



# WS-Trust V1.0

## Working Draft, 09 January 2006

**Artifact Identifier:**

ws-sx-spec-draft-v1-r0-ws-trust

**Location:**

Current: [docs.oasis-open.org/ws-sx/200512/ws-trust](http://docs.oasis-open.org/ws-sx/200512/ws-trust)

This Version: [docs.oasis-open.org/ws-sx/200512/ws-trust](http://docs.oasis-open.org/ws-sx/200512/ws-trust)

Previous Version: N/A

**Artifact Type:**

specification

**Technical Committee:**

OASIS Web Service Secure Exchange TC

**Chair(s):**

Kelvin Lawrence, IBM

Chris Kaler, Microsoft

**Editor(s):**

Anthony Nadalin, IBM

Martin Gudgin, Microsoft

Abbie Barbir, Nortel

Hans Granqvist, VeriSign

**OASIS Conceptual Model topic area:**

[Topic Area]

**Related work:**

N/A

**Abstract:**

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

**Status:**

This document was last revised or approved by the Web Service Secure Exchange 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 [www.oasis-open.org/committees/ws-sx](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 ([www.oasis-open.org/committees/ws-sx/ipr.php](http://www.oasis-open.org/committees/ws-sx/ipr.php)).

The non-normative errata page for this specification is located at [www.oasis-open.org/committees/ws-sx](http://www.oasis-open.org/committees/ws-sx).

---

## Notices

Copyright © OASIS Open 2005. All Rights Reserved.

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

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

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

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

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

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

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

---

# Table of Contents

1	Introduction.....	5
1.1	Goals and Non-Goals.....	5
1.2	Requirements.....	6
1.3	Namespace.....	6
1.4	Schema and WSDL Files.....	7
1.5	Terminology.....	7
1.5.1	Notational Conventions.....	8
1.6	Normative References.....	8
1.7	Non-Normative References.....	9
2	Web Services Trust Model.....	10
2.1	Models for Trust Brokering and Assessment.....	12
2.2	Token Acquisition.....	12
2.3	Out-of-Band Token Acquisition.....	12
2.4	Trust Bootstrap.....	12
3	Security Token Service Framework.....	14
3.1	Requesting a Security Token.....	14
3.2	Returning a Security Token.....	15
3.3	Binary Secrets.....	17
3.4	Composition.....	17
4	Issuance Binding.....	18
4.1	Requesting a Security Token.....	18
4.2	Returning a Security Token.....	21
4.2.1	wsp:AppliesTo in RST and RSTR.....	22
4.2.2	Requested References.....	23
4.2.3	Keys and Entropy.....	23
4.2.4	Returning Computed Keys.....	24
4.2.5	Sample Response with Encrypted Secret.....	25
4.2.6	Sample Response with Unencrypted Secret.....	25
4.2.7	Sample Response with Token Reference.....	25
4.2.8	Sample Response without Proof-of-Possession Token.....	26
4.3	Returning Multiple Security Tokens.....	26
4.3.1	Zero or One Proof-of-Possession Token Case.....	26
4.3.2	More Than One Proof-of-Possession Tokens Case.....	27
4.4	Returning Security Tokens in Headers.....	28
5	Renewal Binding.....	30
6	Cancel Binding.....	33
7	Validation Binding.....	35
8	Negotiation and Challenge Extensions.....	38
8.1	Negotiation and Challenge Framework.....	39
8.2	Signature Challenges.....	39
8.3	Binary Exchanges and Negotiations.....	40
8.4	Key Exchange Tokens.....	41
8.5	Custom Exchanges.....	42

8.6	Signature Challenge Example .....	42
8.7	Custom Exchange Example .....	44
8.8	Protecting Exchanges.....	45
8.9	Authenticating Exchanges .....	46
9	Key and Token Parameter Extensions.....	48
9.1	On-Behalf-Of Parameters .....	48
9.2	Key and Encryption Requirements .....	48
9.3	Delegation and Forwarding Requirements .....	52
9.4	Policies.....	52
9.5	Authorized Token Participants.....	53
10	Key Exchange Token Binding .....	55
11	Error Handling .....	57
12	Security Considerations .....	58
A.	Key Exchange .....	60
A.1	Ephemeral Encryption Keys.....	60
A.2	Requestor-Provided Keys .....	61
A.3	Issuer-Provided Keys .....	61
A.4	Composite Keys .....	61
A.5	Key Transfer and Distribution.....	62
A.5.1	Direct Key Transfer .....	62
A.5.2	Brokered Key Distribution .....	63
A.5.3	Delegated Key Transfer .....	63
A.5.4	Authenticated Request/Reply Key Transfer.....	64
A.6	Perfect Forward Secrecy.....	65
B.	WSDL .....	67
C.	Acknowledgements .....	69
D.	Non-Normative Text .....	71
E.	Revision History.....	72

---

# 1 Introduction

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

6  
7 In order to secure a communication between two parties, the two parties must exchange  
8 security credentials (either directly or indirectly). However, each party needs to determine  
9 if they can "trust" the asserted credentials of the other party.

10  
11 In this specification we define extensions to [WS-Security] that provide:

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

14  
15 Using these extensions, applications can engage in secure communication designed to work  
16 with the general Web services framework, including WSDL service descriptions, UDDI  
17 businessServices and bindingTemplates, and [SOAP] messages.

18  
19 To achieve this, this specification introduces a number of elements that are used to request  
20 security tokens and broker trust relationships.

21  
22 This specification defines a number of extensions; compliant services are NOT REQUIRED to  
23 implement everything defined in this specification. However, if a service implements an  
24 aspect of the specification, it MUST comply with the requirements specified (e.g. related  
25 "MUST" statements).

26  
27 Section 12 is non-normative.

## 28 1.1 Goals and Non-Goals

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

33  
34 This specification is intended to provide a flexible set of mechanisms that can be used to  
35 support a range of security protocols; this specification intentionally does not describe  
36 explicit fixed security protocols.

37  
38 As with every security protocol, significant efforts must be applied to ensure that specific  
39 profiles and message exchanges constructed using WS-Trust are not vulnerable to attacks  
40 (or at least that the attacks are understood).

41

42 The following are explicit non-goals for this document:

- 43 • Password authentication
- 44 • Token revocation
- 45 • Management of trust policies

46

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

- 48 • Establishing a security context token
- 49 • Key derivation

## 50 1.2 Requirements

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

- 53 • Requesting and obtaining security tokens
- 54 • Managing trusts and establishing trust relationships
- 55 • Establishing and assessing trust relationships

## 56 1.3 Namespace

57 The [XML namespace] [URI] that MUST be used by implementations of this specification is:

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

59 The following namespaces are used in this document:

Prefix	Namespace
S11	<a href="http://schemas.xmlsoap.org/soap/envelope/">http://schemas.xmlsoap.org/soap/envelope/</a>
S12	<a href="http://www.w3.org/2003/05/soap-envelope">http://www.w3.org/2003/05/soap-envelope</a>
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>
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>
wst	<a href="http://docs.oasis-open.org/ws-sx/ws-trust/200512">http://docs.oasis-open.org/ws-sx/ws-trust/200512</a>
ds	<a href="http://www.w3.org/2000/09/xmldsig#">http://www.w3.org/2000/09/xmldsig#</a>
xenc	<a href="http://www.w3.org/2001/04/xmlenc#">http://www.w3.org/2001/04/xmlenc#</a>
wsp	<a href="http://schemas.xmlsoap.org/ws/2004/09/policy">http://schemas.xmlsoap.org/ws/2004/09/policy</a>
wsa	<a href="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://schemas.xmlsoap.org/ws/2004/08/addressing</a>

XS

<http://www.w3.org/2001/XMLSchema>

---

## 60 1.4 Schema and WSDL Files

---

61 The schema for this specification can be located at:

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

---

63

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

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

---

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

---

## 69 1.5 Terminology

---

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

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

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

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

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

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

88 **Trust Engine** – The *trust engine* of a Web service is a conceptual component that evaluates  
89 the security-related aspects of a message as described in [section 3](#) below.

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

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

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

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

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

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

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

112 Namespace URIs of the general form "some-URI" represents some application-dependent or context-  
113 dependent URI as defined in [RFC2396].

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

## 116 1.5.1 Notational Conventions

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

120

121 Namespace URIs of the general form "some-URI" represents some application-dependent or  
122 context-dependent URI as defined in [RFC2396].

## 123 1.6 Normative References

124 [RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*,  
125 <http://www.ietf.org/rfc/rfc2119.txt>, IETF RFC 2119, March 1997.

126 [RFC2246] IETF Standard, "The TLS Protocol," January 1999.

127 [SOAP] W3C Note, "SOAP: Simple Object Access Protocol 1.1," 08 May 2000.

128 [SOAP12] W3C Recommendation, "SOAP 1.2 Part 1: Messaging Framework," 24 June  
129 2003.

130 [URI] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers (URI):  
131 Generic Syntax," RFC 2396, MIT/LCS, U.C. Irvine, Xerox Corporation, August  
132 1998.

133 [WS-Addressing] "Web Services Addressing (WS-Addressing)," BEA, IBM, Microsoft, SAP, Sun  
134 Microsystems, Inc., August 2004.

135 [WS-Federation] "Web Services Federation Language," BEA, IBM, Microsoft, RSA Security,  
136 VeriSign, July 2003.

137 [WS-Policy] "Web Services Policy Framework," BEA, IBM, Microsoft, SAP, Sonic Software,  
138 Verisign, September 2004.

139 [WS-PolicyAttachment] "Web Services Policy Attachment," BEA, IBM, Microsoft, SAP, Sonic  
140 Software, Verisign, September 2004.

141 [WS-Security] OASIS, "Web Services Security: SOAP Message Security," 15 March 2004.

142 [WS-SecurityPolicy] "Web Services Security Policy Language," IBM, Microsoft, RSA Security,  
143 VeriSign, December 2002.

144 [XML-C14N] W3C Candidate Recommendation, "Canonical XML Version 1.0," 26 October  
145 2000.

---

146	[XML-Encrypt]	W3C Recommendation, " <a href="#">XML Encryption Syntax and Processing</a> ," 10
147		December, 2002.
148	[XML-ns]	W3C Recommendation, " <a href="#">Namespaces in XML</a> ," 14 January 1999.
149	[XML-Schema1]	W3C Recommendation, " <a href="#">XML Schema Part 1: Structures</a> ," 2 May 2001.
150	[XML-Schema2]	W3C Recommendation, " <a href="#">XML Schema Part 2: Datatypes</a> ," 2 May 2001.
151	[XML-Signature]	W3C Candidate Recommendation, " <a href="#">XML-Signature Syntax and Processing</a> ," 31
152		October 2000.
153	[X509]	S. Santesson, et al, " <a href="#">Internet X.509 Public Key Infrastructure Qualified Certificates</a>
154		<a href="#">Profile</a> ."
155	[Kerberos]	J. Kohl and C. Neuman, "The Kerberos Network Authentication Service (V5),"
156		<a href="#">RFC 1510</a> , September 1993.

---

## 157 **1.7 Non-Normative References**

---

158 **[Reference]** [Full reference citation]

---

## 159 2 Web Services Trust Model

---

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

---

166  
167 Authentication of requests is based on a combination of optional network and transport-  
168 provided security and information (claims) proven in the message. Requestors can  
169 authenticate recipients using network and transport-provided security, claims proven in  
170 messages, and encryption of the request using a key known to the recipient.

---

171  
172 One way to demonstrate authorized use of a security token is to include a digital signature  
173 using the associated secret key (from a proof-of-possession token). This allows a requestor  
174 to prove a required set of claims by associating security tokens (e.g., PKIX, X.509  
175 certificates) with the messages.

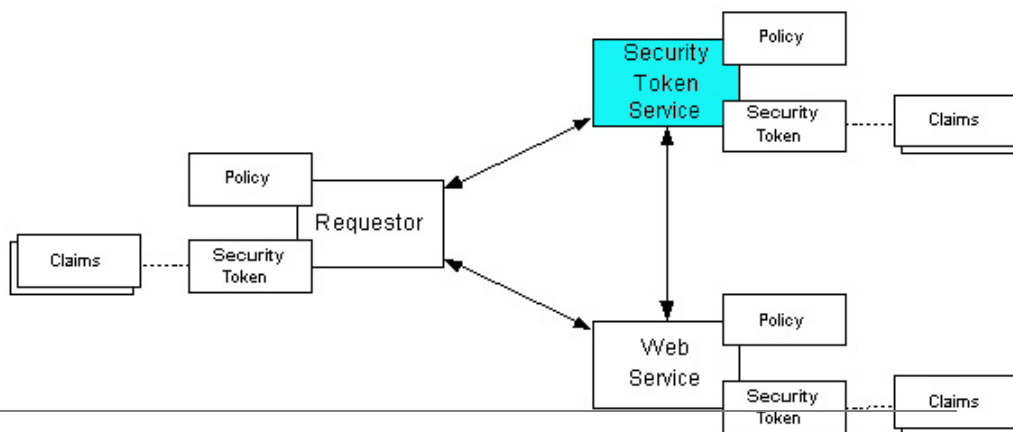
---

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

188  
189 This model is illustrated in the figure below, showing that any requestor may also be a  
190 service, and that the Security Token Service is a Web service (that is, it may express policy  
191 and require security tokens).

---

192



193

194 This general security model – claims, policies, and security tokens – subsumes and supports  
 195 several more specific models such as identity-based authorization, access control lists, and  
 196 capabilities-based authorization. It allows use of existing technologies such as X.509 public-  
 197 key certificates, XML-based tokens, Kerberos shared-secret tickets, and even password  
 198 digests. The general model in combination with the [WS-Security] and [WS-Policy]  
 199 primitives is sufficient to construct higher-level key exchange, authentication, policy-based  
 200 access control, auditing, and complex trust relationships.

201

202 In the figure above the arrows represent possible communication paths; the requestor may  
 203 obtain a token from the security token service, or it may have been obtained indirectly. The  
 204 requestor then demonstrates authorized use of the token to the Web service. The Web  
 205 service either trusts the issuing security token service or may request a token service to  
 206 validate the token (or the Web service may validate the token itself).

207

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

- 212 1. Verify that the claims in the token are sufficient to comply with the policy and that  
 213 the message conforms to the policy.
- 214 2. Verify that the attributes of the claimant are proven by the signatures. In brokered  
 215 trust models, the signature may not verify the identity of the claimant – it may verify  
 216 the identity of the intermediary, who may simply assert the identity of the claimant.  
 217 The claims are either proven or not based on policy.
- 218 3. Verify that the issuers of the security tokens (including all related and issuing  
 219 security token) are trusted to issue the claims they have made. The trust engine  
 220 may need to externally verify or broker tokens (that is, send tokens to a security  
 221 token service in order to exchange them for other security tokens that it can use  
 222 directly in its evaluation).

223

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

226 In this specification we define how security tokens are requested and obtained from security  
 227 token services and how these services may broker trust and trust policies so that services  
 228 can perform step 3.

---

229 Network and transport protection mechanisms such as IPsec or TLS/SSL can be used in  
230 conjunction with this specification to support different security requirements and scenarios.  
231 If available, requestors should consider using a network or transport security mechanism to  
232 perform pre-authentication of the recipient when requesting, validating, or renewing  
233 security tokens as an added level of security.

---

234  
235 The [WS-Federation] specification builds on this specification to define mechanisms for  
236 brokering and federating trust, identity, and claims. Examples are provided in [WS-  
237 Federation] illustrating different trust scenarios and usage patterns.

