

1262 The following example illustrates a request to cancel a custom token.

```
1263 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
```

```

1264     <S11:Header>
1265         <wsse:Security>
1266             ...
1267         </wsse:Security>
1268     </S11:Header>
1269     <S11:Body>
1270         <wst:RequestSecurityToken>
1271             <wst:RequestType>
1272                 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
1273             </wst:RequestType>
1274             <wst:CancelTarget>
1275                 ...
1276             </wst:CancelTarget>
1277         </wst:RequestSecurityToken>
1278     </S11:Body>
1279 </S11:Envelope>

```

1280 The following example illustrates a response to cancel a custom token.

```

1281 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
1282     <S11:Header>
1283         <wsse:Security>
1284             ...
1285         </wsse:Security>
1286     </S11:Header>
1287     <S11:Body>
1288         <wst:RequestSecurityTokenResponse>
1289             <wst:RequestedTokenCancelled/>
1290         </wst:RequestSecurityTokenResponse>
1291     </S11:Body>
1292 </S11:Envelope>

```

## 1293 6.1 STS-initiated Cancel Binding

1294 Using the token request framework, this section defines an OPTIONAL binding for requesting security  
1295 tokens to be cancelled by the STS:

1296 **STS-initiated Cancel** – When a previously issued token becomes invalid on the STS, the STS-  
1297 initiated Cancel binding can be used to cancel the token, terminating its use. After canceling a  
1298 token, a STS MUST not validate or renew the token. This binding can be only used when STS  
1299 can send one-way messages to the original token requestor.

1300  
1301 For this binding, the following actions are defined to enable specific processing context to be conveyed to  
1302 the recipient:

```
1303 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/STSCancel
```

1304 For this binding, the <wst:RequestType> element uses the following URI:

```
1305 http://docs.oasis-open.org/ws-sx/ws-trust/200512/STSCancel
```

1306 Extensions MAY be specified in the request, but the semantics are not defined by this binding.

1307  
1308 The request MUST prove authorized use of the token being cancelled unless the recipient trusts the  
1309 requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its  
1310 identity to the issuer so that appropriate authorization occurs.

1311 In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key  
1312 bearing tokens MUST be signed.

1313

1314 A cancelled token is no longer valid for authentication and authorization usages.

1315

1316 The mechanism to determine the availability of STS-initiated Cancel binding on the STS is out of scope of  
1317 this specification. Similarly, how the client communicates its endpoint address to the STS so that it can  
1318 send the STSCancel messages to the client is out of scope of this specification. This functionality is  
1319 implementation specific and can be solved by different mechanisms that are not in scope for this  
1320 specification.

1321

1322 This is a one-way operation, no response is returned from the recipient of the message.

1323

1324 The syntax of the request is as follows:

```
1325 <wst:RequestSecurityToken xmlns:wst="...">  
1326   <wst:RequestType>...</wst:RequestType>  
1327   ...  
1328   <wst:CancelTarget>...</wst:CancelTarget>  
1329 </wst:RequestSecurityToken>
```

1330 */wst:RequestSecurityToken/wst:CancelTarget*

1331 This REQUIRED element identifies the token being cancelled. Typically this contains a  
1332 <wsse:SecurityTokenReference> pointing at the token, but it could also carry the token  
1333 directly.

1334 The following example illustrates a request to cancel a custom token.

```
1335 <?xml version="1.0" encoding="utf-8"?>  
1336 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">  
1337   <S11:Header>  
1338     <wsse:Security>  
1339       ...  
1340     </wsse:Security>  
1341   </S11:Header>  
1342   <S11:Body>  
1343     <wst:RequestSecurityToken>  
1344       <wst:RequestType>  
1345         http://docs.oasis-open.org/ws-sx/ws-trust/200512/STSCancel  
1346       </wst:RequestType>  
1347       <wst:CancelTarget>  
1348         ...  
1349       </wst:CancelTarget>  
1350     </wst:RequestSecurityToken>  
1351   </S11:Body>  
1352 </S11:Envelope>
```

---

## 7 Validation Binding

1353

1354 Using the token request framework, this section defines bindings for requesting security tokens to be  
1355 validated:

1356 **Validate** – The validity of the specified security token is evaluated and a result is returned. The  
1357 result MAY be a status, a new token, or both.

1358

1359 It should be noted that for this binding, a SOAP Envelope MAY be specified as a "security token" if the  
1360 requestor desires the envelope to be validated. In such cases the recipient SHOULD understand how to  
1361 process a SOAP envelope and adhere to SOAP processing semantics (e.g., mustUnderstand) of the  
1362 version of SOAP used in the envelope. Otherwise, the recipient SHOULD fault.

1363 For this binding, the following actions are defined to enable specific processing context to be conveyed to  
1364 the recipient:

```
1365 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate  
1366 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Validate  
1367 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/ValidateFinal
```

1368

1369 For this binding, the `<wst:RequestType>` element contains the following URI:

```
1370 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate
```

1371

1372 The request provides a token upon which the request is based and OPTIONAL tokens. As well, the  
1373 OPTIONAL `<wst:TokenType>` element in the request can indicate desired type response token. This  
1374 MAY be any supported token type or it MAY be the following URI indicating that only status is desired:

```
1375 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Status
```

1376

1377 For some use cases a status token is returned indicating the success or failure of the validation. In other  
1378 cases a security token MAY be returned and used for authorization. This binding assumes that the  
1379 validation requestor and provider are known to each other and that the general issuance parameters  
1380 beyond requesting a token type, which is OPTIONAL, are not needed (note that other bindings and  
1381 profiles could define different semantics).

1382

1383 For this binding an applicability scope (e.g., `<wsp:AppliesTo>`) need not be specified. It is assumed  
1384 that the applicability of the validation response relates to the provided information (e.g. security token) as  
1385 understood by the issuing service.

1386

1387 The validation binding does not allow the use of exchanges.

1388

1389 The RSTR for this binding carries the following element even if a token is returned (note that the base  
1390 elements described above are included here italicized for completeness):

```
1391 <wst:RequestSecurityToken xmlns:wst="...">  
1392   <wst:TokenType>...</wst:TokenType>  
1393   <wst:RequestType>...</wst:RequestType>  
1394   <wst:ValidateTarget>... </wst:ValidateTarget>  
1395   ...
```

1396

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

1397

1398

```
<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>
```

1399

1400

1401

1402

1403

1404

1405

1406

1407

1408

*/wst:RequestSecurityToken/wst:ValidateTarget*

1409

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

1410

1411

1412

*/wst:RequestSecurityTokenResponse/wst:Status*

1413

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

1414

1415

1416

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

1417

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

1418

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

1419

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

1420

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

1421

1422

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

1423

1424

```
<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>
```

1425

1426

1427

1428

1429

1430

1431

1432

1433

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

1434

1435

1436



```
1437     <wst:Status>
1438         <wst:Code>
1439             http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
1440         </wst:Code>
1441     </wst:Status>
1442     ...
1443 </wst:RequestSecurityTokenResponse>
```

1444 The following illustrates the syntax of a validation request and response. In this example a custom token  
1445 is requested indicating authorized rights in addition to the status.

```
1446 <wst:RequestSecurityToken xmlns:wst="...">
1447     <wst:TokenType>
1448         http://example.org/mySpecialToken
1449     </wst:TokenType>
1450     <wst:RequestType>
1451         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate
1452     </wst:RequestType>
1453 </wst:RequestSecurityToken>
```

```
1454
1455 <wst:RequestSecurityTokenResponse xmlns:wst="...">
1456     <wst:TokenType>
1457         http://example.org/mySpecialToken
1458     </wst:TokenType>
1459     <wst:Status>
1460         <wst:Code>
1461             http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
1462         </wst:Code>
1463     </wst:Status>
1464     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1465     ...
1466 </wst:RequestSecurityTokenResponse>
```

## 8 Negotiation and Challenge Extensions

1467

1468 The general security token service framework defined above allows for a simple request and response for  
1469 security tokens (possibly asynchronous). However, there are many scenarios where a set of exchanges  
1470 between the parties is REQUIRED prior to returning (e.g., issuing) a security token. This section  
1471 describes the extensions to the base WS-Trust mechanisms to enable exchanges for negotiation and  
1472 challenges.

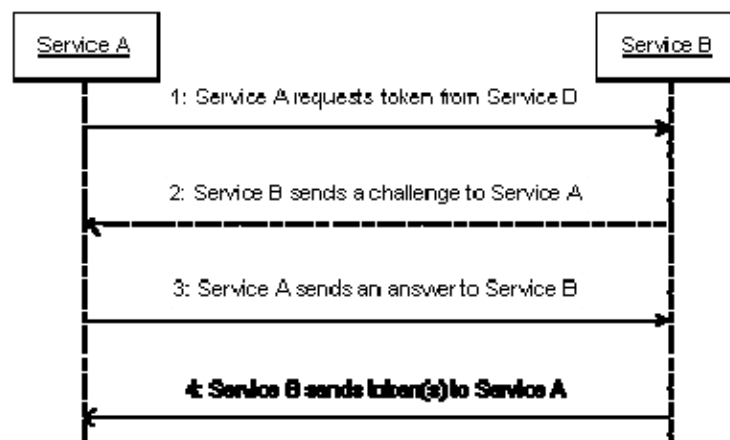
1473

1474 There are potentially different forms of exchanges, but one specific form, called "challenges", provides  
1475 mechanisms in addition to those described in [WS-Security] for authentication. This section describes  
1476 how general exchanges are issued and responded to within this framework. Other types of exchanges  
1477 include, but are not limited to, negotiation, tunneling of hardware-based processing, and tunneling of  
1478 legacy protocols.

1479

1480 The process is straightforward (illustrated here using a challenge):

1481



1482

- 1483 1. A requestor sends, for example, a `<wst:RequestSecurityToken>` message with a  
1484 timestamp.
- 1485 2. The recipient does not trust the timestamp and issues a  
1486 `<wst:RequestSecurityTokenResponse>` message with an embedded challenge.
- 1487 3. The requestor sends a `<wst:RequestSecurityTokenResponse>` message with an answer to  
1488 the challenge.
- 1489 4. The recipient issues a `<wst:RequestSecurityTokenResponseCollection>` message with  
1490 the issued security token and OPTIONAL proof-of-possession token.

1491

1492 It should be noted that the requestor might challenge the recipient in either step 1 or step 3. In which  
1493 case, step 2 or step 4 contains an answer to the initiator's challenge. Similarly, it is possible that steps 2  
1494 and 3 could iterate multiple times before the process completes (step 4).

