WS-Trust 1.3

Committee Draft 01, 06 September 2006

Artifact Identifier:
ws-trust-1.3-spec-cd-01

Location:
Current: docs.oasis-open.org/ws-sx/ws-trust/200512
This Version: docs.oasis-open.org/ws-sx/ws-trust/200512
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
Marc Goodner, Microsoft
Martin Gudgin, Microsoft
Abbie Barbir, Nortel
Hans Granqvist, VeriSign

OASIS Conceptual Model topic area:
[Topic Area]

Related work:
N/A

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

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

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

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

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

Notices

Copyright © OASIS Open 2006. All Rights Reserved.

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

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

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

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

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

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

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

Table of Contents

1 Introduction 5
1.1 Goals and Non-Goals 5
1.2 Requirements 6
1.3 Namespace 6
1.4 Schema and WSDL Files 7
1.5 Terminology 7
1.5.1 Notational Conventions 8
1.6 Normative References 9
1.7 Non-Normative References 10
2 Web Services Trust Model 11
2.1 Models for Trust Brokering and Assessment 12
2.2 Token Acquisition 12
2.3 Out-of-Band Token Acquisition 13
2.4 Trust Bootstrap 13
3 Security Token Service Framework 14
3.1 Requesting a Security Token 14
3.2 Returning a Security Token 15
3.3 Binary Secrets 17
3.4 Composition 17
4 Issuance Binding 19
4.1 Requesting a Security Token 19
4.2 Request Security Token Collection 21
4.2.1 Processing Rules 23
4.3 Returning a Security Token 23
4.3.1 wsp:AppliesTo in RST and RSTR 25
4.3.2 Requested References 26
4.3.3 Keys and Entropy 26
4.3.4 Returning Computed Keys 27
4.3.5 Sample Response with Encrypted Secret 27
4.3.6 Sample Response with Unencrypted Secret 28
4.3.7 Sample Response with Token Reference 28
4.3.8 Sample Response without Proof-of-Possession Token 29
4.3.9 Zero or One Proof-of-Possession Token Case 29
4.3.10 More Than One Proof-of-Possession Tokens Case 29
4.4 Returning Security Tokens in Headers 30
5 Renewal Binding 33
6 Cancel Binding 36
6.1 STS-initiated Cancel Binding 37
7 Validation Binding 39
8 Negotiation and Challenge Extensions 42
8.1 Negotiation and Challenge Framework 43
8.2 Signature Challenges 43
8.3 Binary Exchanges and Negotiations 44

Copyright © OASIS Open 2006. All Rights Reserved.

8.4 Key Exchange Tokens 45
8.5 Custom Exchanges 46
8.6 Signature Challenge Example 46
8.7 Custom Exchange Example 48
8.8 Protecting Exchanges 49
8.9 Authenticating Exchanges 50
9 Key and Token Parameter Extensions 51
9.1 On-Behalf-Of Parameters 51
9.2 Key and Encryption Requirements 51
9.3 Delegation and Forwarding Requirements 56
9.4 Policies 57
9.5 Authorized Token Participants 57
10 Key Exchange Token Binding 59
11 Error Handling 61
12 Security Considerations 62
A. Key Exchange 64
A.1 Ephemeral Encryption Keys 64
A.2 Requestor-Provided Keys 64
A.3 Issuer-Provided Keys 65
A.4 Composite Keys 65
A.5 Key Transfer and Distribution 66
A.5.1 Direct Key Transfer 66
A.5.2 Brokered Key Distribution 66
A.5.3 Delegated Key Transfer 67
A.5.4 Authenticated Request/Reply Key Transfer 68
A.6 Perfect Forward Secrecy 69
B. WSDL 70
C. Acknowledgements 72

Copyright © OASIS Open 2006. All Rights Reserved.

1 Introduction

0 [WS-Security] defines the basic mechanisms for providing secure messaging. This specification uses
1 these base mechanisms and defines additional primitives and extensions for security token exchange to
2 enable the issuance and dissemination of credentials within different trust domains.
3
4 In order to secure a communication between two parties, the two parties must exchange security
5 credentials (either directly or indirectly). However, each party needs to determine if they can "trust" the
6 asserted credentials of the other party.
7
8 In this specification we define extensions to [WS-Security] that provide:
9 Methods for issuing, renewing, and validating security tokens.
10 Ways to establish assess the presence of, and broker trust relationships.
11
12 Using these extensions, applications can engage in secure communication designed to work with the
13 general Web services framework, including WSDL service descriptions, UDDI businessServices and
14 bindingTemplates, and [SOAP] [SOAP2] messages.
15
16 To achieve this, this specification introduces a number of elements that are used to request security
17 tokens and broker trust relationships.
18
19 This specification defines a number of extensions; compliant services are NOT REQUIRED to implement
20 everything defined in this specification. However, if a service implements an aspect of the specification, it
21 MUST comply with the requirements specified (e.g. related "MUST" statements).
22
23 Section 12 is non-normative.

24 1.1 Goals and Non-Goals

25 The goal of WS-Trust is to enable applications to construct trusted [SOAP] message exchanges. This
26 trust is represented through the exchange and brokering of security tokens. This specification provides a
27 protocol agnostic way to issue, renew, and validate these security tokens.
28
29 This specification is intended to provide a flexible set of mechanisms that can be used to support a range
30 of security protocols; this specification intentionally does not describe explicit fixed security protocols.
31
32 As with every security protocol, significant efforts must be applied to ensure that specific profiles and
33 message exchanges constructed using WS-Trust are not vulnerable to attacks (or at least that the attacks
34 are understood).
35
36 The following are explicit non-goals for this document:
37 Password authentication
38 Token revocation

Copyright © OASIS Open 2006. All Rights Reserved.

39 Management of trust policies

40

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

42 Establishing a security context token

43 Key derivation

44 1.2 Requirements

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

47 Requesting and obtaining security tokens

48 Establishing, managing and assessing trust relationships

49 1.3 Namespace

50 The [URI] that MUST be used by implementations of this specification is:

51
```
http://docs.oasis-open.org/ws-sx/ws-trust/200512
```

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

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

Prefix	Namespace	Specification(s)
S11	http://schemas.xmlsoap.org/soap/envelope/	[SOAP]
S12	http://www.w3.org/2003/05/soap-envelope	[SOAP12]
wsu	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd	[WS-Security]
wsse	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd	[WS-Security]
wst	http://docs.oasis-open.org/ws-sx/ws-trust/200512	This specification
ds	http://www.w3.org/2000/09/xmldsig#	[XML-Signature]
xenc	http://www.w3.org/2001/04/xmlenc#	[XML-Encrypt]
wsp	http://schemas.xmlsoap.org/ws/2004/09/policy	[WS-Policy]
wsa	http://www.w3.org/2005/08/addressing	[WS-Addressing]
xs	http://www.w3.org/2001/XMLSchema	[XML-Schema1] [XML-Schema2]

Copyright © OASIS Open 2006. All Rights Reserved.

55 ## 1.4 Schema and WSDL Files

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

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

58

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

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

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

64 ## 1.5 Terminology

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

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

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

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

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

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

81 **Trust Engine** – The *trust engine* of a Web service is a conceptual component that evaluates the security-
82 related aspects of a message as described in section 2 below.

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

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

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

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

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

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

Copyright © OASIS Open 2006. All Rights Reserved.

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

101 1.5.1 Notational Conventions

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

105

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

108

109 This specification uses the following syntax to define outlines for messages:

110 The syntax appears as an XML instance, but values in italics indicate data types instead of literal
111 values.

112 Characters are appended to elements and attributes to indicate cardinality:

113 - "?" (0 or 1)
114 - "*" (0 or more)
115 - "+" (1 or more)

116 The character "|" is used to indicate a choice between alternatives.

117 The characters "(" and ")" are used to indicate that contained items are to be treated as a group
118 with respect to cardinality or choice.

119 The characters "[" and "]" are used to call out references and property names.

120 Ellipses (i.e., "...") indicate points of extensibility. Additional children and/or attributes MAY be
121 added at the indicated extension points but MUST NOT contradict the semantics of the parent
122 and/or owner, respectively. By default, if a receiver does not recognize an extension, the receiver
123 SHOULD ignore the extension; exceptions to this processing rule, if any, are clearly indicated
124 below.