---

## 238 **2.1 Models for Trust Brokering and Assessment**

---

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

---

## 242 **2.2 Token Acquisition**

---

243 As part of a message flow, a request may be made of a security token service to exchange  
244 a security token (or some proof) of one form for another. The exchange request can be  
245 made either by a requestor or by another party on the requestor's behalf. If the security  
246 token service trusts the provided security token (for example, because it trusts the issuing  
247 authority of the provided security token), and the request can prove possession of that  
248 security token, then the exchange is processed by the security token service.

---

249  
250 The previous paragraph illustrates an example of token acquisition in a direct trust  
251 relationship. In the case of a delegated request (one in which another party provides the  
252 request on behalf of the requestor rather than the requestor presenting it themselves), the  
253 security token service generating the new token may not need to trust the authority that  
254 issued the original token provided by the original requestor since it does trust the security  
255 token service that is engaging in the exchange for a new security token. The basis of the  
256 trust is the relationship between the two security token services.

---

## 257 **2.3 Out-of-Band Token Acquisition**

---

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

---

## 264 **2.4 Trust Bootstrap**

---

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

---

270 There are several different mechanisms that can be used to bootstrap trust for a service.  
271 These mechanisms are non-normative and are not required in any way. That is, services

---

272 are free to bootstrap trust and establish trust among a domain of services or extend this  
273 trust to other domains using any mechanism.

---

274

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

---

278

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

---

284

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

---

## 290 3 Security Token Service Framework

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

293  
294 A requestor sends a request, and if the policy permits and the recipient's requirements are  
295 met, then the requestor receives a security token response. This process uses the  
296 `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>` elements  
297 respectively. These elements are passed as the payload to specific WSDL ports (described  
298 in [section 2.3](#)) that are implemented by security token services.

299  
300 This framework does not define specific actions; each binding defines its own actions.  
301 When requesting and returning security tokens additional parameters can be included in  
302 requests, or provided in responses to indicate server-determined (or used) values. If a  
303 requestor specifies a specific value that isn't supported by the recipient, then the recipient  
304 MAY fault with a `wst:InvalidRequest` (or a more specific fault code), or they MAY return a  
305 token with their chosen parameters that the requestor may then choose to discard because  
306 it doesn't meet their needs.

307  
308 The requesting and returning of security tokens can be used for a variety of purposes.  
309 Bindings define how this framework is used for specific usage patterns. Other specifications  
310 may define specific bindings and profiles of this mechanism for additional purposes.

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

316  
317 The [WS-Security] specification defines and illustrates time references in terms of the  
318 `dateTime` type defined in XML Schema. It is RECOMMENDED that all time references use  
319 this type. It is further RECOMMENDED that all references be in UTC time. Requestors and  
320 receivers SHOULD NOT rely on other applications supporting time resolution finer than  
321 milliseconds. Implementations MUST NOT generate time instants that specify leap seconds.  
322 Also, any required clock synchronization is outside the scope of this document.

323  
324 The following sections describe the basic structure of token request and response elements  
325 identifying the general mechanisms and most common sub-elements. Specific bindings  
326 extend these elements with binding-specific sub-elements. That is, sections 5.1 and 5.2  
327 should be viewed as patterns or templates on which specific bindings build.

328 It should be noted that all time references use the XML Schema `dateTime` type and use  
329 universal time.

---

### 330 3.1 Requesting a Security Token

331 The `<wst:RequestSecurityToken>` element (RST) is used to request a security token (for  
332 any purpose). This element SHOULD be signed by the requestor, using tokens

333 contained/referenced in the request that are relevant to the request. If using a signed  
334 request, the requestor MUST prove any required claims to the satisfaction of the security  
335 token service.

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

338 The syntax for this element is as follows:

```
339     <wst:RequestSecurityToken Context="...">  
340         <wst:TokenType>...</wst:TokenType>  
341         <wst:RequestType>...</wst:RequestType>  
342         ...  
343     </wst:RequestSecurityToken>
```

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

345 */wst:RequestSecurityToken*

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

347 */wst:RequestSecurityToken/@Context*

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

352 */wst:RequestSecurityToken/wst:TokenType*

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

357 */wst:RequestSecurityToken/wst:RequestType*

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

364 */wst:RequestSecurityToken/{any}*

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

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

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

## 372 **3.2 Returning a Security Token**

373 The `<wst:RequestSecurityTokenResponse>` element (RSTR) is used to return a security  
374 token or response to a security token request.

375

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

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

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

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

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

401 The syntax for this element is as follows:

```
402 <wst:RequestSecurityTokenResponse Context="...">  
403   <wst:TokenType>...</wst:TokenType>  
404   <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>  
405  
406 </wst:RequestSecurityTokenResponse>
```

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

408 */wst:RequestSecurityTokenResponse*

409 This is the response to a security token request.

410 */wst:RequestSecurityTokenResponse/@Context*

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

416 */wst:RequestSecurityTokenResponse/wst:TokenType*

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

418 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

419 This optional element is used to return the requested security token. Normally the requested  
420 security token is the contents of this element but a security token reference MAY be used instead.  
421 For example, if the requested security token is used in securing the message, then the security  
422 token is placed into the <wsse:Security> header (as described in [WS-Security]) and a  
423 <wsse:SecurityTokenReference> element is placed inside of the  
424 <wst:RequestedSecurityToken> element to reference the token in the <wsse:Security>  
425 header. The response MAY contain a token reference where the token is located at a URI  
426 outside of the message. In such cases the recipient is assumed to know how to fetch the token

427 from the URI address or specified endpoint reference. It should be noted that when the token is  
428 not returned as part of the message it cannot be secured, so a secure communication  
429 mechanism SHOULD be used to obtain the token.

430 */wst:RequestSecurityTokenResponse/{any}*

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

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

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

### 436 3.3 Binary Secrets

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

445 *.../wst:BinarySecret*

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

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

449 This optional attribute indicates the type of secret being encoded. The pre-defined values are  
450 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)

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

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

### 454 3.4 Composition

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

---

## 4 Issuance Binding

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

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

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

466 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue`  
467 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Issue`

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

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

470  
471 The mechanisms defined in this specification apply to both symmetric and asymmetric keys.  
472 It should be noted that in practice, asymmetric key usage often differs as it is common to  
473 reuse existing asymmetric keys rather than regenerate due to the time cost and desire to  
474 map to a common public key. In such cases a request might be made for an asymmetric  
475 token providing the public key and proving ownership of the private key. The public key is  
476 then used in the issued token.

477  
478 As an example, a Kerberos KDC could provide the services defined in this specification to  
479 make tokens available; similarly, so can a public key infrastructure. In such cases, the  
480 issuing authority is the security token service. A public key directory is not really a security  
481 token service per se; however, such a service MAY implement token retrieval as a form of  
482 issuance. As well, it is possible to bridge environments (security technologies) using PKI for  
483 authentication or bootstrapping to a symmetric key.

484  
485 This binding provides a general token issuance action that can be used for any type of token  
486 being requested. Other bindings MAY use separate actions if they have specialized  
487 semantics.

488  
489 This binding supports the optional use of exchanges during the token acquisition process as  
490 well as the optional use of the key extensions described in a later section. Subsequent  
491 profiles are needed to describe specific behaviors (and exclusions) when different  
492 combinations are used (e.g. multiple simultaneous exchanges).

---

### 4.1 Requesting a Security Token

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

498 `<wst:RequestSecurityToken>`  
499 `<wst:TokenType>...</wst:TokenType>`  
500 `<wst:RequestType>...</wst:RequestType>`

```

501
502     <wsp:AppliesTo>...</wsp:AppliesTo>
503     <wst:Claims Dialect="...">...</wst:Claims>
504     <wst:Entropy>
505         <wst:BinarySecret>...</wst:BinarySecret>
506     </wst:Entropy>
507     <wst:Lifetime>
508         <wsu:Created>...</wsu:Created>
509         <wsu:Expires>...</wsu:Expires>
510     </wst:Lifetime>
511 </wst:RequestSecurityToken>

```

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

513 */wst:RequestSecurityToken/wst:TokenType*

514 If this optional element is not specified in an issue request, it is RECOMMENDED that the  
515 optional element <wsp:AppliesTo> be used to indicate the target where this token will be used  
516 (similar to the Kerberos target service model). This assumes that a token type can be inferred  
517 from the target scope specified. That is, either the <wst:TokenType> or the  
518 <wsp:AppliesTo> element SHOULD be defined within a request. If both the  
519 <wst:TokenType> and <wsp:AppliesTo> elements are defined, the <wsp:AppliesTo>  
520 element takes precedence (for the current request only) in case the target scope requires a  
521 specific type of token.

522 */wst:RequestSecurityToken/wsp:AppliesTo*

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

530 */wst:RequestSecurityToken/wst:Claims*

531 This optional element requests a specific set of claims. In most cases, this element contains  
532 claims identified as required in a service's policy. Refer to [WS-Policy] for examples of how a  
533 service uses policy to specify claim requirements. The @Dialect attribute specifies a URI to  
534 indicate the syntax of the claims. No URIs are predefined; refer to profiles and other  
535 specifications to define these URIs.

536 */wst:RequestSecurityToken/wst:Entropy*

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

541 */wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret*

542 This optional element specifies a base64 encoded sequence of octets representing the  
543 requestor's entropy. The value can contain either a symmetric or the private key of an  
544 asymmetric key pair, or any suitable key material. The format is assumed to be understood by  
545 the requestor because the value space may be (a) fixed, (b) indicated via policy, (c) inferred from  
546 the indicated token aspects and/or algorithms, or (d) determined from the returned token. (See  
547 Section 5.3)

548 */wst:RequestSecurityToken/wst:Lifetime*

549 This optional element is used to specify the desired valid time range (time window during which  
550 the token is valid for use) for the returned security token. That is, to request a specific time  
551 interval for using the token. The issuer is not obligated to honor this range – they may return a

552 more (or less) restrictive interval. It is RECOMMENDED that the issuer return this element with  
553 issued tokens (in the RSTR) so the requestor knows the actual validity period without having to  
554 parse the returned token.

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

556 This optional element represents the creation time of the security token. Within the SOAP  
557 processing model, creation is the instant that the infoset is serialized for transmission. The  
558 creation time of the token SHOULD NOT differ substantially from its transmission time. The  
559 difference in time should be minimized. If this time occurs in the future then this is a request for a  
560 post-dated token. If this attribute isn't specified, then the current time is used as an initial period.

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

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

567

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

```
574 <S11:Envelope xmlns:S11="..." xmlns:wsua="..." xmlns:wsse="..."
575     xmlns:xenc="..." xmlns:wst="...">
576   <S11:Header>
577     ...
578     <wsse:Security>
579       <xenc:ReferenceList>...</xenc:ReferenceList>
580       <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>
581       <ds:Signature xmlns:ds="...">
582         ...
583         <ds:KeyInfo>
584           <wsse:SecurityTokenReference>
585             <wsse:Reference URI="#myToken"/>
586           </wsse:SecurityTokenReference>
587         </ds:KeyInfo>
588       </ds:Signature>
589     </wsse:Security>
590     ...
591   </S11:Header>
592   <S11:Body wsua:Id="req">
593     <wst:RequestSecurityToken>
594       <wst:TokenType>
595         http://example.org/mySpecialToken
596       </wst:TokenType>
597       <wst:RequestType>
598         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
599       </wst:RequestType>
600     </wst:RequestSecurityToken>
601   </S11:Body>
602 </S11:Envelope>
```

## 603 4.2 Returning a Security Token

604 When returning a security token, the following optional elements MAY be included in the  
605 response. The syntax for these elements is as follows (note that the base elements  
606 described above are included here italicized for completeness):

```
607     <wst:RequestSecurityTokenResponse>  
608         <wst:TokenType>...</wst:TokenType>  
609         <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>  
610         ...  
611         <wsp:AppliesTo>...</wsp:AppliesTo>  
612         <wst:RequestedAttachedReference>  
613             ...  
614             </wst:RequestedAttachedReference>  
615         <wst:RequestedUnattachedReference>  
616             ...  
617             </wst:RequestedUnattachedReference>  
618         <wst:RequestedProofToken>...</wst:RequestedProofToken>  
619         <wst:Entropy>  
620             <wst:BinarySecret>...</wst:BinarySecret>  
621             </wst:Entropy>  
622         <wst:Lifetime>...</wst:Lifetime>  
623     </wst:RequestSecurityTokenResponse>
```

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

625 */wst:RequestSecurityTokenResponse/wsp:AppliesTo*

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

630 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

631 This optional element is used to return the requested security token. This element is optional, but  
632 it is REQUIRED that at least one of `<wst:RequestedSecurityToken>` or  
633 `<wst:RequestedProofToken>` be returned unless there is an error or part of an on-going  
634 message exchange (e.g. negotiation). If returning more than one security token see section 6.3,  
635 Returning Multiple Security Tokens.

636 */wst:RequestSecurityTokenResponse/wst:RequestedAttachedReference*

637 Since returned tokens are considered opaque to the requestor, this optional element is specified  
638 to indicate how to reference the returned token when that token doesn't support references using  
639 URI fragments (XML ID). This element contains a `<wsse:SecurityTokenReference>`  
640 element that can be used *verbatim* to reference the token (when the token is placed inside a  
641 message). Typically tokens allow the use of `wsu:id` so this element isn't required. Note that a  
642 token MAY support multiple reference mechanisms; this indicates the issuer's preferred  
643 mechanism. When encrypted tokens are returned, this element is not needed since the  
644 `<xenc:EncryptedData>` element supports an ID reference. If this element is not present in the  
645 RSTR then the recipient can assume that the returned token (when present in a message)  
646 supports references using URI fragments.

647 */wst:RequestSecurityTokenResponse/wst:RequestedUnattachedReference*

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

654 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken*

655 This optional element is used to return the proof-of-possession token associated with the  
656 requested security token. Normally the proof-of-possession token is the contents of this element  
657 but a security token reference MAY be used instead. The token (or reference) is specified as the  
658 contents of this element. For example, if the proof-of-possession token is used as part of the  
659 securing of the message, then it is placed in the <wsse:Security> header and a  
660 <wsse:SecurityTokenReference> element is used inside of the  
661 <wst:RequestedProofToken> element to reference the token in the <wsse:Security>  
662 header. This element is optional, but it is REQUIRED that at least one of  
663 <wst:RequestedSecurityToken> or <wst:RequestedProofToken> be returned unless  
664 there is an error.

665 */wst:RequestSecurityTokenResponse/wst:Entropy*

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

670 */wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret*

671 This optional element specifies a base64 encoded sequence of octets represent the responder's  
672 entropy. (See Section 5.3)

673 */wst:RequestSecurityTokenResponse/wst:Lifetime*

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

## 677 4.2.1 wsp:AppliesTo in RST and RSTR

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

- 682 • If neither the requestor nor the issuer specifies a scope then the scope of the issued  
683 token is implied.
- 684 • If the requestor specifies a scope and the issuer does not then the scope of the token  
685 is assumed to be that specified by the requestor.
- 686 • If the requestor does not specify a scope and the issuer does specify a scope then  
687 the scope of the token is as defined by the issuers scope
- 688 • If both requestor and issuer specify a scope then there are two possible outcomes:
  - 689 ○ If both the issuer and requestor specify the same scope then the issued token  
690 has that scope.
  - 691 ○ If the issuer specifies a wider scope than the requestor then the issued token  
692 has the scope specified by the issuer.

693

694 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.