1495

1496 The two services can use [WS-SecurityPolicy] to state their requirements and preferences for security  
1497 tokens and encryption and signing algorithms (general policy intersection). This section defines  
1498 mechanisms for legacy and more sophisticated types of negotiations.

## 1499 8.1 Negotiation and Challenge Framework

1500 The general mechanisms defined for requesting and returning security tokens are extensible. This  
1501 section describes the general model for extending these to support negotiations and challenges.

1502

1503 The exchange model is as follows:

- 1504 1. A request is initiated with a `<wst:RequestSecurityToken>` that identifies the details of the  
1505 request (and MAY contain initial negotiation/challenge information)
- 1506 2. A response is returned with a `<wst:RequestSecurityTokenResponse>` that contains  
1507 additional negotiation/challenge information. Optionally, this MAY return token information in the  
1508 form of a `<wst:RequestSecurityTokenResponseCollection>` (if the exchange is two legs  
1509 long).
- 1510 3. If the exchange is not complete, the requestor uses a  
1511 `<wst:RequestSecurityTokenResponse>` that contains additional negotiation/challenge  
1512 information.
- 1513 4. The process repeats at step 2 until the negotiation/challenge is complete (a token is returned or a  
1514 Fault occurs). In the case where token information is returned in the final leg, it is returned in the  
1515 form of a `<wst:RequestSecurityTokenResponseCollection>`.

1516

1517 The negotiation/challenge information is passed in binding/profile-specific elements that are placed inside  
1518 of the `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>` elements.

1519

1520 It is RECOMMENDED that at least the `<wsu:Timestamp>` element be included in messages (as per  
1521 [\[WS-Security\]](#)) as a way to ensure freshness of the messages in the exchange. Other types of  
1522 challenges MAY also be included. For example, a `<wsp:Policy>` element may be used to negotiate  
1523 desired policy behaviors of both parties. Multiple challenges and responses MAY be included.

## 1524 8.2 Signature Challenges

1525 Exchange requests are issued by including an element that describes the exchange (e.g. challenge) and  
1526 responses contain an element describing the response. For example, signature challenges are  
1527 processed using the `<wst:SignChallenge>` element. The response is returned in a  
1528 `<wst:SignChallengeResponse>` element. Both the challenge and the response elements are  
1529 specified within the `<wst:RequestSecurityTokenResponse>` element. Some forms of negotiation  
1530 MAY specify challenges along with responses to challenges from the other party. It should be noted that  
1531 the requestor MAY provide exchange information (e.g. a challenge) to the recipient in the initial request.  
1532 Consequently, these elements are also allowed within a `<wst:RequestSecurityToken>` element.

1533

1534 The syntax of these elements is as follows:

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

1538

```
1539 <wst:SignChallengeResponse xmlns:wst="...">  
1540   <wst:Challenge ...>...</wst:Challenge>  
1541 </wst:SignChallengeResponse>
```

1542

1543 The following describes the attributes and tags listed in the schema above:

1544 *.../wst:SignChallenge*

1545 This OPTIONAL element describes a challenge that requires the other party to sign a specified  
1546 set of information.

1547 *.../wst:SignChallenge/wst:Challenge*

1548 This REQUIRED string element describes the value to be signed. In order to prevent certain  
1549 types of attacks (such as man-in-the-middle), it is strongly RECOMMENDED that the challenge  
1550 be bound to the negotiation. For example, the challenge SHOULD track (such as using a digest  
1551 of) any relevant data exchanged such as policies, tokens, replay protection, etc. As well, if the  
1552 challenge is happening over a secured channel, a reference to the channel SHOULD also be  
1553 included. Furthermore, the recipient of a challenge SHOULD verify that the data tracked  
1554 (digested) matches their view of the data exchanged. The exact algorithm MAY be defined in  
1555 profiles or agreed to by the parties.

1556 *.../SignChallenge/{any}*

1557 This is an extensibility mechanism to allow additional negotiation types to be used.

1558 *.../wst:SignChallenge/@{any}*

1559 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added  
1560 to the element.

1561 *.../wst:SignChallengeResponse*

1562 This OPTIONAL element describes a response to a challenge that requires the signing of a  
1563 specified set of information.

1564 *.../wst:SignChallengeResponse/wst:Challenge*

1565 If a challenge was issued, the response MUST contain the challenge element exactly as  
1566 received. As well, while the RSTR response SHOULD always be signed, if a challenge was  
1567 issued, the RSTR MUST be signed (and the signature coupled with the message to prevent  
1568 replay).

1569 *.../wst:SignChallengeResponse/{any}*

1570 This is an extensibility mechanism to allow additional negotiation types to be used.

1571 *.../wst:SignChallengeResponse/@{any}*

1572 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added  
1573 to the element.

## 1574 **8.3 Binary Exchanges and Negotiations**

1575 Exchange requests MAY also utilize existing binary formats passed within the WS-Trust framework. A  
1576 generic mechanism is provided for this that includes a URI attribute to indicate the type of binary  
1577 exchange.

1578  
1579 The syntax of this element is as follows:

```
1580 <wst:BinaryExchange ValueType="..." EncodingType="..." xmlns:wst="...">  
1581 </wst:BinaryExchange>
```

1582 The following describes the attributes and tags listed in the schema above (note that the ellipses below  
1583 indicate that this element MAY be placed in different containers. For this specification, these are limited  
1584 to <wst:RequestSecurityToken> and <wst:RequestSecurityTokenResponse>):

1585 *.../wst:BinaryExchange*

1586 This OPTIONAL element is used for a security negotiation that involves exchanging binary blobs  
1587 as part of an existing negotiation protocol. The contents of this element are blob-type-specific  
1588 and are encoded using base64 (unless otherwise specified).

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

1590 This REQUIRED attribute specifies a URI to identify the type of negotiation (and the value space  
1591 of the blob – the element's contents).

1592 *.../wst:BinaryExchange/@EncodingType*

1593 This REQUIRED attribute specifies a URI to identify the encoding format (if different from base64)  
1594 of the negotiation blob. Refer to [WS-Security] for sample encoding format URIs.

1595 *.../wst:BinaryExchange/@{any}*

1596 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added  
1597 to the element.

1598 Some binary exchanges result in a shared state/context between the involved parties. It is  
1599 RECOMMENDED that at the conclusion of the exchange, a new token and proof-of-possession token be  
1600 returned. A common approach is to use the negotiated key as a "secure channel" mechanism to secure  
1601 the new token and proof-of-possession token.

1602 For example, an exchange might establish a shared secret *Sx* that can then be used to sign the final  
1603 response and encrypt the proof-of-possession token.

## 1604 8.4 Key Exchange Tokens

1605 In some cases it MAY be necessary to provide a key exchange token so that the other party (either  
1606 requestor or issuer) can provide entropy or key material as part of the exchange. Challenges MAY NOT  
1607 always provide a usable key as the signature may use a signing-only certificate.

1608

1609 The section describes two OPTIONAL elements that can be included in RST and RSTR elements to  
1610 indicate that a Key Exchange Token (KET) is desired, or to provide a KET.

1611 The syntax of these elements is as follows (Note that the ellipses below indicate that this element MAY be  
1612 placed in different containers. For this specification, these are limited to

1613 `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>`):

1614 

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

1615

1616 

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

1617

1618 The following describes the attributes and tags listed in the schema above:

1619 *.../wst:RequestKET*

1620 This OPTIONAL element is used to indicate that the receiving party (either the original requestor  
1621 or issuer) SHOULD provide a KET to the other party on the next leg of the exchange.

1622 *.../wst:KeyExchangeToken*

1623 This OPTIONAL element is used to provide a key exchange token. The contents of this element  
1624 either contain the security token to be used for key exchange or a reference to it.

## 1625 8.5 Custom Exchanges

1626 Using the extensibility model described in this specification, any custom XML-based exchange can be  
1627 defined in a separate binding/profile document. In such cases elements are defined which are carried in  
1628 the RST and RSTR elements.

1629

1630 It should be noted that it is NOT REQUIRED that exchange elements be symmetric. That is, a specific  
1631 exchange mechanism MAY use multiple elements at different times, depending on the state of the  
1632 exchange.

## 1633 8.6 Signature Challenge Example

1634 Here is an example exchange involving a signature challenge. In this example, a service requests a  
1635 custom token using a X.509 certificate for authentication. The issuer uses the exchange mechanism to  
1636 challenge the requestor to sign a random value (to ensure message freshness). The requestor provides  
1637 a signature of the requested data and, once validated, the issuer then issues the requested token.

1638

1639 The first message illustrates the initial request that is signed with the private key associated with the  
1640 requestor's X.509 certificate:

```
1641 <S11:Envelope xmlns:S11="..." xmlns:wss="..."
1642     xmlns:wsu="..." xmlns:wst="...">
1643   <S11:Header>
1644     ...
1645     <wss:Security>
1646       <wss:BinarySecurityToken
1647         wsu:Id="reqToken"
1648         ValueType="...X509v3">
1649         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...
1650       </wss:BinarySecurityToken>
1651       <ds:Signature xmlns:ds="...">
1652         ...
1653         <ds:KeyInfo>
1654           <wss:SecurityTokenReference>
1655             <wss:Reference URI="#reqToken"/>
1656           </wss:SecurityTokenReference>
1657         </ds:KeyInfo>
1658       </ds:Signature>
1659     </wss:Security>
1660     ...
1661   </S11:Header>
1662   <S11:Body>
1663     <wst:RequestSecurityToken>
1664       <wst:TokenType>
1665         http://example.org/mySpecialToken
1666       </wst:TokenType>
1667       <wst:RequestType>
1668         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1669       </wst:RequestType>
1670     </wst:RequestSecurityToken>
1671   </S11:Body>
1672 </S11:Envelope>
```

1673

1674 The issuer (recipient) service doesn't trust the sender's timestamp (or one wasn't specified) and issues a  
1675 challenge using the exchange framework defined in this specification. This message is signed using the  
1676 private key associated with the issuer's X.509 certificate and contains a random challenge that the  
1677 requestor must sign:

1678  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1700  
1701

```
<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>
```

1702  
1703  
1704  
1705

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:

1706  
1707  
1708  
1709  
1710  
1711  
1712  
1713  
1714  
1715  
1716  
1717  
1718  
1719  
1720  
1721  
1722  
1723  
1724  
1725  
1726  
1727  
1728  
1729

```
<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>
```

1730  
1731  
1732  
1733

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.