125 XML namespace prefixes (see Table 1) are used to indicate the namespace of the element being
126 defined.

127

128 Elements and Attributes defined by this specification are referred to in the text of this document using
129 XPath 1.0 expressions. Extensibility points are referred to using an extended version of this syntax:

130 An element extensibility point is referred to using {any} in place of the element name. This
131 indicates that any element name can be used, from any namespace other than the namespace of
132 this specification.

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

136

137 In this document reference is made to the `wsu:Id` attribute and the `wsu:Created` and `wsu:Expires`
138 elements in a utility schema (http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
139 1.0.xsd). The `wsu:Id` attribute and the `wsu:Created` and `wsu:Expires` elements were added to the
140 utility schema with the intent that other specifications requiring such an ID type attribute or timestamp
141 element could reference it (as is done here).

142

Copyright © OASIS Open 2006. All Rights Reserved.

143 ## 1.6 Normative References

144 [RFC2119] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels",
145 RFC 2119, Harvard University, March 1997.
146 http://www.ietf.org/rfc/rfc2119.txt.
147 [RFC2246] IETF Standard, "The TLS Protocol", January 1999.
148 http://www.ietf.org/rfc/rfc2246.txt.
149 [SOAP] W3C Note, "SOAP: Simple Object Access Protocol 1.1", 08 May 2000.
150 http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.
151 [SOAP12] W3C Recommendation, "SOAP 1.2 Part 1: Messaging Framework", 24
152 June 2003.
153 http://www.w3.org/TR/2003/REC-soap12-part1-20030624/.
154 [URI] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers
155 (URI): Generic Syntax", RFC 3986, MIT/LCS, Day Software, Adobe
156 Systems, January 2005.
157 http://www.ietf.org/rfc/rfc3986.txt
158 [WS-Addressing] W3C Recommendation, "Web Services Addressing (WS-Addressing)", 9
159 May 2006.
160 http://www.w3.org/TR/2006/REC-ws-addr-core-20060509.
161 [WS-Policy] W3C Member Submission, "Web Services Policy 1.2 - Framework", 25
162 April 2006.
163 http://www.w3.org/Submission/2006/SUBM-WS-Policy-20060425/
164 [WS-PolicyAttachment] W3C Member Submission, "Web Services Policy 1.2 - Attachment", 25
165 April 2006.
166 http://www.w3.org/Submission/2006/SUBM-WS-PolicyAttachment-
167 20060425/
168 [WS-Security] OASIS Standard, "OASIS Web Services Security: SOAP Message Security
169 1.0 (WS-Security 2004)", March 2004.
170 http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-
171 security-1.0.pdf
172 OASIS Standard, "OASIS Web Services Security: SOAP Message Security
173 1.1 (WS-Security 2004)", February 2006.
174 http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-
175 spec-os-SOAPMessageSecurity.pdf
176 [XML-C14N] W3C Recommendation, "Canonical XML Version 1.0", 15 March 2001.
177 http://www.w3.org/TR/2001/REC-xml-c14n-20010315.
178 [XML-Encrypt] W3C Recommendation, "XML Encryption Syntax and Processing", 10
179 December 2002.
180 http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/.
181 [XML-Schema1] W3C Recommendation, "XML Schema Part 1: Structures Second Edition",
182 28 October 2004.
183 http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.
184 [XML-Schema2] W3C Recommendation, "XML Schema Part 2: Datatypes Second Edition",
185 28 October 2004.
186 http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/
187 [XML-Signature] W3C Recommendation, "XML-Signature Syntax and Processing", 12
188 February 2002.
189 http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/.

190

Copyright © OASIS Open 2006. All Rights Reserved.

191
1.7 Non-Normative References

192 193	[Kerberos]	J. Kohl and C. Neuman, "The Kerberos Network 149 Authentication Service (V5)," RFC 1510, September 1993.
194		http://www.ietf.org/rfc/rfc1510.txt
195 196	[WS-Federation]	"Web Services Federation Language," BEA, IBM, Microsoft, RSA Security, VeriSign, July 2003.
197	[WS-SecurityPolicy]	OASIS Committee Draft, “WS-SecurityPolicy 1.2”, September 2006
198		http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512
199 200	[X509]	S. Santesson, et al,"Internet X.509 Public Key Infrastructure Qualified Certificates Profile."
201 202		http://www.itu.int/rec/recommendation.asp?type=items&lang=e&parent=T-REC-X.509-200003-I

Copyright © OASIS Open 2006. All Rights Reserved.

2 Web Services Trust Model

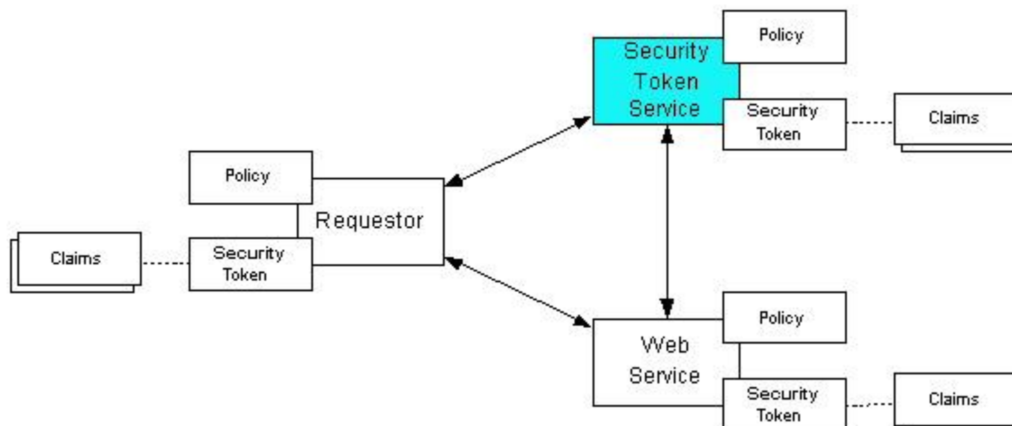
The Web service security model defined in WS-Trust is based on a process in which a Web service can require that an incoming message prove a set of claims (e.g., name, key, permission, capability, etc.). If a message arrives without having the required proof of claims, the service SHOULD ignore or reject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

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

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

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

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



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

Copyright © OASIS Open 2006. All Rights Reserved.

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

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

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

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

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

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

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

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

2.1 Models for Trust Brokering and Assessment

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

2.2 Token Acquisition

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

Copyright © OASIS Open 2006. All Rights Reserved.

280 or by another party on the requestor's behalf. If the security token service trusts the provided security
281 token (for example, because it trusts the issuing authority of the provided security token), and the request
282 can prove possession of that security token, then the exchange is processed by the security token
283 service.
284
285 The previous paragraph illustrates an example of token acquisition in a direct trust relationship. In the
286 case of a delegated request (one in which another party provides the request on behalf of the requestor
287 rather than the requestor presenting it themselves), the security token service generating the new token
288 may not need to trust the authority that issued the original token provided by the original requestor since it
289 does trust the security token service that is engaging in the exchange for a new security token. The basis
290 of the trust is the relationship between the two security token services.

291 2.3 Out-of-Band Token Acquisition

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

297 2.4 Trust Bootstrap

298 An administrator or other trusted authority may designate that all tokens of a certain type are trusted (e.g.
299 all Kerberos tokens from a specific realm or all X.509 tokens from a specific CA). The security token
300 service maintains this as a trust axiom and can communicate this to trust engines to make their own trust
301 decisions (or revoke it later), or the security token service may provide this function as a service to
302 trusting services.
303 There are several different mechanisms that can be used to bootstrap trust for a service. These
304 mechanisms are non-normative and are not required in any way. That is, services are free to bootstrap
305 trust and establish trust among a domain of services or extend this trust to other domains using any
306 mechanism.
307
308 **Fixed trust roots** – The simplest mechanism is where the recipient has a fixed set of trust relationships.
309 It will then evaluate all requests to determine if they contain security tokens from one of the trusted roots.
310
311 **Trust hierarchies** – Building on the trust roots mechanism, a service may choose to allow hierarchies of
312 trust so long as the trust chain eventually leads to one of the known trust roots. In some cases the
313 recipient may require the sender to provide the full hierarchy. In other cases, the recipient may be able to
314 dynamically fetch the tokens for the hierarchy from a token store.
315
316 **Authentication service** – Another approach is to use an authentication service. This can essentially be
317 thought of as a fixed trust root where the recipient only trusts the authentication service. Consequently,
318 the recipient forwards tokens to the authentication service, which replies with an authoritative statement
319 (perhaps a separate token or a signed document) attesting to the authentication.

