



# WS-Trust 1.4

## OASIS Committee Draft

9 July 2008

**Specification URIs:**

**This Version:**

<http://docs.oasis-open.org/ws-sx/ws-trust/200802/ws-trust-1.4-cd-01.doc>  
<http://docs.oasis-open.org/ws-sx/ws-trust/200802/ws-trust-1.4-cd-01.pdf>  
<http://docs.oasis-open.org/ws-sx/ws-trust/200802/ws-trust-1.4-cd-01.html>

**Previous 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>

**Latest Version:**

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

**Artifact Type:**

specification

**Technical Committee:**

OASIS Web Service Secure Exchange TC

**Chair(s):**

Kelvin Lawrence, IBM  
Chris Kaler, Microsoft

**Editor(s):**

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

**Related work:**

N/A

**Declared XML namespace(s):**

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

**Abstract:**

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

**Status:**

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

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the

“Send A Comment” button on the Technical Committee’s web page at <http://www.oasis-open.org/committees/ws-sx>.

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

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

---

## Notices

Copyright © OASIS® 1993–2008. All Rights Reserved. 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
1.8	Conformance.....	11
2	Web Services Trust Model.....	13
2.1	Models for Trust Brokering and Assessment.....	14
2.2	Token Acquisition.....	14
2.3	Out-of-Band Token Acquisition.....	15
2.4	Trust Bootstrap.....	15
3	Security Token Service Framework.....	16
3.1	Requesting a Security Token.....	16
3.2	Returning a Security Token.....	17
3.3	Binary Secrets.....	19
3.4	Composition.....	19
4	Issuance Binding.....	20
4.1	Requesting a Security Token.....	20
4.2	Request Security Token Collection.....	22
4.2.1	Processing Rules.....	24
4.3	Returning a Security Token Collection.....	24
4.4	Returning a Security Token.....	25
4.4.1	wsp:AppliesTo in RST and RSTR.....	26
4.4.2	Requested References.....	27
4.4.3	Keys and Entropy.....	27
4.4.4	Returning Computed Keys.....	28
4.4.5	Sample Response with Encrypted Secret.....	29
4.4.6	Sample Response with Unencrypted Secret.....	29
4.4.7	Sample Response with Token Reference.....	30
4.4.8	Sample Response without Proof-of-Possession Token.....	30
4.4.9	Zero or One Proof-of-Possession Token Case.....	30
4.4.10	More Than One Proof-of-Possession Tokens Case.....	31
4.5	Returning Security Tokens in Headers.....	32
5	Renewal Binding.....	34
6	Cancel Binding.....	37
6.1	STS-initiated Cancel Binding.....	38
7	Validation Binding.....	40
8	Negotiation and Challenge Extensions.....	43
8.1	Negotiation and Challenge Framework.....	44

8.2	Signature Challenges .....	44
8.3	User Interaction Challenge .....	45
8.3.1	Challenge Format .....	46
8.3.2	PIN and OTP Challenges .....	49
8.4	Binary Exchanges and Negotiations .....	50
8.5	Key Exchange Tokens .....	50
8.6	Custom Exchanges .....	51
8.7	Signature Challenge Example .....	51
8.8	Challenge Examples .....	53
8.8.1	Text and choice challenge .....	53
8.8.2	PIN challenge .....	55
8.8.3	PIN challenge with optimized response .....	57
8.9	Custom Exchange Example .....	58
8.10	Protecting Exchanges .....	59
8.11	Authenticating Exchanges .....	59
9	Key and Token Parameter Extensions .....	61
9.1	On-Behalf-Of Parameters .....	61
9.2	Key and Encryption Requirements .....	61
9.3	Delegation and Forwarding Requirements .....	66
9.4	Policies .....	67
9.5	Authorized Token Participants .....	68
10	Key Exchange Token Binding .....	69
11	Error Handling .....	71
12	Security Considerations .....	72
A.	Key Exchange .....	74
A.1	Ephemeral Encryption Keys .....	74
A.2	Requestor-Provided Keys .....	74
A.3	Issuer-Provided Keys .....	75
A.4	Composite Keys .....	75
A.5	Key Transfer and Distribution .....	76
A.5.1	Direct Key Transfer .....	76
A.5.2	Brokered Key Distribution .....	76
A.5.3	Delegated Key Transfer .....	77
A.5.4	Authenticated Request/Reply Key Transfer .....	78
A.6	Perfect Forward Secrecy .....	79
B.	WSDL .....	80
C.	Acknowledgements .....	82

---

# 1 Introduction

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

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

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

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

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

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

Section 12 is non-normative.

## 1.1 Goals and Non-Goals

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

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

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

The following are explicit non-goals for this document:

- Password authentication
- Token revocation
- Management of trust policies

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

- Establishing a security context token

- 41 • Key derivation

## 42 1.2 Requirements

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

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

## 47 1.3 Namespace

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

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

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

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

55 *Table 1: Prefixes and XML Namespaces used in this specification.*

Prefix	Namespace	Specification(s)
S11	<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
wst14	<a href="http://docs.oasis-open.org/ws-sx/ws-trust/200802">http://docs.oasis-open.org/ws-sx/ws-trust/200802</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> or <a href="http://www.w3.org/ns/ws-policy">http://www.w3.org/ns/ws-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]

## 56 1.4 Schema and WSDL Files

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

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

- An attribute extensibility point is referred to using `@{any}` in place of the attribute name. This indicates that any attribute name can be used, from any namespace other than the namespace of this specification.

In this document reference is made to the `wsu:Id` attribute and the `wsu:Created` and `wsu:Expires` elements in a utility schema (<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd>). The `wsu:Id` attribute and the `wsu:Created` and `wsu:Expires` elements were added to the utility schema with the intent that other specifications requiring such an ID type attribute or timestamp element could reference it (as is done here).

## 1.6 Normative References

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

184	[XML-C14N]	W3C Recommendation, "Canonical XML Version 1.0", 15 March 2001. <a href="http://www.w3.org/TR/2001/REC-xml-c14n-20010315">http://www.w3.org/TR/2001/REC-xml-c14n-20010315</a>
185		
186	[XML-Encrypt]	W3C Recommendation, "XML Encryption Syntax and Processing", 10 December 2002. <a href="http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/">http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/</a>
187		
188		
189	[XML-Schema1]	W3C Recommendation, "XML Schema Part 1: Structures Second Edition", 28 October 2004. <a href="http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/">http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/</a>
190		
191		
192	[XML-Schema2]	W3C Recommendation, "XML Schema Part 2: Datatypes Second Edition", 28 October 2004. <a href="http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/">http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/</a>
193		
194		
195	[XML-Signature]	W3C Recommendation, "XML-Signature Syntax and Processing", 12 February 2002. <a href="http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/">http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/</a>
196		
197		
198		

## 199 1.7 Non-Normative References

200	[Kerberos]	J. Kohl and C. Neuman, "The Kerberos Network 149 Authentication Service (V5)," RFC 1510, September 1993. <a href="http://www.ietf.org/rfc/rfc1510.txt">http://www.ietf.org/rfc/rfc1510.txt</a>
201		
202		
203	[WS-Federation]	"Web Services Federation Language," BEA, IBM, Microsoft, RSA Security, VeriSign, July 2003.
204		
205	[WS-SecurityPolicy]	OASIS Committee Draft, "WS-SecurityPolicy 1.2", September 2006 <a href="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512">http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512</a>
206		
207	[X509]	S. Santesson, et al, "Internet X.509 Public Key Infrastructure Qualified Certificates Profile." <a href="http://www.itu.int/rec/recommendation.asp?type=items&amp;lang=e&amp;parent=T-REC-X.509-200003-1">http://www.itu.int/rec/recommendation.asp?type=items&amp;lang=e&amp;parent=T-REC-X.509-200003-1</a>
208		
209		
210		

## 211 1.8 Conformance

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

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

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

226 This specification defines a number of extensions; compliant services are NOT REQUIRED to implement  
227 OPTIONAL features defined in this specification. However, if a service implements an aspect of the  
228 specification, it MUST comply with the requirements specified (e.g. related "MUST" statements). If an  
229 OPTIONAL message is not supported, then the implementation SHOULD Fault just as it would for any

230 other unrecognized/unsupported message. If an OPTIONAL message is supported, then the  
231 implementation MUST satisfy all of the MUST and REQUIRED sections of the message.  
232

233

## 2 Web Services Trust Model

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

239

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

244

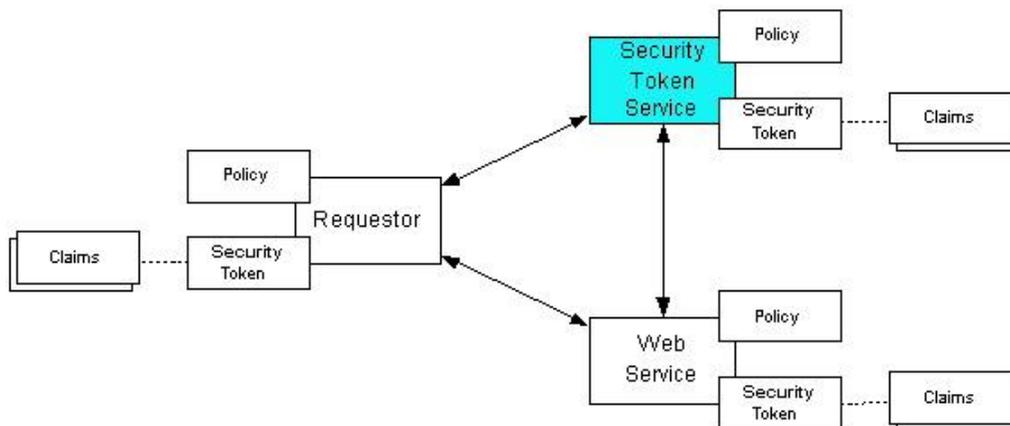
245 One way to demonstrate authorized use of a security token is to include a digital signature using the  
246 associated secret key (from a proof-of-possession token). This allows a requestor to prove a required set  
247 of claims by associating security tokens (e.g., PKIX, X.509 certificates) with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

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

258

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

261



262

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

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

269  
270 In the figure above the arrows represent possible communication paths; the requestor MAY obtain a  
271 token from the security token service, or it MAY have been obtained indirectly. The requestor then  
272 demonstrates authorized use of the token to the Web service. The Web service either trusts the issuing  
273 security token service or MAY request a token service to validate the token (or the Web service MAY  
274 validate the token itself).

275  
276 In summary, the Web service has a policy applied to it, receives a message from a requestor that possibly  
277 includes security tokens, and MAY have some protection applied to it using [\[WS-Security\]](#) mechanisms.  
278 The following key steps are performed by the trust engine of a Web service (note that the order of  
279 processing is non-normative):

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

290  
291 If these conditions are met, and the requestor is authorized to perform the operation, then the service can  
292 process the service request.

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

299  
300 The [\[WS-Federation\]](#) specification builds on this specification to define mechanisms for brokering and  
301 federating trust, identity, and claims. Examples are provided in [\[WS-Federation\]](#) illustrating different trust  
302 scenarios and usage patterns.

## 303 **2.1 Models for Trust Brokering and Assessment**

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

## 307 **2.2 Token Acquisition**

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

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

314

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

## 321 **2.3 Out-of-Band Token Acquisition**

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

## 327 **2.4 Trust Bootstrap**

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

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

337

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

340

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

345

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

---

## 3 Security Token Service Framework

350

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

352

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

358

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

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

365

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

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

373

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

380

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

### 3.1 Requesting a Security Token

385

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

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

391 The syntax for this element is as follows:

392  
393  
394  
395  
396  
397

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

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

399 */wst:RequestSecurityToken*

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

401 */wst:RequestSecurityToken/@Context*

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

406 */wst:RequestSecurityToken/wst:TokenType*

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

411 */wst:RequestSecurityToken/wst:RequestType*

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

418 */wst:RequestSecurityToken/wst:SecondaryParameters*

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

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

426 */wst:RequestSecurityToken/{any}*

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

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

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

## 434 3.2 Returning a Security Token

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

439

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

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

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

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

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

462 The syntax for this element is as follows:

```
463 <wst:RequestSecurityTokenResponse Context="..." xmlns:wst="...">  
464   <wst:TokenType>...</wst:TokenType>  
465   <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>  
466   ...  
467 </wst:RequestSecurityTokenResponse>
```

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

469 */wst:RequestSecurityTokenResponse*

470 This is the response to a security token request.

471 */wst:RequestSecurityTokenResponse/@Context*

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

477 */wst:RequestSecurityTokenResponse/wst:TokenType*

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

479 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

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

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

491 */wst:RequestSecurityTokenResponse/{any}*

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

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

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

### 497 3.3 Binary Secrets

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

505 *.../wst:BinarySecret*

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

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

509 This OPTIONAL attribute indicates the type of secret being encoded. The pre-defined values are  
 510 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)

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

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

### 514 3.4 Composition

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

---

## 519 4 Issuance Binding

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

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

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

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

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

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

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

538

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

542

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

545

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

### 549 4.1 Requesting a Security Token

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

```
553 <wst:RequestSecurityToken xmlns:wst="...">  
554   <wst:TokenType>...</wst:TokenType>  
555   <wst:RequestType>...</wst:RequestType>  
556   ...  
557   <wsp:AppliesTo>...</wsp:AppliesTo>  
558   <wst:Claims Dialect="...">...</wst:Claims>  
559   <wst:Entropy>  
560     <wst:BinarySecret>...</wst:BinarySecret>  
561   </wst:Entropy>  
562   <wst:Lifetime>
```

563  
564  
565  
566

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

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

568 */wst:RequestSecurityToken/wst:TokenType*

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

577 */wst:RequestSecurityToken/wsp:AppliesTo*

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

585 */wst:RequestSecurityToken/wst:Claims*

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

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

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

593 */wst:RequestSecurityToken/wst:Entropy*

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

598 */wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret*

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

605 */wst:RequestSecurityToken/wst:Lifetime*

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

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

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

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

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

625

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

```
632 <S11:Envelope xmlns:S11="..." xmlns:wsua="..." xmlns:wsse="..."  
633   xmlns:xenc="..." xmlns:wst="...">  
634   <S11:Header>  
635     ...  
636     <wsse:Security>  
637       <xenc:ReferenceList>...</xenc:ReferenceList>  
638       <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>  
639       <ds:Signature xmlns:ds="...">  
640         ...  
641         <ds:KeyInfo>  
642           <wsse:SecurityTokenReference>  
643             <wsse:Reference URI="#myToken"/>  
644           </wsse:SecurityTokenReference>  
645         </ds:KeyInfo>  
646       </ds:Signature>  
647     </wsse:Security>  
648     ...  
649   </S11:Header>  
650   <S11:Body wsua:Id="req">  
651     <wst:RequestSecurityToken>  
652       <wst:TokenType>  
653         http://example.org/mySpecialToken  
654       </wst:TokenType>  
655       <wst:RequestType>  
656         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
657       </wst:RequestType>  
658     </wst:RequestSecurityToken>  
659   </S11:Body>  
660 </S11:Envelope>
```

## 661 4.2 Request Security Token Collection

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

666

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

671

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

675

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

677

```
678 <wst:RequestSecurityTokenCollection xmlns:wst="...">
679
680   <!-- identity token request -->
681   <wst:RequestSecurityToken Context="http://www.example.com/1">
682     <wst:TokenType>
683       http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
684       1.1#SAMLV2.0
685     </wst:TokenType>
686     <wst:RequestType>http://docs.oasis-open.org/ws-sx/ws-
687     trust/200512/BatchIssue</wst:RequestType>
688     <wsp:AppliesTo xmlns:wsp="..." xmlns:wsa="...">
689       <wsa:EndpointReference>
690         <wsa:Address>http://manufacturer.example.com/</wsa:Address>
691       </wsa:EndpointReference>
692     </wsp:AppliesTo>
693     <wsp:PolicyReference xmlns:wsp="..."
694     URI='http://manufacturer.example.com/IdentityPolicy' />
695   </wst:RequestSecurityToken>
696
697   <!-- certification claim token request -->
698   <wst:RequestSecurityToken Context="http://www.example.com/2">
699     <wst:TokenType>
700       http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
701       1.1#SAMLV2.0
702     </wst:TokenType>
703     <wst:RequestType>http://docs.oasis-open.org/ws-sx/ws-trust/200512
704     /BatchIssue</wst:RequestType>
705     <wst:Claims xmlns:wsp="...">
706       http://manufacturer.example.com/certification
707     </wst:Claims>
708     <wsp:PolicyReference
709     URI='http://certificationbody.example.org/certificationPolicy' />
710   </wst:RequestSecurityToken>
711 </wst:RequestSecurityTokenCollection>
```

712

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

714

715 */wst:RequestSecurityTokenCollection*

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

## 719 4.2.1 Processing Rules

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

722

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

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

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

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

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

- 733 • <http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchIssue>
- 734 • <http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchValidate>
- 735 • <http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchRenew>
- 736 • <http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchCancel>

737

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

740

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

743

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

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

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

751

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

754

## 755 4.3 Returning a Security Token Collection

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

760 The syntax for this element is as follows:

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

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

765 */wst:RequestSecurityTokenResponseCollection*

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

768 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

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

## 770 4.4 Returning a Security Token

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

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

```
774 <wst:RequestSecurityTokenResponse xmlns:wst="...">
775 <wst:TokenType>...</wst:TokenType>
776 <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
777 ...
778 <wsp:AppliesTo xmlns:wsp="...">...</wsp:AppliesTo>
779 <wst:RequestedAttachedReference>
780 ...
781 </wst:RequestedAttachedReference>
782 <wst:RequestedUnattachedReference>
783 ...
784 </wst:RequestedUnattachedReference>
785 <wst:RequestedProofToken>...</wst:RequestedProofToken>
786 <wst:Entropy>
787 <wst:BinarySecret>...</wst:BinarySecret>
788 </wst:Entropy>
789 <wst:Lifetime>...</wst:Lifetime>
790 </wst:RequestSecurityTokenResponse>
```

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

792 */wst:RequestSecurityTokenResponse/wsp:AppliesTo*

793 This OPTIONAL element specifies the scope to which this security token applies. Refer to [\[WS-PolicyAttachment\]](#) for more information. Note that if an `<wsp:AppliesTo>` was specified in the  
794 request, the same scope SHOULD be returned in the response (if a `<wsp:AppliesTo>` is  
795 returned).  
796

797 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

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

803 */wst:RequestSecurityTokenResponse/wst:RequestedAttachedReference*

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

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

814 */wst:RequestSecurityTokenResponse/wst:RequestedUnattachedReference*

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

821 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken*

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

832 */wst:RequestSecurityTokenResponse/wst:Entropy*

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

837 */wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret*

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

840 */wst:RequestSecurityTokenResponse/wst:Lifetime*

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

#### 844 **4.4.1 wsp:AppliesTo in RST and RSTR**

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

- 849 • If neither the requestor nor the issuer specifies a scope then the scope of the issued token is  
850 implied.
- 851 • If the requestor specifies a scope and the issuer does not then the scope of the token is assumed  
852 to be that specified by the requestor.
- 853 • If the requestor does not specify a scope and the issuer does specify a scope then the scope of  
854 the token is as defined by the issuers scope
- 855 • If both requestor and issuer specify a scope then there are two possible outcomes:
  - 856 ○ If both the issuer and requestor specify the same scope then the issued token has that  
857 scope.