1734  
1735  
1736

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

```

1737     ...
1738     <wsse:Security>
1739         <wsse:BinarySecurityToken
1740             wsu:Id="issuerToken"
1741             ValueType="...X509v3">
1742                 DFJHuedsujfnrnv45JZc0...
1743         </wsse:BinarySecurityToken>
1744         <ds:Signature xmlns:ds="...">
1745             ...
1746         </ds:Signature>
1747     </wsse:Security>
1748     ...
1749 </S11:Header>
1750 <S11:Body>
1751     <wst:RequestSecurityTokenResponseCollection>
1752     <wst:RequestSecurityTokenResponse>
1753     <wst:RequestedSecurityToken>
1754     <xyz:CustomToken xmlns:xyz="...">
1755         ...
1756     </xyz:CustomToken>
1757 </wst:RequestedSecurityToken>
1758 <wst:RequestedProofToken>
1759 <xenc:EncryptedKey Id="newProof">
1760     ...
1761 </xenc:EncryptedKey>
1762 </wst:RequestedProofToken>
1763 </wst:RequestSecurityTokenResponse>
1764 </wst:RequestSecurityTokenResponseCollection>
1765 </S11:Body>
1766 </S11:Envelope>

```

## 1767 8.7 Custom Exchange Example

1768 Here is another illustrating the syntax for a token request using a custom XML exchange. For brevity,  
1769 only the RST and RSTR elements are illustrated. Note that the framework allows for an arbitrary number  
1770 of exchanges, although this example illustrates the use of four legs. The request uses a custom  
1771 exchange element and the requestor signs only the non-mutable element of the message:

```

1772     <wst:RequestSecurityToken xmlns:wst="...">
1773     <wst:TokenType>
1774         http://example.org/mySpecialToken
1775     </wst:TokenType>
1776     <wst:RequestType>
1777         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1778     </wst:RequestType>
1779     <xyz:CustomExchange xmlns:xyz="...">
1780         ...
1781     </xyz:CustomExchange>
1782 </wst:RequestSecurityToken>

```

1783  
1784 The issuer service (recipient) responds with another leg of the custom exchange and signs the response  
1785 (non-mutable aspects) with its token:

```

1786     <wst:RequestSecurityTokenResponse xmlns:wst="...">
1787     <xyz:CustomExchange xmlns:xyz="...">
1788         ...
1789     </xyz:CustomExchange>
1790 </wst:RequestSecurityTokenResponse>

```

1791



1792 The requestor receives the issuer's exchange and issues a response that is signed using the requestor's  
 1793 token and continues the custom exchange. The signature covers all non-mutable aspects of the  
 1794 message to prevent certain types of security attacks:

```
1795 <wst:RequestSecurityTokenResponse xmlns:wst="...">
1796   <xyz:CustomExchange xmlns:xyz="...">
1797     ...
1798   </xyz:CustomExchange>
1799 </wst:RequestSecurityTokenResponse>
```

1800  
 1801 The issuer processes the exchange and determines that the exchange is complete and that a token  
 1802 should be issued. Consequently it issues the requested token(s) and the associated proof-of-possession  
 1803 token. The proof-of-possession token is encrypted for the requestor using the requestor's public key.

```
1804 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
1805   <wst:RequestSecurityTokenResponse>
1806     <wst:RequestedSecurityToken>
1807       <xyz:CustomToken xmlns:xyz="...">
1808         ...
1809       </xyz:CustomToken>
1810     </wst:RequestedSecurityToken>
1811     <wst:RequestedProofToken>
1812       <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
1813         ...
1814       </xenc:EncryptedKey>
1815     </wst:RequestedProofToken>
1816     <wst:RequestedProofToken>
1817       <xenc:EncryptedKey xmlns:xenc="...">...</xenc:EncryptedKey>
1818     </wst:RequestedProofToken>
1819   </wst:RequestSecurityTokenResponse>
1820 </wst:RequestSecurityTokenResponseCollection>
```

1821 It should be noted that other example exchanges include the issuer returning a final custom exchange  
 1822 element, and another example where a token isn't returned.

## 1823 8.8 Protecting Exchanges

1824 There are some attacks, such as forms of man-in-the-middle, that can be applied to token requests  
 1825 involving exchanges. It is RECOMMENDED that the exchange sequence be protected. This MAY be  
 1826 built into the exchange messages, but if metadata is provided in the RST or RSTR elements, then it is  
 1827 subject to attack.

1828  
 1829 Consequently, it is RECOMMENDED that keys derived from exchanges be linked cryptographically to the  
 1830 exchange. For example, a hash can be computed by computing the SHA1 of the exclusive  
 1831 canonicalization [XML-C14N] of all RST and RSTR elements in messages exchanged. This value can  
 1832 then be combined with the exchanged secret(s) to create a new master secret that is bound to the data  
 1833 both parties sent/received.

1834  
 1835 To this end, the following computed key algorithm is defined to be OPTIONALLY used in these scenarios:

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

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

## 1836 8.9 Authenticating Exchanges

1837 After an exchange both parties have a shared knowledge of a key (or keys) that can then be used to  
 1838 secure messages. However, in some cases it may be desired to have the issuer prove to the requestor  
 1839 that it knows the key (and that the returned metadata is valid) prior to the requestor using the data.  
 1840 However, until the exchange is actually completed it MAY be (and is often) inappropriate to use the  
 1841 computed keys. As well, using a token that hasn't been returned to secure a message may complicate  
 1842 processing since it crosses the boundary of the exchange and the underlying message security. This  
 1843 means that it MAY NOT be appropriate to sign the final leg of the exchange using the key derived from  
 1844 the exchange.

1845  
 1846 For this reason an authenticator is defined that provides a way for the issuer to verify the hash as part of  
 1847 the token issuance. Specifically, when an authenticator is returned, the  
 1848 `<wst:RequestSecurityTokenResponseCollection>` element is returned. This contains one  
 1849 RSTR with the token being returned as a result of the exchange and a second RSTR that contains the  
 1850 authenticator (this order SHOULD be used). When an authenticator is used, RSTRs MUST use the  
 1851 `@Context` element so that the authenticator can be correlated to the token issuance. The authenticator is  
 1852 separated from the RSTR because otherwise computation of the RST/RSTR hash becomes more  
 1853 complex. The authenticator is represented using the `<wst:Authenticator>` element as illustrated  
 1854 below:

```

1855 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
1856   <wst:RequestSecurityTokenResponse Context="...">
1857     ...
1858   </wst:RequestSecurityTokenResponse>
1859   <wst:RequestSecurityTokenResponse Context="...">
1860     <wst:Authenticator>
1861       <wst:CombinedHash>...</wst:CombinedHash>
1862       ...
1863     </wst:Authenticator>
1864   </wst:RequestSecurityTokenResponse>
1865 </wst:RequestSecurityTokenResponseCollection>
  
```

1866  
 1867 The following describes the attributes and elements listed in the schema overview above (the ... notation  
 1868 below represents the path RSTRC/RSTR and is used for brevity):

1869 *.../wst:Authenticator*

1870 This OPTIONAL element provides verification (authentication) of a computed hash.

1871 *.../wst:Authenticator/wst:CombinedHash*

1872 This OPTIONAL element proves the hash and knowledge of the computed key. This is done by  
 1873 providing the base64 encoding of the first 256 bits of the P\_SHA1 digest of the computed key and the  
 1874 concatenation of the hash determined for the computed key and the string "AUTH-HASH".  
 1875 Specifically, P\_SHA1(*computed-key*, H + "AUTH-HASH")<sub>0-255</sub>. The octets for the "AUTH-HASH"  
 1876 string are the UTF-8 octets.

1877

1878 This `<wst:CombinedHash>` element is OPTIONAL (and an open content model is used) to allow for  
1879 different authenticators in the future.

---

## 9 Key and Token Parameter Extensions

1880

1881 This section outlines additional parameters that can be specified in token requests and responses.  
1882 Typically they are used with issuance requests, but since all types of requests MAY issue security tokens  
1883 they could apply to other bindings.

### 9.1 On-Behalf-Of Parameters

1884

1885 In some scenarios the requestor is obtaining a token on behalf of another party. These parameters  
1886 specify the issuer and original requestor of the token being used as the basis of the request. The syntax  
1887 is as follows (note that the base elements described above are included here italicized for completeness):

```
1888 <wst:RequestSecurityToken xmlns:wst="...">  
1889   <wst:TokenType>...</wst:TokenType>  
1890   <wst:RequestType>...</wst:RequestType>  
1891   ...  
1892   <wst:OnBehalfOf>...</wst:OnBehalfOf>  
1893   <wst:Issuer>...</wst:Issuer>  
1894 </wst:RequestSecurityToken>
```

1895

1896 The following describes the attributes and elements listed in the schema overview above:

1897 */wst:RequestSecurityToken/wst:OnBehalfOf*

1898 This OPTIONAL element indicates that the requestor is making the request on behalf of another.  
1899 The identity on whose behalf the request is being made is specified by placing a security token,  
1900 <wsse:SecurityTokenReference> element, or <wsa:EndpointReference> element  
1901 within the <wst:OnBehalfOf> element. The requestor MAY provide proof of possession of the  
1902 key associated with the OnBehalfOf identity by including a signature in the RST security header  
1903 generated using the OnBehalfOf token that signs the primary signature of the RST (i.e. endorsing  
1904 supporting token concept from WS-SecurityPolicy). Additional signed supporting tokens  
1905 describing the OnBehalfOf context MAY also be included within the RST security header.

1906 */wst:RequestSecurityToken/wst:Issuer*

1907 This OPTIONAL element specifies the issuer of the security token that is presented in the  
1908 message. This element's type is an endpoint reference as defined in [\[WS-Addressing\]](#).

1909

1910 In the following illustrates the syntax for a proxy that is requesting a security token on behalf of another  
1911 requestor or end-user.

```
1912 <wst:RequestSecurityToken xmlns:wst="...">  
1913   <wst:TokenType>...</wst:TokenType>  
1914   <wst:RequestType>...</wst:RequestType>  
1915   ...  
1916   <wst:OnBehalfOf>endpoint-reference</wst:OnBehalfOf>  
1917 </wst:RequestSecurityToken>
```

### 9.2 Key and Encryption Requirements

1918

1919 This section defines extensions to the <wst:RequestSecurityToken> element for requesting specific  
1920 types of keys or algorithms or key and algorithms as specified by a given policy in the return token(s). In  
1921 some cases the service may support a variety of key types, sizes, and algorithms. These parameters  
1922 allow a requestor to indicate its desired values. It should be noted that the issuer's policy indicates if input

1923 values must be adhered to and faults generated for invalid inputs, or if the issuer will provide alternative  
1924 values in the response.