Copyright © OASIS Open 2006. All Rights Reserved.

3 Security Token Service Framework

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

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

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

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

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

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

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

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

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

3.1 Requesting a Security Token

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

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

Copyright © OASIS Open 2006. All Rights Reserved.

The syntax for this element is as follows:

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

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

/wst:RequestSecurityToken

This is a request to have a security token issued.

/wst:RequestSecurityToken/@Context

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

/wst:RequestSecurityToken/wst:TokenType

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

/wst:RequestSecurityToken/wst:RequestType

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

/wst:RequestSecurityToken/wst:SecondaryParameters

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

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

/wst:RequestSecurityToken/{any}

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

/wst:RequestSecurityToken/@{any}

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

3.2 Returning a Security Token

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

Copyright © OASIS Open 2006. All Rights Reserved.

element (RSTRC) MUST be used to return a security token or response to a security token request on the final response.

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

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

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

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

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

The syntax for this element is as follows:

```
<wst:RequestSecurityTokenResponse Context="..." xmlns:wst="...">
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
    ...
</wst:RequestSecurityTokenResponse>
```

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

/wst:RequestSecurityTokenResponse

This is the response to a security token request.

/wst:RequestSecurityTokenResponse/@Context

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

/wst:RequestSecurityTokenResponse/wst:TokenType

This optional element specifies the type of security token returned.

/wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken

This optional element is used to return the requested security token. Normally the requested security token is the contents of this element but a security token reference MAY be used instead. For example, if the requested security token is used in securing the message, then the security token is placed into the `<wsse:Security>` header (as described in [WS-Security]) and a

Copyright © OASIS Open 2006. All Rights Reserved.

454 <wsse:SecurityTokenReference> element is placed inside of the
455 <wst:RequestedSecurityToken> element to reference the token in the <wsse:Security>
456 header. The response MAY contain a token reference where the token is located at a URI
457 outside of the message. In such cases the recipient is assumed to know how to fetch the token
458 from the URI address or specified endpoint reference. It should be noted that when the token is
459 not returned as part of the message it cannot be secured, so a secure communication
460 mechanism SHOULD be used to obtain the token.

461 */wst:RequestSecurityTokenResponse/{any}*

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

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

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

467 3.3 Binary Secrets

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

475 *.../wst:BinarySecret*

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

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

479 This optional attribute indicates the type of secret being encoded. The pre-defined values are
480 listed in the table below:

URI	Meaning
http://docs.oasis-open.org/ws-sx/ws-trust/200512/AsymmetricKey	The private portion of a public key token is returned – this URI assumes both parties agree on the format of the octets; other bindings and profiles MAY define additional URIs with specific formats
http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey	A symmetric key token is returned (default)
http://docs.oasis-open.org/ws-sx/ws-trust/200512/Nonce	A raw nonce value (typically passed as entropy or key material)

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

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

484 3.4 Composition

485 The sections below, as well as other documents, describe a set of bindings using the model framework
486 described in the above sections. Each binding describes the amount of extensibility and composition with

Copyright © OASIS Open 2006. All Rights Reserved.

487 other parts of WS-Trust that is permitted. Additional profile documents MAY further restrict what can be
488 specified in a usage of a binding.

Copyright © OASIS Open 2006. All Rights Reserved.

4 Issuance Binding

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

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

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

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

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

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

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

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

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

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

4.1 Requesting a Security Token

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

```
<wst:RequestSecurityToken xmlns:wst="...">
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>...</wst:RequestType>
    ...
    <wsp:AppliesTo>...</wsp:AppliesTo>
    <wst:Claims Dialect="...">...</wst:Claims>
    <wst:Entropy>
          <wst:BinarySecret>...</wst:BinarySecret>
     </wst:Entropy>
    <wst:Lifetime>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
533                 <wsu:Created>...</wsu:Created>
534                 <wsu:Expires>...</wsu:Expires>
535             </wst:Lifetime>
536         </wst:RequestSecurityToken>
```

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

538 */wst:RequestSecurityToken/wst:TokenType*

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

547 */wst:RequestSecurityToken/wsp:AppliesTo*

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

555 */wst:RequestSecurityToken/wst:Claims*

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

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

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

563 */wst:RequestSecurityToken/wst:Entropy*

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

568 */wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret*

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

575 */wst:RequestSecurityToken/wst:Lifetime*

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

Copyright © OASIS Open 2006. All Rights Reserved.

/wst:RequestSecurityToken/wst:Lifetime/wsu:Created

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

/wst:RequestSecurityToken/wst:Lifetime/wsu:Expires

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

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

```
<S11:Envelope xmlns:S11="..." xmlns:wsu="..." xmlns:wsse="..."
        xmlns:xenc="..." xmlns:wst="...">
    <S11:Header>
        ...
        <wsse:Security>
            <xenc:ReferenceList>...</xenc:ReferenceList>
            <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>
            <ds:Signature xmlns:ds="...">
                ...
             <ds:KeyInfo>
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#myToken"/>
                </wsse:SecurityTokenReference>
             </ds:KeyInfo>
            </ds:Signature>
        </wsse:Security>
        ...
    </S11:Header>
    <S11:Body wsu:Id="req">
        <wst:RequestSecurityToken>
            <wst:TokenType>
                http://example.org/mySpecialToken
            </wst:TokenType>
            <wst:RequestType>
                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
            </wst:RequestType>
        </wst:RequestSecurityToken>
    </S11:Body>
</S11:Envelope>
```

4.2 Request Security Token Collection

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

Copyright © OASIS Open 2006. All Rights Reserved.

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

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

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

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

    <!-- identity token request -->
    <wst:RequestSecurityToken Context="http://www.example.com/1">
        <wst:TokenType>
          http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0
        </wst:TokenType>
        <wst:RequestType>http://docs.oasis-open.org/ws-sx/ws-
trust/200512/BatchIssue</wst:RequestType>
        <wsp:AppliesTo xmlns:wsp="..." xmlns:wsa="...">
            <wsa:EndpointReference>
               <wsa:Address>http://manufacturer.example.com/</wsa:Address>
            </wsa:EndpointReference>
        </wsp:AppliesTo>
        <wsp:PolicyReference xmlns:wsp="..."
URI='http://manufacturer.example.com/IdentityPolicy' />
    </wst:RequestSecurityToken>

    <!-- certification claim token request -->
    <wst:RequestSecurityToken Context="http://www.example.com/2">
        <wst:TokenType>
        http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0
        </wst:TokenType>
        <wst:RequestType>http://docs.oasis-open.org/ws-sx/ws-trust/200512
/BatchIssue</wst:RequestType>
        <wsp:Claims xmlns:wsp="...">
          http://manufacturer.example.com/certification
        </wsp:Claims>
        <wsp:PolicyReference
URI='http://certificationbody.example.org/certificationPolicy’ />
    </wst:RequestSecurityToken>
  </wst:RequestSecurityTokenCollection>
```

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

/wst:RequestSecurityTokenCollection

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

Copyright © OASIS Open 2006. All Rights Reserved.

4.2.1 Processing Rules

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

1. The single `RequestSecurityTokenResponseCollection` response MUST contain at least one RSTR element corresponding to each RST element in the request. A RSTR element corresponds to an RST element if it has the same Context attribute value as the RST element.
   Note: Each request may generate more than one RSTR sharing the same Context attribute value
   a. Specifically there is no notion of a deferred response
   b. If any RST request results in an error, then no RSTRs will be returned and a SOAP Fault will be generated as the entire response.
2. Every RST in the request MUST use an action URI value in the RequestType element that is a batch version corresponding to the non-batch version, in particular one of the following:

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

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

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

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

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

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

3. All Signatures MUST reference the entire RSTC. One or more Signatures referencing the entire collection MAY be used.
4. No negotiation or other multi-leg authentication mechanisms are allowed in batch requests or responses to batch requests; the communication with STS is limited to one RSTC request and one RSTRC response.
5. This mechanism requires that every RST in a RSTC is to be handled by the single endpoint processing the RSTC.

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

4.3 Returning a Security Token

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