## 695 4.2.2 Requested References

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

- 700 • If a `<wst:RequestedAttachedReference>` element is NOT returned in the RSTR then the  
701 client SHOULD assume that the token can be referenced by ID. Alternatively, the  
702 client MAY use token-specific knowledge to construct an STR.
- 703 • If a `<wst:RequestedAttachedReference>` element is returned in the RSTR then the  
704 token cannot be referred to by ID. The supplied STR MUST be used to refer to the  
705 token.
- 706 • If a `<wst:RequestedUnattachedReference>` element is returned then the server MAY  
707 reference the token using the supplied STR when sending responses back to the  
708 client. Thus the client MUST be prepared to resolve the supplied STR to the  
709 appropriate token. Note: the server SHOULD NOT send the token back to the client  
710 as the token is often tailored specifically to the server (i.e. it may be encrypted for  
711 the server). References to the token in subsequent messages, whether sent by the  
712 client or the server, that omit the token MUST use the supplied STR.

## 713 4.2.3 Keys and Entropy

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

- 716 • In the case of specific keys, a `<wst:RequestedProofToken>` element is included in  
717 the response which indicates the specific key(s) to use unless the key was provided  
718 by the requestor (in which case there is no need to return it).
- 719 • In the case of partial, the `<wst:Entropy>` element is included in the response, which  
720 indicates partial key material from the issuer (not the full key) that is combined (by  
721 each party) with the requestor's entropy to determine the resulting key(s). In this  
722 case a `<wst:ComputedKey>` element is returned inside the  
723 `<wst:RequestedProofToken>` to indicate how the key is computed.
- 724 • In the case of omitted, an existing key is used or the resulting token is not directly  
725 associated with a key.

726  
727 The decision as to which path to take is based on what the requestor provides, what the  
728 issuer provides, and the issuer's policy.

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

738

739 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

## 740 4.2.4 Returning Computed Keys

741 As previously described, in some scenarios the key(s) resulting from a token request are not  
742 directly returned and must be computed. One example of this is when both parties provide  
743 entropy and are combined to make the shared secret. To indicate a computed key, the  
744 <wst:ComputedKey> element is returned inside the <wst:RequestedProofToken> to indicate  
745 how the key is computed. The following is a syntax overview of the <wst:ComputedKey>  
746 element:

```

747 ...
748 <wst:RequestSecurityTokenResponse>
749 <wst:RequestedProofToken>
750 <wst:ComputedKey>...</wst:ComputedKey>
751 </wst:RequestedProofToken>
752 </wst:RequestSecurityTokenResponse>
753 ...

```

754

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

756 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken/wst:ComputedKey*

757 The value of this element is a URI describing how to compute the key. While this can be  
758 extended by defining new URIs in other bindings and profiles, the following URI pre-defines one  
759 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: key = P_SHA1 (Ent <sub>REQ</sub> , Ent <sub>RES</sub> )

#### 760 4.2.5 Sample Response with Encrypted Secret

761 The following is a sample security token response. In this example the token requested in  
762 [section 6.1](#) is returned. Additionally a proof-of-possession token element is returned  
763 containing the secret key associated with the `<wst:RequestedSecurityToken>` encrypted  
764 for the requestor (note that this assumes that the requestor has a shared secret with the  
765 issuer or a public key).

```
766 ...
767     <wst:RequestSecurityTokenResponse>
768         <wst:RequestedSecurityToken>
769             <xyz:CustomToken xmlns:xyz="...">
770
771             </xyz:CustomToken>
772         </wst:RequestedSecurityToken>
773         <wst:RequestedProofToken>
774             <xenc:EncryptedKey Id="newProof">
775
776             </xenc:EncryptedKey>
777         </wst:RequestedProofToken>
778     </wst:RequestSecurityTokenResponse>
779 ...
```

#### 780 4.2.6 Sample Response with Unencrypted Secret

781 The example below is an alternative form where the secret is passed in the clear because  
782 the transport is providing confidentiality:

```
783 ...
784     <wst:RequestSecurityTokenResponse>
785         <wst:RequestedSecurityToken>
786             <xyz:CustomToken xmlns:xyz="...">
787
788             </xyz:CustomToken>
789         </wst:RequestedSecurityToken>
790         <wst:RequestedProofToken>
791             <wst:BinarySecret>...</wst:BinarySecret>
792         </wst:RequestedProofToken>
793     </wst:RequestSecurityTokenResponse>
794 ...
```

#### 795 4.2.7 Sample Response with Token Reference

796 If the returned token doesn't allow the use of the `wsu:Id` attribute, then a  
797 `<wst:RequestedTokenReference>` is returned as illustrated below. In this example, the  
798 returned token has a URI which is referenced.

```
799 ...
800     <wst:RequestSecurityTokenResponse>
801         <wst:RequestedSecurityToken>
802             <xyz:CustomToken ID="urn:fabrikam123:5445" xmlns:xyz="...">
803
804             ...
805         </wst:RequestedSecurityToken>
806     </wst:RequestSecurityTokenResponse>
807 ...
```

```

804         </xyz:CustomToken>
805     </wst:RequestedSecurityToken>
806     <wst:RequestedTokenReference>
807         <wsse:SecurityTokenReference>
808             <wsse:Reference URI="urn:fabrikam123:5445"/>
809         </wsse:SecurityTokenReference>
810     </wst:RequestedTokenReference>
811     ...
812 </wst:RequestSecurityTokenResponse>
813 ...

```

814

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

## 820 4.2.8 Sample Response without Proof-of-Possession Token

821 The example below illustrates a response that doesn't include a proof-of-possession token.  
822 For example, if the basis of the request were a public key token and another public key  
823 token is returned with the same public key, the proof-of-possession token from the original  
824 token is reused (no new proof-of-possession token is required).

```

825     ...
826     <wst:RequestSecurityTokenResponse>
827         <wst:RequestedSecurityToken>
828             <xyz:CustomToken xmlns:xyz="...">
829                 ...
830             </xyz:CustomToken>
831         </wst:RequestedSecurityToken>
832     </wst:RequestSecurityTokenResponse>
833     ...

```

## 834 4.3 Returning Multiple Security Tokens

835 In some cases a response MAY provide multiple tokens. These can be divided into two  
836 cases: zero or one proof-of-possession tokens and responses with more than one proof-of-  
837 possession tokens.

### 838 4.3.1 Zero or One Proof-of-Possession Token Case

839 In the zero or single proof-of-possession token case, a primary token and one or more  
840 tokens are returned. The returned tokens either use the same proof-of-possession token  
841 (one is returned), or no proof-of-possession token is returned. The tokens are returned  
842 (one each) in the response. The following example illustrates this case. In this example a  
843 supporting authorization token is returned that has no separate proof-of-possession token  
844 as it is secured using the same proof-of-possession token that was returned.

```

845     ...
846     <wst:RequestSecurityTokenResponse>
847         <wst:RequestedSecurityToken>
848             <xyz:CustomToken xmlns:xyz="...">
849                 ...
850             </xyz:CustomToken>
851         </wst:RequestedSecurityToken>
852         <wst:RequestedProofToken>

```

```

853         <xenc:EncryptedKey Id="newProof">
854             ...
855         </xenc:EncryptedKey>
856     </wst:RequestedProofToken>
857 </wst:RequestSecurityTokenResponse>
858     ...

```

### 859 4.3.2 More Than One Proof-of-Possession Tokens Case

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

```

864 <wst:RequestSecurityTokenResponseCollection>
865     <wst:RequestSecurityTokenResponse>
866         ...
867     </wst:RequestSecurityTokenResponse>
868     <wst:RequestSecurityTokenResponse>
869         ...
870     </wst:RequestSecurityTokenResponse>
871     ...
872 </wst:RequestSecurityTokenResponseCollection>

```

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

874 */wst:RequestSecurityTokenResponseCollection*

875 This element is used to provide multiple RSTR responses, each of which has separate key  
876 information.

877 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

878 Two or more RSTR elements are returned in the collection.

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

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

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

```

883 ...
884 <wst:RequestSecurityTokenResponseCollection>
885     <wst:RequestSecurityTokenResponse>
886         <wst:RequestedSecurityToken>
887             <xyz:CustomToken xmlns:xyz="...">
888                 ...
889             </xyz:CustomToken>
890         </wst:RequestedSecurityToken>
891         <wst:RequestedProofToken>
892             <xenc:EncryptedKey Id="newProofA">
893                 ...
894             </xenc:EncryptedKey>
895         </wst:RequestedProofToken>
896     </wst:RequestSecurityTokenResponse>
897     <wst:RequestSecurityTokenResponse>
898         <wst:RequestedSecurityToken>
899             <abc:CustomToken xmlns:abc="...">
900                 ...
901             </abc:CustomToken>
902         </wst:RequestedSecurityToken>
903         <wst:RequestedProofToken>
904             <xenc:EncryptedKey Id="newProofB">
905                 ...

```

```
906         </xenc:EncryptedKey>
907         </wst:RequestedProofToken>
908     </wst:RequestSecurityTokenResponse>
909 </wst:RequestSecurityTokenResponseCollection>
910 ...
```

## 911 4.4 Returning Security Tokens in Headers

912 In certain situations it is useful to issue one or more security tokens as part of a protocol  
913 other than RST/RSTR. This typically requires that the tokens be passed in a SOAP header.  
914 The tokens present in that element can then be referenced from elsewhere in the message.  
915 This section defines a specific header element, whose type is the same as that of the  
916 <wst:RequestSecurityTokenCollection> element (see Section 6.3), that can be used to carry  
917 issued tokens (and associated proof tokens, references etc.) in a message.

```
918 <wst:IssuedTokens>
919   <wst:RequestSecurityTokenResponse>
920   ...
921   </wst:RequestSecurityTokenResponse>+
922 </wst:IssuedTokens>
```

924  
925 The following describes the attributes and elements listed in the schema overview above:

926 */wst:IssuedTokens*

927 This header element carries one or more issued security tokens.

928 */wst:IssuedTokens/wst:RequestSecurityTokenResponse*

929 This element MUST appear at least once. Its meaning and semantics are as defined in  
930 Section 6.2.

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

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

933

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

938

939 It is RECOMMENDED that the <wst:IssuedTokens> header be signed to protect the integrity  
940 of the issued tokens and of the issuance itself. If confidentiality protection of the  
941 <wst:IssuedTokens> header is required then the entire header MUST be encrypted using the  
942 <wsse11:EncryptedHeader> construct. This helps facilitate re-issuance by the receiving party  
943 as that party can re-encrypt the entire header for another party rather than having to  
944 extract and re-encrypt portions of the header.

945

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

```
948 <S:Envelope>
949 <S:Header>
950 <wst:IssuedTokens>
951 <wst:RequestSecurityTokenResponse>
952 <wsp:AppliesTo>
953 <x:SomeContext1 />
954 </wsp:AppliesTo>
955 <wst:RequestedSecurityToken>
956 ...
957 </wst:RequestedSecurityToken>
958 ...
959 </wst:RequestSecurityTokenResponse>
960 <wst:RequestSecurityTokenResponse>
961 <wsp:AppliesTo>
962 <x:SomeContext1 />
963 </wsp:AppliesTo>
964 <wst:RequestedSecurityToken>
965 ...
966 </wst:RequestedSecurityToken>
967 ...
968 </wst:RequestSecurityTokenResponse>
969 </wst:IssuedTokens>
970 <wst:IssuedTokens s:role='http://example.org/somerole' >
971 <wst:RequestSecurityTokenResponse>
972 <wsp:AppliesTo>
973 <x:SomeContext2 />
974 </wsp:AppliesTo>
975 <wst:RequestedSecurityToken>
976 ...
977 </wst:RequestedSecurityToken>
978 ...
979 </wst:RequestSecurityTokenResponse>
980 </wst:IssuedTokens>
981 </S:Header>
982 <S:Body>
983 ...
984 </S:Body>
985 </S:Envelope>
```

---

## 986 5 Renewal Binding

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

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

991  
992 For this binding, the following actions are defined to enable specific processing context to be  
993 conveyed to the recipient:

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

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

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

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

1000  
1001 Other extensions MAY be specified in the request (and the response), but the key semantics  
1002 (size, type, algorithms, scope, etc.) MUST NOT be altered during renewal. Token services  
1003 MAY use renewal as an opportunity to rekey, so the renewal responses MAY include a new  
1004 proof-of-possession token as well as entropy and key exchange elements.

1005  
1006 The request MUST prove authorized use of the token being renewed unless the recipient  
1007 trusts the requestor to make third-party renewal requests. In such cases, the third-party  
1008 requestor MUST prove its identity to the issuer so that appropriate authorization occurs.

1009  
1010 The original proof information SHOULD be proven during renewal.

1011  
1012 The renewal binding allows the use of exchanges during the renewal process. Subsequent  
1013 profiles MAY define restriction around the usage of exchanges.

1014  
1015 During renewal, all key bearing tokens used in the renewal request MUST have an  
1016 associated signature. All non-key bearing tokens MUST be signed. Signature confirmation  
1017 is RECOMMENDED on the renewal response.

1018  
1019 The renewal binding also defines several extensions to the request and response elements.  
1020 The syntax for these extension elements is as follows (note that the base elements  
1021 described above are included here italicized for completeness):

1022 `<wst:RequestSecurityToken>`  
1023 `<wst:TokenType>...</wst:TokenType>`  
1024 `<wst:RequestType>...</wst:RequestType>`  
1025 `...`  
1026 `<wst:RenewTarget>...</wst:RenewTarget>`  
1027 `<wst:AllowPostdating/>`  
1028 `<wst:Renewing Allow=... OK=.../>`

1029	<code>&lt;/wst:RequestSecurityToken&gt;</code>
1030	<code>/wst:RequestSecurityToken/wst:RenewTarget</code>
1031	This required element identifies the token being renewed. This MAY contain a
1032	<code>&lt;wsse:SecurityTokenReference&gt;</code> pointing at the token to be renewed or it MAY directly contain
1033	the token to be renewed.
1034	<code>/wst:RequestSecurityToken/wst:AllowPostdating</code>
1035	This optional element indicates that returned tokens should allow requests for postdated tokens.
1036	That is, this allows for tokens to be issued that are not immediately valid (e.g., a token that can be
1037	used the next day).
1038	<code>/wst:RequestSecurityToken/wst:Renewing</code>
1039	This optional element is used to specify renew semantics for types that support this operation.
1040	<code>/wst:RequestSecurityToken/wst:Renewing/@Allow</code>
1041	This optional Boolean attribute is used to request a renewable token. If not specified, the default
1042	value is <i>true</i> . A renewable token is one whose lifetime can be extended. This is done using a
1043	renewal request. The recipient MAY allow renewals without demonstration of authorized use of
1044	the token or they MAY fault.
1045	<code>/wst:RequestSecurityToken/wst:Renewing/@OK</code>
1046	This optional Boolean attribute is used to indicate that a renewable token is acceptable if the
1047	requested duration exceeds the limit of the issuance service. That is, if <i>true</i> then tokens can be
1048	renewed after their expiration. It should be noted that the token is NOT valid after expiration for
1049	any operation except renewal. The default for this attribute is <i>false</i> . It NOT RECOMMENDED to
1050	use this as it can leave you open to certain types of security attacks. Issuers MAY restrict the
1051	period after expiration during which time the token can be renewed. This window is governed by
1052	the issuer's policy.
1053	The following example illustrates a request for a custom token that can be renewed.
1054	<code>&lt;wst:RequestSecurityToken&gt;</code>
1055	<code>&lt;wst:TokenType&gt;</code>
1056	<code>http://example.org/mySpecialToken</code>
1057	<code>&lt;/wst:TokenType&gt;</code>
1058	<code>&lt;wst:RequestType&gt;</code>
1059	<code>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue</code>
1060	<code>&lt;/wst:RequestType&gt;</code>
1061	<code>&lt;wst:Renewing/&gt;</code>
1062	<code>&lt;/wst:RequestSecurityToken&gt;</code>
1063	
1064	The following example illustrates a subsequent renewal request and response (note that for
1065	brevity only the request and response are illustrated). Note that the response includes an
1066	indication of the lifetime of the renewed token.
1067	<code>&lt;wst:RequestSecurityToken&gt;</code>
1068	<code>&lt;wst:TokenType&gt;</code>
1069	<code>http://example.org/mySpecialToken</code>
1070	<code>&lt;/wst:TokenType&gt;</code>
1071	<code>&lt;wst:RequestType&gt;</code>
1072	<code>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Renew</code>
1073	<code>&lt;/wst:RequestType&gt;</code>
1074	<code>&lt;wst:RenewTarget&gt;</code>
1075	<code>... reference to previously issued token ...</code>
1076	<code>&lt;/wst:RenewTarget&gt;</code>
1077	<code>&lt;/wst:RequestSecurityToken&gt;</code>
1078	
1079	<code>&lt;wst:RequestSecurityTokenResponse&gt;</code>

```
1080     <wst:TokenType>
1081         http://example.org/mySpecialToken
1082     </wst:TokenType>
1083     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1084     <wst:Lifetime>...</wst:Lifetime>
1085     ...
1086 </wst:RequestSecurityTokenResponse>
```

---

## 1087 6 Cancel Binding

1088 Using the token request framework, this section defines bindings for requesting security  
1089 tokens to be cancelled:

1090 **Cancel** – When a previously issued token is no longer needed, the Cancel binding can be used  
1091 to cancel the token, terminating its use.