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

863

864 The following table summarizes the above rules:

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

#### 865 4.4.2 Requested References

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

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

#### 881 4.4.3 Keys and Entropy

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

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

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

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

893

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

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

904

905 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

#### 906 4.4.4 Returning Computed Keys

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

```

912 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
913   <wst:RequestSecurityTokenResponse>
914     <wst:RequestedProofToken>
915       <wst:ComputedKey>...</wst:ComputedKey>
916     </wst:RequestedProofToken>
917   </wst:RequestSecurityTokenResponse>
918 </wst:RequestSecurityTokenResponseCollection>
  
```

919

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

921 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken/wst:ComputedKey*  
 922 The value of this element is a URI describing how to compute the key. While this can be  
 923 extended by defining new URIs in other bindings and profiles, the following URI pre-defines one  
 924 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.

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

#### 926 4.4.5 Sample Response with Encrypted Secret

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

```

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

#### 945 4.4.6 Sample Response with Unencrypted Secret

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

```

948 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
949   <wst:RequestSecurityTokenResponse>
950     <wst:RequestedSecurityToken>
951       <xyz:CustomToken xmlns:xyz="...">
952         ...
953       </xyz:CustomToken>
954     </wst:RequestedSecurityToken>
955     <wst:RequestedProofToken>
956       <wst:BinarySecret>...</wst:BinarySecret>
957     </wst:RequestedProofToken>
958   </wst:RequestSecurityTokenResponse>
959 </wst:RequestSecurityTokenResponseCollection>
  
```

#### 960 4.4.7 Sample Response with Token Reference

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

```
964 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
965   <wst:RequestSecurityTokenResponse>  
966     <wst:RequestedSecurityToken>  
967       <xyz:CustomToken ID="urn:fabrikam123:5445" xmlns:xyz="...">  
968         ...  
969       </xyz:CustomToken>  
970     </wst:RequestedSecurityToken>  
971     <wst:RequestedAttachedReference>  
972       <wsse:SecurityTokenReference xmlns:wsse="...">  
973         <wsse:Reference URI="urn:fabrikam123:5445"/>  
974       </wsse:SecurityTokenReference>  
975     </wst:RequestedAttachedReference>  
976     ...  
977   </wst:RequestSecurityTokenResponse>  
978 </wst:RequestSecurityTokenResponseCollection>
```

979  
980 In the example above, the recipient may place the returned custom token directly into a message and  
981 include a signature using the provided proof-of-possession token. The specified reference is then placed  
982 into the `<ds:KeyInfo>` of the signature and directly references the included token without requiring the  
983 requestor to understand the details of the custom token format.

#### 984 4.4.8 Sample Response without Proof-of-Possession Token

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

```
989 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
990   <wst:RequestSecurityTokenResponse>  
991     <wst:RequestedSecurityToken>  
992       <xyz:CustomToken xmlns:xyz="...">  
993         ...  
994       </xyz:CustomToken>  
995     </wst:RequestedSecurityToken>  
996   </wst:RequestSecurityTokenResponse>  
997 </wst:RequestSecurityTokenResponseCollection>
```

#### 999 4.4.9 Zero or One Proof-of-Possession Token Case

1000 In the zero or single proof-of-possession token case, a primary token and one or more tokens are  
1001 returned. The returned tokens either use the same proof-of-possession token (one is returned), or no  
1002 proof-of-possession token is returned. The tokens are returned (one each) in the response. The  
1003 following example illustrates this case. The following illustrates the syntax of a supporting security token  
1004 is returned that has no separate proof-of-possession token as it is secured using the same proof-of-  
1005 possession token that was returned.

```
1006  
1007 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
1008   <wst:RequestSecurityTokenResponse>  
1009     <wst:RequestedSecurityToken>
```

```

1010         <xyz:CustomToken xmlns:xyz="...">
1011             ...
1012         </xyz:CustomToken>
1013     </wst:RequestedSecurityToken>
1014     <wst:RequestedProofToken>
1015         <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
1016             ...
1017         </xenc:EncryptedKey>
1018     </wst:RequestedProofToken>
1019 </wst:RequestSecurityTokenResponse>
1020 </wst:RequestSecurityTokenResponseCollection>

```

#### 1021 4.4.10 More Than One Proof-of-Possession Tokens Case

1022 The second case is where multiple security tokens are returned that have separate proof-of-possession  
1023 tokens. As a result, the proof-of-possession tokens, and possibly lifetime and other key parameters  
1024 elements, MAY be different. To address this scenario, the body MAY be specified using the syntax  
1025 illustrated below:

```

1026 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
1027     <wst:RequestSecurityTokenResponse>
1028         ...
1029     </wst:RequestSecurityTokenResponse>
1030     <wst:RequestSecurityTokenResponse>
1031         ...
1032     </wst:RequestSecurityTokenResponse>
1033     ...
1034 </wst:RequestSecurityTokenResponseCollection>

```

1035 The following describes the attributes and elements listed in the schema overview above:

1036 */wst:RequestSecurityTokenResponseCollection*

1037 This element is used to provide multiple RSTR responses, each of which has separate key  
1038 information. One or more RSTR elements are returned in the collection. This MUST always be  
1039 used on the final response to the RST.

1040 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

1041 Each RequestSecurityTokenResponse element is an individual RSTR.

1042 */wst:RequestSecurityTokenResponseCollection/{any}*

1043 This is an extensibility mechanism to allow additional elements, based on schemas, to be added.

1044 */wst:RequestSecurityTokenResponseCollection/@{any}*

1045 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added.

1046 The following illustrates the syntax of a response that includes multiple tokens each, in a separate RSTR,  
1047 each with their own proof-of-possession token.

```

1048 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
1049     <wst:RequestSecurityTokenResponse>
1050         <wst:RequestedSecurityToken>
1051             <xyz:CustomToken xmlns:xyz="...">
1052                 ...
1053             </xyz:CustomToken>
1054         </wst:RequestedSecurityToken>
1055         <wst:RequestedProofToken>
1056             <xenc:EncryptedKey Id="newProofA">
1057                 ...
1058             </xenc:EncryptedKey>
1059         </wst:RequestedProofToken>
1060     </wst:RequestSecurityTokenResponse>
1061     <wst:RequestSecurityTokenResponse>

```

1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1070  
1071  
1072  
1073

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

1074

## 4.5 Returning Security Tokens in Headers

1075  
1076  
1077  
1078  
1079  
1080

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

1081  
1082  
1083  
1084  
1085

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

1086

1087 The following describes the attributes and elements listed in the schema overview above:

1088 */wst:IssuedTokens*

1089 This header element carries one or more issued security tokens. This element schema is defined using the RequestSecurityTokenResponse schema type.

1091 */wst:IssuedTokens/wst:RequestSecurityTokenResponse*

1092 This element MUST appear at least once. Its meaning and semantics are as defined in Section 4.2.

1093 */wst:IssuedTokens/{any}*

1094 This is an extensibility mechanism to allow additional elements, based on schemas, to be added.

1095 */wst:IssuedTokens/@{any}*

1096 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added.

1097

1098 There MAY be multiple instances of the `<wst:IssuedTokens>` header in a given message. Such  
1099 instances MAY be targeted at the same actor/role. Intermediaries MAY add additional  
1100 `<wst:IssuedTokens>` header elements to a message. Intermediaries SHOULD NOT modify any  
1101 `<wst:IssuedTokens>` header already present in a message.

1102

1103 It is RECOMMENDED that the `<wst:IssuedTokens>` header be signed to protect the integrity of the  
1104 issued tokens and of the issuance itself. If confidentiality protection of the `<wst:IssuedTokens>` header is  
1105 REQUIRED then the entire header MUST be encrypted using the `<wsse11:EncryptedHeader>` construct.  
1106 This helps facilitate re-issuance by the receiving party as that party can re-encrypt the entire header for  
1107 another party rather than having to extract and re-encrypt portions of the header.

1108

1109 The following example illustrates a response that includes multiple `<wst:IssuedTokens>` headers.

```
1110 <?xml version="1.0" encoding="utf-8"?>
1111 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsp="..." xmlns:ds="..."
1112 xmlns:x="...">
1113   <S11:Header>
1114     <wst:IssuedTokens>
1115       <wst:RequestSecurityTokenResponse>
1116         <wsp:AppliesTo>
1117           <x:SomeContext1 />
1118         </wsp:AppliesTo>
1119         <wst:RequestedSecurityToken>
1120           ...
1121         </wst:RequestedSecurityToken>
1122         ...
1123       </wst:RequestSecurityTokenResponse>
1124       <wst:RequestSecurityTokenResponse>
1125         <wsp:AppliesTo>
1126           <x:SomeContext1 />
1127         </wsp:AppliesTo>
1128         <wst:RequestedSecurityToken>
1129           ...
1130         </wst:RequestedSecurityToken>
1131         ...
1132       </wst:RequestSecurityTokenResponse>
1133     </wst:IssuedTokens>
1134     <wst:IssuedTokens S11:role="http://example.org/someroles" >
1135       <wst:RequestSecurityTokenResponse>
1136         <wsp:AppliesTo>
1137           <x:SomeContext2 />
1138         </wsp:AppliesTo>
1139         <wst:RequestedSecurityToken>
1140           ...
1141         </wst:RequestedSecurityToken>
1142         ...
1143       </wst:RequestSecurityTokenResponse>
1144     </wst:IssuedTokens>
1145   </S11:Header>
1146   <S11:Body>
1147     ...
1148   </S11:Body>
1149 </S11:Envelope>
```

---

## 5 Renewal Binding

1150

1151 Using the token request framework, this section defines bindings for requesting security tokens to be  
1152 renewed:

1153 **Renew** – A previously issued token with expiration is presented (and possibly proven) and the  
1154 same token is returned with new expiration semantics.

1155

1156 For this binding, the following actions are defined to enable specific processing context to be conveyed to  
1157 the recipient:

1158

```
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
```

1159

1161 For this binding, the `<wst:RequestType>` element uses the following URI:

1162

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

1163 For this binding the token to be renewed is identified in the `<wst:RenewTarget>` element and the  
1164 OPTIONAL `<wst:Lifetime>` element MAY be specified to request a specified renewal duration.

1165

1166 Other extensions MAY be specified in the request (and the response), but the key semantics (size, type,  
1167 algorithms, scope, etc.) MUST NOT be altered during renewal. Token services MAY use renewal as an  
1168 opportunity to rekey, so the renewal responses MAY include a new proof-of-possession token as well as  
1169 entropy and key exchange elements.

1170

1171 The request MUST prove authorized use of the token being renewed unless the recipient trusts the  
1172 requestor to make third-party renewal requests. In such cases, the third-party requestor MUST prove its  
1173 identity to the issuer so that appropriate authorization occurs.

1174

1175 The original proof information SHOULD be proven during renewal.

1176

1177 The renewal binding allows the use of exchanges during the renewal process. Subsequent profiles MAY  
1178 define restriction around the usage of exchanges.

1179

1180 During renewal, all key bearing tokens used in the renewal request MUST have an associated signature.  
1181 All non-key bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the renewal  
1182 response.

1183

1184 The renewal binding also defines several extensions to the request and response elements. The syntax  
1185 for these extension elements is as follows (note that the base elements described above are included  
1186 here italicized for completeness):

1187

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

1188

1189

1190

1191

1192

1193 `<wst:Renewing Allow="..." OK="..." />`  
1194 `</wst:RequestSecurityToken>`

1195 `/wst:RequestSecurityToken/wst:RenewTarget`

1196 This REQUIRED element identifies the token being renewed. This MAY contain a  
1197 `<wsse:SecurityTokenReference>` pointing at the token to be renewed or it MAY directly contain  
1198 the token to be renewed.

1199 `/wst:RequestSecurityToken/wst:AllowPostdating`

1200 This OPTIONAL element indicates that returned tokens SHOULD allow requests for postdated  
1201 tokens. That is, this allows for tokens to be issued that are not immediately valid (e.g., a token  
1202 that can be used the next day).

1203 `/wst:RequestSecurityToken/wst:Renewing`

1204 This OPTIONAL element is used to specify renew semantics for types that support this operation.

1205 `/wst:RequestSecurityToken/wst:Renewing/@Allow`

1206 This OPTIONAL Boolean attribute is used to request a renewable token. If not specified, the  
1207 default value is *true*. A renewable token is one whose lifetime can be extended. This is done  
1208 using a renewal request. The recipient MAY allow renewals without demonstration of authorized  
1209 use of the token or they MAY fault.

1210 `/wst:RequestSecurityToken/wst:Renewing/@OK`

1211 This OPTIONAL Boolean attribute is used to indicate that a renewable token is acceptable if the  
1212 requested duration exceeds the limit of the issuance service. That is, if *true* then tokens can be  
1213 renewed after their expiration. It should be noted that the token is NOT valid after expiration for  
1214 any operation except renewal. The default for this attribute is *false*. It NOT RECOMMENDED to  
1215 use this as it can leave you open to certain types of security attacks. Issuers MAY restrict the  
1216 period after expiration during which time the token can be renewed. This window is governed by  
1217 the issuer's policy.

1218 The following example illustrates a request for a custom token that can be renewed.

```
1219 <wst:RequestSecurityToken xmlns:wst="...">  
1220   <wst:TokenType>  
1221     http://example.org/mySpecialToken  
1222   </wst:TokenType>  
1223   <wst:RequestType>  
1224     http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
1225   </wst:RequestType>  
1226   <wst:Renewing/>  
1227 </wst:RequestSecurityToken>
```

1228  
1229 The following example illustrates a subsequent renewal request and response (note that for brevity only  
1230 the request and response are illustrated). Note that the response includes an indication of the lifetime of  
1231 the renewed token.