1925

1926 Although illustrated using the `<wst:RequestSecurityToken>` element, these options can also be  
1927 returned in a `<wst:RequestSecurityTokenResponse>` element.

1928 The syntax for these OPTIONAL elements is as follows (note that the base elements described above are  
1929 included here italicized for completeness):

1930

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

1931

1932

1933

1934

1935

1936

1937

1938

1939

1940

1941

1942

1943

1944

1945

1946

1947

1948

1949 The following describes the attributes and elements listed in the schema overview above:

1950 */wst:RequestSecurityToken/wst:AuthenticationType*

1951 This OPTIONAL URI element indicates the type of authentication desired, specified as a URI.

1952 This specification does not predefine classifications; these are specific to token services as is the  
1953 relative strength evaluations. The relative assessment of strength is up to the recipient to  
1954 determine. That is, requestors SHOULD be familiar with the recipient policies. For example, this  
1955 might be used to indicate which of the four U.S. government authentication levels is REQUIRED.

1956 */wst:RequestSecurityToken/wst:KeyType*

1957 This OPTIONAL URI element indicates the type of key desired in the security token. The  
1958 predefined values are identified in the table below. Note that some security token formats have  
1959 fixed key types. It should be noted that new algorithms can be inserted by defining URIs in other  
1960 specifications and profiles.

URI	Meaning
<a href="http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey">http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey</a>	A public key token is requested
<a href="http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey">http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey</a>	A symmetric key token is requested (default)
<a href="http://docs.oasis-open.org/ws-sx/wstrust/200512/Bearer">http://docs.oasis-open.org/ws-sx/wstrust/200512/Bearer</a>	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.

1961 */wst:RequestSecurityToken/wst:KeySize*

- 1962 This OPTIONAL integer element indicates the size of the key REQUIRED specified in number of  
 1963 bits. This is a request, and, as such, the requested security token is not obligated to use the  
 1964 requested key size. That said, the recipient SHOULD try to use a key at least as strong as the  
 1965 specified value if possible. The information is provided as an indication of the desired strength of  
 1966 the security.
- 1967 */wst:RequestSecurityToken/wst:SignatureAlgorithm*
- 1968 This OPTIONAL URI element indicates the desired signature algorithm used within the returned  
 1969 token. This is specified as a URI indicating the algorithm (see [XML-Signature] for typical signing  
 1970 algorithms).
- 1971 */wst:RequestSecurityToken/wst:EncryptionAlgorithm*
- 1972 This OPTIONAL URI element indicates the desired encryption algorithm used within the returned  
 1973 token. This is specified as a URI indicating the algorithm (see [XML-Encrypt] for typical  
 1974 encryption algorithms).
- 1975 */wst:RequestSecurityToken/wst:CanonicalizationAlgorithm*
- 1976 This OPTIONAL URI element indicates the desired canonicalization method used within the  
 1977 returned token. This is specified as a URI indicating the method (see [XML-Signature] for typical  
 1978 canonicalization methods).
- 1979 */wst:RequestSecurityToken/wst:ComputedKeyAlgorithm*
- 1980 This OPTIONAL URI element indicates the desired algorithm to use when computed keys are  
 1981 used for issued tokens.
- 1982 */wst:RequestSecurityToken/wst:Encryption*
- 1983 This OPTIONAL element indicates that the requestor desires any returned secrets in issued  
 1984 security tokens to be encrypted for the specified token. That is, so that the owner of the specified  
 1985 token can decrypt the secret. Normally the security token is the contents of this element but a  
 1986 security token reference MAY be used instead. If this element isn't specified, the token used as  
 1987 the basis of the request (or specialized knowledge) is used to determine how to encrypt the key.
- 1988 */wst:RequestSecurityToken/wst:ProofEncryption*
- 1989 This OPTIONAL element indicates that the requestor desires any returned secrets in proof-of-  
 1990 possession tokens to be encrypted for the specified token. That is, so that the owner of the  
 1991 specified token can decrypt the secret. Normally the security token is the contents of this element  
 1992 but a security token reference MAY be used instead. If this element isn't specified, the token  
 1993 used as the basis of the request (or specialized knowledge) is used to determine how to encrypt  
 1994 the key.
- 1995 */wst:RequestSecurityToken/wst:KeyWrapAlgorithm*
- 1996 This OPTIONAL URI element indicates the desired algorithm to use for key wrapping when STS  
 1997 encrypts the issued token for the relying party using an asymmetric key.
- 1998 */wst:RequestSecurityToken/wst:UseKey*
- 1999 If the requestor wishes to use an existing key rather than create a new one, then this OPTIONAL  
 2000 element can be used to reference the security token containing the desired key. This element  
 2001 either contains a security token or a `<wsse:SecurityTokenReference>` element that  
 2002 references the security token containing the key that SHOULD be used in the returned token. If  
 2003 `<wst:KeyType>` is not defined and a key type is not implicitly known to the service, it MAY be  
 2004 determined from the token (if possible). Otherwise this parameter is meaningless and is ignored.  
 2005 Requestors SHOULD demonstrate authorized use of the public key provided.
- 2006 */wst:RequestSecurityToken/wst:UseKey/@Sig*
- 2007 In order to *authenticate* the key referenced, a signature MAY be used to prove the referenced  
 2008 token/key. If specified, this OPTIONAL attribute indicates the ID of the corresponding signature

2009 (by URI reference). When this attribute is present, a key need not be specified inside the element  
2010 since the referenced signature will indicate the corresponding token (and key).

2011 */wst:RequestSecurityToken/wst:SignWith*

2012 This OPTIONAL URI element indicates the desired signature algorithm to be used with the issued  
2013 security token (typically from the policy of the target site for which the token is being requested.  
2014 While any of these OPTIONAL elements MAY be included in RSTRs, this one is a likely  
2015 candidate if there is some doubt (e.g., an X.509 cert that can only use DSS).

2016 */wst:RequestSecurityToken/wst:EncryptWith*

2017 This OPTIONAL URI element indicates the desired encryption algorithm to be used with the  
2018 issued security token (typically from the policy of the target site for which the token is being  
2019 requested.) While any of these OPTIONAL elements MAY be included in RSTRs, this one is a  
2020 likely candidate if there is some doubt.

2021 The following summarizes the various algorithm parameters defined above. T is the issued token, P is the  
2022 proof key.  
2023

2024 **SignatureAlgorithm** - The signature algorithm to use to sign T

2025 **EncryptionAlgorithm** - The encryption algorithm to use to encrypt T

2026 **CanonicalizationAlgorithm** - The canonicalization algorithm to use when signing T

2027 **ComputedKeyAlgorithm** - The key derivation algorithm to use if using a symmetric key for P  
2028 where P is computed using client, server, or combined entropy

2029 **Encryption** - The token/key to use when encrypting T

2030 **ProofEncryption** - The token/key to use when encrypting P

2031 **UseKey** - This is P. This is generally used when the client supplies a public-key that it wishes to  
2032 be embedded in T as the proof key

2033 **SignWith** - The signature algorithm the client intends to employ when using P to  
2034 sign

2035 The encryption algorithms further differ based on whether the issued token contains asymmetric key or  
2036 symmetric key. Furthermore, they differ based on what type of key is used to protect the issued token  
2037 from the STS to the relying party. The following cases can occur:

2038 T contains symmetric key/STS uses symmetric key to encrypt T for RP

2039 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP  
2040 when using the proof key (e.g. AES256)

2041 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to  
2042 encrypt the T (e.g. AES256)

2043

2044 T contains symmetric key/STS uses asymmetric key to encrypt T for RP

2045 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP  
2046 when using the proof key (e.g. AES256)

2047 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to  
2048 encrypt T for RP (e.g. AES256)

2049 **KeyWrapAlgorithm** – used to indicate the KeyWrap algorithm that the STS SHOULD use to  
2050 wrap the generated key that is used to encrypt the T for RP

2051

2052 T contains asymmetric key/STS uses symmetric key to encrypt T for RP

2053 **EncryptWith** – used to indicate the KeyWrap algorithm that the client will use to

2054 protect the symmetric key that is used to protect messages to RP when using the proof key (e.g.  
2055 RSA-OAEP-MGF1P)

2056 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to  
2057 encrypt T for RP (e.g. AES256)

2058

2059 T contains asymmetric key/STS uses asymmetric key to encrypt T for RP

2060 **EncryptWith** - used to indicate the KeyWrap algorithm that the client will use to  
2061 protect symmetric key that is used to protect message to RP when using the proof  
2062 key (e.g. RSA-OAEP-MGF1P)

2063 **EncryptionAlgorithm** - used to indicate the symmetric algorithm that the STS SHOULD use to  
2064 encrypt T for RP (e.g. AES256)

2065 **KeyWrapAlgorithm** – used to indicate the KeyWrap algorithm that the STS SHOULD use to  
2066 wrap the generated key that is used to encrypt the T for RP

2067

2068 The example below illustrates a request that utilizes several of these parameters. A request is made for a  
2069 custom token using a username and password as the basis of the request. For security, this token is  
2070 encrypted (see "encUsername") for the recipient using the recipient's public key and referenced in the  
2071 encryption manifest. The message is protected by a signature using a public key from the sender and  
2072 authorized by the username and password.