1092

1093 For this binding, the following actions are defined to enable specific processing context to be  
1094 conveyed to the recipient:

1095 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel`  
1096 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Cancel`

1097 For this binding, the `<RequestType>` element uses the following URI:

1098 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel`

1099 Extensions MAY be specified in the request (and the response), but the semantics are not  
1100 defined by this binding.

1101

1102 The request MUST prove authorized use of the token being cancelled unless the recipient  
1103 trusts the requestor to make third-party cancel requests. In such cases, the third-party  
1104 requestor MUST prove its identity to the issuer so that appropriate authorization occurs.

1105 In a cancel request, all key bearing tokens specified MUST have an associated signature. All  
1106 non-key bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the  
1107 closure response.

1108

1109 A cancelled token is no longer valid for authentication and authorization usages.

1110 On success a cancel response is returned. This is an RSTR message with the  
1111 `<wst:RequestedTokenCancelled>` element in the body. On failure, a Fault is raised. It  
1112 should be noted that the cancel RSTR is informational. That is, the security token is  
1113 cancelled once the cancel request is processed.

1114

1115 The syntax of the request is as follows:

1116 `<wst:RequestSecurityToken>`  
1117 `<wst:RequestType>...</wst:RequestType>`  
1118 `...`  
1119 `<wst:CancelTarget>...</wst:CancelTarget>`  
1120 `</wst:RequestSecurityToken>`

1121 *`/wst:RequestSecurityToken/wst:CancelTarget`*

1122 This required element identifies the token being cancelled. Typically this contains a  
1123 `<wsse:SecurityTokenReference>` pointing at the token, but it could also carry the token  
1124 directly.

1125 The following example illustrates a request to cancel a custom token.

1126 `<S11:Envelope>`  
1127 `<S11:Header>`  
1128 `<wsse:Security>`  
1129 `...`

1130	</wsse:Security>
1131	</S11:Header>
1132	<S11:Body>
1133	<wst:RequestSecurityToken>
1134	<wst:RequestType>
1135	http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
1136	</wst:RequestType>
1137	<wst:CancelTarget>
1138	...
1139	</wst:CancelTarget>
1140	</wst:RequestSecurityToken>
1141	</S11:Body>
1142	</S11:Envelope>
1143	
1144	<S11:Envelope>
1145	<S11:Header>
1146	<wsse:Security>
1147	...
1148	</wsse:Security>
1149	</S11:Header>
1150	<S11:Body>
1151	<wst:RequestSecurityTokenResponse>
1152	<wst:RequestedTokenCancelled/>
1153	</wst:RequestSecurityTokenResponse>
1154	</S11:Body>
1155	</S11:Envelope>

---

## 1156 7 Validation Binding

---

1157 Using the token request framework, this section defines bindings for requesting security  
1158 tokens to be validated:

1159 **Validate** – The validity of the specified security token is evaluated and a result is returned. The  
1160 result may be a status, a new token, or both.

1161  
1162 It should be noted that for this binding, a SOAP Envelope MAY be specified as a "security  
1163 token" if the requestor desires the envelope to be validated. In such cases the recipient  
1164 SHOULD understand how to process a SOAP envelope and adhere to SOAP processing  
1165 semantics (e.g., mustUnderstand) of the version of SOAP used in the envelope. Otherwise,  
1166 the recipient SHOULD fault.

1167 For this binding, the following actions are defined to enable specific processing context to be  
1168 conveyed to the recipient:

1169 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate`  
1170 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Validate`

1171  
1172 For this binding, the <RequestType> element contains the following URI:

1173 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate`

1174  
1175 The request provides a token upon which the request is based and optional tokens. As well,  
1176 the optional <wst:TokenType> element in the request can indicate desired type response  
1177 token. This may be any supported token type or it may be the following URI indicating that  
1178 only status is desired:

1179 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Status`

1180  
1181 For some use cases a status token is returned indicating the success or failure of the  
1182 validation. In other cases an authorization token MAY be returned. This binding assumes  
1183 that the validation requestor and provider are known to each other and that the general  
1184 issuance parameters beyond requesting a token type, which is optional, are not needed  
1185 (note that other bindings and profiles could define different semantics).

1186  
1187 For this binding an applicability scope (e.g., <wsp:AppliesTo>) need not be specified. It is  
1188 assumed that the applicability of the validation response relates to the provided information  
1189 (e.g. security token) as understood by the issuing service.

1190  
1191 The validation binding does not allow the use of exchanges.

1192  
1193 The RSTR for this binding carries the following element even if a token is returned (note  
1194 that the base elements described above are included here italicized for completeness):

1195 `<wst:RequestSecurityToken>`  
1196 `<wst:TokenType>...</wst:TokenType>`  
1197 `<wst:RequestType>...</wst:RequestType>`

```

1198
1199     </wst:RequestSecurityToken>
1200
1201     <wst:RequestSecurityTokenResponse>
1202         <wst:TokenType>...</wst:TokenType>
1203         <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1204
1205         <wst:Status>
1206             <wst:Code>...</wst:Code>
1207             <wst:Reason>...</wst:Reason>
1208         </wst:Status>
1209     </wst:RequestSecurityTokenResponse>

```

1210 */wst:RequestSecurityTokenResponse/wst:Status*

1211 When a validation request is made, this element MUST be in the response. The code value  
1212 indicates the results of the validation in a machine-readable form. The accompanying text  
1213 element allows for human textual display.

1214 */wst:RequestSecurityTokenResponse/wst:Status/wst:Code*

1215 This required URI value provides a machine-readable status code. The following URIs are  
1216 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 request 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 request did not successfully validate the input

1217 */wst:RequestSecurityTokenResponse/wst:Status/wst:Reason*

1218 This optional string provides human-readable text relating to the status code.

1219

1220 The following example illustrates a validation request and response. In this example no  
1221 token is requested, just a status.

```

1222     <wst:RequestSecurityToken>
1223         <wst:TokenType>
1224             http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Status
1225         </wst:TokenType>
1226         <wst:RequestType>
1227             http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate
1228         </wst:RequestType>
1229     </wst:RequestSecurityToken>
1230
1231     <wst:RequestSecurityTokenResponse>
1232         <wst:TokenType>
1233             http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Status
1234         </wst:TokenType>
1235         <wst:Status>
1236             <wst:Code>
1237                 http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
1238             </wst:Code>
1239         </wst:Status>
1240
1241     </wst:RequestSecurityTokenResponse>

```

1242

1243 The following example illustrates a validation request and response. In this example a  
1244 custom token is requested indicating authorized rights in addition to the status.

```
1245     <wst:RequestSecurityToken>
1246         <wst:TokenType>
1247             http://example.org/mySpecialToken
1248         </wst:TokenType>
1249         <wst:RequestType>
1250             http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate
1251         </wst:RequestType>
1252     </wst:RequestSecurityToken>
1253
1254     <wst:RequestSecurityTokenResponse>
1255         <wst:TokenType>
1256             http://example.org/mySpecialToken
1257         </wst:TokenType>
1258         <wst:Status>
1259             <wst:Code>
1260                 http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
1261             </wst:Code>
1262         </wst:Status>
1263         <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1264         ...
1265     </wst:RequestSecurityTokenResponse>
```

---

## 1266 8 Negotiation and Challenge Extensions

---

1267 The general security token service framework defined above allows for a simple request and  
1268 response for security tokens (possibly asynchronous). However, there are many scenarios  
1269 where a set of exchanges between the parties is required prior to returning (e.g., issuing) a  
1270 security token. This section describes the extensions to the base WS-Trust mechanisms to  
1271 enable exchanges for negotiation and challenges.

---

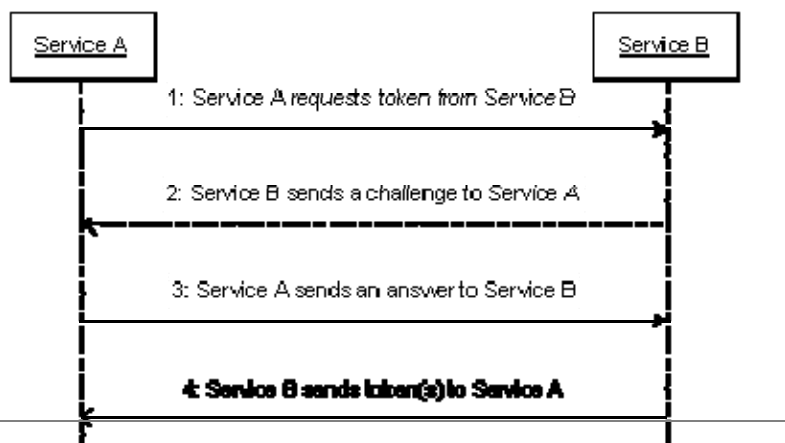
1272  
1273 There are potentially different forms of exchanges, but one specific form, called  
1274 "challenges", provides mechanisms in addition to those described in [WS-Security] for  
1275 authentication. This section describes how general exchanges are issued and responded to  
1276 within this framework. Other types of exchanges include, but are not limited to, negotiation,  
1277 tunneling of hardware-based processing, and tunneling of legacy protocols.

---

1278  
1279 The process is straightforward (illustrated here using a challenge):

---

1280



1281

1282 1. A requestor sends, for example, a `<wst:RequestSecurityToken>` message with a  
1283 timestamp.

---

1284 2. The recipient does not trust the timestamp and issues a  
1285 `<wst:RequestSecurityTokenResponse>` message with an embedded challenge.

---

1286 3. The requestor sends a `<wst:RequestSecurityTokenReponse>` message with an  
1287 answer to the challenge.

---

1288 4. The recipient issues a `<wst:RequestSecurityTokenResponse>` message with the  
1289 issued security token and optional proof-of-possession token.

---

1290

1291 It should be noted that the requestor might challenge the recipient in either step 1 or step  
1292 3. In which case, step 2 or step 4 contains an answer to the initiator's challenge. Similarly,  
1293 it is possible that steps 2 and 3 could iterate multiple times before the process completes  
1294 (step 4).

---

1295

1296 The two services can use [WS-SecurityPolicy] to state their requirements and preferences  
1297 for security tokens and encryption and signing algorithms (general policy intersection). This  
1298 section defines mechanisms for legacy and more sophisticated types of negotiations.

## 1299 **8.1 Negotiation and Challenge Framework**

1300 The general mechanisms defined for requesting and returning security tokens are  
1301 extensible. This section describes the general model for extending these to support  
1302 negotiations and challenges.

1303

1304 The exchange model is as follows:

- 1305 1. A request is initiated with a `<wst:RequestSecurityToken>` that identifies the details  
1306 of the request (and may contain initial negotiation/challenge information)
- 1307 2. A response is returned with a `<wst:RequestSecurityTokenResponse>` that contains  
1308 additional negotiation/challenge information. Optionally, this may return token  
1309 information (if the exchange is two legs long).
- 1310 3. If the exchange is not complete, the requestor uses a  
1311 `<wst:RequestSecurityTokenResponse>` that contains additional  
1312 negotiation/challenge information.
- 1313 4. The process repeats at step 2 until the negotiation/challenge is complete (a token is  
1314 returned or a Fault occurs).

1315

1316 The negotiation/challenge information is passed in binding/profile-specific elements that are  
1317 placed inside of the `<wst:RequestSecurityToken>` and  
1318 `<wst:RequestSecurityTokenResponse>` elements.

1319

1320 It is RECOMMENDED that at least the `<wsu:Timestamp>` element be included in messages  
1321 (as per [WS-Security]) as a way to ensure freshness of the messages in the exchange.  
1322 Other types of challenges MAY also be included. For example, a `<wsp:Policy>` element  
1323 may be used to negotiate desired policy behaviors of both parties. Multiple challenges and  
1324 responses MAY be included.

## 1325 **8.2 Signature Challenges**

1326 Exchange requests are issued by including an element that describes the exchange (e.g.  
1327 challenge) and responses contain an element describing the response. For example,  
1328 signature challenges are processed using the `<SignChallenge>` element. The response is  
1329 returned in a `<SignChallengeResponse>` element. Both the challenge and the response  
1330 elements are specified within the `<wst:RequestSecurityTokenResponse>` element. Some  
1331 forms of negotiation MAY specify challenges along with responses to challenges from the  
1332 other party. It should be noted that the requestor MAY provide exchange information (e.g.  
1333 a challenge) to the recipient in the initial request. Consequently, these elements are also  
1334 allowed within a `<wst:RequestSecurityToken>` element.

1335

1336 The syntax of these elements is as follows:

```
1337 <wst:SignChallenge>  
1338   <wst:Challenge ...>... </wst:Challenge>  
1339 </wst:SignChallenge>
```

1340

```
1341 <wst:SignChallengeResponse>
1342 <wst:Challenge ...>...</wst:Challenge>
1343 </wst:SignChallengeResponse>
```

1344

1345 The following describes the attributes and tags listed in the schema above:

1346 *.../wst:SignChallenge*

1347 This optional element describes a challenge that requires the other party to sign a specified set of  
1348 information.

1349 *.../wst:SignChallenge/ws:LChallenge*

1350 This required string element describes the value to be signed. In order to prevent certain types of  
1351 attacks (such as man-in-the-middle), it is strongly RECOMMENDED that the challenge be bound  
1352 to the negotiation. For example, the challenge SHOULD track (such as using a digest of) any  
1353 relevant data exchanged such as policies, tokens, replay protection, etc. As well, if the challenge  
1354 is happening over a secured channel, a reference to the channel SHOULD also be included.  
1355 Furthermore, the recipient of a challenge SHOULD verify that the data tracked (digested)  
1356 matches their view of the data exchanged. The exact algorithm MAY be defined in profiles or  
1357 agreed to by the parties.

1358 *.../SignChallenge/{any}*

1359 This is an extensibility mechanism to allow additional negotiation types to be used.

1360 *.../wst:SignChallenge/@{any}*

1361 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added  
1362 to the element.