```
<wst:RequestSecurityTokenResponse xmlns:wst="...">
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
    ...
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
731            <wsp:AppliesTo xmlns:wsp="...”>...</wsp:AppliesTo>
732            <wst:RequestedAttachedReference>
733            ...
734            </wst:RequestedAttachedReference>
735            <wst:RequestedUnattachedReference>
736            ...
737           </wst:RequestedUnattachedReference>
738            <wst:RequestedProofToken>...</wst:RequestedProofToken>
739            <wst:Entropy>
740                <wst:BinarySecret>...</wst:BinarySecret>
741            </wst:Entropy>
742            <wst:Lifetime>...</wst:Lifetime>
743        </wst:RequestSecurityTokenResponse>
```

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

745 */wst:RequestSecurityTokenResponse/wsp:AppliesTo*

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

750 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

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

756 */wst:RequestSecurityTokenResponse/wst:RequestedAttachedReference*

757 Since returned tokens are considered opaque to the requestor, this optional element is specified
758 to indicate how to reference the returned token when that token doesn't support references using
759 URI fragments (XML ID). This element contains a `<wsse:SecurityTokenReference>`
760 element that can be used *verbatim* to reference the token (when the token is placed inside a
761 message). Typically tokens allow the use of *wsu:Id* so this element isn't required. Note that a
762 token MAY support multiple reference mechanisms; this indicates the issuer's preferred
763 mechanism. When encrypted tokens are returned, this element is not needed since the
764 `<xenc:EncryptedData>` element supports an ID reference. If this element is not present in the
765 RSTR then the recipient can assume that the returned token (when present in a message)
766 supports references using URI fragments.

767 */wst:RequestSecurityTokenResponse/wst:RequestedUnattachedReference*

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

774 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken*

775 This optional element is used to return the proof-of-possession token associated with the
776 requested security token. Normally the proof-of-possession token is the contents of this element
777 but a security token reference MAY be used instead. The token (or reference) is specified as the
778 contents of this element. For example, if the proof-of-possession token is used as part of the
779 securing of the message, then it is placed in the `<wsse:Security>` header and a
780 `<wsse:SecurityTokenReference>` element is used inside of the
781 `<wst:RequestedProofToken>` element to reference the token in the `<wsse:Security>`
782 header. This element is optional, but it is REQUIRED that at least one of

Copyright © OASIS Open 2006. All Rights Reserved.

783 <wst:RequestedSecurityToken> or <wst:RequestedProofToken> be returned unless
784 there is an error.

785 */wst:RequestSecurityTokenResponse/wst:Entropy*

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

790 */wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret*

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

793 */wst:RequestSecurityTokenResponse/wst:Lifetime*

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

797 ### 4.3.1 wsp:AppliesTo in RST and RSTR

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

802 If neither the requestor nor the issuer specifies a scope then the scope of the issued token is
803 implied.

804 If the requestor specifies a scope and the issuer does not then the scope of the token is assumed
805 to be that specified by the requestor.

806 If the requestor does not specify a scope and the issuer does specify a scope then the scope of
807 the token is as defined by the issuers scope

808 If both requestor and issuer specify a scope then there are two possible outcomes:

809 o If both the issuer and requestor specify the same scope then the issued token has that
810 scope.

811 o If the issuer specifies a wider scope than the requestor then the issued token has the
812 scope specified by the issuer.

813

814 The following table summarizes the above rules:

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

Copyright © OASIS Open 2006. All Rights Reserved.

815
4.3.2 Requested References

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

820 If a `<wst:RequestedAttachedReference>` element is NOT returned in the RSTR then the client
821 SHOULD assume that the token can be referenced by ID. Alternatively, the client MAY use token-
822 specific knowledge to construct an STR.

823 If a `<wst:RequestedAttachedReference>` element is returned in the RSTR then the token
824 cannot be referred to by ID. The supplied STR MUST be used to refer to the token.

825 If a `<wst:RequestedUnattachedReference>` element is returned then the server MAY reference
826 the token using the supplied STR when sending responses back to the client. Thus the client
827 MUST be prepared to resolve the supplied STR to the appropriate token. Note: the server
828 SHOULD NOT send the token back to the client as the token is often tailored specifically to the
829 server (i.e. it may be encrypted for the server). References to the token in subsequent messages,
830 whether sent by the client or the server, that omit the token MUST use the supplied STR.

831
4.3.3 Keys and Entropy

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

833 In the case of specific keys, a `<wst:RequestedProofToken>` element is included in the
834 response which indicates the specific key(s) to use unless the key was provided by the requestor
835 (in which case there is no need to return it).

836 In the case of partial, the `<wst:Entropy>` element is included in the response, which indicates
837 partial key material from the issuer (not the full key) that is combined (by each party) with the
838 requestor's entropy to determine the resulting key(s). In this case a `<wst:ComputedKey>`
839 element is returned inside the `<wst:RequestedProofToken>` to indicate how the key is
840 computed.

841 In the case of omitted, an existing key is used or the resulting token is not directly associated with
842 a key.

843

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

846 If the requestor does not provide entropy or issuer rejects the requestor's entropy, a proof-of-
847 possession token MUST be returned with an issuer-provided key.

848 If the requestor provides entropy and the responder doesn't (issuer uses the requestor's key),
849 then a proof-of-possession token need not be returned.

850 If both the requestor and the issuer provide entropy, then the partial form is used. Ideally both
851 entropies are specified as encrypted values and the resultant key is never used (only keys
852 derived from it are used). As noted above, the `<wst:ComputedKey>` element is returned inside
853 the `<wst:RequestedProofToken>` to indicate how the key is computed.

854

855 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.

Copyright © OASIS Open 2006. All Rights Reserved.

	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

4.3.4 Returning Computed Keys

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

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

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

/wst:RequestSecurityTokenResponse/wst:RequestedProofToken/wst:ComputedKey

The value of this element is a URI describing how to compute the key. While this can be extended by defining new URIs in other bindings and profiles, the following URI pre-defines one 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_{REQ}, Ent_{RES}) It is RECOMMENDED that EntREQ be a string of length at least 128 bits.

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

4.3.5 Sample Response with Encrypted Secret

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

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
```


Copyright © OASIS Open 2006. All Rights Reserved.

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

4.3.6 Sample Response with Unencrypted Secret

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

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

4.3.7 Sample Response with Token Reference

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

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
          <xyz:CustomToken ID="urn:fabrikam123:5445" xmlns:xyz="...">
              ...
          </xyz:CustomToken>
      </wst:RequestedSecurityToken>
      <wst:RequestedTokenReference>
          <wsse:SecurityTokenReference xmlns:wsse="...”>
             <wsse:Reference URI="urn:fabrikam123:5445"/>
          </wsse:SecurityTokenReference>
      </wst:RequestedTokenReference>
      ...
  </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

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

Copyright © OASIS Open 2006. All Rights Reserved.

4.3.8 Sample Response without Proof-of-Possession Token

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

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
          <xyz:CustomToken xmlns:xyz="...">
              ...
          </xyz:CustomToken>
      </wst:RequestedSecurityToken>
  </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

4.3.9 Zero or One Proof-of-Possession Token Case

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

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

4.3.10 More Than One Proof-of-Possession Tokens Case

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

```
  <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
      <wst:RequestSecurityTokenResponse>
          ...
      </wst:RequestSecurityTokenResponse>
      <wst:RequestSecurityTokenResponse>
          ...
      </wst:RequestSecurityTokenResponse>
      ...
  </wst:RequestSecurityTokenResponseCollection>
```


Copyright © OASIS Open 2006. All Rights Reserved.