2073

2074 The requestor would like the custom token to contain a 1024-bit public key whose value can be found in  
2075 the key provided with the "proofSignature" signature (the key identified by "requestProofToken"). The  
2076 token should be signed using RSA-SHA1 and encrypted for the token identified by  
2077 "requestEncryptionToken". The proof should be encrypted using the token identified by  
2078 "requestProofToken".

```
2079 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."  
2080     xmlns:wst="..." xmlns:ds="..." xmlns:xenc="...">  
2081   <S11:Header>  
2082     ...  
2083     <wsse:Security>  
2084       <xenc:ReferenceList>...</xenc:ReferenceList>  
2085       <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>  
2086       <wsse:BinarySecurityToken wsu:Id="requestEncryptionToken"  
2087         ValueType="...SomeTokenType" xmlns:x="...">  
2088         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
2089       </wsse:BinarySecurityToken>  
2090       <wsse:BinarySecurityToken wsu:Id="requestProofToken"  
2091         ValueType="...SomeTokenType" xmlns:x="...">  
2092         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
2093       </wsse:BinarySecurityToken>  
2094       <ds:Signature Id="proofSignature">  
2095         ... signature proving requested key ...  
2096         ... key info points to the "requestedProofToken" token ...  
2097       </ds:Signature>  
2098     </wsse:Security>  
2099     ...  
2100   </S11:Header>  
2101   <S11:Body wsu:Id="req">  
2102     <wst:RequestSecurityToken>  
2103       <wst:TokenType>  
2104         http://example.org/mySpecialToken  
2105       </wst:TokenType>  
2106     <wst:RequestType>
```



```

2107         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
2108     </wst:RequestType>
2109     <wst:KeyType>
2110         http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey
2111     </wst:KeyType>
2112     <wst:KeySize>1024</wst:KeySize>
2113     <wst:SignatureAlgorithm>
2114         http://www.w3.org/2000/09/xmldsig#rsa-sha1
2115     </wst:SignatureAlgorithm>
2116     <wst:Encryption>
2117         <Reference URI="#requestEncryptionToken"/>
2118     </wst:Encryption>
2119     <wst:ProofEncryption>
2120         <wsse:Reference URI="#requestProofToken"/>
2121     </wst:ProofEncryption>
2122     <wst:UseKey Sig="#proofSignature"/>
2123     </wst:RequestSecurityToken>
2124 </S11:Body>
2125 </S11:Envelope>

```

### 2126 9.3 Delegation and Forwarding Requirements

2127 This section defines extensions to the `<wst:RequestSecurityToken>` element for indicating  
2128 delegation and forwarding requirements on the requested security token(s).

2129 The syntax for these extension elements is as follows (note that the base elements described above are  
2130 included here italicized for completeness):

```

2131     <wst:RequestSecurityToken xmlns:wst="...">
2132         <wst:TokenType>...</wst:TokenType>
2133         <wst:RequestType>...</wst:RequestType>
2134         ...
2135         <wst:DelegateTo>...</wst:DelegateTo>
2136         <wst:Forwardable>...</wst:Forwardable>
2137         <wst:Delegatable>...</wst:Delegatable>
2138     </wst:RequestSecurityToken>

```

2139 */wst:RequestSecurityToken/wst:DelegateTo*

2140 This OPTIONAL element indicates that the requested or issued token be delegated to another  
2141 identity. The identity receiving the delegation is specified by placing a security token or  
2142 `<wsse:SecurityTokenReference>` element within the `<wst:DelegateTo>` element.

2143 */wst:RequestSecurityToken/wst:Forwardable*

2144 This OPTIONAL element, of type `xs:boolean`, specifies whether the requested security token  
2145 SHOULD be marked as "Forwardable". In general, this flag is used when a token is normally  
2146 bound to the requestor's machine or service. Using this flag, the returned token MAY be used  
2147 from any source machine so long as the key is correctly proven. The default value of this flag is  
2148 true.

2149 */wst:RequestSecurityToken/wst:Delegatable*

2150 This OPTIONAL element, of type `xs:boolean`, specifies whether the requested security token  
2151 SHOULD be marked as "Delegatable". Using this flag, the returned token MAY be delegated to  
2152 another party. This parameter SHOULD be used in conjunction with `<wst:DelegateTo>`. The  
2153 default value of this flag is false.

2154  
2155 The following illustrates the syntax of a request for a custom token that can be delegated to the indicated  
2156 recipient (specified in the binary security token) and used in the specified interval.

```

2157     <wst:RequestSecurityToken xmlns:wst="...">

```

```

2158     <wst:TokenType>
2159         http://example.org/mySpecialToken
2160     </wst:TokenType>
2161     <wst:RequestType>
2162         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
2163     </wst:RequestType>
2164     <wst:DelegateTo>
2165         <wsse:BinarySecurityToken
2166 xmlns:wsse="...">...</wsse:BinarySecurityToken>
2167     </wst:DelegateTo>
2168     <wst:Delegatable>true</wst:Delegatable>
2169 </wst:RequestSecurityToken>

```

## 2170 9.4 Policies

2171 This section defines extensions to the `<wst:RequestSecurityToken>` element for passing policies.

2172

2173 The syntax for these extension elements is as follows (note that the base elements described above are  
2174 included here italicized for completeness):

```

2175     <wst:RequestSecurityToken xmlns:wst="...">
2176         <wst:TokenType>...</wst:TokenType>
2177         <wst:RequestType>...</wst:RequestType>
2178         ...
2179         <wsp:Policy xmlns:wsp="...">...</wsp:Policy>
2180         <wsp:PolicyReference xmlns:wsp="...">...</wsp:PolicyReference>
2181     </wst:RequestSecurityToken>

```

2182

2183 The following describes the attributes and elements listed in the schema overview above:

2184 */wst:RequestSecurityToken/wsp:Policy*

2185 This OPTIONAL element specifies a policy (as defined in [WS-Policy]) that indicates desired  
2186 settings for the requested token. The policy specifies defaults that can be overridden by the  
2187 elements defined in the previous sections.

2188 */wst:RequestSecurityToken/wsp:PolicyReference*

2189 This OPTIONAL element specifies a reference to a policy (as defined in [WS-Policy]) that  
2190 indicates desired settings for the requested token. The policy specifies defaults that can be  
2191 overridden by the elements defined in the previous sections.

2192

2193 The following illustrates the syntax of a request for a custom token that provides a set of policy  
2194 statements about the token or its usage requirements.

```

2195     <wst:RequestSecurityToken xmlns:wst="...">
2196         <wst:TokenType>
2197             http://example.org/mySpecialToken
2198         </wst:TokenType>
2199         <wst:RequestType>
2200             http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
2201         </wst:RequestType>
2202         <wsp:Policy xmlns:wsp="...">
2203             ...
2204         </wsp:Policy>
2205     </wst:RequestSecurityToken>

```

## 2206 9.5 Authorized Token Participants

2207 This section defines extensions to the `<wst:RequestSecurityToken>` element for passing information  
2208 about which parties are authorized to participate in the use of the token. This parameter is typically used  
2209 when there are additional parties using the token or if the requestor needs to clarify the actual parties  
2210 involved (for some profile-specific reason).

2211 It should be noted that additional participants will need to prove their identity to recipients in addition to  
2212 proving their authorization to use the returned token. This typically takes the form of a second signature  
2213 or use of transport security.

2214

2215 The syntax for these extension elements is as follows (note that the base elements described above are  
2216 included here italicized for completeness):

```
2217 <wst:RequestSecurityToken xmlns:wst="...">  
2218   <wst:TokenType>...</wst:TokenType>  
2219   <wst:RequestType>...</wst:RequestType>  
2220   ...  
2221   <wst:Participants>  
2222     <wst:Primary>...</wst:Primary>  
2223     <wst:Participant>...</wst:Participant>  
2224   </wst:Participants>  
2225 </wst:RequestSecurityToken>
```

2226

2227 The following describes elements and attributes used in a `<wsc:SecurityContextToken>` element.

2228 */wst:RequestSecurityToken/wst:Participants/*

2229 This OPTIONAL element specifies the participants sharing the security token. Arbitrary types  
2230 MAY be used to specify participants, but a typical case is a security token or an endpoint  
2231 reference (see [WS-Addressing]).

2232 */wst:RequestSecurityToken/wst:Participants/wst:Primary*

2233 This OPTIONAL element specifies the primary user of the token (if one exists).

2234 */wst:RequestSecurityToken/wst:Participants/wst:Participant*

2235 This OPTIONAL element specifies participant (or multiple participants by repeating the element)  
2236 that play a (profile-dependent) role in the use of the token or who are allowed to use the token.

2237 */wst:RequestSecurityToken/wst:Participants/{any}*

2238 This is an extensibility option to allow other types of participants and profile-specific elements to  
2239 be specified.

---

## 2240 10 Key Exchange Token Binding

2241 Using the token request framework, this section defines a binding for requesting a key exchange token  
2242 (KET). That is, if a requestor desires a token that can be used to encrypt key material for a recipient.

2243  
2244 For this binding, the following actions are defined to enable specific processing context to be conveyed to  
2245 the recipient:

```
2246 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET  
2247 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KET  
2248 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal
```

2249  
2250 For this binding, the `RequestType` element contains the following URI:

```
2251 http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET
```

2252  
2253 For this binding very few parameters are specified as input. OPTIONALLY the `<wst:TokenType>`  
2254 element can be specified in the request can indicate desired type response token carrying the key for key  
2255 exchange; however, this isn't commonly used.

2256 The applicability scope (e.g. `<wsp:AppliesTo>`) MAY be specified if the requestor desires a key  
2257 exchange token for a specific scope.

2258  
2259 It is RECOMMENDED that the response carrying the key exchange token be secured (e.g., signed by the  
2260 issuer or someone who can speak on behalf of the target for which the KET applies).

2261  
2262 Care should be taken when using this binding to prevent possible man-in-the-middle and substitution  
2263 attacks. For example, responses to this request SHOULD be secured using a token that can speak for  
2264 the desired endpoint.

2265  
2266 The RSTR for this binding carries the `<RequestedSecurityToken>` element even if a token is returned  
2267 (note that the base elements described above are included here italicized for completeness):

```
2268 <wst:RequestSecurityToken xmlns:wst="...">  
2269   <wst:TokenType>...</wst:TokenType>  
2270   <wst:RequestType>...</wst:RequestType>  
2271   ...  
2272 </wst:RequestSecurityToken>
```

```
2273  
2274 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2275   <wst:RequestSecurityTokenResponse>  
2276     <wst:TokenType>...</wst:TokenType>  
2277     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>  
2278     ...  
2279   </wst:RequestSecurityTokenResponse>  
2280 </wst:RequestSecurityTokenResponseCollection>
```

2281  
2282 The following illustrates the syntax for requesting a key exchange token. In this example, the KET is  
2283 returned encrypted for the requestor since it had the credentials available to do that. Alternatively the

2284 request could be made using transport security (e.g. TLS) and the key could be returned directly using  
2285 <wst:BinarySecret>.

```
2286 <wst:RequestSecurityToken xmlns:wst="...">  
2287   <wst:RequestType>  
2288     http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET  
2289   </wst:RequestType>  
2290 </wst:RequestSecurityToken>
```

2291

```
2292 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2293   <wst:RequestSecurityTokenResponse>  
2294     <wst:RequestedSecurityToken>  
2295       <xenc:EncryptedKey xmlns:xenc="...">...</xenc:EncryptedKey>  
2296     </wst:RequestedSecurityToken>  
2297   </wst:RequestSecurityTokenResponse>  
2298 </wst:RequestSecurityTokenResponseCollection>
```

2299

## 11 Error Handling

2300 There are many circumstances where an *error* can occur while processing security information. Errors  
2301 use the SOAP Fault mechanism. Note that the reason text provided below is RECOMMENDED, but  
2302 alternative text MAY be provided if more descriptive or preferred by the implementation. The tables  
2303 below are defined in terms of SOAP 1.1. For SOAP 1.2, the Fault/Code/Value is env:Sender (as defined  
2304 in SOAP 1.2) and the Fault/Code/Subcode/Value is the *faultcode* below and the Fault/Reason/Text is the  
2305 *faultstring* below. It should be noted that profiles MAY provide second-level detail fields, but they should  
2306 be careful not to introduce security vulnerabilities when doing so (e.g., by providing too detailed  
2307 information).