1363 *.../wst:SignChallengeResponse*

1364 This optional element describes a response to a challenge that requires the signing of a specified  
1365 set of information.

1366 *.../wst:SignChallengeResponse/wst:Challenge*

1367 If a challenge was issued, the response MUST contain the challenge element exactly as  
1368 received. As well, while the RSTR response SHOULD always be signed, if a challenge was  
1369 issued, the RSTR MUST be signed (and the signature coupled with the message to prevent  
1370 replay).

1371 *.../wst:SignChallengeResponse/{any}*

1372 This is an extensibility mechanism to allow additional negotiation types to be used.

1373 *.../wst:SignChallengeResponse/@{any}*

1374 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added  
1375 to the element.

### 1376 8.3 Binary Exchanges and Negotiations

1377 Exchange requests may also utilize existing binary formats passed within the WS-Trust  
1378 framework. A generic mechanism is provided for this that includes a URI attribute to  
1379 indicate the type of binary exchange.

1380

1381 The syntax of this element is as follows:

```
1382 <wst:BinaryExchange ValueType="..." EncodingType="...">
1383 ...
1384 </wst:BinaryExchange>
```

1385 The following describes the attributes and tags listed in the schema above (note that the  
1386 ellipses below indicate that this element may be placed in different containers. For this  
1387 specification, these are limited to <wst:RequestSecurityToken> and  
1388 <wst:RequestSecurityTokenResponse>):

1389 *.../wst:BinaryExchange*

1390 This optional element is used for a security negotiation that involves exchanging binary blobs as  
1391 part of an existing negotiation protocol. The contents of this element are blob-type-specific and  
1392 are encoded using base64 (unless otherwise specified).

1393 *.../wst:BinaryExchange/@ValueType*

1394 This required attribute specifies a URI to identify the type of negotiation (and the value space of  
1395 the blob – the element's contents).

1396 *.../wst:BinaryExchange/@EncodingType*

1397 This required attribute specifies a URI to identify the encoding format (if different from base64) of  
1398 the negotiation blob. Refer to [WS-Security] for sample encoding format URIs.

1399 *.../wst:BinaryExchange/@{any}*

1400 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added  
1401 to the element.

1402 Some binary exchanges result in a shared state/context between the involved parties. It is  
1403 RECOMMENDED that at the conclusion of the exchange, a new token and proof-of-  
1404 possession token be returned. A common approach is to use the negotiated key as a  
1405 "secure channel" mechanism to secure the new token and proof-of-possession token.  
1406 For example, an exchange might establish a shared secret Sx that can then be used to sign  
1407 the final response and encrypt the proof-of-possession token.

## 1408 **8.4 Key Exchange Tokens**

1409 In some cases it may be necessary to provide a key exchange token so that the other party  
1410 (either requestor or issuer) can provide entropy or key material as part of the exchange.  
1411 Challenges may not always provide a usable key as the signature may use a signing-only  
1412 certificate.

1413

1414 The section describes two optional elements that can be included in RST and RSTR elements  
1415 to indicate that a Key Exchange Token (KET) is desired, or to provide a KET.

1416 The syntax of these elements is as follows (Note that the ellipses below indicate that this  
1417 element may be placed in different containers. For this specification, these are limited to  
1418 <wst:RequestSecurityToken> and <wst:RequestSecurityTokenResponse>):

```
1419 <wst:RequestKET/>  
1420 <wst:KeyExchangeToken>...</wst:KeyExchangeToken>
```

1421

1422 The following describes the attributes and tags listed in the schema above:

1423 *.../wst:RequestKET*

1424 This optional element is used to indicate that the receiving party (either the original requestor or  
1425 issuer) should provide a KET to the other party on the next leg of the exchange.

1426 *.../wst:KeyExchangeToken*

1427 This optional element is used to provide a key exchange token. The contents of this element  
1428 either contain the security token to be used for key exchange or a reference to it.

---

## 1429 8.5 Custom Exchanges

---

1430 Using the extensibility model described in this specification, any custom XML-based  
1431 exchange can be defined in a separate binding/profile document. In such cases elements  
1432 are defined which are carried in the RST and RSTR elements.

---

1433  
1434 It should be noted that it is NOT REQUIRED that exchange elements be symmetric. That is,  
1435 a specific exchange mechanism MAY use multiple elements at different times, depending on  
1436 the state of the exchange.

---

## 1437 8.6 Signature Challenge Example

---

1438 Here is an example exchange involving a signature challenge. In this example, a service  
1439 requests a custom token using a X.509 certificate for authentication. The issuer uses the  
1440 exchange mechanism to challenge the requestor to sign a random value (to ensure message  
1441 freshness). The requestor provides a signature of the requested data and, once validated,  
1442 the issuer then issues the requested token.

---

1443  
1444 The first message illustrates the initial request that is signed with the requestor's X.509  
1445 certificate:

```
1446 <S11:Envelope xmlns:S11="..." xmlns:wsse="..."
1447     xmlns:wsu="..." xmlns:wst="...">
1448   <S11:Header>
1449     ...
1450     <wsse:Security>
1451       <wsse:BinarySecurityToken
1452         wsu:Id="reqToken"
1453         ValueType="X509v3">
1454         MTEZzCCA9CgAwIBAgTQEmtJZc0...
1455       </wsse:BinarySecurityToken>
1456       <ds:Signature xmlns:ds="...">
1457         ...
1458         <ds:KeyInfo>
1459           <wsse:SecurityTokenReference>
1460             <wsse:Reference URI="#reqToken"/>
1461           </wsse:SecurityTokenReference>
1462         </ds:KeyInfo>
1463       </ds:Signature>
1464     </wsse:Security>
1465     ...
1466   </S11:Header>
1467   <S11:Body>
1468     <wst:RequestSecurityToken>
1469       <wst:TokenType>
1470         http://example.org/mySpecialToken
1471       </wst:TokenType>
1472       <wst:RequestType>
1473         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1474       </wst:RequestType>
1475     </wst:RequestSecurityToken>
1476   </S11:Body>
1477 </S11:Envelope>
```

---

1478  
1479 The issuer (recipient) service doesn't trust the sender's timestamp (or one wasn't specified)  
1480 and issues a challenge using the exchange framework defined in this specification. This

1481 message is signed using the issuer's X.509 certificate and contains a random challenge that  
1482 the requestor must sign:

```
1483 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."  
1484     xmlns:wst="...">  
1485   <S11:Header>  
1486     ...  
1487     <wsse:Security>  
1488       <wsse:BinarySecurityToken  
1489         wsu:Id="issuerToken"  
1490         ValueType="...X509v3">  
1491         DEJHuedsujfnrnv45JZc0  
1492       </wsse:BinarySecurityToken>  
1493       <ds:Signature xmlns:ds="...">  
1494         ...  
1495       </ds:Signature>  
1496     </wsse:Security>  
1497   </S11:Header>  
1498   <S11:Body>  
1500     <wst:RequestSecurityTokenResponse>  
1501       <wst:SignChallenge>  
1502         <wst:Challenge>Huehf... </wst:Challenge>  
1503       </wst:SignChallenge>  
1504     </wst:RequestSecurityTokenResponse>  
1505   </S11:Body>  
1506 </S11:Envelope>
```

1507  
1508 The requestor receives the issuer's challenge and issues a response that is signed using the  
1509 requestor's X.509 certificate and contains the challenge. The signature only covers the non-  
1510 mutable elements of the message to prevent certain types of security attacks:

```
1511 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."  
1512     xmlns:wst="...">  
1513   <S11:Header>  
1514     ...  
1515     <wsse:Security>  
1516       <wsse:BinarySecurityToken  
1517         wsu:Id="reqToken"  
1518         ValueType="...X509v3">  
1519         MIEZzCCA9CgAwIBAgIQEmtJZc0...  
1520       </wsse:BinarySecurityToken>  
1521       <ds:Signature xmlns:ds="...">  
1522         ...  
1523       </ds:Signature>  
1524     </wsse:Security>  
1525   </S11:Header>  
1526   <S11:Body>  
1528     <wst:RequestSecurityTokenResponse>  
1529       <wst:SignChallengeResponse>  
1530         <wst:Challenge>Huehf... </wst:Challenge>  
1531       </wst:SignChallengeResponse>  
1532     </wst:RequestSecurityTokenResponse>  
1533   </S11:Body>  
1534 </S11:Envelope>
```

1535  
1536 The issuer validates the requestor's signature responding to the challenge and issues the  
1537 requested token(s) and the associated proof-of-possession token. The proof-of-possession  
1538 token is encrypted for the requestor using the requestor's public key.

```

1539 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
1540     xmlns:wst="..." xmlns:xenc="...">
1541   <S11:Header>
1542     ...
1543     <wsse:Security>
1544       <wsse:BinarySecurityToken
1545         wsu:Id="issuerToken"
1546         ValueType="...X509v3">
1547         DFJHuedsujfnrnv45JZc0...
1548       </wsse:BinarySecurityToken>
1549       <ds:Signature xmlns:ds="...">
1550         ...
1551       </ds:Signature>
1552     </wsse:Security>
1553     ...
1554   </S11:Header>
1555   <S11:Body>
1556     <wst:RequestSecurityTokenResponse>
1557       <wst:RequestedSecurityToken>
1558         <xyz:CustomToken xmlns:xyz="...">
1559           ...
1560         </xyz:CustomToken>
1561       </wst:RequestedSecurityToken>
1562       <wst:RequestedProofToken>
1563         <xenc:EncryptedKey Id="newProof">
1564           ...
1565         </xenc:EncryptedKey>
1566       </wst:RequestedProofToken>
1567     </wst:RequestSecurityTokenResponse>
1568   </S11:Body>
1569 </S11:Envelope>

```

## 1570 8.7 Custom Exchange Example

1571 Here is another example illustrating a token request using a custom XML exchange. For  
1572 brevity, only the RST and RSTR elements are illustrated. Note that the framework allows  
1573 for an arbitrary number of exchanges, although this example illustrates the use of four legs.  
1574 The request uses a custom exchange element and the requestor signs only the non-mutable  
1575 element of the message:

```

1576 <wst:RequestSecurityToken>
1577   <wst:TokenType>
1578     http://example.org/mySpecialToken
1579   </wst:TokenType>
1580   <wst:RequestType>
1581     http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1582   </wst:RequestType>
1583   <xyz:CustomExchange xmlns:xyz="...">
1584     ...
1585   </xyz:CustomExchange>
1586 </wst:RequestSecurityToken>

```

1587  
1588 The issuer service (recipient) responds with another leg of the custom exchange and signs  
1589 the response (non-mutable aspects) with its token:

```

1590 ...
1591 <wst:RequestSecurityTokenResponse>
1592   <xyz:CustomExchange xmlns:xyz="...">
1593     ...
1594   </xyz:CustomExchange>
1595 </wst:RequestSecurityTokenResponse>

```

1596

...

1597

1598 The requestor receives the issuer's exchange and issues a response that is signed using the  
1599 requestor's token and continues the custom exchange. The signature covers all non-  
1600 mutable aspects of the message to prevent certain types of security attacks:

1601

```
1602 <wst:RequestSecurityTokenResponse>
1603   <xyz:CustomExchange xmlns:xyz="...">
1604     ...
1605   </xyz:CustomExchange>
1606 </wst:RequestSecurityTokenResponse>
1607 ...
```

1608

1609 The issuer processes the exchange and determines that the exchange is complete and that  
1610 a token should be issued. Consequently it issues the requested token(s) and the associated  
1611 proof-of-possession token. The proof-of-possession token is encrypted for the requestor  
1612 using the requestor's public key.

1613

```
1614 <wst:RequestSecurityTokenResponse>
1615   <wst:RequestedSecurityToken>
1616     <xyz:CustomToken xmlns:xyz="...">
1617       ...
1618     </xyz:CustomToken>
1619   </wst:RequestedSecurityToken>
1620   <wst:RequestedProofToken>
1621     <xenc:EncryptedKey Id="newProof">
1622       ...
1623     </xenc:EncryptedKey>
1624   </wst:RequestedProofToken>
1625 <wst:RequestedProofToken>
1626   <xenc:EncryptedKey>...</xenc:EncryptedKey>
1627 </wst:RequestedProofToken>
1628 </wst:RequestSecurityTokenResponse>
1629 ...
```

1630 It should be noted that other example exchanges include the issuer returning a final custom  
1631 exchange element, and another example where a token isn't returned.

## 1632 8.8 Protecting Exchanges

1633 There are some attacks, such as forms of man-in-the-middle, that can be applied to token  
1634 requests involving exchanges. It is RECOMMENDED that the exchange sequence be  
1635 protected. This may be built into the exchange messages, but if metadata is provided in the  
1636 RST or RSTR elements, then it is subject to attack.

1637

1638 Consequently, it is RECOMMENDED that keys derived from exchanges be linked  
1639 cryptographically to the exchange. For example, a hash can be computed by computing the  
1640 SHA1 of the exclusive canonicalization of all RST and RSTR elements in messages  
1641 exchanged. This value can then be combined with the exchanged secret(s) to create a new  
1642 master secret that is bound to the data both parties sent/received.

1643

1644 To this end, the following computed key algorithm is defined to be optionally used in these  
1645 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>	<p>The key is computed using P_SHA1 as follows:</p> $H = \text{SHA1}(\text{ExclC14N}(\text{RST} \dots \text{RSTRs}))$ $X = \text{encrypting } H \text{ using negotiated key and mechanism}$ $\text{Key} = \text{P\_SHA1}(X, H + \text{"CK-HASH"})$ <p>The octets for the "CK-HASH" string are the UTF-8 octets.</p>

## 1646 8.9 Authenticating Exchanges

1647 After an exchange both parties have a shared knowledge of a key (or keys) that can then be  
1648 used to secure messages. However, in some cases it may be desired to have the issuer  
1649 prove to the requestor that it knows the key (and that the returned metadata is valid) prior  
1650 to the requestor using the data. However, until the exchange is actually completed it may  
1651 (and is often) inappropriate to use the computed keys. As well, using a token that hasn't  
1652 been returned to secure a message may complicate processing since it crosses the  
1653 boundary of the exchange and the underlying message security. This means that it may not  
1654 be appropriate to sign the final leg of the exchange using the key derived from the  
1655 exchange.

1656

1657 For this reason an authenticator is defined that provides a way for the issuer to verify the  
1658 hash as part of the token issuance. Specifically, when an authenticator is returned, the  
1659 `<wst:RequestSecurityTokenResponseCollection>` element is returned. This contains one  
1660 RSTR with the token being returned as a result of the exchange and a second RSTR that  
1661 contains the authenticator (this order SHOULD be used). When an authenticator is used,  
1662 RSTRs MUST use the *@Context* element so that the authenticator can be correlated to the  
1663 token issuance. The authenticator is separated from the RSTR because otherwise  
1664 computation of the RST/RSTR hash becomes more complex. The authenticator is  
1665 represented using the `<wst:Authenticator>` element as illustrated below:

```

1666 ...
1667 <wst:RequestSecurityTokenResponseCollection>
1668   <wst:RequestSecurityTokenResponse Context="...">
1669     ...
1670   </wst:RequestSecurityTokenResponse>
1671   <wst:RequestSecurityTokenResponse Context="...">
1672     <wst:Authenticator>
1673       <wst:CombinedHash>...</wst:CombinedHash>
1674     ...
1675     </wst:Authenticator>
1676   </wst:RequestSecurityTokenResponse>
1677 </wst:RequestSecurityTokenResponseCollection>
1678 ...