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

986 */wst:RequestSecurityTokenResponseCollection*

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

990 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

991 Each RequestSecurityTokenResponse element is an individual RSTR.

992 */wst:RequestSecurityTokenResponseCollection/{any}*

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

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

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

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

```
998     <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
999         <wst:RequestSecurityTokenResponse>
1000            <wst:RequestedSecurityToken>
1001                <xyz:CustomToken xmlns:xyz="...">
1002                    ...
1003                </xyz:CustomToken>
1004            </wst:RequestedSecurityToken>
1005            <wst:RequestedProofToken>
1006                <xenc:EncryptedKey Id="newProofA">
1007                    ...
1008                </xenc:EncryptedKey>
1009            </wst:RequestedProofToken>
1010        </wst:RequestSecurityTokenResponse>
1011        <wst:RequestSecurityTokenResponse>
1012            <wst:RequestedSecurityToken>
1013                <abc:CustomToken xmlns:abc="...">
1014                    ...
1015                </abc:CustomToken>
1016            </wst:RequestedSecurityToken>
1017            <wst:RequestedProofToken>
1018                <xenc:EncryptedKey Id="newProofB xmlns:xenc="...”>
1019                    ...
1020                </xenc:EncryptedKey>
1021            </wst:RequestedProofToken>
1022        </wst:RequestSecurityTokenResponse>
1023    </wst:RequestSecurityTokenResponseCollection>
```

1024 4.4 Returning Security Tokens in Headers

1025 In certain situations it is useful to issue one or more security tokens as part of a protocol other than
1026 RST/RSTR. This typically requires that the tokens be passed in a SOAP header. The tokens present in
1027 that element can then be referenced from elsewhere in the message. This section defines a specific
1028 header element, whose type is the same as that of the `<wst:RequestSecurityTokenCollection>`
1029 element (see Section 4.3), that can be used to carry issued tokens (and associated proof tokens,
1030 references etc.) in a message.

```
1031    <wst:IssuedTokens xmlns:wst="...”>
1032      <wst:RequestSecurityTokenResponse>
1033      ...
1034      </wst:RequestSecurityTokenResponse>+
1035    </wst:IssuedTokens>
```


Copyright © OASIS Open 2006. All Rights Reserved.

1036

1037 The following describes the attributes and elements listed in the schema overview above:

1038 */wst:IssuedTokens*

1039 This header element carries one or more issued security tokens. This element schema is defined
1040 using the RequestSecurityTokenResponse schema type.

1041 */wst:IssuedTokens/wst:RequestSecurityTokenResponse*

1042 This element MUST appear at least once. Its meaning and semantics are as defined in Section 4.2.

1043 */wst:IssuedTokens/{any}*

1044 This is an extensibility mechanism to allow additional elements, based on schemas, to be added.

1045 */wst:IssuedTokens/@{any}*

1046 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added.

1047

1048 There MAY be multiple instances of the `<wst:IssuedTokens>` header in a given message. Such
1049 instances MAY be targeted at the same actor/role. Intermediaries MAY add additional
1050 `<wst:IssuedTokens>` header elements to a message. Intermediaries SHOULD NOT modify any
1051 `<wst:IssuedTokens>` header already present in a message.

1052

1053 It is RECOMMENDED that the `<wst:IssuedTokens>` header be signed to protect the integrity of the
1054 issued tokens and of the issuance itself. If confidentiality protection of the `<wst:IssuedTokens>` header is
1055 required then the entire header MUST be encrypted using the `<wsse11:EncryptedHeader>` construct.
1056 This helps facilitate re-issuance by the receiving party as that party can re-encrypt the entire header for
1057 another party rather than having to extract and re-encrypt portions of the header.

1058

1059 The following example illustrates a response that includes multiple `<wst:IssuedTokens>` headers.

```
1060 <?xml version="1.0" encoding="utf-8"?>
1061 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsp="..." xmlns:ds="..."
1062 xmlns:x="...">
1063    <S11:Header>
1064     <wst:IssuedTokens>
1065       <wst:RequestSecurityTokenResponse>
1066         <wsp:AppliesTo>
1067           <x:SomeContext1 />
1068         </wsp:AppliesTo>
1069         <wst:RequestedSecurityToken>
1070         ...
1071         </wst:RequestedSecurityToken>
1072         ...
1073       </wst:RequestSecurityTokenResponse>
1074       <wst:RequestSecurityTokenResponse>
1075         <wsp:AppliesTo>
1076           <x:SomeContext1 />
1077         </wsp:AppliesTo>
1078         <wst:RequestedSecurityToken>
1079         ...
1080         </wst:RequestedSecurityToken>
1081         ...
1082       </wst:RequestSecurityTokenResponse>
1083     </wst:IssuedTokens>
1084     <wst:IssuedTokens S11:role="http://example.org/somerole" >
1085       <wst:RequestSecurityTokenResponse>
1086         <wsp:AppliesTo>
1087             <x:SomeContext2 />
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
1088          </wsp:AppliesTo>
1089          <wst:RequestedSecurityToken>
1090          ...
1091          </wst:RequestedSecurityToken>
1092          ...
1093        </wst:RequestSecurityTokenResponse>
1094      </wst:IssuedTokens>
1095    </S11:Header>
1096    <S11:Body>
1097    ...
1098    </S11:Body>
1099  </S11:Envelope>
```


Copyright © OASIS Open 2006. All Rights Reserved.

5 Renewal Binding

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

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

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

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

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

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

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

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

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

The original proof information SHOULD be proven during renewal.

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

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

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

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


Copyright © OASIS Open 2006. All Rights Reserved.

```
1143            <wst:Renewing Allow=”...” OK=”...”/>
1144        </wst:RequestSecurityToken>
```

1145 */wst:RequestSecurityToken/wst:RenewTarget*

1146 This required element identifies the token being renewed. This MAY contain a
1147 `<wsse:SecurityTokenReference>` pointing at the token to be renewed or it MAY directly contain
1148 the token to be renewed.

1149 */wst:RequestSecurityToken/wst:AllowPostdating*

1150 This optional element indicates that returned tokens should allow requests for postdated tokens.
1151 That is, this allows for tokens to be issued that are not immediately valid (e.g., a token that can be
1152 used the next day).

1153 */wst:RequestSecurityToken/wst:Renewing*

1154 This optional element is used to specify renew semantics for types that support this operation.

1155 */wst:RequestSecurityToken/wst:Renewing/@Allow*

1156 This optional Boolean attribute is used to request a renewable token. If not specified, the default
1157 value is *true*. A renewable token is one whose lifetime can be extended. This is done using a
1158 renewal request. The recipient MAY allow renewals without demonstration of authorized use of
1159 the token or they MAY fault.

1160 */wst:RequestSecurityToken/wst:Renewing/@OK*

1161 This optional Boolean attribute is used to indicate that a renewable token is acceptable if the
1162 requested duration exceeds the limit of the issuance service. That is, if *true* then tokens can be
1163 renewed after their expiration. It should be noted that the token is NOT valid after expiration for
1164 any operation except renewal. The default for this attribute is *false*. It NOT RECOMMENDED to
1165 use this as it can leave you open to certain types of security attacks. Issuers MAY restrict the
1166 period after expiration during which time the token can be renewed. This window is governed by
1167 the issuer's policy.

1168 The following example illustrates a request for a custom token that can be renewed.

```
1169        <wst:RequestSecurityToken xmlns:wst="...”>
1170            <wst:TokenType>
1171                http://example.org/mySpecialToken
1172            </wst:TokenType>
1173            <wst:RequestType>
1174                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1175            </wst:RequestType>
1176            <wst:Renewing/>
1177        </wst:RequestSecurityToken>
```

1178

1179 The following example illustrates a subsequent renewal request and response (note that for brevity only
1180 the request and response are illustrated). Note that the response includes an indication of the lifetime of
1181 the renewed token.

```
1182        <wst:RequestSecurityToken xmlns:wst="...”>
1183            <wst:TokenType>
1184                http://example.org/mySpecialToken
1185            </wst:TokenType>
1186            <wst:RequestType>
1187                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Renew
1188            </wst:RequestType>
1189            <wst:RenewTarget>
1190                ... reference to previously issued token ...
1191            </wst:RenewTarget>
1192        </wst:RequestSecurityToken>
1193
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
1194    <wst:RequestSecurityTokenResponse xmlns:wst="...”>
1195        <wst:TokenType>
1196            http://example.org/mySpecialToken
1197        </wst:TokenType>
1198        <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1199        <wst:Lifetime>...</wst:Lifetime>
1200        ...
1201    </wst:RequestSecurityTokenResponse>
```


Copyright © OASIS Open 2006. All Rights Reserved.