<b>Error that occurred (faultstring)</b>	<b>Fault code (faultcode)</b>
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 <a href="#">RequestSecurityToken</a> 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

2308

---

## 12 Security Considerations

2309 As stated in the Goals section of this document, this specification is meant to provide extensible  
2310 framework and flexible syntax, with which one could implement various security mechanisms. This  
2311 framework and syntax by itself does not provide any guarantee of security. When implementing and using  
2312 this framework and syntax, one must make every effort to ensure that the result is not vulnerable to any  
2313 one of a wide range of attacks.

2314

2315 It is not feasible to provide a comprehensive list of security considerations for such an extensible set of  
2316 mechanisms. A complete security analysis must be conducted on specific solutions based on this  
2317 specification. Below we illustrate some of the security concerns that often come up with protocols of this  
2318 type, but we stress that this *is not an exhaustive list of concerns*.

2319

2320 The following statements about signatures and signing apply to messages sent on unsecured channels.

2321

2322 It is critical that all the security-sensitive message elements must be included in the scope of the  
2323 message signature. As well, the signatures for conversation authentication must include a timestamp,  
2324 nonce, or sequence number depending on the degree of replay prevention required as described in [[WS-  
2325 Security](#)] and the UsernameToken Profile. Also, conversation establishment should include the policy so  
2326 that supported algorithms and algorithm priorities can be validated.

2327

2328 It is required that security token issuance messages be signed to prevent tampering. If a public key is  
2329 provided, the request should be signed by the corresponding private key to prove ownership. As well,  
2330 additional steps should be taken to eliminate replay attacks (refer to [[WS-Security](#)] for additional  
2331 information). Similarly, all token references should be signed to prevent any tampering.

2332

2333 Security token requests are susceptible to denial-of-service attacks. Care should be taken to mitigate  
2334 such attacks as is warranted by the service.

2335

2336 For security, tokens containing a symmetric key or a password should only be sent to parties who have a  
2337 need to know that key or password.

2338

2339 For privacy, tokens containing personal information (either in the claims, or indirectly by identifying who is  
2340 currently communicating with whom) should only be sent according to the privacy policies governing  
2341 these data at the respective organizations.

2342

2343 For some forms of multi-message exchanges, the exchanges are susceptible to attacks whereby  
2344 signatures are altered. To address this, it is suggested that a signature confirmation mechanism be used.  
2345 In such cases, each leg should include the confirmation of the previous leg. That is, leg 2 includes  
2346 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  
2347 correctness of the message outside of the message body.

2348

2349 There are many other security concerns that one may need to consider in security protocols. The list  
2350 above should not be used as a "check list" instead of a comprehensive security analysis.

2351

2352 It should be noted that use of unsolicited RSTRs implies that the recipient is prepared to accept such  
2353 issuances. Recipients should ensure that such issuances are properly authorized and recognize their  
2354 use could be used in denial-of-service attacks.

2355 In addition to the consideration identified here, readers should also review the security considerations in  
2356 [\[WS-Security\]](#).

2357

2358 Both token cancellation bindings defined in this specification require that the STS MUST NOT validate or  
2359 renew the token after it has been successfully canceled. The STS must take care to ensure that the token  
2360 is properly invalidated before confirming the cancel request or sending the cancel notification to the client.  
2361 This can be more difficult if the token validation or renewal logic is physically separated from the issuance  
2362 and cancellation logic. It is out of scope of this spec how the STS propagates the token cancellation to its  
2363 other components. If STS cannot ensure that the token was properly invalidated it MUST NOT send the  
2364 cancel notification or confirm the cancel request to the client.



---

## 2365 A. Key Exchange

2366 Key exchange is an integral part of token acquisition. There are several mechanisms by which keys are  
2367 exchanged using [WS-Security] and WS-Trust. This section highlights and summarizes these  
2368 mechanisms. Other specifications and profiles MAY provide additional details on key exchange.

2369  
2370 Care must be taken when employing a key exchange to ensure that the mechanism does not provide an  
2371 attacker with a means of discovering information that could only be discovered through use of secret  
2372 information (such as a private key).

2373  
2374 It is therefore important that a shared secret should only be considered as trustworthy as its source. A  
2375 shared secret communicated by means of the direct encryption scheme described in section I.1 is  
2376 acceptable if the encryption key is provided by a completely trustworthy key distribution center (this is the  
2377 case in the Kerberos model). Such a key would not be acceptable for the purposes of decrypting  
2378 information from the source that provided it since an attacker might replay information from a prior  
2379 transaction in the hope of learning information about it.

2380  
2381 In most cases the other party in a transaction is only imperfectly trustworthy. In these cases both parties  
2382 SHOULD contribute entropy to the key exchange by means of the <wst:entropy> element.

### 2383 A.1 Ephemeral Encryption Keys

2384 The simplest form of key exchange can be found in [WS-Security] for encrypting message data. As  
2385 described in [WS-Security] and [XML-Encrypt], when data is encrypted, a temporary key can be used to  
2386 perform the encryption which is, itself, then encrypted using the <xenc:EncryptedKey> element.

2387  
2388 The illustrates the syntax for encrypting a temporary key using the public key in an issuer name and serial  
2389 number:

```
2390 <xenc:EncryptedKey xmlns:xenc="...">  
2391   ...  
2392   <ds:KeyInfo xmlns:ds="...">  
2393     <wsse:SecurityTokenReference xmlns:wsse="...">  
2394       <ds:X509IssuerSerial>  
2395         <ds:X509IssuerName>  
2396           DC=ACMECorp, DC=com  
2397         </ds:X509IssuerName>  
2398         <ds:X509SerialNumber>12345678</ds:X509SerialNumber>  
2399       </ds:X509IssuerSerial>  
2400     </wsse:SecurityTokenReference>  
2401   </ds:KeyInfo>  
2402   ...  
2403 </xenc:EncryptedKey>
```

### 2404 A.2 Requestor-Provided Keys

2405 When a request sends a message to an issuer to request a token, the client can provide proposed key  
2406 material using the <wst:Entropy> element. If the issuer doesn't contribute any key material, this is  
2407 used as the secret (key). This information is encrypted for the issuer either using  
2408 <xenc:EncryptedKey> or by using a transport security. If the requestor provides key material that the

2409 recipient doesn't accept, then the issuer SHOULD reject the request. Note that the issuer need not return  
2410 the key provided by the requestor.

2411

2412 The following illustrates the syntax of a request for a custom security token and includes a secret that is  
2413 to be used for the key. In this example the entropy is encrypted for the issuer (if transport security was  
2414 used for confidentiality then the <wst:Entropy> element would contain a <wst:BinarySecret>  
2415 element):

```
2416 <wst:RequestSecurityToken xmlns:wst="...">  
2417 <wst:TokenType>  
2418   http://example.org/mySpecialToken  
2419 </wst:TokenType>  
2420 <wst:RequestType>  
2421   http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
2422 </wst:RequestType>  
2423 <wst:Entropy>  
2424   <xenc:EncryptedData xmlns:xenc="...">...</xenc:EncryptedData>  
2425 </wst:Entropy>  
2426 </wst:RequestSecurityToken>
```

### 2427 **A.3 Issuer-Provided Keys**

2428 If a requestor fails to provide key material, then issued proof-of-possession tokens contain an issuer-  
2429 provided secret that is encrypted for the requestor (either using <xenc:EncryptedKey> or by using a  
2430 transport security).

2431

2432 The following illustrates the syntax of a token being returned with an associated proof-of-possession  
2433 token that is encrypted using the requestor's public key.

```
2434 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2435 <wst:RequestSecurityTokenResponse>  
2436   <wst:RequestedSecurityToken>  
2437     <xyz:CustomToken xmlns:xyz="...">  
2438       ...  
2439     </xyz:CustomToken>  
2440   </wst:RequestedSecurityToken>  
2441   <wst:RequestedProofToken>  
2442     <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">  
2443       ...  
2444     </xenc:EncryptedKey>  
2445   </wst:RequestedProofToken>  
2446 </wst:RequestSecurityTokenResponse>  
2447 </wst:RequestSecurityTokenResponseCollection>
```

### 2448 **A.4 Composite Keys**

2449 The safest form of key exchange/generation is when both the requestor and the issuer contribute to the  
2450 key material. In this case, the request sends encrypted key material. The issuer then returns additional  
2451 encrypted key material. The actual secret (key) is computed using a function of the two pieces of data.  
2452 Ideally this secret is never used and, instead, keys derived are used for message protection.

2453

2454 The following example illustrates a server, having received a request with requestor entropy returning its  
2455 own entropy, which is used in conjunction with the requestor's to generate a key. In this example the  
2456 entropy is not encrypted because the transport is providing confidentiality (otherwise the  
2457 <wst:Entropy> element would have an <xenc:EncryptedData> element).

2458  
2459  
2460  
2461  
2462  
2463  
2464  
2465  
2466  
2467  
2468  
2469

```
<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>
```

## 2470 **A.5 Key Transfer and Distribution**

2471 There are also a few mechanisms where existing keys are transferred to other parties.

### 2472 **A.5.1 Direct Key Transfer**

2473 If one party has a token and key and wishes to share this with another party, the key can be directly  
2474 transferred. This is accomplished by sending an RSTR (either in the body or header) to the other party.  
2475 The RSTR contains the token and a proof-of-possession token that contains the key encrypted for the  
2476 recipient.