```
1232 <wst:RequestSecurityToken xmlns:wst="...">  
1233   <wst:TokenType>  
1234     http://example.org/mySpecialToken  
1235   </wst:TokenType>  
1236   <wst:RequestType>  
1237     http://docs.oasis-open.org/ws-sx/ws-trust/200512/Renew  
1238   </wst:RequestType>  
1239   <wst:RenewTarget>  
1240     ... reference to previously issued token ...  
1241   </wst:RenewTarget>  
1242 </wst:RequestSecurityToken>  
1243
```

```
1244 <wst:RequestSecurityTokenResponse xmlns:wst="...">
1245   <wst:TokenType>
1246     http://example.org/mySpecialToken
1247   </wst:TokenType>
1248   <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1249   <wst:Lifetime>...</wst:Lifetime>
1250   ...
1251 </wst:RequestSecurityTokenResponse>
```

---

## 6 Cancel Binding

1252

1253 Using the token request framework, this section defines bindings for requesting security tokens to be  
1254 cancelled:

1255 **Cancel** – When a previously issued token is no longer needed, the Cancel binding can be used  
1256 to cancel the token, terminating its use. After canceling a token at the issuer, a STS MUST not  
1257 validate or renew the token. A STS MAY initiate the revocation of a token, however, revocation is  
1258 out of scope of this specification and a client MUST NOT rely on it. If a client needs to ensure the  
1259 validity of a token, it MUST validate the token at the issuer.

1260

1261 For this binding, the following actions are defined to enable specific processing context to be conveyed to  
1262 the recipient:

1263  
1264  
1265

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

1266 For this binding, the `<wst:RequestType>` element uses the following URI:

1267

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

1268 Extensions MAY be specified in the request (and the response), but the semantics are not defined by this  
1269 binding.

1270

1271 The request MUST prove authorized use of the token being cancelled unless the recipient trusts the  
1272 requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its  
1273 identity to the issuer so that appropriate authorization occurs.

1274 In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key  
1275 bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the closure response.

1276

1277 A cancelled token is no longer valid for authentication and authorization usages.

1278 On success a cancel response is returned. This is an RSTR message with the  
1279 `<wst:RequestedTokenCancelled>` element in the body. On failure, a Fault is raised. It should be  
1280 noted that the cancel RSTR is informational. That is, the security token is cancelled once the cancel  
1281 request is processed.

1282

1283 The syntax of the request is as follows:

1284  
1285  
1286  
1287  
1288

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

1289 `/wst:RequestSecurityToken/wst:CancelTarget`

1290 This REQUIRED element identifies the token being cancelled. Typically this contains a  
1291 `<wsse:SecurityTokenReference>` pointing at the token, but it could also carry the token  
1292 directly.

1293 The following example illustrates a request to cancel a custom token.

1294

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

1295  
1296  
1297  
1298  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308  
1309  
1310

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

1311 The following example illustrates a response to cancel a custom token.

1312  
1313  
1314  
1315  
1316  
1317  
1318  
1319  
1320  
1321  
1322  
1323

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

## 1324 6.1 STS-initiated Cancel Binding

1325 Using the token request framework, this section defines an OPTIONAL binding for requesting security  
1326 tokens to be cancelled by the STS:

1327 **STS-initiated Cancel** – When a previously issued token becomes invalid on the STS, the STS-  
1328 initiated Cancel binding can be used to cancel the token, terminating its use. After canceling a  
1329 token, a STS MUST not validate or renew the token. This binding can be only used when STS  
1330 can send one-way messages to the original token requestor.

1331

1332 For this binding, the following actions are defined to enable specific processing context to be conveyed to  
1333 the recipient:

1334

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

1335 For this binding, the <wst:RequestType> element uses the following URI:

1336

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

1337 Extensions MAY be specified in the request, but the semantics are not defined by this binding.

1338

1339 The request MUST prove authorized use of the token being cancelled unless the recipient trusts the  
1340 requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its  
1341 identity to the issuer so that appropriate authorization occurs.

1342 In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key  
1343 bearing tokens MUST be signed.

1344

1345 A cancelled token is no longer valid for authentication and authorization usages.

1346

1347 The mechanism to determine the availability of STS-initiated Cancel binding on the STS is out of scope of  
1348 this specification. Similarly, how the client communicates its endpoint address to the STS so that it can  
1349 send the STSCancel messages to the client is out of scope of this specification. This functionality is  
1350 implementation specific and can be solved by different mechanisms that are not in scope for this  
1351 specification.

1352

1353 This is a one-way operation, no response is returned from the recipient of the message.

1354

1355 The syntax of the request is as follows:

```
1356 <wst:RequestSecurityToken xmlns:wst="...">  
1357   <wst:RequestType>...</wst:RequestType>  
1358   ...  
1359   <wst:CancelTarget>...</wst:CancelTarget>  
1360 </wst:RequestSecurityToken>
```

1361 */wst:RequestSecurityToken/wst:CancelTarget*

1362 This REQUIRED element identifies the token being cancelled. Typically this contains a  
1363 <wsse:SecurityTokenReference> pointing at the token, but it could also carry the token  
1364 directly.

1365 The following example illustrates a request to cancel a custom token.

```
1366 <?xml version="1.0" encoding="utf-8"?>  
1367 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">  
1368   <S11:Header>  
1369     <wsse:Security>  
1370       ...  
1371     </wsse:Security>  
1372   </S11:Header>  
1373   <S11:Body>  
1374     <wst:RequestSecurityToken>  
1375       <wst:RequestType>  
1376         http://docs.oasis-open.org/ws-sx/ws-trust/200512/STSCancel  
1377       </wst:RequestType>  
1378       <wst:CancelTarget>  
1379         ...  
1380       </wst:CancelTarget>  
1381     </wst:RequestSecurityToken>  
1382   </S11:Body>  
1383 </S11:Envelope>
```

---

## 7 Validation Binding

1384

1385 Using the token request framework, this section defines bindings for requesting security tokens to be  
1386 validated:

1387 **Validate** – The validity of the specified security token is evaluated and a result is returned. The  
1388 result MAY be a status, a new token, or both.

1389

1390 It should be noted that for this binding, a SOAP Envelope MAY be specified as a "security token" if the  
1391 requestor desires the envelope to be validated. In such cases the recipient SHOULD understand how to  
1392 process a SOAP envelope and adhere to SOAP processing semantics (e.g., mustUnderstand) of the  
1393 version of SOAP used in the envelope. Otherwise, the recipient SHOULD fault.

1394 For this binding, the following actions are defined to enable specific processing context to be conveyed to  
1395 the recipient:

1396

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

1397

1398

1399

1400 For this binding, the `<wst:RequestType>` element contains the following URI:

1401

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

1402

1403 The request provides a token upon which the request is based and OPTIONAL tokens. As well, the  
1404 OPTIONAL `<wst:TokenType>` element in the request can indicate desired type response token. This  
1405 MAY be any supported token type or it MAY be the following URI indicating that only status is desired:

1406

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

1407

1408 For some use cases a status token is returned indicating the success or failure of the validation. In other  
1409 cases a security token MAY be returned and used for authorization. This binding assumes that the  
1410 validation requestor and provider are known to each other and that the general issuance parameters  
1411 beyond requesting a token type, which is OPTIONAL, are not needed (note that other bindings and  
1412 profiles could define different semantics).

1413

1414 For this binding an applicability scope (e.g., `<wsp:AppliesTo>`) need not be specified. It is assumed  
1415 that the applicability of the validation response relates to the provided information (e.g. security token) as  
1416 understood by the issuing service.

1417

1418 The validation binding does not allow the use of exchanges.

1419

1420 The RSTR for this binding carries the following element even if a token is returned (note that the base  
1421 elements described above are included here italicized for completeness):

1422

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

1423

1424

1425

1426

1427

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

1428

1429

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

1430

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

1431

```
<wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
```

1432

```
...
```

1433

```
<wst:Status>
```

1434

```
<wst:Code>...</wst:Code>
```

1435

```
<wst:Reason>...</wst:Reason>
```

1436

```
</wst:Status>
```

1437

```
</wst:RequestSecurityTokenResponse>
```

1438

1439 */wst:RequestSecurityToken/wst:ValidateTarget*

1440 This REQUIRED element identifies the token being validated. Typically this contains a  
1441 `<wsse:SecurityTokenReference>` pointing at the token, but could also carry the token  
1442 directly.

1443 */wst:RequestSecurityTokenResponse/wst:Status*

1444 When a validation request is made, this element MUST be in the response. The code value  
1445 indicates the results of the validation in a machine-readable form. The accompanying text  
1446 element allows for human textual display.

1447 */wst:RequestSecurityTokenResponse/wst:Status/wst:Code*

1448 This REQUIRED URI value provides a machine-readable status code. The following URIs are  
1449 predefined, but others MAY be used.

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

1450 */wst:RequestSecurityTokenResponse/wst:Status/wst:Reason*

1451 This OPTIONAL string provides human-readable text relating to the status code.

1452

1453 The following illustrates the syntax of a validation request and response. In this example no token is  
1454 requested, just a status.

1455

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

1456

```
<wst:TokenType>
```

1457

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

1458

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

1459

```
<wst:RequestType>
```

1460

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

1461

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

1462

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

1463

1464

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

1465

```
<wst:TokenType>
```

1466

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

1467

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

```
1468     <wst:Status>
1469         <wst:Code>
1470             http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
1471         </wst:Code>
1472     </wst:Status>
1473     ...
1474 </wst:RequestSecurityTokenResponse>
```

1475 The following illustrates the syntax of a validation request and response. In this example a custom token  
1476 is requested indicating authorized rights in addition to the status.

```
1477 <wst:RequestSecurityToken xmlns:wst="...">
1478     <wst:TokenType>
1479         http://example.org/mySpecialToken
1480     </wst:TokenType>
1481     <wst:RequestType>
1482         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate
1483     </wst:RequestType>
1484 </wst:RequestSecurityToken>
```

```
1485
1486 <wst:RequestSecurityTokenResponse xmlns:wst="...">
1487     <wst:TokenType>
1488         http://example.org/mySpecialToken
1489     </wst:TokenType>
1490     <wst:Status>
1491         <wst:Code>
1492             http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
1493         </wst:Code>
1494     </wst:Status>
1495     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1496     ...
1497 </wst:RequestSecurityTokenResponse>
```

---

## 8 Negotiation and Challenge Extensions

1498

1499 The general security token service framework defined above allows for a simple request and response for  
1500 security tokens (possibly asynchronous). However, there are many scenarios where a set of exchanges  
1501 between the parties is REQUIRED prior to returning (e.g., issuing) a security token. This section  
1502 describes the extensions to the base WS-Trust mechanisms to enable exchanges for negotiation and  
1503 challenges.

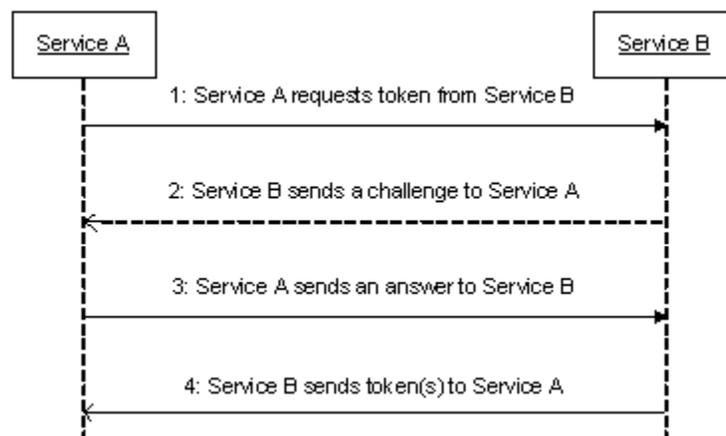
1504

1505 There are potentially different forms of exchanges, but one specific form, called "challenges", provides  
1506 mechanisms in addition to those described in [WS-Security] for authentication. This section describes  
1507 how general exchanges are issued and responded to within this framework. Other types of exchanges  
1508 include, but are not limited to, negotiation, tunneling of hardware-based processing, and tunneling of  
1509 legacy protocols.

1510

1511 The process is straightforward (illustrated here using a challenge):

1512



1513

- 1514 1. A requestor sends, for example, a `<wst:RequestSecurityToken>` message with a  
1515 timestamp.
- 1516 2. The recipient does not trust the timestamp and issues a  
1517 `<wst:RequestSecurityTokenResponse>` message with an embedded challenge.
- 1518 3. The requestor sends a `<wst:RequestSecurityTokenResponse>` message with an answer to  
1519 the challenge.
- 1520 4. The recipient issues a `<wst:RequestSecurityTokenResponseCollection>` message with  
1521 the issued security token and OPTIONAL proof-of-possession token.

1522

1523 It should be noted that the requestor might challenge the recipient in either step 1 or step 3. In which  
1524 case, step 2 or step 4 contains an answer to the initiator's challenge. Similarly, it is possible that steps 2  
1525 and 3 could iterate multiple times before the process completes (step 4).

1526

1527 The two services can use [WS-SecurityPolicy] to state their requirements and preferences for security  
1528 tokens and encryption and signing algorithms (general policy intersection). This section defines  
1529 mechanisms for legacy and more sophisticated types of negotiations.

## 1530 8.1 Negotiation and Challenge Framework

1531 The general mechanisms defined for requesting and returning security tokens are extensible. This  
1532 section describes the general model for extending these to support negotiations and challenges.

1533

1534 The exchange model is as follows:

- 1535 1. A request is initiated with a `<wst:RequestSecurityToken>` that identifies the details of the  
1536 request (and MAY contain initial negotiation/challenge information)
- 1537 2. A response is returned with a `<wst:RequestSecurityTokenResponse>` that contains  
1538 additional negotiation/challenge information. Optionally, this MAY return token information in the  
1539 form of a `<wst:RequestSecurityTokenResponseCollection>` (if the exchange is two legs  
1540 long).
- 1541 3. If the exchange is not complete, the requestor uses a  
1542 `<wst:RequestSecurityTokenResponse>` that contains additional negotiation/challenge  
1543 information.
- 1544 4. The process repeats at step 2 until the negotiation/challenge is complete (a token is returned or a  
1545 Fault occurs). In the case where token information is returned in the final leg, it is returned in the  
1546 form of a `<wst:RequestSecurityTokenResponseCollection>`.

1547

1548 The negotiation/challenge information is passed in binding/profile-specific elements that are placed inside  
1549 of the `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>` elements.

1550

1551 It is RECOMMENDED that at least the `<wsu:Timestamp>` element be included in messages (as per  
1552 [\[WS-Security\]](#)) as a way to ensure freshness of the messages in the exchange. Other types of  
1553 challenges MAY also be included. For example, a `<wsp:Policy>` element may be used to negotiate  
1554 desired policy behaviors of both parties. Multiple challenges and responses MAY be included.

## 1555 8.2 Signature Challenges

1556 Exchange requests are issued by including an element that describes the exchange (e.g. challenge) and  
1557 responses contain an element describing the response. For example, signature challenges are  
1558 processed using the `<wst:SignChallenge>` element. The response is returned in a  
1559 `<wst:SignChallengeResponse>` element. Both the challenge and the response elements are  
1560 specified within the `<wst:RequestSecurityTokenResponse>` element. Some forms of negotiation  
1561 MAY specify challenges along with responses to challenges from the other party. It should be noted that  
1562 the requestor MAY provide exchange information (e.g. a challenge) to the recipient in the initial request.  
1563 Consequently, these elements are also allowed within a `<wst:RequestSecurityToken>` element.

1564

1565 The syntax of these elements is as follows:

```
1566 <wst:SignChallenge xmlns:wst="...">  
1567   <wst:Challenge ...>...</wst:Challenge>  
1568 </wst:SignChallenge>
```

1569

```
1570 <wst:SignChallengeResponse xmlns:wst="...">  
1571   <wst:Challenge ...>...</wst:Challenge>  
1572 </wst:SignChallengeResponse>
```

1573

1574 The following describes the attributes and tags listed in the schema above:

1575 *.../wst:SignChallenge*

1576 This OPTIONAL element describes a challenge that requires the other party to sign a specified  
1577 set of information.

1578 *.../wst:SignChallenge/wst:Challenge*

1579 This REQUIRED string element describes the value to be signed. In order to prevent certain  
1580 types of attacks (such as man-in-the-middle), it is strongly RECOMMENDED that the challenge  
1581 be bound to the negotiation. For example, the challenge SHOULD track (such as using a digest  
1582 of) any relevant data exchanged such as policies, tokens, replay protection, etc. As well, if the  
1583 challenge is happening over a secured channel, a reference to the channel SHOULD also be  
1584 included. Furthermore, the recipient of a challenge SHOULD verify that the data tracked  
1585 (digested) matches their view of the data exchanged. The exact algorithm MAY be defined in  
1586 profiles or agreed to by the parties.

1587 *.../SignChallenge/{any}*

1588 This is an extensibility mechanism to allow additional negotiation types to be used.

1589 *.../wst:SignChallenge/@{any}*

1590 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added  
1591 to the element.

1592 *.../wst:SignChallengeResponse*

1593 This OPTIONAL element describes a response to a challenge that requires the signing of a  
1594 specified set of information.

1595 *.../wst:SignChallengeResponse/wst:Challenge*

1596 If a challenge was issued, the response MUST contain the challenge element exactly as  
1597 received. As well, while the RSTR response SHOULD always be signed, if a challenge was  
1598 issued, the RSTR MUST be signed (and the signature coupled with the message to prevent  
1599 replay).

1600 *.../wst:SignChallengeResponse/{any}*

1601 This is an extensibility mechanism to allow additional negotiation types to be used.

1602 *.../wst:SignChallengeResponse/@{any}*

1603 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added  
1604 to the element.

### 1605 **8.3 User Interaction Challenge**

1606 User interaction challenge requests are issued by including the <InteractiveChallenge> element. The  
1607 response is returned in a <InteractiveChallengeResponse> element. Both the challenge and response  
1608 elements are specified within the <wst:RequestSecurityTokenResponse> element. In some instances, the  
1609 requestor may issue a challenge to the recipient or provide a response to an anticipated challenge from  
1610 the recipient in the initial request. Consequently, these elements are also allowed within a  
1611 <wst:RequestSecurityToken> element. The challenge/response exchange between client and server  
1612 MAY be iterated over multiple legs before a final response is issued.

1613 Implementations SHOULD take into account the possibility that messages in either direction may be lost  
1614 or duplicated. In the absence of a lower level protocol guaranteeing delivery of every message in order  
1615 and exactly once, which retains the ordering of requests and responses traveling in opposite directions,  
1616 implementations SHOULD observe the following procedures:

1617 The STS SHOULD:

1618 1. Never send a new request while an existing request is pending,

- 1619 2. Timeout requests and retransmit them.
- 1620 3. Silently discard responses when no request is pending.
- 1621
- 1622 The service consumer MAY:
- 1623 1. Respond to a repeated request with the same information
- 1624 2. Retain user input until the Challenge Iteration is complete in case it is necessary to repeat the
- 1625 response.
- 1626 Note that the xml:lang attribute may be used where allowed via attribute extensibility to specify a
- 1627 language of localized elements and attributes using the language codes specified in [RFC 3066].

### 1628 8.3.1 Challenge Format

1629 The syntax of the user interaction challenge element is as follows:

```

1630 <wst14:InteractiveChallenge xmlns:wst14="..." ...>
1631   <wst14:Title ...> xs:string </wst14:Title> ?
1632   <wst14:TextChallenge RefId="xs:anyURI" Label="xs:string"?
1633     MaxLen="xs:int"? HideText="xs:boolean"? ...>
1634     <wst14:Image MimeType="xs:string"> xs:base64Binary </wst14:Image> ?
1635   </wst14:TextChallenge> *
1636   <wst14:ChoiceChallenge RefId="xs:anyURI" Label="xs:string"?
1637     ExactlyOne="xs:boolean"? ...>
1638     <wst14:Choice RefId="xs:anyURI" Label="xs:string"? ...>
1639       <wst14:Image MimeType="xs:string"> xs:base64Binary </wst14:Image> ?
1640     </wst14:Choice> +
1641   </wst14:ChoiceChallenge> *
1642   < wst14:ContextData RefId="xs:anyURI"> xs:any </wst14:ContextData> *
1643   ...
1644 </wst14:InteractiveChallenge>

```

1645 The following describes the attributes and elements listed in the schema outlined above:

- 1646
- 1647 *.../wst14:InteractiveChallenge*
- 1648 A container element for a challenge that requires interactive user input.
- 1649 *.../wst14:InteractiveChallenge/wst14:Title*
- 1650 An OPTIONAL element that specifies an overall title text to be displayed to the user (e.g. a title
- 1651 describing the purpose or nature of the challenge). How the preferred language of the requestor
- 1652 is communicated to the STS is left up to implementations.
- 1653 *.../wst14:InteractiveChallenge/wst14:TextChallenge*
- 1654 An OPTIONAL element that specifies a challenge that requires textual input from the user.
- 1655 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@RefId*
- 1656 A REQUIRED attribute that specifies a reference identifier for this challenge element which is
- 1657 used to correlate the corresponding element in the response to the challenge.
- 1658 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@MaxLen*
- 1659 An OPTIONAL attribute that specifies the maximum length of the text string that is sent as the
- 1660 response to this text challenge. This value serves as a hint for the user interface software at the
- 1661 requestor which manifests the end-user experience for this challenge.
- 1662 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@HideText*
- 1663 An OPTIONAL attribute that specifies that the response to this text challenge MUST receive
- 1664 treatment as hidden text in any user interface. For example, the text entry may be displayed as a

1665 series of asterisks in the user interface. This attribute serves as a hint for the user interface  
1666 software at the requestor which manifests the end-user experience for this challenge.

1667 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@Label*

1668 An OPTIONAL attribute that specifies a label for the text challenge item (e.g. a label for a text  
1669 entry field) which will be shown to the user. How the preferred language of the requestor is  
1670 communicated to the STS is left up to implementations.

1671 *.../wst14:InteractiveChallenge/wst14:TextChallenge/Image*

1672 An OPTIONAL element that contains a base64 encoded inline image specific to the text  
1673 challenge item to be shown to the user (e.g. an image that the user must see to respond  
1674 successfully to the challenge). The image presented is intended as an additional label to a  
1675 challenge element which could be CAPTCHA, selection of a previously established image secret  
1676 or any other means by which images can be used to challenge a user to interact in a way to  
1677 satisfy a challenge.

1678 *.../wst14:InteractiveChallenge/wst14:TextChallenge/Image/@MimeType*

1679 A REQUIRED attribute that specifies a MIME type (e.g., image/gif, image/jpg) indicating the  
1680 format of the image.

1681 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge*

1682 An OPTIONAL element that specifies a challenge that requires a choice among multiple items by  
1683 the user.

1684 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/@RefId*

1685 A REQUIRED attribute that specifies a reference identifier for this challenge element which is  
1686 used to correlate the corresponding element in the response to the challenge.

1687 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/@Label*

1688 An OPTIONAL attribute that specifies a title label for the choice challenge item (e.g., a text  
1689 header describing the list of choices as a whole) which will be shown to the user. How the  
1690 preferred language of the requestor is communicated to the STS is left up to implementations.

1691 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/@ExactlyOne*

1692 An OPTIONAL attribute that specifies if exactly once choice must be selected by the user from  
1693 among the child element choices. The absence of this attribute implies the value "false" which  
1694 means multiple choices can be selected.

1695 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice*

1696 A REQUIRED element that specifies a single choice item within the choice challenge.

1697 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/@RefId*

1698 A REQUIRED attribute that specifies a reference identifier for this specific choice item which is  
1699 used to correlate the corresponding element in the response to the challenge.

1700 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/@Label*

1701 An OPTIONAL attribute that specifies a text label for the choice item (e.g., text describing the  
1702 individual choice) which will be shown to the user. How the preferred language of the requestor is  
1703 communicated to the STS is left up to implementations.

1704 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/wst14:Image*

1705 An OPTIONAL element that contains a base64 encoded inline image specific to the choice item  
1706 to be shown to the user (e.g. an image that the user must see to respond successfully to the  
1707 challenge). The image presented is intended as an additional label to a challenge element which  
1708 could be CAPTCHA, selection of a previously established image secret or any other means by  
1709 which images can be used to challenge a user to interact in a way to satisfy a challenge.

1710 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/wst14:Image/@MimeType*  
1711 A REQUIRED attribute that specifies a MIME type (e.g., image/gif, image/jpg) indicating the  
1712 format of the image.

1713 *.../wst14:InteractiveChallenge/wst14:ContextData*  
1714 An OPTIONAL element that specifies a value that MUST be reflected back in the response to the  
1715 challenge (e.g., cookie). The element may contain any value. The actual content is opaque to the  
1716 requestor; it is not required to understand its structure or semantics. This can be used by an STS,  
1717 for instance, to store information between the challenge/response exchanges that would  
1718 otherwise be lost if the STS were to remain stateless.

1719 *.../wst14:InteractiveChallenge/wst14:ContextData/@RefId*  
1720 A REQUIRED attribute that specifies a reference identifier for this context element which is used  
1721 to correlate the corresponding element in the response to the challenge.

1722 *.../wst14:InteractiveChallenge/{any}*  
1723 This is an extensibility mechanism to allow additional elements to be specified.

1724 *.../wst14:InteractiveChallenge/@{any}*  
1725 This is an extensibility mechanism to allow additional attributes to be specified.

1726

1727 The syntax of the user interaction challenge response element is as follows:

```
1728 <wst14:InteractiveChallengeResponse xmlns:wst14="..." ...>  
1729   <wst14:TextChallengeResponse RefId="xs:anyURI" ...>  
1730     xs:string  
1731   </wst14:TextChallengeResponse> *  
1732   <wst14:ChoiceChallengeResponse RefId="xs:anyURI"> *  
1733     <wst14:ChoiceSelected RefId="xs:anyURI" /> *  
1734   </wst14:ChoiceChallengeResponse>  
1735   <wst14:ContextData RefId="xs:anyURI"> xs:any </wst14:ContextData> *  
1736   ...  
1737 </wst14:InteractiveChallengeResponse>
```

1738 The following describes the attributes and elements listed in the schema outlined above:

1739

1740 *.../wst14:InteractiveChallengeResponse*

1741 A container element for the response to a challenge that requires interactive user input.

1742 *.../wst14:InteractiveChallengeResponse/wst14:TextChallengeResponse*

1743 This element value contains the user input as the response to the original text challenge issued.

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

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

1747 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse*

1748 A container element for the response to a choice challenge.

1749 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse/@RefId*

1750 A required attribute that specifies the reference identifier for the choice challenge element in the  
1751 original challenge which can be used for correlation.

1752 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse/wst14:ChoiceSelected*

1753 A required element that specifies a choice item selected by the user from the choice challenge.

1754 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse/wst14:ChoiceSelected/@RefId*

1755 A required attribute that specifies the reference identifier for the choice item in the original choice  
1756 challenge which can be used for correlation.

1757 *.../wst14:InteractiveChallengeResponse/wst14:ContextData*

1758 An optional element that carries a context data item from the original challenge that is simply  
1759 reflected back.

1760 *.../wst14:InteractiveChallengeResponse/wst14:ContextData/@RefId*

1761 A required attribute that specifies the reference identifier for the context data element in the  
1762 original challenge which can be used for correlation.

1763 *.../wst14:InteractiveChallengeResponse/{any}*

1764 This is an extensibility mechanism to allow additional elements to be specified.

1765 *.../wst14:InteractiveChallengeResponse/@{any}*

1766 This is an extensibility mechanism to allow additional attributes to be specified.

1767 In order to prevent certain types of attacks, such as man-in-the-middle or replay of response, the  
1768 challenge SHOULD be bound to the response. For example, an STS may use the <ContextData>  
1769 element in the challenge to include a digest of any relevant replay protection data and verify that the  
1770 same data is reflected back by the requestor.

1771 Text provided by the STS which is intended for display SHOULD NOT contain script, markup or other  
1772 unprintable characters. Image data provided by the STS SHOULD NOT contain imbedded commands or  
1773 other content except an image to be displayed.

1774 Service consumers MUST ignore any script, markup or other unprintable characters when displaying text  
1775 sent by the STS. Service consumers MUST insure that image data does not contain imbedded  
1776 commands or other content before displaying the image.

### 1777 **8.3.2 PIN and OTP Challenges**

1778 In some situations, some additional authentication step may be required, but the Consumer cannot  
1779 determine this in advance of making the request. Two common cases that require user interaction are:

- 1780 • a challenge for a secret PIN,
- 1781 • a challenge for a one-time-password (OTP).

1782  
1783 This challenge may be issued by an STS using the “text challenge” format within a user interaction  
1784 challenge specified in the section above. A requestor responds to the challenge with the PIN/OTP value  
1785 along with the corresponding @RefId attribute value for the text challenge which is used by the STS to  
1786 correlate the response to the original challenge. This pattern of exchange requires that the requestor  
1787 must receive the challenge first and thus learn the @RefId attribute value to include in the response.

1788  
1789 There are cases where a requestor may know a priori that the STS challenges for a single PIN/OTP and,  
1790 as an optimization, provide the response to the anticipated challenge in the initial request. The following  
1791 distinguished URIs are defined for use as the value of the @RefId attribute of a  
1792 <TextChallengeResponse> element to represent PIN and OTP responses using the optimization pattern.

1793

1794 <http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN>  
1795 <http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/OTP>

1796

1797 An STS may choose not to support the optimization pattern above for PIN/OTP response. In some cases,  
1798 an OTP challenge from the STS may include a dynamic random value that the requestor must feed into  
1799 the OTP generating module before an OTP response is computed. In such cases, the optimized response  
1800 pattern may not be usable.

## 1801 8.4 Binary Exchanges and Negotiations

1802 Exchange requests MAY also utilize existing binary formats passed within the WS-Trust framework. A  
1803 generic mechanism is provided for this that includes a URI attribute to indicate the type of binary  
1804 exchange.

1805

1806 The syntax of this element is as follows:

```
1807 <wst:BinaryExchange ValueType="..." EncodingType="..." xmlns:wst="...">  
1808 </wst:BinaryExchange>
```

1809 The following describes the attributes and tags listed in the schema above (note that the ellipses below  
1810 indicate that this element MAY be placed in different containers. For this specification, these are limited  
1811 to <wst:RequestSecurityToken> and <wst:RequestSecurityTokenResponse>):

1812 *.../wst:BinaryExchange*

1813 This OPTIONAL element is used for a security negotiation that involves exchanging binary blobs  
1814 as part of an existing negotiation protocol. The contents of this element are blob-type-specific  
1815 and are encoded using base64 (unless otherwise specified).

1816 *.../wst:BinaryExchange/@ValueType*

1817 This REQUIRED attribute specifies a URI to identify the type of negotiation (and the value space  
1818 of the blob – the element's contents).

1819 *.../wst:BinaryExchange/@EncodingType*

1820 This REQUIRED attribute specifies a URI to identify the encoding format (if different from base64)  
1821 of the negotiation blob. Refer to [WS-Security] for sample encoding format URIs.

1822 *.../wst:BinaryExchange/@{any}*

1823 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added  
1824 to the element.

1825 Some binary exchanges result in a shared state/context between the involved parties. It is  
1826 RECOMMENDED that at the conclusion of the exchange, a new token and proof-of-possession token be  
1827 returned. A common approach is to use the negotiated key as a "secure channel" mechanism to secure  
1828 the new token and proof-of-possession token.

1829 For example, an exchange might establish a shared secret *S<sub>x</sub>* that can then be used to sign the final  
1830 response and encrypt the proof-of-possession token.

## 1831 8.5 Key Exchange Tokens

1832 In some cases it MAY be necessary to provide a key exchange token so that the other party (either  
1833 requestor or issuer) can provide entropy or key material as part of the exchange. Challenges MAY NOT  
1834 always provide a usable key as the signature may use a signing-only certificate.

1835

1836 The section describes two OPTIONAL elements that can be included in RST and RSTR elements to  
1837 indicate that a Key Exchange Token (KET) is desired, or to provide a KET.

1838 The syntax of these elements is as follows (Note that the ellipses below indicate that this element MAY be  
1839 placed in different containers. For this specification, these are limited to

1840 <wst:RequestSecurityToken> and <wst:RequestSecurityTokenResponse>):

```
1841 <wst:RequestKET xmlns:wst="..." />
```

1842

1843

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

1844

1845 The following describes the attributes and tags listed in the schema above:

1846 *.../wst:RequestKET*

1847 This OPTIONAL element is used to indicate that the receiving party (either the original requestor  
1848 or issuer) SHOULD provide a KET to the other party on the next leg of the exchange.

1849 *.../wst:KeyExchangeToken*

1850 This OPTIONAL element is used to provide a key exchange token. The contents of this element  
1851 either contain the security token to be used for key exchange or a reference to it.

## 1852 8.6 Custom Exchanges

1853 Using the extensibility model described in this specification, any custom XML-based exchange can be  
1854 defined in a separate binding/profile document. In such cases elements are defined which are carried in  
1855 the RST and RSTR elements.

1856

1857 It should be noted that it is NOT REQUIRED that exchange elements be symmetric. That is, a specific  
1858 exchange mechanism MAY use multiple elements at different times, depending on the state of the  
1859 exchange.

## 1860 8.7 Signature Challenge Example

1861 Here is an example exchange involving a signature challenge. In this example, a service requests a  
1862 custom token using a X.509 certificate for authentication. The issuer uses the exchange mechanism to  
1863 challenge the requestor to sign a random value (to ensure message freshness). The requestor provides  
1864 a signature of the requested data and, once validated, the issuer then issues the requested token.

1865

1866 The first message illustrates the initial request that is signed with the private key associated with the  
1867 requestor's X.509 certificate:

1868

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..."  
1869   xmlns:wsu="..." xmlns:wst="...">  
1870   <S11:Header>  
1871     ...  
1872     <wsse:Security>  
1873       <wsse:BinarySecurityToken  
1874         wsu:Id="reqToken"  
1875         ValueType="...X509v3">  
1876         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
1877       </wsse:BinarySecurityToken>  
1878       <ds:Signature xmlns:ds="...">  
1879         ...  
1880       <ds:KeyInfo>  
1881         <wsse:SecurityTokenReference>  
1882           <wsse:Reference URI="#reqToken"/>  
1883         </wsse:SecurityTokenReference>  
1884       </ds:KeyInfo>  
1885     </ds:Signature>  
1886   </wsse:Security>  
1887   ...  
1888 </S11:Header>  
1889 <S11:Body>  
1890   <wst:RequestSecurityToken>  
1891     <wst:TokenType>
```

```

1892         http://example.org/mySpecialToken
1893     </wst:TokenType>
1894     <wst:RequestType>
1895         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1896     </wst:RequestType>
1897 </wst:RequestSecurityToken>
1898 </S11:Body>
1899 </S11:Envelope>

```

1900

1901 The issuer (recipient) service doesn't trust the sender's timestamp (or one wasn't specified) and issues a

1902 challenge using the exchange framework defined in this specification. This message is signed using the

1903 private key associated with the issuer's X.509 certificate and contains a random challenge that the

1904 requestor must sign:

```

1905 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsp="..."
1906     xmlns:wst="...">
1907   <S11:Header>
1908     ...
1909     <wsse:Security>
1910       <wsse:BinarySecurityToken
1911         wsu:Id="issuerToken"
1912         ValueType="...X509v3">
1913         DFJHuedsujfnrnv45JZc0...
1914       </wsse:BinarySecurityToken>
1915       <ds:Signature xmlns:ds="...">
1916         ...
1917       </ds:Signature>
1918     </wsse:Security>
1919     ...
1920   </S11:Header>
1921   <S11:Body>
1922     <wst:RequestSecurityTokenResponse>
1923       <wst:SignChallenge>
1924         <wst:Challenge>Huehf...</wst:Challenge>
1925       </wst:SignChallenge>
1926     </wst:RequestSecurityTokenResponse>
1927   </S11:Body>
1928 </S11:Envelope>

```

1929

1930 The requestor receives the issuer's challenge and issues a response that is signed using the requestor's

1931 X.509 certificate and contains the challenge. The signature only covers the non-mutable elements of the

1932 message to prevent certain types of security attacks:

```

1933 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsp="..."
1934     xmlns:wst="...">
1935   <S11:Header>
1936     ...
1937     <wsse:Security>
1938       <wsse:BinarySecurityToken
1939         wsu:Id="reqToken"
1940         ValueType="...X509v3">
1941         MIEZzCCA9CgAwIBAgIQEmtJZc0...
1942       </wsse:BinarySecurityToken>
1943       <ds:Signature xmlns:ds="...">
1944         ...
1945       </ds:Signature>
1946     </wsse:Security>
1947     ...
1948   </S11:Header>
1949   <S11:Body>
1950     <wst:RequestSecurityTokenResponse>

```

```

1951         <wst:SignChallengeResponse>
1952             <wst:Challenge>Huehf...</wst:Challenge>
1953         </wst:SignChallengeResponse>
1954     </wst:RequestSecurityTokenResponse>
1955 </S11:Body>
1956 </S11:Envelope>

```

1957

1958 The issuer validates the requestor's signature responding to the challenge and issues the requested

1959 token(s) and the associated proof-of-possession token. The proof-of-possession token is encrypted for

1960 the requestor using the requestor's public key.

```

1961 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
1962     xmlns:wst="..." xmlns:xenc="...">
1963     <S11:Header>
1964         ...
1965         <wsse:Security>
1966             <wsse:BinarySecurityToken
1967                 wsu:Id="issuerToken"
1968                 ValueType="...X509v3">
1969                 DFJHuedsujfnrnv45JZc0...
1970             </wsse:BinarySecurityToken>
1971             <ds:Signature xmlns:ds="...">
1972                 ...
1973             </ds:Signature>
1974         </wsse:Security>
1975         ...
1976     </S11:Header>
1977     <S11:Body>
1978         <wst:RequestSecurityTokenResponseCollection>
1979             <wst:RequestSecurityTokenResponse>
1980                 <wst:RequestedSecurityToken>
1981                     <xyz:CustomToken xmlns:xyz="...">
1982                         ...
1983                     </xyz:CustomToken>
1984                 </wst:RequestedSecurityToken>
1985                 <wst:RequestedProofToken>
1986                     <xenc:EncryptedKey Id="newProof">
1987                         ...
1988                     </xenc:EncryptedKey>
1989                 </wst:RequestedProofToken>
1990             </wst:RequestSecurityTokenResponse>
1991         </wst:RequestSecurityTokenResponseCollection>
1992     </S11:Body>
1993 </S11:Envelope>

```

## 1994 8.8 Challenge Examples

### 1995 8.8.1 Text and choice challenge

1996 Here is an example of a user interaction challenge using both text and choice challenges. In this example,

1997 a user requests a custom token using a username/password for authentication. The STS uses the

1998 challenge mechanism to challenge the user for additional information in the form of a secret question (i.e.,

1999 Mother's maiden name) and an age group choice. The challenge additionally includes one contextual

2000 data item that needs to be reflected back in the response. The user interactively provides the requested

2001 data and, once validated, the STS issues the requested token. All messages are sent over a protected

2002 transport using SSLv3.