1202 # 6 Cancel Binding

1203 Using the token request framework, this section defines bindings for requesting security tokens to be
1204 cancelled:

1205 **Cancel** – When a previously issued token is no longer needed, the Cancel binding can be used
1206 to cancel the token, terminating its use. After canceling a token at the issuer, a STS MUST not
1207 validate or renew the token. A STS MAY initiate the revocation of a token, however, revocation is
1208 out of scope of this specification and a client MUST NOT rely on it. If a client needs to ensure the
1209 validity of a token, it must validate the token at the issuer.

1210

1211 For this binding, the following actions are defined to enable specific processing context to be conveyed to
1212 the recipient:

```
1213  http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel
1214  http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Cancel
1215  http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal
```

1216 For this binding, the <wst:RequestType> element uses the following URI:

```
1217  http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
```

1218 Extensions MAY be specified in the request (and the response), but the semantics are not defined by this
1219 binding.

1220

1221 The request MUST prove authorized use of the token being cancelled unless the recipient trusts the
1222 requestor to make third-party cancel requests. In such cases, the third-party requestor MUST prove its
1223 identity to the issuer so that appropriate authorization occurs.

1224 In a cancel request, all key bearing tokens specified MUST have an associated signature. All non-key
1225 bearing tokens MUST be signed. Signature confirmation is RECOMMENDED on the closure response.

1226

1227 A cancelled token is no longer valid for authentication and authorization usages.

1228 On success a cancel response is returned. This is an RSTR message with the
1229 <wst:RequestedTokenCancelled> element in the body. On failure, a Fault is raised. It should be
1230 noted that the cancel RSTR is informational. That is, the security token is cancelled once the cancel
1231 request is processed.

1232

1233 The syntax of the request is as follows:

```
1234  <wst:RequestSecurityToken xmlns:wst="...”>
1235      <wst:RequestType>...</wst:RequestType>
1236      ...
1237      <wst:CancelTarget>...</wst:CancelTarget>
1238  </wst:RequestSecurityToken>
```

1239 */wst:RequestSecurityToken/wst:CancelTarget*

1240 This required element identifies the token being cancelled. Typically this contains a
1241 <wsse:SecurityTokenReference> pointing at the token, but it could also carry the token
1242 directly.

1243 The following example illustrates a request to cancel a custom token.

```
1244 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
```


Copyright © OASIS Open 2006. All Rights Reserved.

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

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

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

6.1 STS-initiated Cancel Binding

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

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

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

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

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

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

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

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

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

Copyright © OASIS Open 2006. All Rights Reserved.

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

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

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

The syntax of the request is as follows:

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

/wst:RequestSecurityToken/wst:CancelTarget

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

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

```
<?xml version="1.0" encoding="utf-8"?>
<S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
  <S11:Header>
    <wsse:Security>
      ...
    </wsse:Security>
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityToken>
        <wst:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/STSCancel
        </wst:RequestType>
        <wst:CancelTarget>
            ...
        </wst:CancelTarget>
    </wst:RequestSecurityToken>
  </S11:Body>
</S11:Envelope>
```


Copyright © OASIS Open 2006. All Rights Reserved.

1334 # 7 Validation Binding

1335 Using the token request framework, this section defines bindings for requesting security tokens to be
1336 validated:

1337 **Validate** – The validity of the specified security token is evaluated and a result is returned. The
1338 result may be a status, a new token, or both.

1339

1340 It should be noted that for this binding, a SOAP Envelope MAY be specified as a "security token" if the
1341 requestor desires the envelope to be validated. In such cases the recipient SHOULD understand how to
1342 process a SOAP envelope and adhere to SOAP processing semantics (e.g., mustUnderstand) of the
1343 version of SOAP used in the envelope. Otherwise, the recipient SHOULD fault.

1344 For this binding, the following actions are defined to enable specific processing context to be conveyed to
1345 the recipient:

```
1346     http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate
1347     http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Validate
1348    http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/ValidateFinal
```

1349

1350 For this binding, the `<wst:RequestType>` element contains the following URI:

```
1351     http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate
```

1352

1353 The request provides a token upon which the request is based and optional tokens. As well, the optional
1354 `<wst:TokenType>` element in the request can indicate desired type response token. This may be any
1355 supported token type or it may be the following URI indicating that only status is desired:

```
1356     http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Status
```

1357

1358 For some use cases a status token is returned indicating the success or failure of the validation. In other
1359 cases a security token MAY be returned and used for authorization. This binding assumes that the
1360 validation requestor and provider are known to each other and that the general issuance parameters
1361 beyond requesting a token type, which is optional, are not needed (note that other bindings and profiles
1362 could define different semantics).

1363

1364 For this binding an applicability scope (e.g., `<wsp:AppliesTo>`) need not be specified. It is assumed
1365 that the applicability of the validation response relates to the provided information (e.g. security token) as
1366 understood by the issuing service.

1367

1368 The validation binding does not allow the use of exchanges.

1369

1370 The RSTR for this binding carries the following element even if a token is returned (note that the base
1371 elements described above are included here italicized for completeness):

```
1372     <wst:RequestSecurityToken xmlns:wst="...”>
1373         <wst:TokenType>...</wst:TokenType>
1374         <wst:RequestType>...</wst:RequestType>
1375        <wst:ValidateTarget>... </wst:ValidateTarget>
```


Copyright © OASIS Open 2006. All Rights Reserved.

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

```
<wst:RequestSecurityTokenResponse xmlns:wst="..."  >
     <wst:TokenType>...</wst:TokenType>
     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
     ...
     <wst:Status>
        <wst:Code>...</wst:Code>
        <wst:Reason>...</wst:Reason>
     </wst:Status>
</wst:RequestSecurityTokenResponse>
```

/wst:RequestSecurityToken/wst:ValidateTarget

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

/wst:RequestSecurityTokenResponse/wst:Status

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

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

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

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

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

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

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

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

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


Copyright © OASIS Open 2006. All Rights Reserved.

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

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

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

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


Copyright © OASIS Open 2006. All Rights Reserved.

1448 # 8 Negotiation and Challenge Extensions

1449 The general security token service framework defined above allows for a simple request and response for
1450 security tokens (possibly asynchronous). However, there are many scenarios where a set of exchanges
1451 between the parties is required prior to returning (e.g., issuing) a security token. This section describes
1452 the extensions to the base WS-Trust mechanisms to enable exchanges for negotiation and challenges.

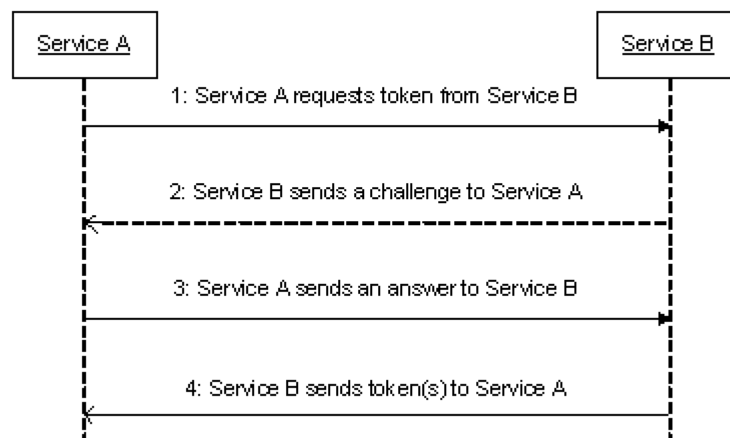
1453

1454 There are potentially different forms of exchanges, but one specific form, called "challenges", provides
1455 mechanisms in addition to those described in [WS-Security] for authentication. This section describes
1456 how general exchanges are issued and responded to within this framework. Other types of exchanges
1457 include, but are not limited to, negotiation, tunneling of hardware-based processing, and tunneling of
1458 legacy protocols.