2477

2478 In the following example a custom token and its associated proof-of-possession token are known to party  
2479 A who wishes to share them with party B. In this example, A is a member in a secure on-line chat  
2480 session and is inviting B to join the conversation. After authenticating B, A sends B an RSTR. The RSTR  
2481 contains the token and the key is communicated as a proof-of-possession token that is encrypted for B:

2482  
2483  
2484  
2485  
2486  
2487  
2488  
2489  
2490  
2491  
2492  
2493  
2494  
2495

```
<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>
```

### 2496 **A.5.2 Brokered Key Distribution**

2497 A third party MAY also act as a broker to transfer keys. For example, a requestor may obtain a token and  
2498 proof-of-possession token from a third-party STS. The token contains a key encrypted for the target  
2499 service (either using the service's public key or a key known to the STS and target service). The proof-of-  
2500 possession token contains the same key encrypted for the requestor (similarly this can use public or  
2501 symmetric keys).

2502

2503 In the following example a custom token and its associated proof-of-possession token are returned from a  
2504 broker B to a requestor R for access to service S. The key for the session is contained within the custom  
2505 token encrypted for S using either a secret known by B and S or using S's public key. The same secret is  
2506 encrypted for R and returned as the proof-of-possession token:

```

2507 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
2508   <wst:RequestSecurityTokenResponse>
2509     <wst:RequestedSecurityToken>
2510       <xyz:CustomToken xmlns:xyz="...">
2511         ...
2512         <xenc:EncryptedKey xmlns:xenc="...">
2513           ...
2514         </xenc:EncryptedKey>
2515         ...
2516       </xyz:CustomToken>
2517     </wst:RequestedSecurityToken>
2518     <wst:RequestedProofToken>
2519       <xenc:EncryptedKey Id="newProof">
2520         ...
2521       </xenc:EncryptedKey>
2522     </wst:RequestedProofToken>
2523   </wst:RequestSecurityTokenResponse>
2524 </wst:RequestSecurityTokenResponseCollection>

```

### 2525 **A.5.3 Delegated Key Transfer**

2526 Key transfer can also take the form of delegation. That is, one party transfers the right to use a key  
2527 without actually transferring the key. In such cases, a delegation token, e.g. XrML, is created that  
2528 identifies a set of rights and a delegation target and is secured by the delegating party. That is, one key  
2529 indicates that another key can use a subset (or all) of its rights. The delegate can provide this token and  
2530 prove itself (using its own key – the delegation target) to a service. The service, assuming the trust  
2531 relationships have been established and that the delegator has the right to delegate, can then authorize  
2532 requests sent subject to delegation rules and trust policies.

2533  
2534 In this example a custom token is issued from party A to party B. The token indicates that B (specifically  
2535 B's key) has the right to submit purchase orders. The token is signed using a secret key known to the  
2536 target service T and party A (the key used to ultimately authorize the requests that B makes to T), and a  
2537 new session key that is encrypted for T. A proof-of-possession token is included that contains the  
2538 session key encrypted for B. As a result, B is *effectively* using A's key, but doesn't actually know the key.

```

2539 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
2540   <wst:RequestSecurityTokenResponse>
2541     <wst:RequestedSecurityToken>
2542       <xyz:CustomToken xmlns:xyz="...">
2543         ...
2544         <xyz:DelegateTo>B</xyz:DelegateTo>
2545         <xyz:DelegateRights>
2546           SubmitPurchaseOrder
2547         </xyz:DelegateRights>
2548         <xenc:EncryptedKey xmlns:xenc="...">
2549           ...
2550         </xenc:EncryptedKey>
2551         <ds:Signature xmlns:ds="...">...</ds:Signature>
2552         ...
2553       </xyz:CustomToken>
2554     </wst:RequestedSecurityToken>
2555     <wst:RequestedProofToken>
2556       <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">
2557         ...
2558       </xenc:EncryptedKey>
2559     </wst:RequestedProofToken>
2560   </wst:RequestSecurityTokenResponse>
2561 </wst:RequestSecurityTokenResponseCollection>

```

## 2562 **A.5.4 Authenticated Request/Reply Key Transfer**

2563 In some cases the RST/RSTR mechanism is not used to transfer keys because it is part of a simple  
2564 request/reply. However, there may be a desire to ensure mutual authentication as part of the key  
2565 transfer. The mechanisms of [WS-Security] can be used to implement this scenario.

2566  
2567 Specifically, the sender wishes the following:

- 2568 • Transfer a key to a recipient that they can use to secure a reply
- 2569 • Ensure that only the recipient can see the key
- 2570 • Provide proof that the sender issued the key

2571  
2572 This scenario could be supported by encrypting and then signing. This would result in roughly the  
2573 following steps:

- 2574 1. Encrypt the message using a generated key
- 2575 2. Encrypt the key for the recipient
- 2576 3. Sign the encrypted form, any other relevant keys, and the encrypted key

2577  
2578 However, if there is a desire to sign prior to encryption then the following general process is used:

- 2579 1. Sign the appropriate message parts using a random key (or ideally a key derived from a random  
2580 key)
- 2581 2. Encrypt the appropriate message parts using the random key (or ideally another key derived from  
2582 the random key)
- 2583 3. Encrypt the random key for the recipient
- 2584 4. Sign just the encrypted key

2585  
2586 This would result in a <wsse:Security> header that looks roughly like the following:

```
2587 <wsse:Security xmlns:wsse="..." xmlns:wssu="..."  
2588     xmlns:ds="..." xmlns:xenc="...">  
2589   <wsse:BinarySecurityToken wssu:Id="myToken">  
2590     ...  
2591   </wsse:BinarySecurityToken>  
2592   <ds:Signature>  
2593     ...signature over #secret using token #myToken...  
2594   </ds:Signature>  
2595   <xenc:EncryptedKey Id="secret">  
2596     ...  
2597   </xenc:EncryptedKey>  
2598   <xenc:ReferenceList>  
2599     ...manifest of encrypted parts using token #secret...  
2600   </xenc:ReferenceList>  
2601   <ds:Signature>  
2602     ...signature over key message parts using token #secret...  
2603   </ds:Signature>  
2604 </wsse:Security>
```

2605  
2606 As well, instead of an <xenc:EncryptedKey> element, the actual token could be passed using  
2607 <xenc:EncryptedData>. The result might look like the following:

```
2608 <wsse:Security xmlns:wsse="..." xmlns:wssu="..."  
2609     xmlns:ds="..." xmlns:xenc="...">
```

```

2610 <wsse:BinarySecurityToken wsu:Id="myToken">
2611   ...
2612 </wsse:BinarySecurityToken>
2613 <ds:Signature>
2614   ...signature over #secret or #Esecret using token #myToken...
2615 </ds:Signature>
2616 <xenc:EncryptedData Id="Esecret">
2617   ...Encrypted version of a token with Id="secret"...
2618 </xenc:EncryptedData>
2619 <xenc:ReferenceList>
2620   ...manifest of encrypted parts using token #secret...
2621 </xenc:ReferenceList>
2622 <ds:Signature>
2623   ...signature over key message parts using token #secret...
2624 </ds:Signature>
2625 </wsse:Security>

```

## 2626 **A.6 Perfect Forward Secrecy**

2627 In some situations it is desirable for a key exchange to have the property of perfect forward secrecy. This  
 2628 means that it is impossible to reconstruct the shared secret even if the private keys of the parties are  
 2629 disclosed.

2630  
 2631 The most straightforward way to attain perfect forward secrecy when using asymmetric key exchange is  
 2632 to dispose of one's key exchange key pair periodically (or even after every key exchange), replacing it  
 2633 with a fresh one. Of course, a freshly generated public key must still be authenticated (using any of the  
 2634 methods normally available to prove the identity of a public key's owner).

2635  
 2636 The perfect forward secrecy property MAY be achieved by specifying a `<wst:entropy>` element that  
 2637 contains an `<xenc:EncryptedKey>` that is encrypted under a public key pair created for use in a single  
 2638 key agreement. The public key does not require authentication since it is only used to provide additional  
 2639 entropy. If the public key is modified, the key agreement will fail. Care should be taken, when using this  
 2640 method, to ensure that the now-secret entropy exchanged via the `<wst:entropy>` element is not  
 2641 revealed elsewhere in the protocol (since such entropy is often assumed to be publicly revealed plaintext,  
 2642 and treated accordingly).

2643  
 2644 Although any public key scheme might be used to achieve perfect forward secrecy (in either of the above  
 2645 methods) it is generally desirable to use an algorithm that allows keys to be generated quickly. The Diffie-  
 2646 Hellman key exchange is often used for this purpose since generation of a key only requires the  
 2647 generation of a random integer and calculation of a single modular exponent.