2003

2004 The requestor sends the initial request that includes the username/password for authentication as follows.

2005

```

2006 <S11:Envelope ...>
2007   <S11:Header>
2008     ...
2009     <wsse:Security>
2010       <wsse:UsernameToken>
2011         <wsse:Username>Zoe</wsse:Username>
2012         <wsse:Password
2013           Type="http://...#PasswordText">ILoveDogs</wsse:Password>
2014       </wsse:UsernameToken>
2015     </wsse:Security>
2016   </S11:Header>
2017   <S11:Body>
2018     <wst:RequestSecurityToken>
2019       <wst:TokenType>http://example.org/customToken</wst:TokenType>
2020       <wst:RequestType>...</wst:RequestType>
2021     </wst:RequestSecurityToken>
2022   </S11:Body>
2023 </S11:Envelope>

```

2024

2025 The STS issues a challenge for additional information using the user interaction challenge mechanism as

2026 follows.

```

2027
2028 <S11:Envelope ...>
2029   <S11:Header>
2030     ...
2031   </S11:Header>
2032   <S11:Body>
2033     <wst:RequestSecurityTokenResponse>
2034       <wst14:InteractiveChallenge xmlns:wst14="..." >
2035         <wst14:Title>
2036           Please answer the following additional questions to login.
2037         </wst14:Title>
2038         <wst14:TextChallenge RefId=http://.../ref#text1
2039           Label="Mother's Maiden Name" MaxLen=80 />
2040         <wst14:ChoiceChallenge RefId="http://.../ref#choiceGroupA"
2041           Label="Your Age Group:" ExactlyOne="true">
2042           <wst14:Choice RefId="http://.../ref#choice1" Label="18-30" />
2043           <wst14:Choice RefId="http://.../ref#choice2" Label="31-40" />
2044           <wst14:Choice RefId="http://.../ref#choice3" Label="41-50" />
2045           <wst14:Choice RefId="http://.../ref#choice4" Label="50+" />
2046         </wst14:ChoiceChallenge>
2047         <wst14:ContextData RefId="http://.../ref#cookie1">
2048           ...
2049         </wst14:ContextData>
2050       </wst14:InteractiveChallenge>
2051     </wst:RequestSecurityTokenResponse>
2052   </S11:Body>
2053 </S11:Envelope>

```

2054

2055 The requestor receives the challenge, provides the necessary user experience for soliciting the required

2056 inputs, and sends a response to the challenge back to the STS as follows.

```

2057
2058 <S11:Envelope ...>
2059   <S11:Header>
2060     ...
2061   </S11:Header>
2062   <S11:Body>
2063     <wst:RequestSecurityTokenResponse>
2064       <wst14:InteractiveChallengeResponse xmlns:wst14="..." >

```

```

2065     <wst14:TextChallengeResponse RefId="http://.../ref#text1">
2066         Goldstein
2067     </wst14:TextChallengeResponse>
2068     <wst14:ChoiceChallengeResponse RefId="http://.../ref#choiceGroupA">
2069         <wst14:ChoiceSelected RefId="http://.../ref#choice3" />
2070     </wst14:ChoiceChallengeResponse>
2071     <wst14:ContextData RefId="http://.../ref#cookie1">
2072         ...
2073     </wst14:ContextData>
2074     </wst14:InteractiveChallengeResponse>
2075 </wst:RequestSecurityTokenResponse>
2076 </S11:Body>
2077 </S11:Envelope>

```

2078

2079 The STS validates the response containing the inputs from the user, and issues the requested token as follows.

2080

2081

```

2082 <S11:Envelope ...>
2083   <S11:Header>
2084     ...
2085   </S11:Header>
2086   <S11:Body>
2087     <wst:RequestSecurityTokenResponseCollection>
2088       <wst:RequestSecurityTokenResponse>
2089         <wst:RequestedSecurityToken>
2090           <xyz:CustomToken xmlns:xyz="...">
2091             ...
2092           </xyz:CustomToken>
2093         </wst:RequestedSecurityToken>
2094         <wst:RequestedProofToken>
2095           ...
2096         </wst:RequestedProofToken>
2097       </wst:RequestSecurityTokenResponse>
2098     </wst:RequestSecurityTokenResponseCollection>
2099   </S11:Body>
2100 </S11:Envelope>

```

2101

## 2102 8.8.2 PIN challenge

2103 Here is an example of a user interaction challenge using a text challenge for a secret PIN. In this  
 2104 example, a user requests a custom token using a username/password for authentication. The STS uses  
 2105 the text challenge mechanism for an additional PIN. The user interactively provides the PIN and, once  
 2106 validated, the STS issues the requested token. All messages are sent over a protected transport using  
 2107 SSLv3.

2108

2109 The requestor sends the initial request that includes the username/password for authentication as follows.

2110

```

2111 <S11:Envelope ...>
2112   <S11:Header>
2113     ...
2114   <wsse:Security>
2115     <wsse:UsernameToken>
2116       <wsse:Username>Zoe</wsse:Username>
2117       <wsse:Password Type="http://...#PasswordText">
2118         ILoveDogs
2119     </wsse:Password>

```

```

2120     </wsse:UsernameToken>
2121     </wsse:Security>
2122 </S11:Header>
2123 <S11:Body>
2124     <wst:RequestSecurityToken>
2125         <wst:TokenType>http://example.org/customToken</wst:TokenType>
2126         <wst:RequestType>...</wst:RequestType>
2127     </wst:RequestSecurityToken>
2128 </S11:Body>
2129 </S11:Envelope>

```

2130  
2131 The STS issues a challenge for a secret PIN using the text challenge mechanism as follows.  
2132

```

2133 <S11:Envelope ...>
2134   <S11:Header>
2135     ...
2136   </S11:Header>
2137   <S11:Body>
2138     <wst:RequestSecurityTokenResponse>
2139       <wst14:InteractiveChallenge xmlns:wst14="..." >
2140         <wst14:TextChallenge
2141           RefId="http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN"
2142           Label="Please enter your PIN" />
2143         </wst14:TextChallenge>
2144       </wst14:InteractiveChallenge>
2145     </wst:RequestSecurityTokenResponse>
2146   </S11:Body>
2147 </S11:Envelope>

```

2148  
2149 The requestor receives the challenge, provides the necessary user experience for soliciting the PIN, and  
2150 sends a response to the challenge back to the STS as follows.  
2151

```

2152 <S11:Envelope ...>
2153   <S11:Header>
2154     ...
2155   </S11:Header>
2156   <S11:Body>
2157     <wst:RequestSecurityTokenResponse>
2158       <wst14:InteractiveChallengeResponse xmlns:wst14="..." >
2159         <wst14:TextChallengeResponse
2160           RefId="http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN">
2161           9988
2162         </wst14:TextChallengeResponse>
2163       </wst14:InteractiveChallengeResponse>
2164     </wst:RequestSecurityTokenResponse>
2165   </S11:Body>
2166 </S11:Envelope>

```

2167  
2168 The STS validates the PIN response, and issues the requested token as follows.  
2169

```

2170 <S11:Envelope ...>
2171   <S11:Header>
2172     ...
2173   </S11:Header>
2174   <S11:Body>
2175     <wst:RequestSecurityTokenResponseCollection>

```

```

2176 <wst:RequestSecurityTokenResponse>
2177 <wst:RequestedSecurityToken>
2178 <xyz:CustomToken xmlns:xyz="...">
2179 ...
2180 </xyz:CustomToken>
2181 </wst:RequestedSecurityToken>
2182 <wst:RequestedProofToken>
2183 ...
2184 </wst:RequestedProofToken>
2185 </wst:RequestSecurityTokenResponse>
2186 </wst:RequestSecurityTokenResponseCollection>
2187 </S11:Body>
2188 </S11:Envelope>

```

2189

### 2190 8.8.3 PIN challenge with optimized response

2191 The following example illustrates using the optimized PIN response pattern for the same exact challenge  
 2192 as in the previous section. This reduces the number of message exchanges to two instead of four. All  
 2193 messages are sent over a protected transport using SSLv3.

2194

2195 The requestor sends the initial request that includes the username/password for authentication as well as  
 2196 the response to the anticipated PIN challenge as follows.

2197

```

2198 <S11:Envelope ...>
2199 <S11:Header>
2200 ...
2201 <wsse:Security>
2202 <wsse:UsernameToken>
2203 <wsse:Username>Zoe</wsse:Username>
2204 <wsse:Password Type="http://...#PasswordText">
2205 ILoveDogs
2206 </wsse:Password>
2207 </wsse:UsernameToken>
2208 </wsse:Security>
2209 </S11:Header>
2210 <S11:Body>
2211 <wst:RequestSecurityToken>
2212 <wst:TokenType>http://example.org/customToken</wst:TokenType>
2213 <wst:RequestType>...</wst:RequestType>
2214 <wst14:InteractiveChallengeResponse xmlns:wst14="..." >
2215 <wst14:TextChallengeResponse
2216 RefId="http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN">
2217 9988
2218 </wst14:TextChallengeResponse>
2219 </wst14:InteractiveChallengeResponse>
2220 </wst:RequestSecurityToken>
2221 </S11:Body>
2222 </S11:Envelope>

```

2223

2224 The STS validates the authentication credential as well as the optimized PIN response, and issues the  
 2225 requested token as follows.

2226

```

2227 <S11:Envelope ...>
2228 <S11:Header>
2229 ...
2230 </S11:Header>

```

```

2231 <S11:Body>
2232   <wst:RequestSecurityTokenResponseCollection>
2233     <wst:RequestSecurityTokenResponse>
2234       <wst:RequestedSecurityToken>
2235         <xyz:CustomToken xmlns:xyz="...">
2236           ...
2237         </xyz:CustomToken>
2238       </wst:RequestedSecurityToken>
2239     <wst:RequestedProofToken>
2240       ...
2241     </wst:RequestedProofToken>
2242   </wst:RequestSecurityTokenResponse>
2243 </wst:RequestSecurityTokenResponseCollection>
2244 </S11:Body>
2245 </S11:Envelope>

```

2246

## 2247 8.9 Custom Exchange Example

2248 Here is another illustrating the syntax for a token request using a custom XML exchange. For brevity,  
 2249 only the RST and RSTR elements are illustrated. Note that the framework allows for an arbitrary number  
 2250 of exchanges, although this example illustrates the use of four legs. The request uses a custom  
 2251 exchange element and the requestor signs only the non-mutable element of the message:

```

2252   <wst:RequestSecurityToken xmlns:wst="...">
2253     <wst:TokenType>
2254       http://example.org/mySpecialToken
2255     </wst:TokenType>
2256     <wst:RequestType>
2257       http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
2258     </wst:RequestType>
2259     <xyz:CustomExchange xmlns:xyz="...">
2260       ...
2261     </xyz:CustomExchange>
2262   </wst:RequestSecurityToken>

```

2263

2264 The issuer service (recipient) responds with another leg of the custom exchange and signs the response  
 2265 (non-mutable aspects) with its token:

```

2266   <wst:RequestSecurityTokenResponse xmlns:wst="...">
2267     <xyz:CustomExchange xmlns:xyz="...">
2268       ...
2269     </xyz:CustomExchange>
2270   </wst:RequestSecurityTokenResponse>

```

2271

2272 The requestor receives the issuer's exchange and issues a response that is signed using the requestor's  
 2273 token and continues the custom exchange. The signature covers all non-mutable aspects of the  
 2274 message to prevent certain types of security attacks:

```

2275   <wst:RequestSecurityTokenResponse xmlns:wst="...">
2276     <xyz:CustomExchange xmlns:xyz="...">
2277       ...
2278     </xyz:CustomExchange>
2279   </wst:RequestSecurityTokenResponse>

```

2280

2281 The issuer processes the exchange and determines that the exchange is complete and that a token  
 2282 should be issued. Consequently it issues the requested token(s) and the associated proof-of-possession  
 2283 token. The proof-of-possession token is encrypted for the requestor using the requestor's public key.

```

2284 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
2285   <wst:RequestSecurityTokenResponse>
2286     <wst:RequestedSecurityToken>
2287       <xyz:CustomToken xmlns:xyz="...">
2288         ...
2289       </xyz:CustomToken>
2290     </wst:RequestedSecurityToken>
2291     <wst:RequestedProofToken>
2292       <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
2293         ...
2294       </xenc:EncryptedKey>
2295     </wst:RequestedProofToken>
2296     <wst:RequestedProofToken>
2297       <xenc:EncryptedKey xmlns:xenc="...">...</xenc:EncryptedKey>
2298     </wst:RequestedProofToken>
2299   </wst:RequestSecurityTokenResponse>
2300 </wst:RequestSecurityTokenResponseCollection>
  
```

2301 It should be noted that other example exchanges include the issuer returning a final custom exchange  
 2302 element, and another example where a token isn't returned.

## 2303 8.10 Protecting Exchanges

2304 There are some attacks, such as forms of man-in-the-middle, that can be applied to token requests  
 2305 involving exchanges. It is RECOMMENDED that the exchange sequence be protected. This MAY be  
 2306 built into the exchange messages, but if metadata is provided in the RST or RSTR elements, then it is  
 2307 subject to attack.

2308  
 2309 Consequently, it is RECOMMENDED that keys derived from exchanges be linked cryptographically to the  
 2310 exchange. For example, a hash can be computed by computing the SHA1 of the exclusive  
 2311 canonicalization [XML-C14N] of all RST and RSTR elements in messages exchanged. This value can  
 2312 then be combined with the exchanged secret(s) to create a new master secret that is bound to the data  
 2313 both parties sent/received.

2314  
 2315 To this end, the following computed key algorithm is defined to be OPTIONALLY used in these scenarios:

URI	Meaning
<a href="http://docs.oasis-open.org/ws-sx/ws-trust/200512/CK/HASH">http://docs.oasis-open.org/ws-sx/ws-trust/200512/CK/HASH</a>	The key is computed using P_SHA1 as follows: $H = \text{SHA1}(\text{ExclC14N}(\text{RST} \dots \text{RSTRs}))$ $X = \text{encrypting } H \text{ using negotiated key and mechanism}$ $\text{Key} = \text{P\_SHA1}(X, H + \text{"CK-HASH"})$ The octets for the "CK-HASH" string are the UTF-8 octets.

## 2316 8.11 Authenticating Exchanges

2317 After an exchange both parties have a shared knowledge of a key (or keys) that can then be used to  
 2318 secure messages. However, in some cases it may be desired to have the issuer prove to the requestor

2319 that it knows the key (and that the returned metadata is valid) prior to the requestor using the data.  
2320 However, until the exchange is actually completed it MAY be (and is often) inappropriate to use the  
2321 computed keys. As well, using a token that hasn't been returned to secure a message may complicate  
2322 processing since it crosses the boundary of the exchange and the underlying message security. This  
2323 means that it MAY NOT be appropriate to sign the final leg of the exchange using the key derived from  
2324 the exchange.