1459

1460 The process is straightforward (illustrated here using a challenge):

1461



1462

1463 1. A requestor sends, for example, a `<wst:RequestSecurityToken>` message with a
1464 timestamp.

1465 2. The recipient does not trust the timestamp and issues a
1466 `<wst:RequestSecurityTokenResponse>` message with an embedded challenge.

1467 3. The requestor sends a `<wst:RequestSecurityTokenReponse>` message with an answer to
1468 the challenge.

1469 4. The recipient issues a `<wst:RequestSecurityTokenResponseCollection>` message with
1470 the issued security token and optional proof-of-possession token.

1471

1472 It should be noted that the requestor might challenge the recipient in either step 1 or step 3. In which
1473 case, step 2 or step 4 contains an answer to the initiator's challenge. Similarly, it is possible that steps 2
1474 and 3 could iterate multiple times before the process completes (step 4).

1475

1476 The two services can use [WS-SecurityPolicy] to state their requirements and preferences for security
1477 tokens and encryption and signing algorithms (general policy intersection). This section defines
1478 mechanisms for legacy and more sophisticated types of negotiations.

Copyright © OASIS Open 2006. All Rights Reserved.

8.1 Negotiation and Challenge Framework

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

The exchange model is as follows:

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

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

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

8.2 Signature Challenges

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

The syntax of these elements is as follows:

```
<wst:SignChallenge xmlns:wst="...”>
    <wst:Challenge ...>...</wst:Challenge>
</wst:SignChallenge>
```

```
<wst:SignChallengeResponse xmlns:wst="...”>
    <wst:Challenge ...>...</wst:Challenge>
</wst:SignChallengeResponse>
```


Copyright © OASIS Open 2006. All Rights Reserved.

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

.../wst:SignChallenge

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

.../wst:SignChallenge/wst:Challenge

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

.../SignChallenge/{any}

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

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

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

.../wst:SignChallengeResponse

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

.../wst:SignChallengeResponse/wst:Challenge

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

.../wst:SignChallengeResponse/{any}

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

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

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

8.3 Binary Exchanges and Negotiations

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

The syntax of this element is as follows:

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

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

.../wst:BinaryExchange

Copyright © OASIS Open 2006. All Rights Reserved.

1566 This optional element is used for a security negotiation that involves exchanging binary blobs as
1567 part of an existing negotiation protocol. The contents of this element are blob-type-specific and
1568 are encoded using base64 (unless otherwise specified).

1569 *.../wst:BinaryExchange/@ValueType*

1570 This required attribute specifies a URI to identify the type of negotiation (and the value space of
1571 the blob – the element's contents).

1572 *.../wst:BinaryExchange/@EncodingType*

1573 This required attribute specifies a URI to identify the encoding format (if different from base64) of
1574 the negotiation blob. Refer to [WS-Security] for sample encoding format URIs.

1575 *.../wst:BinaryExchange/@{any}*

1576 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added
1577 to the element.

1578 Some binary exchanges result in a shared state/context between the involved parties. It is
1579 RECOMMENDED that at the conclusion of the exchange, a new token and proof-of-possession token be
1580 returned. A common approach is to use the negotiated key as a "secure channel" mechanism to secure
1581 the new token and proof-of-possession token.

1582 For example, an exchange might establish a shared secret Sx that can then be used to sign the final
1583 response and encrypt the proof-of-possession token.

1584 8.4 Key Exchange Tokens

1585 In some cases it may be necessary to provide a key exchange token so that the other party (either
1586 requestor or issuer) can provide entropy or key material as part of the exchange. Challenges may not
1587 always provide a usable key as the signature may use a signing-only certificate.

1588

1589 The section describes two optional elements that can be included in RST and RSTR elements to indicate
1590 that a Key Exchange Token (KET) is desired, or to provide a KET.

1591 The syntax of these elements is as follows (Note that the ellipses below indicate that this element may be
1592 placed in different containers. For this specification, these are limited to
1593 `<wst:RequestSecurityToken>` and `<wst:RequestSecurityTokenResponse>`):

1594
```
<wst:RequestKET xmlns:wst="..." />
```

1595

1596
```
<wst:KeyExchangeToken xmlns:wst="...”>...</wst:KeyExchangeToken>
```

1597

1598 The following describes the attributes and tags listed in the schema above:

1599 *.../wst:RequestKET*

1600 This optional element is used to indicate that the receiving party (either the original requestor or
1601 issuer) should provide a KET to the other party on the next leg of the exchange.

1602 *.../wst:KeyExchangeToken*

1603 This optional element is used to provide a key exchange token. The contents of this element
1604 either contain the security token to be used for key exchange or a reference to it.

Copyright © OASIS Open 2006. All Rights Reserved.

1605 ## 8.5 Custom Exchanges

1606 Using the extensibility model described in this specification, any custom XML-based exchange can be
1607 defined in a separate binding/profile document. In such cases elements are defined which are carried in
1608 the RST and RSTR elements.

1609

1610 It should be noted that it is NOT REQUIRED that exchange elements be symmetric. That is, a specific
1611 exchange mechanism MAY use multiple elements at different times, depending on the state of the
1612 exchange.

1613 ## 8.6 Signature Challenge Example

1614 Here is an example exchange involving a signature challenge. In this example, a service requests a
1615 custom token using a X.509 certificate for authentication. The issuer uses the exchange mechanism to
1616 challenge the requestor to sign a random value (to ensure message freshness). The requestor provides
1617 a signature of the requested data and, once validated, the issuer then issues the requested token.

1618

1619 The first message illustrates the initial request that is signed with the private key associated with the
1620 requestor's X.509 certificate:

```
1621 <S11:Envelope xmlns:S11="..." xmlns:wsse="..."
1622         xmlns:wsu="..." xmlns:wst="...">
1623     <S11:Header>
1624         ...
1625         <wsse:Security>
1626             <wsse:BinarySecurityToken
1627                     wsu:Id="reqToken"
1628                     ValueType="...X509v3">
1629                 MIIEZzCCA9CgAwIBAgIQEmtJZc0...
1630             </wsse:BinarySecurityToken>
1631             <ds:Signature xmlns:ds="...">
1632                 ...
1633               <ds:KeyInfo>
1634                 <wsse:SecurityTokenReference>
1635                     <wsse:Reference URI="#reqToken"/>
1636                 </wsse:SecurityTokenReference>
1637               </ds:KeyInfo>
1638             </ds:Signature>
1639         </wsse:Security>
1640         ...
1641     </S11:Header>
1642     <S11:Body>
1643         <wst:RequestSecurityToken>
1644             <wst:TokenType>
1645                 http://example.org/mySpecialToken
1646             </wst:TokenType>
1647             <wst:RequestType>
1648                 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1649             </wst:RequestType>
1650         </wst:RequestSecurityToken>
1651     </S11:Body>
1652 </S11:Envelope>
```

1653

1654 The issuer (recipient) service doesn't trust the sender's timestamp (or one wasn't specified) and issues a
1655 challenge using the exchange framework defined in this specification. This message is signed using the
1656 private key associated with the issuer's X.509 certificate and contains a random challenge that the
1657 requestor must sign:

Copyright © OASIS Open 2006. All Rights Reserved.