```

1679

1680 The following describes the attributes and elements listed in the schema overview above  
1681 (the ... notation below represents the path RSTRC/RSTR and is used for brevity):

1682 `.../wst:Authenticator`

1683 This optional element provides verification (authentication) of a computed hash.

---

1684	<i>.../wst:Authenticator/wst:CombinedHash</i>
1685	This optional element proves the hash and knowledge of the computed key. This is done by
1686	providing the base64 encoding of the first 256 bits of the P_SHA1 digest of the computed key and
1687	the concatenation of the hash determined for the computed key and the string "AUTH-HASH".
1688	Specifically, P_SHA1( <i>computed-key</i> , H + "AUTH-HASH") <sub>0-255</sub> . The octets for the "AUTH-HASH"
1689	string are the UTF-8 octets.
1690	
1691	This <wst:CombinedHash> element is optional (and an open content model is used) to allow
1692	for different authenticators in the future.

---

---

## 1693 9 Key and Token Parameter Extensions

1694 This section outlines additional parameters that can be specified in token requests and  
1695 responses. Typically they are used with issuance requests, but since all types of requests  
1696 may issue security tokens they could apply to binding.

---

### 1697 9.1 On-Behalf-Of Parameters

1698 In some scenarios the requestor is obtaining a token on behalf of another party. These  
1699 parameters specify the issuer and original requestor of the token being used as the basis of  
1700 the request. The syntax is as follows (note that the base elements described above are  
1701 included here italicized for completeness):

```
1702 <wst:RequestSecurityToken>  
1703   <wst:TokenType>...</wst:TokenType>  
1704   <wst:RequestType>...</wst:RequestType>  
1705   ...  
1706   <wst:OnBehalfOf>...</wst:OnBehalfOf>  
1707   <wst:Issuer>...</wst:Issuer>  
1708 </wst:RequestSecurityToken>
```

1709

1710 The following describes the attributes and elements listed in the schema overview above:

1711 */wst:RequestSecurityToken/wst:OnBehalfOf*

1712 This optional element indicates that the requestor is making the request on behalf of another.  
1713 The identity on whose behalf the request is being made is specified by placing a security token,  
1714 <wsse:SecurityTokenReference> element, or <wsa:EndpointReference> element  
1715 within the <wst:OnBehalfOf> element.

1716 */wst:RequestSecurityToken/wst:Issuer*

1717 This optional element specifies the issuer of the security token that is presented in the message.  
1718 This element's type is an endpoint reference as defined in [WS-Addressing].

1719

1720 In the following example a proxy is requesting a security token on behalf of another  
1721 requestor or end-user.

```
1722 <wst:RequestSecurityToken>  
1723   <wst:TokenType>...</wst:TokenType>  
1724   <wst:RequestType>...</wst:RequestType>  
1725   ...  
1726   <wst:OnBehalfOf>endpoint-reference</wst:OnBehalfOf>  
1727 </wst:RequestSecurityToken>
```

---

### 1728 9.2 Key and Encryption Requirements

1729 This section defines extensions to the <wst:RequestSecurityToken> element for requesting  
1730 specific types of keys or algorithms or key and algorithms as specified by a given policy in  
1731 the return token(s). In some cases the service may support a variety of key types, sizes,  
1732 and algorithms. These parameters allow a requestor to indicate its desired values. It  
1733 should be noted that the issuer's policy indicates if input values must be adhered to and  
1734 faults generated for invalid inputs, or if the issuer will provide alternative values in the  
1735 response.

1736

1737 Although illustrated using the `<wst:RequestSecurityToken>` element, these options can  
1738 also be returned in a `<wst:RequestSecurityTokenResponse>` element.

1739 The syntax for these optional elements is as follows (note that the base elements described  
1740 above are included here italicized for completeness):

```

1741     <wst:RequestSecurityToken>
1742         <wst:TokenType>...</wst:TokenType>
1743         <wst:RequestType>...</wst:RequestType>
1744         ...
1745         <wst:AuthenticationType>...</wst:AuthenticationType>
1746         <wst:KeyType>...</wst:KeyType>
1747         <wst:KeySize>...</wst:KeySize>
1748         <wst:SignatureAlgorithm>...</wst:SignatureAlgorithm>
1749         <wst:EncryptionAlgorithm>...</wst:EncryptionAlgorithm>
1750         <wst:CanonicalizationAlgorithm>...</wst:CanonicalizationAlgorithm>
1751         <wst:ComputedKeyAlgorithm>...</wst:ComputedKeyAlgorithm>
1752         <wst:Encryption>...</wst:Encryption>
1753         <wst:ProofEncryption>...</wst:ProofEncryption>
1754         <wst:UseKey Sig=...>...</wst:UseKey>
1755         <wst:SignWith>...</wst:SignWith>
1756         <wst:EncryptWith>...</wst:EncryptWith>
1757     </wst:RequestSecurityToken>

```

1758

1759 The following describes the attributes and elements listed in the schema overview above:

1760 */wst:RequestSecurityToken/wst:AuthenticationType*

1761 This optional URI element indicates the type of authentication desired, specified as a URI. This  
1762 specification does not predefine classifications; these are specific to token services as is the  
1763 relative strength evaluations. The relative assessment of strength is up to the recipient to  
1764 determine. That is, requestors should be familiar with the recipient policies. For example, this  
1765 might be used to indicate which of the four U.S. government authentication levels is required.

1766 */wst:RequestSecurityToken/wst:KeyType*

1767 This optional URI element indicates the type of key desired in the security token. The predefined  
1768 values are identified in the table below. Note that some security token formats have fixed key  
1769 types. It should be noted that new algorithms can be inserted by defining URIs in other  
1770 specifications and profiles.

URI	Meaning
http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey	A public key token is requested
http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey	A symmetric key token is requested (default)

1771 */wst:RequestSecurityToken/wst:KeySize*

1772 This optional integer element indicates the size of the key required specified in number of bits.  
1773 This is a request, and, as such, the requested security token is not obligated to use the requested  
1774 key size. That said, the recipient SHOULD try to use a key at least as strong as the specified  
1775 value if possible. The information is provided as an indication of the desired strength of the  
1776 security.

1777 */wst:RequestSecurityToken/wst:SignatureAlgorithm*

1778 This optional URI element indicates the desired signature algorithm used within the returned  
1779 token. This is specified as a URI indicating the algorithm (see [XML Signature] for typical signing  
1780 algorithms).

1781	<i>/wst:RequestSecurityToken/wst:EncryptionAlgorithm</i>
1782	This optional URI element indicates the desired encryption algorithm used within the returned token. This is specified as a URI indicating the algorithm (see <a href="#">XML-Encrypt</a> for typical encryption algorithms).
1783	
1784	
1785	<i>/wst:RequestSecurityToken/wst:CanonicalizationAlgorithm</i>
1786	This optional URI element indicates the desired canonicalization method used within the returned token. This is specified as a URI indicating the method (see <a href="#">XML Signature</a> for typical canonicalization methods).
1787	
1788	
1789	<i>/wst:RequestSecurityToken/wst:ComputedKeyAlgorithm</i>
1790	This optional URI element indicates the desired algorithm to use when computed keys are used for issued tokens.
1791	
1792	<i>/wst:RequestSecurityToken/wst:Encryption</i>
1793	This optional element indicates that the requestor desires any returned secrets in issued security tokens to be encrypted for the specified token. That is, so that the owner of the specified token can decrypt the secret. Normally the security token is the contents of this element but a security token reference MAY be used instead. If this element isn't specified, the token used as the basis of the request (or specialized knowledge) is used to determine how to encrypt the key.
1794	
1795	
1796	
1797	
1798	<i>/wst:RequestSecurityToken/wst:ProofEncryption</i>
1799	This optional element indicates that the requestor desires any returned secrets in proof-of-possession tokens to be encrypted for the specified token. That is, so that the owner of the specified token can decrypt the secret. Normally the security token is the contents of this element but a security token reference MAY be used instead. If this element isn't specified, the token used as the basis of the request (or specialized knowledge) is used to determine how to encrypt the key.
1800	
1801	
1802	
1803	
1804	
1805	<i>/wst:RequestSecurityToken/wst:UseKey</i>
1806	If the requestor wishes to use an existing key rather than create a new one, then this optional element can be used to reference the security token containing the desired key. This element either contains a security token or a <code>&lt;wsse:SecurityTokenReference&gt;</code> element that references the security token containing the key that should be used in the returned token. If <code>&lt;wst:KeyType&gt;</code> is not defined and a key type is not implicitly known to the service, it MAY be determined from the token (if possible). Otherwise this parameter is meaningless and is ignored. Requestors SHOULD demonstrate authorized use of the public key provided.
1807	
1808	
1809	
1810	
1811	
1812	
1813	<i>/wst:RequestSecurityToken/wst:UseKey/@Sig</i>
1814	In order to <i>authenticate</i> the key referenced, a signature MAY be used to prove the referenced token/key. If specified, this optional attribute indicates the ID of the corresponding signature (by URI reference). When this attribute is present, a key need not be specified inside the element since the referenced signature will indicate the corresponding token (and key).
1815	
1816	
1817	
1818	<i>/wst:RequestSecurityToken/wst:SignWith</i>
1819	This optional URI element indicates the desired signature algorithm to be used with the issued security token (typically from the policy of the target site for which the token is being requested. While any of these optional elements MAY be included in RSTRs, this one is a likely candidate if there is some doubt (e.g., an X.509 cert that can only use DSS).
1820	
1821	
1822	
1823	<i>/wst:RequestSecurityToken/wst:EncryptWith</i>
1824	This optional URI element indicates the desired encryption algorithm to be used with the issued security token (typically from the policy of the target site for which the token is being requested.) While any of these optional elements MAY be included in RSTRs, this one is a likely candidate if there is some doubt.
1825	
1826	
1827	

1828 The example below illustrates a request that utilizes several of these parameters. A request  
1829 is made for a custom token using a username and password as the basis of the request.  
1830 For security, this token is encrypted (see "encUsername") for the recipient using the  
1831 recipient's public key and referenced in the encryption manifest. The message is protected  
1832 by a signature using a public key from the sender and authorized by the username and  
1833 password.

1834  
1835 The requestor would like the custom token to contain a 1024-bit public key whose value can  
1836 be found in the key provided with the "proofSignature" signature (the key identified by  
1837 "requestProofToken"). The token should be signed using RSA-SHA1 and encrypted for the  
1838 token identified by "requestEncryptionToken". The proof should be encrypted using the  
1839 token identified by "requestProofToken".

```
1840 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
1841     xmlns:wst="..." xmlns:ds="..." xmlns:xenc="...">
1842   <S11:Header>
1843     ...
1844     <wsse:Security>
1845       <xenc:ReferenceList>...</xenc:ReferenceList>
1846       <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>
1847       <wsse:BinarySecurityToken wsu:Id="requestEncryptionToken"
1848         ValueType="...SomeTokenType" xmlns:x="...">
1849         MTEEZzCCA9CgAwIBAgIQEmtJZc0...
1850       </wsse:BinarySecurityToken>
1851       <wsse:BinarySecurityToken wsu:Id="requestProofToken"
1852         ValueType="...SomeTokenType" xmlns:x="...">
1853         MTEEZzCCA9CgAwIBAgIQEmtJZc0...
1854       </wsse:BinarySecurityToken>
1855       <ds:Signature Id="proofSignature">
1856         ... signature proving requested key ...
1857         ... key info points to the "requestedProofToken" token ...
1858       </ds:Signature>
1859     </wsse:Security>
1860     ...
1861   </S11:Header>
1862   <S11:Body wsu:Id="req">
1863     <wst:RequestSecurityToken>
1864       <wst:TokenType>
1865         http://example.org/mySpecialToken
1866       </wst:TokenType>
1867       <wst:RequestType>
1868         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1869       </wst:RequestType>
1870       <wst:KeyType>
1871         http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey
1872       </wst:KeyType>
1873       <wst:KeySize>1024</wst:KeySize>
1874       <wst:SignatureAlgorithm>
1875         http://www.w3.org/2000/09/xmldsig#rsa-sha1
1876       </wst:SignatureAlgorithm>
1877       <wst:Encryption>
1878         <Reference URI="#requestEncryptionToken">
1879       </wst:Encryption>
1880       <wst:ProofEncryption>
1881         <wsse:Reference URI="#requestProofToken"/>
1882       </wst:ProofEncryption>
1883       <wst:UseKey Sig="#proofSignature"/>
1884     </wst:RequestSecurityToken>
1885   </S11:Body>
1886 </S11:Envelope>
```

---

## 1887 9.3 Delegation and Forwarding Requirements

---

1888 This section defines extensions to the `<wst:RequestSecurityToken>` element for indicating  
1889 delegation and forwarding requirements on the requested security token(s).

1890 The syntax for these extension elements is as follows (note that the base elements  
1891 described above are included here italicized for completeness):

```
1892 <wst:RequestSecurityToken>  
1893   <wst:TokenType>...</wst:TokenType>  
1894   <wst:RequestType>...</wst:RequestType>  
1895  
1896   <wst:DelegateTo>...</wst:DelegateTo>  
1897   <wst:Forwardable>...</wst:Forwardable>  
1898   <wst:Delegatable>...</wst:Delegatable>  
1899 </wst:RequestSecurityToken>
```

1900 */wst:RequestSecurityToken/wst:DelegateTo*

1901 This optional element indicates that the requested or issued token be delegated to another  
1902 identity. The identity receiving the delegation is specified by placing a security token or  
1903 `<wsse:SecurityTokenReference>` element within the `<wst:DelegateTo>` element.

1904 */wst:RequestSecurityToken/wst:Forwardable*

1905 This optional element, of type `xs:boolean`, specifies whether the requested security token should  
1906 be marked as "Forwardable". In general, this flag is used when a token is normally bound to the  
1907 requestor's machine or service. Using this flag, the returned token MAY be used from any source  
1908 machine so long as the key is correctly proven. The default value of this flag is true.

1909 */wst:RequestSecurityToken/wst:Delegatable*

1910 This optional element, of type `xs:boolean`, specifies whether the requested security token should  
1911 be marked as "Delegatable". Using this flag, the returned token MAY be delegated to another  
1912 party. This parameter SHOULD be used in conjunction with `<wst:DelegateTo>`. The default  
1913 value of this flag is false.

1914

1915 The following example illustrates a request for a custom token that can be delegated to the  
1916 indicated recipient (specified in the binary security token) and used in the specified interval.

```
1917 <wst:RequestSecurityToken>  
1918   <wst:TokenType>  
1919     http://example.org/mySpecialToken  
1920   </wst:TokenType>  
1921   <wst:RequestType>  
1922     http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
1923   </wst:RequestType>  
1924   <wst:DelegateTo>  
1925     <wsse:BinarySecurityToken>...</wsse:BinarySecurityToken>  
1926   </wst:DelegateTo>  
1927   <wst:Delegatable>true</wst:Delegatable>  
1928 </wst:RequestSecurityToken>
```

---

## 1929 9.4 Policies

---

1930 This section defines extensions to the `<wst:RequestSecurityToken>` element for passing  
1931 policies.

1932

1933 The syntax for these extension elements is as follows (note that the base elements  
1934 described above are included here italicized for completeness):

```

1935 <wst:RequestSecurityToken>
1936 <wst:TokenType>...</wst:TokenType>
1937 <wst:RequestType>...</wst:RequestType>
1938 ...
1939 <wsp:Policy>...</wsp:Policy>
1940 <wsp:PolicyReference>...</wsp:PolicyReference>
1941 </wst:RequestSecurityToken>

```

1942

1943 The following describes the attributes and elements listed in the schema overview above:

1944 */wst:RequestSecurityToken/wsp:Policy*

1945 This optional element specifies a policy (as defined in [WS-Policy]) that indicates desired settings

1946 for the requested token. The policy specifies defaults that can be overridden by the elements

1947 defined in the previous sections.