2325  
2326 For this reason an authenticator is defined that provides a way for the issuer to verify the hash as part of  
2327 the token issuance. Specifically, when an authenticator is returned, the  
2328 `<wst:RequestSecurityTokenResponseCollection>` element is returned. This contains one  
2329 RSTR with the token being returned as a result of the exchange and a second RSTR that contains the  
2330 authenticator (this order SHOULD be used). When an authenticator is used, RSTRs MUST use the  
2331 `@Context` element so that the authenticator can be correlated to the token issuance. The authenticator is  
2332 separated from the RSTR because otherwise computation of the RST/RSTR hash becomes more  
2333 complex. The authenticator is represented using the `<wst:Authenticator>` element as illustrated  
2334 below:

```
2335 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2336   <wst:RequestSecurityTokenResponse Context="...">  
2337     ...  
2338   </wst:RequestSecurityTokenResponse>  
2339   <wst:RequestSecurityTokenResponse Context="...">  
2340     <wst:Authenticator>  
2341       <wst:CombinedHash>...</wst:CombinedHash>  
2342     ...  
2343     </wst:Authenticator>  
2344   </wst:RequestSecurityTokenResponse>  
2345 </wst:RequestSecurityTokenResponseCollection>
```

2346  
2347 The following describes the attributes and elements listed in the schema overview above (the ... notation  
2348 below represents the path RSTRC/RSTR and is used for brevity):

2349 `.../wst:Authenticator`

2350 This OPTIONAL element provides verification (authentication) of a computed hash.

2351 `.../wst:Authenticator/wst:CombinedHash`

2352 This OPTIONAL element proves the hash and knowledge of the computed key. This is done by  
2353 providing the base64 encoding of the first 256 bits of the P\_SHA1 digest of the computed key and  
2354 the concatenation of the hash determined for the computed key and the string "AUTH-HASH".  
2355 Specifically,  $P\_SHA1(\textit{computed-key}, H + \textit{"AUTH-HASH"})_{0-255}$ . The octets for the "AUTH-HASH"  
2356 string are the UTF-8 octets.

2357  
2358 This `<wst:CombinedHash>` element is OPTIONAL (and an open content model is used) to allow for  
2359 different authenticators in the future.

2360

## 9 Key and Token Parameter Extensions

2361

This section outlines additional parameters that can be specified in token requests and responses.

2362

Typically they are used with issuance requests, but since all types of requests MAY issue security tokens

2363

they could apply to other bindings.

2364

### 9.1 On-Behalf-Of Parameters

2365

In some scenarios the requestor is obtaining a token on behalf of another party. These parameters

2366

specify the issuer and original requestor of the token being used as the basis of the request. The syntax

2367

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

2368

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

2369

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

2370

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

2371

```
  ...
```

2372

```
  <wst:OnBehalfOf>...</wst:OnBehalfOf>
```

2373

```
  <wst:Issuer>...</wst:Issuer>
```

2374

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

2375

2376

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

2377

*/wst:RequestSecurityToken/wst:OnBehalfOf*

2378

This OPTIONAL element indicates that the requestor is making the request on behalf of another.

2379

The identity on whose behalf the request is being made is specified by placing a security token,

2380

<wsse:SecurityTokenReference> element, or <wsa:EndpointReference> element

2381

within the <wst:OnBehalfOf> element. The requestor MAY provide proof of possession of the

2382

key associated with the OnBehalfOf identity by including a signature in the RST security header

2383

generated using the OnBehalfOf token that signs the primary signature of the RST (i.e. endorsing

2384

supporting token concept from WS-SecurityPolicy). Additional signed supporting tokens

2385

describing the OnBehalfOf context MAY also be included within the RST security header.

2386

*/wst:RequestSecurityToken/wst:Issuer*

2387

This OPTIONAL element specifies the issuer of the security token that is presented in the

2388

message. This element's type is an endpoint reference as defined in [\[WS-Addressing\]](#).

2389

2390

In the following illustrates the syntax for a proxy that is requesting a security token on behalf of another

2391

requestor or end-user.

2392

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

2393

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

2394

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

2395

```
  ...
```

2396

```
  <wst:OnBehalfOf>endpoint-reference</wst:OnBehalfOf>
```

2397

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

2398

### 9.2 Key and Encryption Requirements

2399

This section defines extensions to the <wst:RequestSecurityToken> element for requesting specific

2400

types of keys or algorithms or key and algorithms as specified by a given policy in the return token(s). In

2401

some cases the service may support a variety of key types, sizes, and algorithms. These parameters

2402

allow a requestor to indicate its desired values. It should be noted that the issuer's policy indicates if input

2403 values must be adhered to and faults generated for invalid inputs, or if the issuer will provide alternative  
2404 values in the response.

2405

2406 Although illustrated using the `<wst:RequestSecurityToken>` element, these options can also be  
2407 returned in a `<wst:RequestSecurityTokenResponse>` element.

2408 The syntax for these OPTIONAL elements is as follows (note that the base elements described above are  
2409 included here italicized for completeness):

```
2410 <wst:RequestSecurityToken xmlns:wst="...">  
2411   <wst:TokenType>...</wst:TokenType>  
2412   <wst:RequestType>...</wst:RequestType>  
2413   ...  
2414   <wst:AuthenticationType>...</wst:AuthenticationType>  
2415   <wst:KeyType>...</wst:KeyType>  
2416   <wst:KeySize>...</wst:KeySize>  
2417   <wst:SignatureAlgorithm>...</wst:SignatureAlgorithm>  
2418   <wst:EncryptionAlgorithm>...</wst:EncryptionAlgorithm>  
2419   <wst:CanonicalizationAlgorithm>...</wst:CanonicalizationAlgorithm>  
2420   <wst:ComputedKeyAlgorithm>...</wst:ComputedKeyAlgorithm>  
2421   <wst:Encryption>...</wst:Encryption>  
2422   <wst:ProofEncryption>...</wst:ProofEncryption>  
2423   <wst:KeyWrapAlgorithm>...</wst:KeyWrapAlgorithm>  
2424   <wst:UseKey Sig="..."> </wst:UseKey>  
2425   <wst:SignWith>...</wst:SignWith>  
2426   <wst:EncryptWith>...</wst:EncryptWith>  
2427 </wst:RequestSecurityToken>
```

2428

2429 The following describes the attributes and elements listed in the schema overview above:

2430 */wst:RequestSecurityToken/wst:AuthenticationType*

2431 This OPTIONAL URI element indicates the type of authentication desired, specified as a URI.

2432 This specification does not predefine classifications; these are specific to token services as is the  
2433 relative strength evaluations. The relative assessment of strength is up to the recipient to  
2434 determine. That is, requestors SHOULD be familiar with the recipient policies. For example, this  
2435 might be used to indicate which of the four U.S. government authentication levels is REQUIRED.

2436 */wst:RequestSecurityToken/wst:KeyType*

2437 This OPTIONAL URI element indicates the type of key desired in the security token. The  
2438 predefined values are identified in the table below. Note that some security token formats have  
2439 fixed key types. It should be noted that new algorithms can be inserted by defining URIs in other  
2440 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.

2441 */wst:RequestSecurityToken/wst:KeySize*

2442 This OPTIONAL integer element indicates the size of the key REQUIRED specified in number of  
2443 bits. This is a request, and, as such, the requested security token is not obligated to use the  
2444 requested key size. That said, the recipient SHOULD try to use a key at least as strong as the  
2445 specified value if possible. The information is provided as an indication of the desired strength of  
2446 the security.

2447 */wst:RequestSecurityToken/wst:SignatureAlgorithm*

2448 This OPTIONAL URI element indicates the desired signature algorithm used within the returned  
2449 token. This is specified as a URI indicating the algorithm (see [XML-Signature] for typical signing  
2450 algorithms).

2451 */wst:RequestSecurityToken/wst:EncryptionAlgorithm*

2452 This OPTIONAL URI element indicates the desired encryption algorithm used within the returned  
2453 token. This is specified as a URI indicating the algorithm (see [XML-Encrypt] for typical  
2454 encryption algorithms).

2455 */wst:RequestSecurityToken/wst:CanonicalizationAlgorithm*

2456 This OPTIONAL URI element indicates the desired canonicalization method used within the  
2457 returned token. This is specified as a URI indicating the method (see [XML-Signature] for typical  
2458 canonicalization methods).

2459 */wst:RequestSecurityToken/wst:ComputedKeyAlgorithm*

2460 This OPTIONAL URI element indicates the desired algorithm to use when computed keys are  
2461 used for issued tokens.

2462 */wst:RequestSecurityToken/wst:Encryption*

2463 This OPTIONAL element indicates that the requestor desires any returned secrets in issued  
2464 security tokens to be encrypted for the specified token. That is, so that the owner of the specified  
2465 token can decrypt the secret. Normally the security token is the contents of this element but a  
2466 security token reference MAY be used instead. If this element isn't specified, the token used as  
2467 the basis of the request (or specialized knowledge) is used to determine how to encrypt the key.

2468 */wst:RequestSecurityToken/wst:ProofEncryption*

2469 This OPTIONAL element indicates that the requestor desires any returned secrets in proof-of-  
2470 possession tokens to be encrypted for the specified token. That is, so that the owner of the  
2471 specified token can decrypt the secret. Normally the security token is the contents of this element  
2472 but a security token reference MAY be used instead. If this element isn't specified, the token  
2473 used as the basis of the request (or specialized knowledge) is used to determine how to encrypt  
2474 the key.

2475 */wst:RequestSecurityToken/wst:KeyWrapAlgorithm*

2476 This OPTIONAL URI element indicates the desired algorithm to use for key wrapping when STS  
2477 encrypts the issued token for the relying party using an asymmetric key.

2478 */wst:RequestSecurityToken/wst:UseKey*

2479 If the requestor wishes to use an existing key rather than create a new one, then this OPTIONAL  
2480 element can be used to reference the security token containing the desired key. This element  
2481 either contains a security token or a <wsse:SecurityTokenReference> element that  
2482 references the security token containing the key that SHOULD be used in the returned token. If  
2483 <wst:KeyType> is not defined and a key type is not implicitly known to the service, it MAY be  
2484 determined from the token (if possible). Otherwise this parameter is meaningless and is ignored.  
2485 Requestors SHOULD demonstrate authorized use of the public key provided.

2486 */wst:RequestSecurityToken/wst:UseKey/@Sig*

2487 In order to *authenticate* the key referenced, a signature MAY be used to prove the referenced  
2488 token/key. If specified, this OPTIONAL attribute indicates the ID of the corresponding signature

2489 (by URI reference). When this attribute is present, a key need not be specified inside the element  
2490 since the referenced signature will indicate the corresponding token (and key).

2491 */wst:RequestSecurityToken/wst:SignWith*

2492 This OPTIONAL URI element indicates the desired signature algorithm to be used with the issued  
2493 security token (typically from the policy of the target site for which the token is being requested.  
2494 While any of these OPTIONAL elements MAY be included in RSTRs, this one is a likely  
2495 candidate if there is some doubt (e.g., an X.509 cert that can only use DSS).

2496 */wst:RequestSecurityToken/wst:EncryptWith*

2497 This OPTIONAL URI element indicates the desired encryption algorithm to be used with the  
2498 issued security token (typically from the policy of the target site for which the token is being  
2499 requested.) While any of these OPTIONAL elements MAY be included in RSTRs, this one is a  
2500 likely candidate if there is some doubt.

2501 The following summarizes the various algorithm parameters defined above. T is the issued token, P is the  
2502 proof key.  
2503

2504 **SignatureAlgorithm** - The signature algorithm to use to sign T

2505 **EncryptionAlgorithm** - The encryption algorithm to use to encrypt T

2506 **CanonicalizationAlgorithm** - The canonicalization algorithm to use when signing T

2507 **ComputedKeyAlgorithm** - The key derivation algorithm to use if using a symmetric key for P  
2508 where P is computed using client, server, or combined entropy

2509 **Encryption** - The token/key to use when encrypting T

2510 **ProofEncryption** - The token/key to use when encrypting P

2511 **UseKey** - This is P. This is generally used when the client supplies a public-key that it wishes to  
2512 be embedded in T as the proof key

2513 **SignWith** - The signature algorithm the client intends to employ when using P to  
2514 sign

2515 The encryption algorithms further differ based on whether the issued token contains asymmetric key or  
2516 symmetric key. Furthermore, they differ based on what type of key is used to protect the issued token  
2517 from the STS to the relying party. The following cases can occur:

2518 T contains symmetric key/STS uses symmetric key to encrypt T for RP

2519 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP  
2520 when using the proof key (e.g. AES256)

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

2523

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

2525 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP  
2526 when using the proof key (e.g. AES256)

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

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

2531

2532 T contains asymmetric key/STS uses symmetric key to encrypt T for RP

2533 **EncryptWith** – used to indicate the KeyWrap algorithm that the client will use to

2534 protect the symmetric key that is used to protect messages to RP when using the proof key (e.g.  
2535 RSA-OAEP-MGF1P)

2536 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to  
2537 encrypt T for RP (e.g. AES256)

2538

2539 T contains asymmetric key/STS uses asymmetric key to encrypt T for RP

2540 **EncryptWith** - used to indicate the KeyWrap algorithm that the client will use to  
2541 protect symmetric key that is used to protect message to RP when using the proof  
2542 key (e.g. RSA-OAEP-MGF1P)

2543 **EncryptionAlgorithm** - used to indicate the symmetric algorithm that the STS SHOULD use to  
2544 encrypt T for RP (e.g. AES256)

2545 **KeyWrapAlgorithm** – used to indicate the KeyWrap algorithm that the STS SHOULD use to  
2546 wrap the generated key that is used to encrypt the T for RP

2547

2548 The example below illustrates a request that utilizes several of these parameters. A request is made for a  
2549 custom token using a username and password as the basis of the request. For security, this token is  
2550 encrypted (see "encUsername") for the recipient using the recipient's public key and referenced in the  
2551 encryption manifest. The message is protected by a signature using a public key from the sender and  
2552 authorized by the username and password.

2553

2554 The requestor would like the custom token to contain a 1024-bit public key whose value can be found in  
2555 the key provided with the "proofSignature" signature (the key identified by "requestProofToken"). The  
2556 token should be signed using RSA-SHA1 and encrypted for the token identified by  
2557 "requestEncryptionToken". The proof should be encrypted using the token identified by  
2558 "requestProofToken".

```
2559 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."  
2560     xmlns:wst="..." xmlns:ds="..." xmlns:xenc="...">  
2561   <S11:Header>  
2562     ...  
2563     <wsse:Security>  
2564       <xenc:ReferenceList>...</xenc:ReferenceList>  
2565       <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>  
2566       <wsse:BinarySecurityToken wsu:Id="requestEncryptionToken"  
2567         ValueType="...SomeTokenType" xmlns:x="...">  
2568         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
2569       </wsse:BinarySecurityToken>  
2570       <wsse:BinarySecurityToken wsu:Id="requestProofToken"  
2571         ValueType="...SomeTokenType" xmlns:x="...">  
2572         MIIIEZzCCA9CgAwIBAgIQEmtJZc0...  
2573       </wsse:BinarySecurityToken>  
2574       <ds:Signature Id="proofSignature">  
2575         ... signature proving requested key ...  
2576         ... key info points to the "requestedProofToken" token ...  
2577       </ds:Signature>  
2578     </wsse:Security>  
2579     ...  
2580   </S11:Header>  
2581   <S11:Body wsu:Id="req">  
2582     <wst:RequestSecurityToken>  
2583       <wst:TokenType>  
2584         http://example.org/mySpecialToken  
2585       </wst:TokenType>  
2586       <wst:RequestType>
```

2587  
2588  
2589  
2590  
2591  
2592  
2593  
2594  
2595  
2596  
2597  
2598  
2599  
2600  
2601  
2602  
2603  
2604  
2605

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

2606 **9.3 Delegation and Forwarding Requirements**

2607 This section defines extensions to the `<wst:RequestSecurityToken>` element for indicating  
2608 delegation and forwarding requirements on the requested security token(s).

2609 The syntax for these extension elements is as follows (note that the base elements described above are  
2610 included here italicized for completeness):

2611  
2612  
2613  
2614  
2615  
2616  
2617  
2618  
2619

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

2620 */wst:RequestSecurityToken/wst:DelegateTo*

2621 This OPTIONAL element indicates that the requested or issued token be delegated to another  
2622 identity. The identity receiving the delegation is specified by placing a security token or  
2623 `<wsse:SecurityTokenReference>` element within the `<wst:DelegateTo>` element.

2624 */wst:RequestSecurityToken/wst:Forwardable*

2625 This OPTIONAL element, of type `xs:boolean`, specifies whether the requested security token  
2626 SHOULD be marked as "Forwardable". In general, this flag is used when a token is normally  
2627 bound to the requestor's machine or service. Using this flag, the returned token MAY be used  
2628 from any source machine so long as the key is correctly proven. The default value of this flag is  
2629 true.

2630 */wst:RequestSecurityToken/wst:Delegatable*

2631 This OPTIONAL element, of type `xs:boolean`, specifies whether the requested security token  
2632 SHOULD be marked as "Delegatable". Using this flag, the returned token MAY be delegated to  
2633 another party. This parameter SHOULD be used in conjunction with `<wst:DelegateTo>`. The  
2634 default value of this flag is false.

2635 */wst:RequestSecurityToken/wst:ActAs*

2636 This OPTIONAL element indicates that the requested token is expected to contain information  
2637 about the identity represented by the content of this element and the token requestor intends to  
2638 use the returned token to act as this identity. The identity that the requestor wants to act-as is

2639 specified by placing a security token or <wsse:SecurityTokenReference> element within the  
2640 <wst:ActAs> element.