2648

## B. WSDL

2649 The WSDL below does not fully capture all the possible message exchange patterns, but captures the  
2650 typical message exchange pattern as described in this document.

```
2651 <?xml version="1.0"?>
2652 <wsdl:definitions
2653     targetNamespace="http://docs.oasis-open.org/ws-sx/ws-
2654 trust/200512/wsdl"
2655     xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/wsdl"
2656     xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
2657     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
2658     xmlns:xs="http://www.w3.org/2001/XMLSchema"
2659     xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
2660 >
2661 <!-- this is the WS-I BP-compliant way to import a schema -->
2662 <wsdl:types>
2663     <xs:schema>
2664         <xs:import
2665             namespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
2666             schemaLocation="http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-
2667 trust.xsd"/>
2668     </xs:schema>
2669 </wsdl:types>
2670
2671 <!-- WS-Trust defines the following GEDs -->
2672 <wsdl:message name="RequestSecurityTokenMsg">
2673     <wsdl:part name="request" element="wst:RequestSecurityToken" />
2674 </wsdl:message>
2675 <wsdl:message name="RequestSecurityTokenResponseMsg">
2676     <wsdl:part name="response"
2677         element="wst:RequestSecurityTokenResponse" />
2678 </wsdl:message>
2679 <wsdl:message name="RequestSecurityTokenCollectionMsg">
2680     <wsdl:part name="requestCollection"
2681         element="wst:RequestSecurityTokenCollection"/>
2682 </wsdl:message>
2683 <wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
2684     <wsdl:part name="responseCollection"
2685         element="wst:RequestSecurityTokenResponseCollection"/>
2686 </wsdl:message>
2687
2688 <!-- This portType an example of a Requestor (or other) endpoint that
2689     Accepts SOAP-based challenges from a Security Token Service -->
2690 <wsdl:portType name="WSSecurityRequestor">
2691     <wsdl:operation name="Challenge">
2692         <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
2693         <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
2694     </wsdl:operation>
2695 </wsdl:portType>
2696
2697 <!-- This portType is an example of an STS supporting full protocol -->
2698 <wsdl:portType name="SecurityTokenService">
2699     <wsdl:operation name="Cancel">
2700         <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
2701 trust/200512/RST/Cancel" message="tns:RequestSecurityTokenMsg"/>
2702         <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
2703 trust/200512/RSTR/CancelFinal" message="tns:RequestSecurityTokenResponseMsg"/>
2704     </wsdl:operation>
2705     <wsdl:operation name="Issue">
```

```

2706     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
2707 trust/200512/RST/Issue" message="tns:RequestSecurityTokenMsg"/>
2708     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
2709 trust/200512/RSTRC/IssueFinal"
2710 message="tns:RequestSecurityTokenResponseCollectionMsg"/>
2711   </wsdl:operation>
2712   <wsdl:operation name="Renew">
2713     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
2714 trust/200512/RST/Renew" message="tns:RequestSecurityTokenMsg"/>
2715     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
2716 trust/200512/RSTR/RenewFinal" message="tns:RequestSecurityTokenResponseMsg"/>
2717   </wsdl:operation>
2718   <wsdl:operation name="Validate">
2719     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
2720 trust/200512/RST/Validate" message="tns:RequestSecurityTokenMsg"/>
2721     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
2722 trust/200512/RSTR/ValidateFinal"
2723 message="tns:RequestSecurityTokenResponseMsg"/>
2724   </wsdl:operation>
2725   <wsdl:operation name="KeyExchangeToken">
2726     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
2727 trust/200512/RST/KET" message="tns:RequestSecurityTokenMsg"/>
2728     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
2729 trust/200512/RSTR/KETFinal" message="tns:RequestSecurityTokenResponseMsg"/>
2730   </wsdl:operation>
2731   <wsdl:operation name="RequestCollection">
2732     <wsdl:input message="tns:RequestSecurityTokenCollectionMsg"/>
2733     <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg"/>
2734   </wsdl:operation>
2735 </wsdl:portType>
2736
2737 <!-- This portType is an example of an endpoint that accepts
2738 Unsolicited RequestSecurityTokenResponse messages -->
2739 <wsdl:portType name="SecurityTokenResponseService">
2740   <wsdl:operation name="RequestSecurityTokenResponse">
2741     <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
2742   </wsdl:operation>
2743 </wsdl:portType>
2744
2745 </wsdl:definitions>

```



2746

---

## C. Acknowledgements

2747 The following individuals have participated in the creation of this specification and are gratefully  
2748 acknowledged:

2749 **Original Authors of the initial contribution:**

2750 Steve Anderson, OpenNetwork  
2751 Jeff Bohren, OpenNetwork  
2752 Toufic Boubez, Layer 7  
2753 Marc Chanliau, Computer Associates  
2754 Giovanni Della-Libera, Microsoft  
2755 Brendan Dixon, Microsoft  
2756 Praerit Garg, Microsoft  
2757 Martin Gudgin (Editor), Microsoft  
2758 Phillip Hallam-Baker, VeriSign  
2759 Maryann Hondo, IBM  
2760 Chris Kaler, Microsoft  
2761 Hal Lockhart, BEA  
2762 Robin Martherus, Oblix  
2763 Hiroshi Maruyama, IBM  
2764 Anthony Nadalin (Editor), IBM  
2765 Nataraj Nagaratnam, IBM  
2766 Andrew Nash, Reactivity  
2767 Rob Philpott, RSA Security  
2768 Darren Platt, Ping Identity  
2769 Hemma Prafullchandra, VeriSign  
2770 Maneesh Sahu, Actional  
2771 John Shewchuk, Microsoft  
2772 Dan Simon, Microsoft  
2773 Davanum Srinivas, Computer Associates  
2774 Elliot Waingold, Microsoft  
2775 David Waite, Ping Identity  
2776 Doug Walter, Microsoft  
2777 Riaz Zolfonoon, RSA Security

2778

2779 **Original Acknowledgments of the initial contribution:**

2780 Paula Austel, IBM  
2781 Keith Ballinger, Microsoft  
2782 Bob Blakley, IBM  
2783 John Brezak, Microsoft  
2784 Tony Cowan, IBM  
2785 Cédric Fournet, Microsoft  
2786 Vijay Gajjala, Microsoft  
2787 HongMei Ge, Microsoft  
2788 Satoshi Hada, IBM  
2789 Heather Hinton, IBM  
2790 Slava Kavsan, RSA Security  
2791 Scott Konersmann, Microsoft  
2792 Leo Laferriere, Computer Associates

- 2793 Paul Leach, Microsoft
- 2794 Richard Levinson, Computer Associates
- 2795 John Linn, RSA Security
- 2796 Michael McIntosh, IBM
- 2797 Steve Millet, Microsoft
- 2798 Birgit Pfitzmann, IBM
- 2799 Fumiko Satoh, IBM
- 2800 Keith Stobie, Microsoft
- 2801 T.R. Vishwanath, Microsoft
- 2802 Richard Ward, Microsoft
- 2803 Hervey Wilson, Microsoft
- 2804
- 2805 **TC Members during the development of this specification:**
- 2806 Don Adams, Tibco Software Inc.
- 2807 Jan Alexander, Microsoft Corporation
- 2808 Steve Anderson, BMC Software
- 2809 Donal Arundel, IONA Technologies
- 2810 Howard Bae, Oracle Corporation
- 2811 Abbie Barbir, Nortel Networks Limited
- 2812 Charlton Barreto, Adobe Systems
- 2813 Mighael Botha, Software AG, Inc.
- 2814 Toufic Boubez, Layer 7 Technologies Inc.
- 2815 Norman Brickman, Mitre Corporation
- 2816 Melissa Brumfield, Booz Allen Hamilton
- 2817 Lloyd Burch, Novell
- 2818 Scott Cantor, Internet2
- 2819 Greg Carpenter, Microsoft Corporation
- 2820 Steve Carter, Novell
- 2821 Ching-Yun (C.Y.) Chao, IBM
- 2822 Martin Chapman, Oracle Corporation
- 2823 Kate Cherry, Lockheed Martin
- 2824 Henry (Hyenvui) Chung, IBM
- 2825 Luc Clement, Systinet Corp.
- 2826 Paul Cotton, Microsoft Corporation
- 2827 Glen Daniels, Sonic Software Corp.
- 2828 Peter Davis, Neustar, Inc.
- 2829 Martijn de Boer, SAP AG
- 2830 Werner Dittmann, Siemens AG
- 2831 Abdeslem DJAOUI, CCLRC-Rutherford Appleton Laboratory
- 2832 Fred Dushin, IONA Technologies
- 2833 Petr Dvorak, Systinet Corp.
- 2834 Colleen Evans, Microsoft Corporation
- 2835 Ruchith Fernando, WSO2
- 2836 Mark Fussell, Microsoft Corporation

2837 Vijay Gajjala, Microsoft Corporation  
2838 Marc Goodner, Microsoft Corporation  
2839 Hans Granqvist, VeriSign  
2840 Martin Gudgin, Microsoft Corporation  
2841 Tony Gullotta, SOA Software Inc.  
2842 Jiandong Guo, Sun Microsystems  
2843 Phillip Hallam-Baker, VeriSign  
2844 Patrick Harding, Ping Identity Corporation  
2845 Heather Hinton, IBM  
2846 Frederick Hirsch, Nokia Corporation  
2847 Jeff Hodges, Neustar, Inc.  
2848 Will Hopkins, BEA Systems, Inc.  
2849 Alex Hristov, Otecia Incorporated  
2850 John Hughes, PA Consulting  
2851 Diane Jordan, IBM  
2852 Venugopal K, Sun Microsystems  
2853 Chris Kaler, Microsoft Corporation  
2854 Dana Kaufman, Forum Systems, Inc.  
2855 Paul Knight, Nortel Networks Limited  
2856 Ramanathan Krishnamurthy, IONA Technologies  
2857 Christopher Kurt, Microsoft Corporation  
2858 Kelvin Lawrence, IBM  
2859 Hubert Le Van Gong, Sun Microsystems  
2860 Jong Lee, BEA Systems, Inc.  
2861 Rich Levinson, Oracle Corporation  
2862 Tommy Lindberg, Dajeil Ltd.  
2863 Mark Little, JBoss Inc.  
2864 Hal Lockhart, BEA Systems, Inc.  
2865 Mike Lyons, Layer 7 Technologies Inc.  
2866 Eve Maler, Sun Microsystems  
2867 Ashok Malhotra, Oracle Corporation  
2868 Anand Mani, CrimsonLogic Pte Ltd  
2869 Jonathan Marsh, Microsoft Corporation  
2870 Robin Martherus, Oracle Corporation  
2871 Miko Matsumura, Infravio, Inc.  
2872 Gary McAfee, IBM  
2873 Michael McIntosh, IBM  
2874 John Merrells, Sxip Networks SRL  
2875 Jeff Mischkinisky, Oracle Corporation  
2876 Prateek Mishra, Oracle Corporation  
2877 Bob Morgan, Internet2  
2878 Vamsi Motukuru, Oracle Corporation

2879 Raajmohan Na, EDS  
2880 Anthony Nadalin, IBM  
2881 Andrew Nash, Reactivity, Inc.  
2882 Eric Newcomer, IONA Technologies  
2883 Duane Nickull, Adobe Systems  
2884 Toshihiro Nishimura, Fujitsu Limited  
2885 Rob Philpott, RSA Security  
2886 Denis Pilipchuk, BEA Systems, Inc.  
2887 Darren Platt, Ping Identity Corporation  
2888 Martin Raeppe, SAP AG  
2889 Nick Ragouzis, Enosis Group LLC  
2890 Prakash Reddy, CA  
2891 Alain Regnier, Ricoh Company, Ltd.  
2892 Irving Reid, Hewlett-Packard  
2893 Bruce Rich, IBM  
2894 Tom Rutt, Fujitsu Limited  
2895 Maneesh Sahu, Actional Corporation  
2896 Frank Siebenlist, Argonne National Laboratory  
2897 Joe Smith, Apani Networks  
2898 Davanum Srinivas, WSO2  
2899 Yakov Sverdlov, CA  
2900 Gene Thurston, AmberPoint  
2901 Victor Valle, IBM  
2902 Asir Vedamuthu, Microsoft Corporation  
2903 Greg Whitehead, Hewlett-Packard  
2904 Ron Williams, IBM  
2905 Corinna Witt, BEA Systems, Inc.  
2906 Kyle Young, Microsoft Corporation