1948 */wst:RequestSecurityToken/wsp:PolicyReference*

1949 This optional element specifies a reference to a policy (as defined in [WS-Policy]) that indicates

1950 desired settings for the requested token. The policy specifies defaults that can be overridden by

1951 the elements defined in the previous sections.

1952

1953 The following example illustrates a request for a custom token that provides a set of policy

1954 statements about the token or its usage requirements.

```

1955 <wst:RequestSecurityToken>
1956 <wst:TokenType>
1957 http://example.org/mySpecialToken
1958 </wst:TokenType>
1959 <wst:RequestType>
1960 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1961 </wst:RequestType>
1962 <wsp:Policy xmlns:wsp="...">
1963 ...
1964 </wsp:Policy>
1965 </wst:RequestSecurityToken>

```

## 1966 9.5 Authorized Token Participants

1967 This section defines extensions to the `<wst:RequestSecurityToken>` element for passing

1968 information about which parties are authorized to participate in the use of the token. This

1969 parameter is typically used when there are additional parties using the token or if the

1970 requestor needs to clarify the actual parties involved (for some profile-specific reason).

1971 It should be noted that additional participants will need to prove their identity to recipients

1972 in addition to proving their authorization to use the returned token. This typically takes the

1973 form of a second signature or use of transport security.

1974

1975 The syntax for these extension elements is as follows (note that the base elements

1976 described above are included here italicized for completeness):

```

1977 <wst:RequestSecurityToken>
1978 <wst:TokenType>...</wst:TokenType>
1979 <wst:RequestType>...</wst:RequestType>
1980 ...
1981 <wst:Participants>
1982 <wst:Primary>...</wst:Primary>
1983 <wst:Participant>...</wst:Participant>
1984 </wst:Participants>

```

1985	<code>&lt;/wst:RequestSecurityToken&gt;</code>
1986	
1987	The following describes elements and attributes used in a <code>&lt;wsc:SecurityContextToken&gt;</code>
1988	element.
1989	<code>/wst:RequestSecurityToken/wst:Participants/</code>
1990	This optional element specifies the participants sharing the security token. Arbitrary types may be
1991	used to specify participants, but a typical case is a security token or an endpoint reference (see
1992	[WS-Addressing]).
1993	<code>/wst:RequestSecurityToken/wst:Participants/wst:Primary</code>
1994	This optional element specifies the primary user of the token (if one exists).
1995	<code>/wst:RequestSecurityToken/wst:Participants/wst:Participant</code>
1996	This optional element specifies participant (or multiple participants by repeating the element) that
1997	play a (profile-dependent) role in the use of the token or who are allowed to use the token.
1998	<code>/wst:RequestSecurityToken/wst:Participants/{any}</code>
1999	This is an extensibility option to allow other types of participants and profile-specific elements to
2000	be specified.

---

## 2001 10 Key Exchange Token Binding

2002 Using the token request framework, this section defines a binding for requesting a key  
2003 exchange token (KET). That is, if a requestor desires a token that can be used to encrypt  
2004 key material for a recipient.

2005

2006 For this binding, the following actions are defined to enable specific processing context to be  
2007 conveyed to the recipient:

2008 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET`  
2009 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KET`

2010

2011 For this binding, the RequestType element contains the following URI:

2012 `http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET`

2013

2014 For this binding very few parameters are specified as input. Optionally the  
2015 `<wst:TokenType>` element can be specified in the request can indicate desired type  
2016 response token carrying the key for key exchange; however, this isn't commonly used.

2017 The applicability scope (e.g. `<wsp:AppliesTo>`) MAY be specified if the requestor desires a  
2018 key exchange token for a specific scope.

2019

2020 It is RECOMMENDED that the response carrying the key exchange token be secured (e.g.,  
2021 signed by the issuer or someone who can speak on behalf of the target for which the KET  
2022 applies).

2023

2024 Care should be taken when using this binding to prevent possible man-in-the-middle and  
2025 substitution attacks. For example, responses to this request SHOULD be secured using a  
2026 token that can speak for the desired endpoint.

2027

2028 The RSTR for this binding carries the `<RequestedSecurityToken>` element even if a token is  
2029 returned (note that the base elements described above are included here italicized for  
2030 completeness):

2031 `<wst:RequestSecurityToken`  
2032 `<wst:TokenType>...</wst:TokenType>`  
2033 `<wst:RequestType>...</wst:RequestType>`  
2034 `...`  
2035 `</wst:RequestSecurityToken>`  
2036  
2037 `<wst:RequestSecurityTokenResponse`  
2038 `<wst:TokenType>...</wst:TokenType>`  
2039 `<wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>`  
2040 `...`  
2041 `</wst:RequestSecurityTokenResponse>`

2042

2043 The following example illustrates requesting a key exchange token. In this example, the  
2044 KET is returned encrypted for the requestor since it had the credentials available to do that.

2045 Alternatively the request could be made using transport security (e.g. TLS) and the key  
2046 could be returned directly using `<wst:BinarySecret>`.

```
2047 <wst:RequestSecurityToken>
2048   <wst:RequestType>
2049     http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET
2050   </wst:RequestType>
2051 </wst:RequestSecurityToken>
2052
2053 <wst:RequestSecurityTokenResponse>
2054   <wst:RequestedSecurityToken>
2055     <xenc:EncryptedKey>...</xenc:EncryptedKey>
2056   </wst:RequestedSecurityToken>
2057 </wst:RequestSecurityTokenResponse>
```

2058

## 11 Error Handling

2059 There are many circumstances where an *error* can occur while processing security  
 2060 information. Errors use the SOAP Fault mechanism. Note that the reason text provided  
 2061 below is RECOMMENDED, but alternative text MAY be provided if more descriptive or  
 2062 preferred by the implementation. The tables below are defined in terms of SOAP 1.1. For  
 2063 SOAP 1.2, the Fault/Code/Value is env:Sender (as defined in SOAP 1.2) and the  
 2064 Fault/Code/Subcode/Value is the *faultcode* below and the Fault/Reason/Text is the  
 2065 *faultstring* below. It should be noted that profiles MAY provide second-level detail fields, but  
 2066 they should be careful not to introduce security vulnerabilities when doing so (e.g., by  
 2067 providing too detailed information).

<b>Error that occurred (faultstring)</b>	<b><i>Fault code (faultcode)</i></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

---

## 2068 12 Security Considerations

2069 As stated in the Goals section of this document, this specification is meant to provide  
2070 extensible framework and flexible syntax, with which one could implement various security  
2071 mechanisms. This framework and syntax by itself does not provide any guarantee of  
2072 security. When implementing and using this framework and syntax, one must make every  
2073 effort to ensure that the result is not vulnerable to any one of a wide range of attacks.

2074  
2075 It is not feasible to provide a comprehensive list of security considerations for such an  
2076 extensible set of mechanisms. A complete security analysis must be conducted on specific  
2077 solutions based on this specification. Below we illustrate some of the security concerns that  
2078 often come up with protocols of this type, but we stress that this *is not an exhaustive list of*  
2079 *concerns*.

2080  
2081 The following statements about signatures and signing apply to messages sent on  
2082 unsecured channels.

2083  
2084 It is critical that all the security-sensitive message elements must be included in the scope  
2085 of the message signature. As well, the signatures for conversation authentication must  
2086 include a timestamp, nonce, or sequence number depending on the degree of replay  
2087 prevention required as described in [WS-Security] and the UsernameToken Profile. Also,  
2088 conversation establishment should include the policy so that supported algorithms and  
2089 algorithm priorities can be validated.

2090  
2091 It is required that security token issuance messages be signed to prevent tampering. If a  
2092 public key is provided, the request should be signed by the corresponding private key to  
2093 prove ownership. As well, additional steps should be taken to eliminate replay attacks  
2094 (refer to [WS-Security] for additional information). Similarly, all token references should be  
2095 signed to prevent any tampering.

2096  
2097 Security token requests are susceptible to denial-of-service attacks. Care should be taken  
2098 to mitigate such attacks as is warranted by the service.

2099  
2100 For security, tokens containing a symmetric key or a password should only be sent to  
2101 parties who have a need to know that key or password.

2102  
2103 For privacy, tokens containing personal information (either in the claims, or indirectly by  
2104 identifying who is currently communicating with whom) should only be sent according to the  
2105 privacy policies governing these data at the respective organizations.

2106  
2107 For some forms of multi-message exchanges, the exchanges are susceptible to attacks  
2108 whereby signatures are altered. To address this, it is suggested that a signature  
2109 confirmation mechanism be used. In such cases, each leg should include the confirmation  
2110 of the previous leg. That is, leg 2 includes confirmation for leg 1, leg 3 for leg 2, leg 4 for

---

2111 leg 3, and so on. In doing so, each side can confirm the correctness of the message outside  
2112 of the message body.

---

2113

---

2114 There are many other security concerns that one may need to consider in security protocols.

2115 The list above should not be used as a "check list" instead of a comprehensive security  
2116 analysis.

---

2117

---

2118 It should be noted that use of unsolicited RSTRs implies that the recipient is prepared to

2119 accept such issuances. Recipients should ensure that such issuances are properly

2120 authorized and recognize their use could be used in denial-of-service attacks.

---

2121 In addition to the consideration identified here, readers should also review the security

2122 considerations in [WS-Security].

---

## 2123 A. Key Exchange

2124 Key exchange is an integral part of token acquisition. There are several mechanisms by  
2125 which keys are exchanged using [WS-Security] and WS-Trust. This section highlights and  
2126 summarizes these mechanisms. Other specifications and profiles may provide additional  
2127 details on key exchange.

2128

2129 Care must be taken when employing a key exchange to ensure that the mechanism does  
2130 not provide an attacker with a means of discovering information that could only be  
2131 discovered through use of secret information (such as a private key).

2132

2133 It is therefore important that a shared secret should only be considered as trustworthy as  
2134 its source. A shared secret communicated by means of the direct encryption scheme  
2135 described in section I.1 is acceptable if the encryption key is provided by a completely  
2136 trustworthy key distribution center (this is the case in the Kerberos model). Such a key  
2137 would not be acceptable for the purposes of decrypting information from the source that  
2138 provided it since an attacker might replay information from a prior transaction in the hope  
2139 of learning information about it.

2140

2141 In most cases the other party in a transaction is only imperfectly trustworthy. In these  
2142 cases both parties should contribute entropy to the key exchange by means of the  
2143 <wst:entropy> element.

---

### 2144 A.1 Ephemeral Encryption Keys

2145 The simplest form of key exchange can be found in [WS-Security] for encrypting message  
2146 data. As described in [WS-Security] and [XML-Encrypt], when data is encrypted, a  
2147 temporary key can be used to perform the encryption which is, itself, then encrypted using  
2148 the <xenc:EncryptedKey> element.

2149

2150 The example below illustrates encrypting a temporary key using the public key in an issuer  
2151 name and serial number:

```
2152     <xenc:EncryptedKey xmlns:xenc="...">  
2153     ...  
2154     <ds:KeyInfo xmlns:ds="...">  
2155         <wsse:SecurityTokenReference xmlns:wsse="...">  
2156             <ds:X509IssuerSerial>  
2157                 <ds:X509IssuerName>  
2158                     DC=ACMECorp, DC=com  
2159                 </ds:X509IssuerName>  
2160                 <ds:X509SerialNumber>12345678</ds:X509SerialNumber>  
2161             </ds:X509IssuerSerial>  
2162         </wsse:SecurityTokenReference>  
2163     </ds:KeyInfo>  
2164     ...  
2165 </xenc:EncryptedKey>
```

---

## 2166 A.2 Requestor-Provided Keys

---

2167 When a request sends a message to an issuer to request a token, the client can provide  
2168 proposed key material using the <wst:Entropy> element. If the issuer doesn't contribute  
2169 any key material, this is used as the secret (key). This information is encrypted for the  
2170 issuer either using <xenc:EncryptedKey> or by using a transport security. If the requestor  
2171 provides key material that the recipient doesn't accept, then the issuer should reject the  
2172 request. Note that the issuer need not return the key provided by the requestor.

2173  
2174 The following example illustrates a request for a custom security token and includes a secret  
2175 that is to be used for the key. In this example the entropy is encrypted for the issuer (if  
2176 transport security was used for confidentiality then the <wst:Entropy> element would  
2177 contain a <wst:BinarySecret> element):

```
2178 ...  
2179     <wst:RequestSecurityToken>  
2180       <wst:TokenType>  
2181         http://example.org/mySpecialToken  
2182       </wst:TokenType>  
2183       <wst:RequestType>  
2184         http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue  
2185       </wst:RequestType>  
2186       <wst:Entropy>  
2187         <xenc:EncryptedData>...</xenc:EncryptedData>  
2188       </wst:Entropy>  
2189     </wst:RequestSecurityToken>  
2190     ...
```

---

## 2191 A.3 Issuer-Provided Keys

---

2192 If a requestor fails to provide key material, then issued proof-of-possession tokens contain  
2193 an issuer-provided secret that is encrypted for the requestor (either using  
2194 <xenc:EncryptedKey> or by using a transport security).

2195  
2196 The following example illustrates a token being returned with an associated proof-of-  
2197 possession token that is encrypted using the requestor's public key.

```
2198 ...  
2199     <wst:RequestSecurityTokenResponse>  
2200       <wst:RequestedSecurityToken>  
2201         <xyz:CustomToken xmlns:xyz="...">  
2202           ...  
2203         </xyz:CustomToken>  
2204       </wst:RequestedSecurityToken>  
2205       <wst:RequestedProofToken>  
2206         <xenc:EncryptedKey Id="newProof">  
2207           ...  
2208         </xenc:EncryptedKey>  
2209       </wst:RequestedProofToken>  
2210     </wst:RequestSecurityTokenResponse>  
2211     ...
```

---

## 2212 A.4 Composite Keys

---

2213 The safest form of key exchange/generation is when both the requestor and the issuer  
2214 contribute to the key material. In this case, the request sends encrypted key material. The  
2215 issuer then returns additional encrypted key material. The actual secret (key) is computed

2216 using a function of the two pieces of data. Ideally this secret is never used and, instead,  
2217 keys derived are used for message protection.

2218

2219 The following example illustrates a server, having received a request with requestor entropy  
2220 returning its own entropy, which is used in conjunction with the requestor's to generate a  
2221 key. In this example the entropy is not encrypted because the transport is providing  
2222 confidentiality (otherwise the <wst:Entropy> element would have an  
2223 <xenc:EncryptedData> element).

```
2224 ...  
2225 <wst:RequestSecurityTokenResponse>  
2226 <wst:RequestedSecurityToken>  
2227 <xyz:CustomToken xmlns:xyz="...">  
2228 ...  
2229 </xyz:CustomToken>  
2230 </wst:RequestedSecurityToken>  
2231 <wst:Entropy>  
2232 <wst:BinarySecret>JIH...</wst:BinarySecret>  
2233 </wst:Entropy>  
2234 </wst:RequestSecurityTokenResponse>  
2235 ...
```

## 2236 A.5 Key Transfer and Distribution

2237 There are also a few mechanisms where existing keys are transferred to other parties.

### 2238 A.5.1 Direct Key Transfer

2239 If one party has a token and key and wishes to share this with another party, the key can  
2240 be directly transferred. This is accomplished by sending an RSTR (either in the body or  
2241 header) to the other party. The RSTR contains the token and a proof-of-possession token  
2242 that contains the key encrypted for the recipient.