2641 The following illustrates the syntax of a request for a custom token that can be delegated to the indicated  
2642 recipient (specified in the binary security token) and used in the specified interval.

```
2643 <wst:RequestSecurityToken xmlns:wst="...">  
2644 <wst:TokenType>  
2645 http://example.org/mySpecialToken  
2646 </wst:TokenType>  
2647 <wst:RequestType>  
2648 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
2649 </wst:RequestType>  
2650 <wst:DelegateTo>  
2651 <wsse:BinarySecurityToken xmlns:wsse="...">  
2652 ...  
2653 </wsse:BinarySecurityToken>  
2654 </wst:DelegateTo>  
2655 <wst:Delegatable>true</wst:Delegatable>  
2656 </wst:RequestSecurityToken>
```

## 2657 9.4 Policies

2658 This section defines extensions to the <wst:RequestSecurityToken> element for passing policies.

2659  
2660 The syntax for these extension elements is as follows (note that the base elements described above are  
2661 included here italicized for completeness):

```
2662 <wst:RequestSecurityToken xmlns:wst="...">  
2663 <wst:TokenType>...</wst:TokenType>  
2664 <wst:RequestType>...</wst:RequestType>  
2665 ...  
2666 <wsp:Policy xmlns:wsp="...">...</wsp:Policy>  
2667 <wsp:PolicyReference xmlns:wsp="...">...</wsp:PolicyReference>  
2668 </wst:RequestSecurityToken>
```

2669  
2670 The following describes the attributes and elements listed in the schema overview above:

2671 */wst:RequestSecurityToken/wsp:Policy*

2672 This OPTIONAL element specifies a policy (as defined in [WS-Policy]) that indicates desired  
2673 settings for the requested token. The policy specifies defaults that can be overridden by the  
2674 elements defined in the previous sections.

2675 */wst:RequestSecurityToken/wsp:PolicyReference*

2676 This OPTIONAL element specifies a reference to a policy (as defined in [WS-Policy]) that  
2677 indicates desired settings for the requested token. The policy specifies defaults that can be  
2678 overridden by the elements defined in the previous sections.

2679  
2680 The following illustrates the syntax of a request for a custom token that provides a set of policy  
2681 statements about the token or its usage requirements.

```
2682 <wst:RequestSecurityToken xmlns:wst="...">  
2683 <wst:TokenType>  
2684 http://example.org/mySpecialToken  
2685 </wst:TokenType>  
2686 <wst:RequestType>  
2687 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
2688 </wst:RequestType>  
2689 <wsp:Policy xmlns:wsp="...">
```

2690  
2691  
2692

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

## 2693 9.5 Authorized Token Participants

2694 This section defines extensions to the `<wst:RequestSecurityToken>` element for passing information  
2695 about which parties are authorized to participate in the use of the token. This parameter is typically used  
2696 when there are additional parties using the token or if the requestor needs to clarify the actual parties  
2697 involved (for some profile-specific reason).

2698 It should be noted that additional participants will need to prove their identity to recipients in addition to  
2699 proving their authorization to use the returned token. This typically takes the form of a second signature  
2700 or use of transport security.

2701

2702 The syntax for these extension elements is as follows (note that the base elements described above are  
2703 included here *italized* for completeness):

```
2704 <wst:RequestSecurityToken xmlns:wst="...">  
2705   <wst:TokenType>...</wst:TokenType>  
2706   <wst:RequestType>...</wst:RequestType>  
2707   ...  
2708   <wst:Participants>  
2709     <wst:Primary>...</wst:Primary>  
2710     <wst:Participant>...</wst:Participant>  
2711   </wst:Participants>  
2712 </wst:RequestSecurityToken>
```

2713

2714 The following describes elements and attributes used in a `<wsc:SecurityContextToken>` element.

2715 */wst:RequestSecurityToken/wst:Participants/*

2716 This OPTIONAL element specifies the participants sharing the security token. Arbitrary types  
2717 MAY be used to specify participants, but a typical case is a security token or an endpoint  
2718 reference (see [\[WS-Addressing\]](#)).

2719 */wst:RequestSecurityToken/wst:Participants/wst:Primary*

2720 This OPTIONAL element specifies the primary user of the token (if one exists).

2721 */wst:RequestSecurityToken/wst:Participants/wst:Participant*

2722 This OPTIONAL element specifies participant (or multiple participants by repeating the element)  
2723 that play a (profile-dependent) role in the use of the token or who are allowed to use the token.

2724 */wst:RequestSecurityToken/wst:Participants/{any}*

2725 This is an extensibility option to allow other types of participants and profile-specific elements to  
2726 be specified.

---

## 2727 10 Key Exchange Token Binding

2728 Using the token request framework, this section defines a binding for requesting a key exchange token  
2729 (KET). That is, if a requestor desires a token that can be used to encrypt key material for a recipient.

2730  
2731 For this binding, the following actions are defined to enable specific processing context to be conveyed to  
2732 the recipient:

```
2733 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET  
2734 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KET  
2735 http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal
```

2736  
2737 For this binding, the `RequestType` element contains the following URI:

```
2738 http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET
```

2739  
2740 For this binding very few parameters are specified as input. **OPTIONALLY** the `<wst:TokenType>`  
2741 element can be specified in the request can indicate desired type response token carrying the key for key  
2742 exchange; however, this isn't commonly used.

2743 The applicability scope (e.g. `<wsp:AppliesTo>`) **MAY** be specified if the requestor desires a key  
2744 exchange token for a specific scope.

2745  
2746 It is **RECOMMENDED** that the response carrying the key exchange token be secured (e.g., signed by the  
2747 issuer or someone who can speak on behalf of the target for which the KET applies).

2748  
2749 Care should be taken when using this binding to prevent possible man-in-the-middle and substitution  
2750 attacks. For example, responses to this request **SHOULD** be secured using a token that can speak for  
2751 the desired endpoint.

2752  
2753 The RSTR for this binding carries the `<RequestedSecurityToken>` element even if a token is returned  
2754 (note that the base elements described above are included here italicized for completeness):

```
2755 <wst:RequestSecurityToken xmlns:wst="...">  
2756   <wst:TokenType>...</wst:TokenType>  
2757   <wst:RequestType>...</wst:RequestType>  
2758   ...  
2759 </wst:RequestSecurityToken>
```

```
2760  
2761 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2762   <wst:RequestSecurityTokenResponse>  
2763     <wst:TokenType>...</wst:TokenType>  
2764     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>  
2765     ...  
2766   </wst:RequestSecurityTokenResponse>  
2767 </wst:RequestSecurityTokenResponseCollection>
```

2768  
2769 The following illustrates the syntax for requesting a key exchange token. In this example, the KET is  
2770 returned encrypted for the requestor since it had the credentials available to do that. Alternatively the

2771 request could be made using transport security (e.g. TLS) and the key could be returned directly using  
2772 <wst:BinarySecret>.

```
2773 <wst:RequestSecurityToken xmlns:wst="...">  
2774 <wst:RequestType>  
2775 http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET  
2776 </wst:RequestType>  
2777 </wst:RequestSecurityToken>
```

```
2778  
2779 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2780 <wst:RequestSecurityTokenResponse>  
2781 <wst:RequestedSecurityToken>  
2782 <xenc:EncryptedKey xmlns:xenc="...">...</xenc:EncryptedKey>  
2783 </wst:RequestedSecurityToken>  
2784 </wst:RequestSecurityTokenResponse>  
2785 </wst:RequestSecurityTokenResponseCollection>
```

2786

## 11 Error Handling

2787  
2788  
2789  
2790  
2791  
2792  
2793  
2794

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

<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

2795

---

## 12 Security Considerations

2796 As stated in the Goals section of this document, this specification is meant to provide extensible  
2797 framework and flexible syntax, with which one could implement various security mechanisms. This  
2798 framework and syntax by itself does not provide any guarantee of security. When implementing and using  
2799 this framework and syntax, one must make every effort to ensure that the result is not vulnerable to any  
2800 one of a wide range of attacks.

2801

2802 It is not feasible to provide a comprehensive list of security considerations for such an extensible set of  
2803 mechanisms. A complete security analysis must be conducted on specific solutions based on this  
2804 specification. Below we illustrate some of the security concerns that often come up with protocols of this  
2805 type, but we stress that this *is not an exhaustive list of concerns*.

2806

2807 The following statements about signatures and signing apply to messages sent on unsecured channels.

2808

2809 It is critical that all the security-sensitive message elements must be included in the scope of the  
2810 message signature. As well, the signatures for conversation authentication must include a timestamp,  
2811 nonce, or sequence number depending on the degree of replay prevention required as described in [[WS-  
2812 Security](#)] and the UsernameToken Profile. Also, conversation establishment should include the policy so  
2813 that supported algorithms and algorithm priorities can be validated.

2814

2815 It is required that security token issuance messages be signed to prevent tampering. If a public key is  
2816 provided, the request should be signed by the corresponding private key to prove ownership. As well,  
2817 additional steps should be taken to eliminate replay attacks (refer to [[WS-Security](#)] for additional  
2818 information). Similarly, all token references should be signed to prevent any tampering.

2819

2820 Security token requests are susceptible to denial-of-service attacks. Care should be taken to mitigate  
2821 such attacks as is warranted by the service.

2822

2823 For security, tokens containing a symmetric key or a password should only be sent to parties who have a  
2824 need to know that key or password.

2825

2826 For privacy, tokens containing personal information (either in the claims, or indirectly by identifying who is  
2827 currently communicating with whom) should only be sent according to the privacy policies governing  
2828 these data at the respective organizations.

2829

2830 For some forms of multi-message exchanges, the exchanges are susceptible to attacks whereby  
2831 signatures are altered. To address this, it is suggested that a signature confirmation mechanism be used.  
2832 In such cases, each leg should include the confirmation of the previous leg. That is, leg 2 includes  
2833 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  
2834 correctness of the message outside of the message body.

2835

2836 There are many other security concerns that one may need to consider in security protocols. The list  
2837 above should not be used as a "check list" instead of a comprehensive security analysis.

2838

2839 It should be noted that use of unsolicited RSTRs implies that the recipient is prepared to accept such  
2840 issuances. Recipients should ensure that such issuances are properly authorized and recognize their  
2841 use could be used in denial-of-service attacks.

2842 In addition to the consideration identified here, readers should also review the security considerations in  
2843 [\[WS-Security\]](#).

2844

2845 Both token cancellation bindings defined in this specification require that the STS MUST NOT validate or  
2846 renew the token after it has been successfully canceled. The STS must take care to ensure that the token  
2847 is properly invalidated before confirming the cancel request or sending the cancel notification to the client.  
2848 This can be more difficult if the token validation or renewal logic is physically separated from the issuance  
2849 and cancellation logic. It is out of scope of this spec how the STS propagates the token cancellation to its  
2850 other components. If STS cannot ensure that the token was properly invalidated it MUST NOT send the  
2851 cancel notification or confirm the cancel request to the client.

---

## 2852 A. Key Exchange

2853 Key exchange is an integral part of token acquisition. There are several mechanisms by which keys are  
2854 exchanged using [\[WS-Security\]](#) and WS-Trust. This section highlights and summarizes these  
2855 mechanisms. Other specifications and profiles MAY provide additional details on key exchange.

2856  
2857 Care must be taken when employing a key exchange to ensure that the mechanism does not provide an  
2858 attacker with a means of discovering information that could only be discovered through use of secret  
2859 information (such as a private key).

2860  
2861 It is therefore important that a shared secret should only be considered as trustworthy as its source. A  
2862 shared secret communicated by means of the direct encryption scheme described in section I.1 is  
2863 acceptable if the encryption key is provided by a completely trustworthy key distribution center (this is the  
2864 case in the Kerberos model). Such a key would not be acceptable for the purposes of decrypting  
2865 information from the source that provided it since an attacker might replay information from a prior  
2866 transaction in the hope of learning information about it.

2867  
2868 In most cases the other party in a transaction is only imperfectly trustworthy. In these cases both parties  
2869 SHOULD contribute entropy to the key exchange by means of the `<wst:entropy>` element.

### 2870 A.1 Ephemeral Encryption Keys

2871 The simplest form of key exchange can be found in [\[WS-Security\]](#) for encrypting message data. As  
2872 described in [\[WS-Security\]](#) and [\[XML-Encrypt\]](#), when data is encrypted, a temporary key can be used to  
2873 perform the encryption which is, itself, then encrypted using the `<xenc:EncryptedKey>` element.

2874  
2875 The illustrates the syntax for encrypting a temporary key using the public key in an issuer name and serial  
2876 number:

```
2877 <xenc:EncryptedKey xmlns:xenc="...">  
2878   ...  
2879   <ds:KeyInfo xmlns:ds="...">  
2880     <wsse:SecurityTokenReference xmlns:wsse="...">  
2881       <ds:X509IssuerSerial>  
2882         <ds:X509IssuerName>  
2883           DC=ACMECorp, DC=com  
2884         </ds:X509IssuerName>  
2885         <ds:X509SerialNumber>12345678</ds:X509SerialNumber>  
2886       </ds:X509IssuerSerial>  
2887     </wsse:SecurityTokenReference>  
2888   </ds:KeyInfo>  
2889   ...  
2890 </xenc:EncryptedKey>
```

### 2891 A.2 Requestor-Provided Keys

2892 When a request sends a message to an issuer to request a token, the client can provide proposed key  
2893 material using the `<wst:Entropy>` element. If the issuer doesn't contribute any key material, this is  
2894 used as the secret (key). This information is encrypted for the issuer either using  
2895 `<xenc:EncryptedKey>` or by using a transport security. If the requestor provides key material that the

2896 recipient doesn't accept, then the issuer SHOULD reject the request. Note that the issuer need not return  
2897 the key provided by the requestor.

2898

2899 The following illustrates the syntax of a request for a custom security token and includes a secret that is  
2900 to be used for the key. In this example the entropy is encrypted for the issuer (if transport security was  
2901 used for confidentiality then the <wst:Entropy> element would contain a <wst:BinarySecret>  
2902 element):

```
2903 <wst:RequestSecurityToken xmlns:wst="...">  
2904 <wst:TokenType>  
2905   http://example.org/mySpecialToken  
2906 </wst:TokenType>  
2907 <wst:RequestType>  
2908   http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
2909 </wst:RequestType>  
2910 <wst:Entropy>  
2911   <xenc:EncryptedData xmlns:xenc="...">...</xenc:EncryptedData>  
2912 </wst:Entropy>  
2913 </wst:RequestSecurityToken>
```

### 2914 **A.3 Issuer-Provided Keys**

2915 If a requestor fails to provide key material, then issued proof-of-possession tokens contain an issuer-  
2916 provided secret that is encrypted for the requestor (either using <xenc:EncryptedKey> or by using a  
2917 transport security).

2918

2919 The following illustrates the syntax of a token being returned with an associated proof-of-possession  
2920 token that is encrypted using the requestor's public key.

```
2921 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">  
2922 <wst:RequestSecurityTokenResponse>  
2923 <wst:RequestedSecurityToken>  
2924 <xyz:CustomToken xmlns:xyz="...">  
2925   ...  
2926 </xyz:CustomToken>  
2927 </wst:RequestedSecurityToken>  
2928 <wst:RequestedProofToken>  
2929 <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">  
2930   ...  
2931 </xenc:EncryptedKey>  
2932 </wst:RequestedProofToken>  
2933 </wst:RequestSecurityTokenResponse>  
2934 </wst:RequestSecurityTokenResponseCollection>
```

### 2935 **A.4 Composite Keys**

2936 The safest form of key exchange/generation is when both the requestor and the issuer contribute to the  
2937 key material. In this case, the request sends encrypted key material. The issuer then returns additional  
2938 encrypted key material. The actual secret (key) is computed using a function of the two pieces of data.  
2939 Ideally this secret is never used and, instead, keys derived are used for message protection.

2940

2941 The following example illustrates a server, having received a request with requestor entropy returning its  
2942 own entropy, which is used in conjunction with the requestor's to generate a key. In this example the  
2943 entropy is not encrypted because the transport is providing confidentiality (otherwise the  
2944 <wst:Entropy> element would have an <xenc:EncryptedData> element).

2945  
2946  
2947  
2948  
2949  
2950  
2951  
2952  
2953  
2954  
2955  
2956

```
<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>
```

## 2957 **A.5 Key Transfer and Distribution**

2958 There are also a few mechanisms where existing keys are transferred to other parties.

### 2959 **A.5.1 Direct Key Transfer**

2960 If one party has a token and key and wishes to share this with another party, the key can be directly  
2961 transferred. This is accomplished by sending an RSTR (either in the body or header) to the other party.  
2962 The RSTR contains the token and a proof-of-possession token that contains the key encrypted for the  
2963 recipient.

2964

2965 In the following example a custom token and its associated proof-of-possession token are known to party  
2966 A who wishes to share them with party B. In this example, A is a member in a secure on-line chat  
2967 session and is inviting B to join the conversation. After authenticating B, A sends B an RSTR. The RSTR  
2968 contains the token and the key is communicated as a proof-of-possession token that is encrypted for B:

2969  
2970  
2971  
2972  
2973  
2974  
2975  
2976  
2977  
2978  
2979  
2980  
2981  
2982

```
<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>
```

### 2983 **A.5.2 Brokered Key Distribution**

2984 A third party MAY also act as a broker to transfer keys. For example, a requestor may obtain a token and  
2985 proof-of-possession token from a third-party STS. The token contains a key encrypted for the target  
2986 service (either using the service's public key or a key known to the STS and target service). The proof-of-  
2987 possession token contains the same key encrypted for the requestor (similarly this can use public or  
2988 symmetric keys).

2989

2990 In the following example a custom token and its associated proof-of-possession token are returned from a  
2991 broker B to a requestor R for access to service S. The key for the session is contained within the custom  
2992 token encrypted for S using either a secret known by B and S or using S's public key. The same secret is  
2993 encrypted for R and returned as the proof-of-possession token:

2994  
2995  
2996  
2997  
2998  
2999  
3000  
3001  
3002  
3003  
3004  
3005  
3006  
3007  
3008  
3009  
3010  
3011

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

3012 **A.5.3 Delegated Key Transfer**

3013 Key transfer can also take the form of delegation. That is, one party transfers the right to use a key  
3014 without actually transferring the key. In such cases, a delegation token, e.g. XrML, is created that  
3015 identifies a set of rights and a delegation target and is secured by the delegating party. That is, one key  
3016 indicates that another key can use a subset (or all) of its rights. The delegate can provide this token and  
3017 prove itself (using its own key – the delegation target) to a service. The service, assuming the trust  
3018 relationships have been established and that the delegator has the right to delegate, can then authorize  
3019 requests sent subject to delegation rules and trust policies.