2243

2244 In the following example a custom token and its associated proof-of-possession token are  
2245 known to party A who wishes to share them with party B. In this example, A is a member  
2246 in a secure on-line chat session and is inviting B to join the conversation. After  
2247 authenticating B, A sends B an RSTR. The RSTR contains the token and the key is  
2248 communicated as a proof-of-possession token that is encrypted for B:

```
2249 ...  
2250 <wst:RequestSecurityTokenResponse>  
2251 <wst:RequestedSecurityToken>  
2252 <xyz:CustomToken xmlns:xyz="...">  
2253 ...  
2254 </xyz:CustomToken>  
2255 </wst:RequestedSecurityToken>  
2256 <wst:RequestedProofToken>  
2257 <xenc:EncryptedKey Id="newProof">  
2258 ...  
2259 </xenc:EncryptedKey>  
2260 </wst:RequestedProofToken>  
2261 </wst:RequestSecurityTokenResponse>  
2262 ...
```

---

## 2263 A.5.2 Brokered Key Distribution

---

2264 A third party may also act as a broker to transfer keys. For example, a requestor may  
2265 obtain a token and proof-of-possession token from a third-party STS. The token contains a  
2266 key encrypted for the target service (either using the service's public key or a key known to  
2267 the STS and target service). The proof-of-possession token contains the same key  
2268 encrypted for the requestor (similarly this can use public or symmetric keys).

2269  
2270 In the following example a custom token and its associated proof-of-possession token are  
2271 returned from a broker B to a requestor R for access to service S. The key for the session is  
2272 contained within the custom token encrypted for S using either a secret known by B and S  
2273 or using S's public key. The same secret is encrypted for R and returned as the proof-of-  
2274 possession token:

```
2275 ...  
2276     <wst:RequestSecurityTokenResponse>  
2277         <wst:RequestedSecurityToken>  
2278             <xyz:CustomToken xmlns:xyz="...">  
2279                 ...  
2280                 <xenc:EncryptedKey xmlns:xenc="...">  
2281                     ...  
2282                 </xenc:EncryptedKey>  
2283             </xyz:CustomToken>  
2284         </wst:RequestedSecurityToken>  
2285         <wst:RequestedProofToken>  
2286             <xenc:EncryptedKey Id="newProof">  
2287                 ...  
2288             </xenc:EncryptedKey>  
2289         </wst:RequestedProofToken>  
2290     </wst:RequestSecurityTokenResponse>  
2291     ...  
2292
```

---

## 2293 A.5.3 Delegated Key Transfer

---

2294 Key transfer can also take the form of delegation. That is, one party transfers the right to  
2295 use a key without actually transferring the key. In such cases, a delegation token, e.g.  
2296 XrML, is created that identifies a set of rights and a delegation target and is secured by the  
2297 delegating party. That is, one key indicates that another key can use a subset (or all) of its  
2298 rights. The delegate can provide this token and prove itself (using its own key – the  
2299 delegation target) to a service. The service, assuming the trust relationships have been  
2300 established and that the delegator has the right to delegate, can then authorize requests  
2301 sent subject to delegation rules and trust policies.

2302  
2303 In this example a custom token is issued from party A to party B. The token indicates that  
2304 B (specifically B's key) has the right to submit purchase orders. The token is signed using a  
2305 secret key known to the target service T and party A (the key used to ultimately authorize  
2306 the requests that B makes to T), and a new session key that is encrypted for T. A proof-of-  
2307 possession token is included that contains the session key encrypted for B. As a result, B is  
2308 *effectively* using A's key, but doesn't actually know the key.

```
2309 ...  
2310     <wst:RequestSecurityTokenResponse>  
2311         <wst:RequestedSecurityToken>  
2312             <xyz:CustomToken xmlns:xyz="...">  
2313                 ...  
2314             </xyz:CustomToken>  
2315         </wst:RequestedSecurityToken>  
2316     </wst:RequestSecurityTokenResponse>  
2317     ...  
2318
```

```

2314         <xyz:DelegateTo>R</xyz:DelegateTo>
2315         <xyz:DelegateRights>
2316             SubmitPurchaseOrder
2317         </xyz:DelegatedRights>
2318         <xenc:EncryptedKey xmlns:xenc="...">
2319             ...
2320         </xenc:EncryptedKey>
2321         <ds:Signature xmlns:ds="...">...</ds:Signature>
2322         ...
2323     </xyz:CustomToken>
2324 </wst:RequestedSecurityToken>
2325 <wst:RequestedProofToken>
2326     <xenc:EncryptedKey Id="newProof">
2327         ...
2328     </xenc:EncryptedKey>
2329 </wst:RequestedProofToken>
2330 </wst:RequestSecurityTokenResponse>
2331 ...

```

#### 2332 **A.5.4 Authenticated Request/Reply Key Transfer**

2333 In some cases the RST/RSTR mechanism is not used to transfer keys because it is part of a  
2334 simple request/reply. However, there may be a desire to ensure mutual authentication as  
2335 part of the key transfer. The mechanisms of [WS-Security] can be used to implement this  
2336 scenario.

2337

2338 Specifically, the sender wishes the following:

- 2339 • Transfer a key to a recipient that they can use to secure a reply
- 2340 • Ensure that only the recipient can see the key
- 2341 • Provide proof that the sender issued the key

2342

2343 This scenario could be supported by encrypting and then signing. This would result in  
2344 roughly the following steps:

- 2345 1. Encrypt the message using a generated key
- 2346 2. Encrypt the key for the recipient
- 2347 3. Sign the encrypted form, any other relevant keys, and the encrypted key

2348

2349 However, if there is a desire to sign prior to encryption then the following general process is  
2350 used:

- 2351 1. Sign the appropriate message parts using a random key (or ideally a key derived  
2352 from a random key)
- 2353 2. Encrypt the appropriate message parts using the random key (or ideally another key  
2354 derived from the random key)
- 2355 3. Encrypt the random key for the recipient
- 2356 4. Sign just the encrypted key

2357

2358 This would result in a <wsse:Security> header that looks roughly like the following:

```

2359 ...
2360 <wsse:Security xmlns:wsse="..." xmlns:wssu="..."
2361     xmlns:ds="..." xmlns:xenc="...">

```

```

2362     <wsse:BinarySecurityToken wsu:Id="myToken" ... >
2363     ...
2364     </wsse:BinarySecurityToken>
2365     <ds:Signature>
2366     ... signature over #secret using token #myToken...
2367     </ds:Signature>
2368     <xenc:EncryptedKey Id="secret">
2369     ...
2370     </xenc:EncryptedKey>
2371     <xenc:ReferenceList>
2372     ... manifest of encrypted parts using token #secret...
2373     </xenc:ReferenceList>
2374     <ds:Signature>
2375     ... signature over key message parts using token #secret...
2376     </ds:Signature>
2377     </wsse:Security>
2378     ...

```

2379

2380 As well, instead of an `<xenc:EncryptedKey>` element, the actual token could be passed

2381 using `<xenc:EncryptedData>`. The result might look like the following:

```

2382     ...
2383     <wsse:Security xmlns:wsse="..." xmlns:wsu="..."
2384     xmlns:ds="..." xmlns:xenc="..." >
2385     <wsse:BinarySecurityToken wsu:Id="myToken" ... >
2386     ...
2387     </wsse:BinarySecurityToken>
2388     <ds:Signature>
2389     ... signature over #secret or #Esecret using token #myToken...
2390     </ds:Signature>
2391     <xenc:EncryptedData Id="Esecret">
2392     ... Encrypted version of a token with Id="secret"...
2393     </xenc:EncryptedData>
2394     <xenc:ReferenceList>
2395     ... manifest of encrypted parts using token #secret...
2396     </xenc:ReferenceList>
2397     <ds:Signature>
2398     ... signature over key message parts using token #secret...
2399     </ds:Signature>
2400     </wsse:Security>
2401     ...

```

## 2402 A.6 Perfect Forward Secrecy

2403 In some situations it is desirable for a key exchange to have the property of perfect forward

2404 secrecy. This means that it is impossible to reconstruct the shared secret even if the private

2405 keys of the parties are disclosed.

2406

2407 The most straightforward way to attain perfect forward secrecy when using asymmetric key

2408 exchange is to dispose of one's key exchange key pair periodically (or even after every key

2409 exchange), replacing it with a fresh one. Of course, a freshly generated public key must still

2410 be authenticated (using any of the methods normally available to prove the identity of a

2411 public key's owner).

2412

2413 The perfect forward secrecy property may be achieved by specifying a `<wst:entropy>`

2414 element that contains an `<xenc:EncryptedKey>` that is encrypted under a public key pair

2415 created for use in a single key agreement. The public key does not require authentication

---

2416 since it is only used to provide additional entropy. If the public key is modified, the key  
2417 agreement will fail. Care should be taken, when using this method, to ensure that the now-  
2418 secret entropy exchanged via the `<wst:entropy>` element is not revealed elsewhere in the  
2419 protocol (since such entropy is often assumed to be publicly revealed plaintext, and treated  
2420 accordingly).

---

2421

---

2422 Although any public key scheme might be used to achieve perfect forward secrecy (in either  
2423 of the above methods) it is generally desirable to use an algorithm that allows keys to be  
2424 generated quickly. The Diffie-Hellman key exchange is often used for this purpose since  
2425 generation of a key only requires the generation of a random integer and calculation of a  
2426 single modular exponent.

---

## 2427 B. WSDL

2428 The WSDL below does not fully capture all the possible message exchange patterns, but  
2429 captures the typical message exchange pattern as described in this document.

```
2430 <?xml version="1.0"?>
2431 <wsdl:definitions
2432     targetNamespace="http://docs.oasis-open.org/ws-sx/ws-
2433     trust/200512/wsdl"
2434     xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/wsdl"
2435     xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
2436     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
2437     xmlns:xs="http://www.w3.org/2001/XMLSchema"
2438 >
2439 <!-- this is the WS-T BP-compliant way to import a schema -->
2440 <wsdl:types>
2441 <xs:schema>
2442 <xs:import
2443     namespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
2444     schemaLocation="http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-
2445     trust.xsd"/>
2446 </xs:schema>
2447 </wsdl:types>
2448
2449 <!-- WS-Trust defines the following GEDs -->
2450 <wsdl:message name="RequestSecurityTokenMsg">
2451 <wsdl:part name="request" element="wst:RequestSecurityToken" />
2452 </wsdl:message>
2453 <wsdl:message name="RequestSecurityTokenResponseMsg">
2454 <wsdl:part name="response"
2455     element="wst:RequestSecurityTokenResponse" />
2456 </wsdl:message>
2457 <wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
2458 <wsdl:part name="responseCollection"
2459     element="wst:RequestSecurityTokenResponseCollection"/>
2460 </wsdl:message>
2461
2462 <!-- This portType models the full request/response the Security Token
2463 Service: -->
2464
2465 <wsdl:portType name="WSSecurityRequestor">
2466 <wsdl:operation name="SecurityTokenResponse">
2467 <wsdl:input
2468     message="tns:RequestSecurityTokenResponseMsg"/>
2469 </wsdl:operation>
2470 <wsdl:operation name="SecurityTokenResponse2">
2471 <wsdl:input
2472     message="tns:RequestSecurityTokenResponseCollectionMsg"/>
2473 </wsdl:operation>
2474 <wsdl:operation name="Challenge">
2475 <wsdl:input message="tns:RequestSecurityTokenResponseMsg" />
2476 <wsdl:output message="tns:RequestSecurityTokenResponseMsg" />
2477 </wsdl:operation>
2478 <wsdl:operation name="Challenge2">
2479 <wsdl:input message="tns:RequestSecurityTokenResponseMsg" />
2480 <wsdl:output
2481     message="tns:RequestSecurityTokenResponseCollectionMsg" />
2482 </wsdl:operation>
2483 </wsdl:portType>
2484
2485 <!-- These portTypes model the individual message exchanges -->
```

```
2486
2487     <wsdl:portType name="SecurityTokenRequestService">
2488         <wsdl:operation name="RequestSecurityToken">
2489             <wsdl:input message="tns:RequestSecurityTokenMsg" />
2490         </wsdl:operation>
2491     </wsdl:portType>
2492
2493     <wsdl:portType name="SecurityTokenService">
2494         <wsdl:operation name="RequestSecurityToken">
2495             <wsdl:input message="tns:RequestSecurityTokenMsg" />
2496             <wsdl:output message="tns:RequestSecurityTokenResponseMsg" />
2497         </wsdl:operation>
2498         <wsdl:operation name="RequestSecurityToken2">
2499             <wsdl:input message="tns:RequestSecurityTokenMsg" />
2500             <wsdl:output
2501                 message="tns:RequestSecurityTokenResponseCollectionMsg" />
2502         </wsdl:operation>
2503     </wsdl:portType>
2504 </wsdl:definitions>
2505
```

---

## 2506 C. Acknowledgements

2507 The following individuals have participated in the creation of this specification and are gratefully  
2508 acknowledged:

### 2509 **Original Authors:**

2510 Steve Anderson, OpenNetwork

2511 Jeff Bohren, OpenNetwork

2512 Toufic Boubez, Layer 7

2513 Marc Chanliau, Computer Associates

2514 Giovanni Della-Libera, Microsoft

2515 Brendan Dixon, Microsoft

2516 Praerit Garg, Microsoft

2517 Martin Gudgin (Editor), Microsoft

2518 Phillip Hallam-Baker, VeriSign

2519 Maryann Hondo, IBM

2520 Chris Kaler, Microsoft

2521 Hal Lockhart, BEA

2522 Robin Martherus, Oblix

2523 Hiroshi Maruyama, IBM

2524 Anthony Nadalin (Editor), IBM

2525 Nataraj Nagaratnam, IBM

2526 Andrew Nash, Reactivity

2527 Rob Philpott, RSA Security

2528 Darren Platt, Ping Identity

2529 Hemma Prafullchandra, VeriSign

2530 Maneesh Sahu, Actional

2531 John Shewchuk, Microsoft

2532 Dan Simon, Microsoft

2533 Davanum Srinivas, Computer Associates

2534 Elliot Waingold, Microsoft

2535 David Waite, Ping Identity

2536 Doug Walter, Microsoft

2537 Riaz Zolfonoon, RSA Security

2538

### 2539 **Original Acknowledgments:**

2540 Paula Austel, IBM

2541 Keith Ballinger, Microsoft

2542 Bob Blakley, IBM

2543 John Brezak, Microsoft

2544 Tony Cowan, IBM

2545 Cédric Fournet, Microsoft

2546 Vijay Gajjala, Microsoft

2547 HongMei Ge, Microsoft

2548 Satoshi Hada, IBM

2549 Heather Hinton, IBM

2550 Slava Kavsan, RSA Security

2551 Scott Konersmann, Microsoft

2552 Leo Laferriere, Computer Associates

2553	Paul Leach, Microsoft
2554	Richard Levinson, Computer Associates
2555	John Linn, RSA Security
2556	Michael McIntosh, IBM
2557	Steve Millet, Microsoft
2558	Birgit Pfitzmann, IBM
2559	Fumiko Satoh, IBM
2560	Keith Stobie, Microsoft
2561	T.R. Vishwanath, Microsoft
2562	Richard Ward, Microsoft
2563	Hervey Wilson, Microsoft



---

2565

## E. Revision History

---

2566 [optional; should not be included in OASIS Standards]

---

2567

Revision	Date	Editor	Changes Made
0.1	12-06-2005	Anthony Nadalin	Initial convergence to OASIS template
0.2	01-09-2006	Martin Gudgin	Updated TOC. Namespaces.

---

2568