3020  
3021 In this example a custom token is issued from party A to party B. The token indicates that B (specifically  
3022 B's key) has the right to submit purchase orders. The token is signed using a secret key known to the  
3023 target service T and party A (the key used to ultimately authorize the requests that B makes to T), and a  
3024 new session key that is encrypted for T. A proof-of-possession token is included that contains the  
3025 session key encrypted for B. As a result, B is *effectively* using A's key, but doesn't actually know the key.

3026  
3027  
3028  
3029  
3030  
3031  
3032  
3033  
3034  
3035  
3036  
3037  
3038  
3039  
3040  
3041  
3042  
3043  
3044  
3045  
3046  
3047  
3048

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

## 3049 **A.5.4 Authenticated Request/Reply Key Transfer**

3050 In some cases the RST/RSTR mechanism is not used to transfer keys because it is part of a simple  
3051 request/reply. However, there may be a desire to ensure mutual authentication as part of the key  
3052 transfer. The mechanisms of [WS-Security] can be used to implement this scenario.

3053

3054 Specifically, the sender wishes the following:

- 3055 • Transfer a key to a recipient that they can use to secure a reply
- 3056 • Ensure that only the recipient can see the key
- 3057 • Provide proof that the sender issued the key

3058

3059 This scenario could be supported by encrypting and then signing. This would result in roughly the  
3060 following steps:

- 3061 1. Encrypt the message using a generated key
- 3062 2. Encrypt the key for the recipient
- 3063 3. Sign the encrypted form, any other relevant keys, and the encrypted key

3064

3065 However, if there is a desire to sign prior to encryption then the following general process is used:

- 3066 1. Sign the appropriate message parts using a random key (or ideally a key derived from a random  
3067 key)
- 3068 2. Encrypt the appropriate message parts using the random key (or ideally another key derived from  
3069 the random key)
- 3070 3. Encrypt the random key for the recipient
- 3071 4. Sign just the encrypted key

3072

3073 This would result in a <wsse:Security> header that looks roughly like the following:

```
3074 <wsse:Security xmlns:wsse="..." xmlns:wsu="..."  
3075     xmlns:ds="..." xmlns:xenc="...">  
3076   <wsse:BinarySecurityToken wsu:Id="myToken">  
3077     ...  
3078   </wsse:BinarySecurityToken>  
3079   <ds:Signature>  
3080     ...signature over #secret using token #myToken...  
3081   </ds:Signature>  
3082   <xenc:EncryptedKey Id="secret">  
3083     ...  
3084   </xenc:EncryptedKey>  
3085   <xenc:ReferenceList>  
3086     ...manifest of encrypted parts using token #secret...  
3087   </xenc:ReferenceList>  
3088   <ds:Signature>  
3089     ...signature over key message parts using token #secret...  
3090   </ds:Signature>  
3091 </wsse:Security>
```

3092

3093 As well, instead of an <xenc:EncryptedKey> element, the actual token could be passed using  
3094 <xenc:EncryptedData>. The result might look like the following:

```
3095 <wsse:Security xmlns:wsse="..." xmlns:wsu="..."  
3096     xmlns:ds="..." xmlns:xenc="...">
```

3097  
3098  
3099  
3100  
3101  
3102  
3103  
3104  
3105  
3106  
3107  
3108  
3109  
3110  
3111  
3112

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

3113 **A.6 Perfect Forward Secrecy**

3114 In some situations it is desirable for a key exchange to have the property of perfect forward secrecy. This  
3115 means that it is impossible to reconstruct the shared secret even if the private keys of the parties are  
3116 disclosed.

3117

3118 The most straightforward way to attain perfect forward secrecy when using asymmetric key exchange is  
3119 to dispose of one's key exchange key pair periodically (or even after every key exchange), replacing it  
3120 with a fresh one. Of course, a freshly generated public key must still be authenticated (using any of the  
3121 methods normally available to prove the identity of a public key's owner).

3122

3123 The perfect forward secrecy property MAY be achieved by specifying a `<wst:entropy>` element that  
3124 contains an `<xenc:EncryptedKey>` that is encrypted under a public key pair created for use in a single  
3125 key agreement. The public key does not require authentication since it is only used to provide additional  
3126 entropy. If the public key is modified, the key agreement will fail. Care should be taken, when using this  
3127 method, to ensure that the now-secret entropy exchanged via the `<wst:entropy>` element is not  
3128 revealed elsewhere in the protocol (since such entropy is often assumed to be publicly revealed plaintext,  
3129 and treated accordingly).

3130

3131 Although any public key scheme might be used to achieve perfect forward secrecy (in either of the above  
3132 methods) it is generally desirable to use an algorithm that allows keys to be generated quickly. The Diffie-  
3133 Hellman key exchange is often used for this purpose since generation of a key only requires the  
3134 generation of a random integer and calculation of a single modular exponent.

3135

## B. WSDL

3136 The WSDL below does not fully capture all the possible message exchange patterns, but captures the  
3137 typical message exchange pattern as described in this document.

```
3138 <?xml version="1.0"?>
3139 <wSDL:definitions
3140     targetNamespace="http://docs.oasis-open.org/ws-sx/ws-
3141     trust/200512/wSDL"
3142     xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/wSDL"
3143     xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
3144     xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
3145     xmlns:xs="http://www.w3.org/2001/XMLSchema"
3146     xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
3147 >
3148 <!-- this is the WS-I BP-compliant way to import a schema -->
3149     <wSDL:types>
3150         <xs:schema>
3151             <xs:import
3152                 namespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
3153                 schemaLocation="http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-
3154                 trust.xsd"/>
3155             </xs:schema>
3156         </wSDL:types>
3157
3158 <!-- WS-Trust defines the following GEDs -->
3159         <wSDL:message name="RequestSecurityTokenMsg">
3160             <wSDL:part name="request" element="wst:RequestSecurityToken" />
3161         </wSDL:message>
3162         <wSDL:message name="RequestSecurityTokenResponseMsg">
3163             <wSDL:part name="response"
3164                 element="wst:RequestSecurityTokenResponse" />
3165         </wSDL:message>
3166         <wSDL:message name="RequestSecurityTokenCollectionMsg">
3167             <wSDL:part name="requestCollection"
3168                 element="wst:RequestSecurityTokenCollection"/>
3169         </wSDL:message>
3170         <wSDL:message name="RequestSecurityTokenResponseCollectionMsg">
3171             <wSDL:part name="responseCollection"
3172                 element="wst:RequestSecurityTokenResponseCollection"/>
3173         </wSDL:message>
3174
3175         <!-- This portType an example of a Requestor (or other) endpoint that
3176             Accepts SOAP-based challenges from a Security Token Service -->
3177         <wSDL:portType name="WSSecurityRequestor">
3178             <wSDL:operation name="Challenge">
3179                 <wSDL:input message="tns:RequestSecurityTokenResponseMsg"/>
3180                 <wSDL:output message="tns:RequestSecurityTokenResponseMsg"/>
3181             </wSDL:operation>
3182         </wSDL:portType>
3183
3184         <!-- This portType is an example of an STS supporting full protocol -->
3185         <wSDL:portType name="SecurityTokenService">
3186             <wSDL:operation name="Cancel">
3187                 <wSDL:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3188                 trust/200512/RST/Cancel" message="tns:RequestSecurityTokenMsg"/>
3189                 <wSDL:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3190                 trust/200512/RSTR/CancelFinal" message="tns:RequestSecurityTokenResponseMsg"/>
3191             </wSDL:operation>
3192             <wSDL:operation name="Issue">
```

```

3193     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3194 trust/200512/RST/Issue" message="tns:RequestSecurityTokenMsg"/>
3195     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3196 trust/200512/RSTRC/IssueFinal"
3197 message="tns:RequestSecurityTokenResponseCollectionMsg"/>
3198   </wsdl:operation>
3199   <wsdl:operation name="Renew">
3200     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3201 trust/200512/RST/Renew" message="tns:RequestSecurityTokenMsg"/>
3202     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3203 trust/200512/RSTR/RenewFinal" message="tns:RequestSecurityTokenResponseMsg"/>
3204   </wsdl:operation>
3205   <wsdl:operation name="Validate">
3206     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3207 trust/200512/RST/Validate" message="tns:RequestSecurityTokenMsg"/>
3208     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3209 trust/200512/RSTR/ValidateFinal
3210 message="tns:RequestSecurityTokenResponseMsg"/>
3211   </wsdl:operation>
3212   <wsdl:operation name="KeyExchangeToken">
3213     <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3214 trust/200512/RST/KET" message="tns:RequestSecurityTokenMsg"/>
3215     <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3216 trust/200512/RSTR/KETFinal" message="tns:RequestSecurityTokenResponseMsg"/>
3217   </wsdl:operation>
3218   <wsdl:operation name="RequestCollection">
3219     <wsdl:input message="tns:RequestSecurityTokenCollectionMsg"/>
3220     <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg"/>
3221   </wsdl:operation>
3222 </wsdl:portType>
3223
3224 <!-- This portType is an example of an endpoint that accepts
3225      Unsolicited RequestSecurityTokenResponse messages -->
3226 <wsdl:portType name="SecurityTokenResponseService">
3227   <wsdl:operation name="RequestSecurityTokenResponse">
3228     <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
3229   </wsdl:operation>
3230 </wsdl:portType>
3231
3232 </wsdl:definitions>
3233

```

3234

---

## C. Acknowledgements

3235 The following individuals have participated in the creation of this specification and are gratefully  
3236 acknowledged:

3237 **Original Authors of the initial contribution:**

3238 Steve Anderson, OpenNetwork  
3239 Jeff Bohren, OpenNetwork  
3240 Toufic Boubez, Layer 7  
3241 Marc Chanliau, Computer Associates  
3242 Giovanni Della-Libera, Microsoft  
3243 Brendan Dixon, Microsoft  
3244 Praerit Garg, Microsoft  
3245 Martin Gudgin (Editor), Microsoft  
3246 Phillip Hallam-Baker, VeriSign  
3247 Maryann Hondo, IBM  
3248 Chris Kaler, Microsoft  
3249 Hal Lockhart, BEA  
3250 Robin Martherus, Oblix  
3251 Hiroshi Maruyama, IBM  
3252 Anthony Nadalin (Editor), IBM  
3253 Nataraj Nagaratnam, IBM  
3254 Andrew Nash, Reactivity  
3255 Rob Philpott, RSA Security  
3256 Darren Platt, Ping Identity  
3257 Hemma Prafullchandra, VeriSign  
3258 Maneesh Sahu, Actional  
3259 John Shewchuk, Microsoft  
3260 Dan Simon, Microsoft  
3261 Davanum Srinivas, Computer Associates  
3262 Elliot Waingold, Microsoft  
3263 David Waite, Ping Identity  
3264 Doug Walter, Microsoft  
3265 Riaz Zolfonoon, RSA Security

3266 **Original Acknowledgments of the initial contribution:**

3267 Paula Austel, IBM  
3268 Keith Ballinger, Microsoft  
3269 Bob Blakley, IBM  
3270 John Brezak, Microsoft  
3271 Tony Cowan, IBM  
3272 Cédric Fournet, Microsoft  
3273 Vijay Gajjala, Microsoft  
3274 HongMei Ge, Microsoft  
3275 Satoshi Hada, IBM  
3276 Heather Hinton, IBM  
3277 Slava Kavsan, RSA Security  
3278 Scott Konersmann, Microsoft  
3279 Leo Laferriere, Computer Associates  
3280

3281 Paul Leach, Microsoft  
3282 Richard Levinson, Computer Associates  
3283 John Linn, RSA Security  
3284 Michael McIntosh, IBM  
3285 Steve Millet, Microsoft  
3286 Birgit Pfitzmann, IBM  
3287 Fumiko Satoh, IBM  
3288 Keith Stobie, Microsoft  
3289 T.R. Vishwanath, Microsoft  
3290 Richard Ward, Microsoft  
3291 Hervey Wilson, Microsoft

3292

3293 **TC Members during the development of this specification:**

3294 Don Adams, Tibco Software Inc.  
3295 Jan Alexander, Microsoft Corporation  
3296 Steve Anderson, BMC Software  
3297 Donal Arundel, IONA Technologies  
3298 Howard Bae, Oracle Corporation  
3299 Abbie Barbir, Nortel Networks Limited  
3300 Charlton Barreto, Adobe Systems  
3301 Mighael Botha, Software AG, Inc.  
3302 Toufic Boubez, Layer 7 Technologies Inc.  
3303 Norman Brickman, Mitre Corporation  
3304 Melissa Brumfield, Booz Allen Hamilton  
3305 Lloyd Burch, Novell  
3306 Geoff Bullen, Microsoft Corporation  
3307 Scott Cantor, Internet2  
3308 Greg Carpenter, Microsoft Corporation  
3309 Steve Carter, Novell  
3310 Ching-Yun (C.Y.) Chao, IBM  
3311 Martin Chapman, Oracle Corporation  
3312 Kate Cherry, Lockheed Martin  
3313 Henry (Hyenvui) Chung, IBM  
3314 Luc Clement, Systinet Corp.  
3315 Paul Cotton, Microsoft Corporation  
3316 Glen Daniels, Sonic Software Corp.  
3317 Peter Davis, Neustar, Inc.  
3318 Martijn de Boer, SAP AG  
3319 Duane DeCouteau, Veterans Health Administration  
3320 Werner Dittmann, Siemens AG  
3321 Abdeslem DJAOUI, CCLRC-Rutherford Appleton Laboratory  
3322 Fred Dushin, IONA Technologies  
3323 Petr Dvorak, Systinet Corp.  
3324 Colleen Evans, Microsoft Corporation

3325 Ruchith Fernando, WSO2  
3326 Mark Fussell, Microsoft Corporation  
3327 Vijay Gajjala, Microsoft Corporation  
3328 Marc Goodner, Microsoft Corporation  
3329 Hans Granqvist, VeriSign  
3330 Martin Gudgin, Microsoft Corporation  
3331 Tony Gullotta, SOA Software Inc.  
3332 Jiandong Guo, Sun Microsystems  
3333 Phillip Hallam-Baker, VeriSign  
3334 Patrick Harding, Ping Identity Corporation  
3335 Heather Hinton, IBM  
3336 Frederick Hirsch, Nokia Corporation  
3337 Jeff Hodges, Neustar, Inc.  
3338 Will Hopkins, BEA Systems, Inc.  
3339 Alex Hristov, Otecia Incorporated  
3340 John Hughes, PA Consulting  
3341 Diane Jordan, IBM  
3342 Venugopal K, Sun Microsystems  
3343 Chris Kaler, Microsoft Corporation  
3344 Dana Kaufman, Forum Systems, Inc.  
3345 Paul Knight, Nortel Networks Limited  
3346 Ramanathan Krishnamurthy, IONA Technologies  
3347 Christopher Kurt, Microsoft Corporation  
3348 Kelvin Lawrence, IBM  
3349 Hubert Le Van Gong, Sun Microsystems  
3350 Jong Lee, BEA Systems, Inc.  
3351 Rich Levinson, Oracle Corporation  
3352 Tommy Lindberg, Dajeil Ltd.  
3353 Mark Little, JBoss Inc.  
3354 Hal Lockhart, BEA Systems, Inc.  
3355 Mike Lyons, Layer 7 Technologies Inc.  
3356 Eve Maler, Sun Microsystems  
3357 Ashok Malhotra, Oracle Corporation  
3358 Anand Mani, CrimsonLogic Pte Ltd  
3359 Jonathan Marsh, Microsoft Corporation  
3360 Robin Martherus, Oracle Corporation  
3361 Miko Matsumura, Infravio, Inc.  
3362 Gary McAfee, IBM  
3363 Michael McIntosh, IBM  
3364 John Merrells, Sxip Networks SRL  
3365 Jeff Mischkinisky, Oracle Corporation  
3366 Prateek Mishra, Oracle Corporation

3367 Bob Morgan, Internet2  
3368 Vamsi Motukuru, Oracle Corporation  
3369 Raajmohan Na, EDS  
3370 Anthony Nadalin, IBM  
3371 Andrew Nash, Reactivity, Inc.  
3372 Eric Newcomer, IONA Technologies  
3373 Duane Nickull, Adobe Systems  
3374 Toshihiro Nishimura, Fujitsu Limited  
3375 Rob Philpott, RSA Security  
3376 Denis Pilipchuk, BEA Systems, Inc.  
3377 Darren Platt, Ping Identity Corporation  
3378 Martin Raeppele, SAP AG  
3379 Nick Ragouzis, Enosis Group LLC  
3380 Prakash Reddy, CA  
3381 Alain Regnier, Ricoh Company, Ltd.  
3382 Irving Reid, Hewlett-Packard  
3383 Bruce Rich, IBM  
3384 Tom Rutt, Fujitsu Limited  
3385 Maneesh Sahu, Actional Corporation  
3386 Frank Siebenlist, Argonne National Laboratory  
3387 Joe Smith, Apani Networks  
3388 Davanum Srinivas, WSO2  
3389 David Staggs, Veterans Health Administration  
3390 Yakov Sverdlov, CA  
3391 Gene Thurston, AmberPoint  
3392 Victor Valle, IBM  
3393 Asir Vedamuthu, Microsoft Corporation  
3394 Greg Whitehead, Hewlett-Packard  
3395 Ron Williams, IBM  
3396 Corinna Witt, BEA Systems, Inc.  
3397 Kyle Young, Microsoft Corporation  
3398