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

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

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

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

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


Copyright © OASIS Open 2006. All Rights Reserved.

```
1717            ...
1718            <wsse:Security>
1719                <wsse:BinarySecurityToken
1720                        wsu:Id="issuerToken"
1721                        ValueType="...X509v3">
1722                    DFJHuedsujfnrnv45JZc0...
1723                </wsse:BinarySecurityToken>
1724                <ds:Signature xmlns:ds="...">
1725                    ...
1726                </ds:Signature>
1727            </wsse:Security>
1728            ...
1729        </S11:Header>
1730        <S11:Body>
1731          <wst:RequestSecurityTokenResponseCollection>
1732            <wst:RequestSecurityTokenResponse>
1733                <wst:RequestedSecurityToken>
1734                    <xyz:CustomToken xmlns:xyz="...">
1735                        ...
1736                    </xyz:CustomToken>
1737                </wst:RequestedSecurityToken>
1738                <wst:RequestedProofToken>
1739                    <xenc:EncryptedKey Id="newProof">
1740                        ...
1741                    </xenc:EncryptedKey>
1742                </wst:RequestedProofToken>
1743            </wst:RequestSecurityTokenResponse>
1744          </wst:RequestSecurityTokenResponseCollection>
1745        </S11:Body>
1746    </S11:Envelope>
```

1747 ## 8.7 Custom Exchange Example

1748 Here is another illustrating the syntax for a token request using a custom XML exchange. For brevity,
1749 only the RST and RSTR elements are illustrated. Note that the framework allows for an arbitrary number
1750 of exchanges, although this example illustrates the use of four legs. The request uses a custom
1751 exchange element and the requestor signs only the non-mutable element of the message:

```
1752        <wst:RequestSecurityToken xmlns:wst="...”>
1753            <wst:TokenType>
1754                http://example.org/mySpecialToken
1755            </wst:TokenType>
1756            <wst:RequestType>
1757                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1758            </wst:RequestType>
1759            <xyz:CustomExchange xmlns:xyz="...">
1760                ...
1761            </xyz:CustomExchange>
1762        </wst:RequestSecurityToken>
```

1763

1764 The issuer service (recipient) responds with another leg of the custom exchange and signs the response
1765 (non-mutable aspects) with its token:

```
1766        <wst:RequestSecurityTokenResponse xmlns:wst="...”>
1767            <xyz:CustomExchange xmlns:xyz="...">
1768                ...
1769            </xyz:CustomExchange>
1770        </wst:RequestSecurityTokenResponse>
```

1771

Copyright © OASIS Open 2006. All Rights Reserved.

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

```
    <wst:RequestSecurityTokenResponse xmlns:wst="...”>
        <xyz:CustomExchange xmlns:xyz="...">
            ...
        </xyz:CustomExchange>
    </wst:RequestSecurityTokenResponse>
```

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

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

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

8.8 Protecting Exchanges

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

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

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

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

Copyright © OASIS Open 2006. All Rights Reserved.

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

1816 ## 8.9 Authenticating Exchanges

1817 After an exchange both parties have a shared knowledge of a key (or keys) that can then be used to
1818 secure messages. However, in some cases it may be desired to have the issuer prove to the requestor
1819 that it knows the key (and that the returned metadata is valid) prior to the requestor using the data.
1820 However, until the exchange is actually completed it may (and is often) inappropriate to use the computed
1821 keys. As well, using a token that hasn't been returned to secure a message may complicate processing
1822 since it crosses the boundary of the exchange and the underlying message security. This means that it
1823 may not be appropriate to sign the final leg of the exchange using the key derived from the exchange.

1824

1825 For this reason an authenticator is defined that provides a way for the issuer to verify the hash as part of
1826 the token issuance. Specifically, when an authenticator is returned, the
1827 `<wst:RequestSecurityTokenResponseCollection>` element is returned. This contains one
1828 RSTR with the token being returned as a result of the exchange and a second RSTR that contains the
1829 authenticator (this order SHOULD be used). When an authenticator is used, RSTRs MUST use the
1830 *@Context* element so that the authenticator can be correlated to the token issuance. The authenticator is
1831 separated from the RSTR because otherwise computation of the RST/RSTR hash becomes more
1832 complex. The authenticator is represented using the `<wst:Authenticator>` element as illustrated
1833 below:

```
1834     <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
1835         <wst:RequestSecurityTokenResponse Context="...">
1836             ...
1837         </wst:RequestSecurityTokenResponse>
1838         <wst:RequestSecurityTokenResponse Context="...">
1839             <wst:Authenticator>
1840                 <wst:CombinedHash>...</wst:CombinedHash>
1841                 ...
1842             </wst:Authenticator>
1843         </wst:RequestSecurityTokenResponse>
1844     </wst:RequestSecurityTokenResponseCollection>
```

1845

1846 The following describes the attributes and elements listed in the schema overview above (the ... notation
1847 below represents the path RSTRC/RSTR and is used for brevity):

1848 *.../wst:Authenticator*

1849 This optional element provides verification (authentication) of a computed hash.

1850 *.../wst:Authenticator/wst:CombinedHash*

1851 This optional element proves the hash and knowledge of the computed key. This is done by
1852 providing the base64 encoding of the first 256 bits of the P_SHA1 digest of the computed key and
1853 the concatenation of the hash determined for the computed key and the string "AUTH-HASH".
1854 Specifically, P_SHA1(*computed-key*, H + "AUTH-HASH")$_{0\text{-}255}$. The octets for the "AUTH-HASH"
1855 string are the UTF-8 octets.

1856

1857 This `<wst:CombinedHash>` element is optional (and an open content model is used) to allow for
1858 different authenticators in the future.

Copyright © OASIS Open 2006. All Rights Reserved.

9 Key and Token Parameter Extensions

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

9.1 On-Behalf-Of Parameters

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

```
<wst:RequestSecurityToken xmlns:wst="...”>
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>...</wst:RequestType>
    ...
    <wst:OnBehalfOf>...</wst:OnBehalfOf>
    <wst:Issuer>...</wst:Issuer>
</wst:RequestSecurityToken>
```

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

/wst:RequestSecurityToken/wst:OnBehalfOf

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

/wst:RequestSecurityToken/wst:Issuer

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

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

```
<wst:RequestSecurityToken xmlns:wst="...”>
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>...</wst:RequestType>
    ...
    <wst:OnBehalfOf>endpoint-reference</wst:OnBehalfOf>
</wst:RequestSecurityToken>
```

9.2 Key and Encryption Requirements

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

Copyright © OASIS Open 2006. All Rights Reserved.

1902 values must be adhered to and faults generated for invalid inputs, or if the issuer will provide alterative
1903 values in the response.

1904

1905 Although illustrated using the `<wst:RequestSecurityToken>` element, these options can also be
1906 returned in a `<wst:RequestSecurityTokenResponse>` element.

1907 The syntax for these optional elements is as follows (note that the base elements described above are
1908 included here italicized for completeness):

```
1909    <wst:RequestSecurityToken xmlns:wst="...”>
1910        <wst:TokenType>...</wst:TokenType>
1911        <wst:RequestType>...</wst:RequestType>
1912        ...
1913        <wst:AuthenticationType>...</wst:AuthenticationType>
1914        <wst:KeyType>...</wst:KeyType>
1915        <wst:KeySize>...</wst:KeySize>
1916        <wst:SignatureAlgorithm>...</wst:SignatureAlgorithm>
1917        <wst:EncryptionAlgorithm>...</wst:EncryptionAlgorithm>
1918        <wst:CanonicalizationAlgorithm>...</wst:CanonicalizationAlgorithm>
1919        <wst:ComputedKeyAlgorithm>...</wst:ComputedKeyAlgorithm>
1920        <wst:Encryption>...</wst:Encryption>
1921        <wst:ProofEncryption>...</wst:ProofEncryption>
1922        <wst:UseKey Sig="..."> </wst:UseKey>
1923        <wst:SignWith>...</wst:SignWith>
1924        <wst:EncryptWith>...</wst:EncryptWith>
1925    </wst:RequestSecurityToken>
```

1926

1927 The following describes the attributes and elements listed in the schema overview above:

1928 */wst:RequestSecurityToken/wst:AuthenticationType*

1929 This optional URI element indicates the type of authentication desired, specified as a URI. This
1930 specification does not predefine classifications; these are specific to token services as is the
1931 relative strength evaluations. The relative assessment of strength is up to the recipient to
1932 determine. That is, requestors should be familiar with the recipient policies. For example, this
1933 might be used to indicate which of the four U.S. government authentication levels is required.

1934 */wst:RequestSecurityToken/wst:KeyType*

1935 This optional URI element indicates the type of key desired in the security token. The predefined
1936 values are identified in the table below. Note that some security token formats have fixed key
1937 types. It should be noted that new algorithms can be inserted by defining URIs in other
1938 specifications and profiles.

URI	Meaning
http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey	A public key token is requested
http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey	A symmetric key token is requested (default)
http://docs.oasis-open.org/ws-sx/wstrust/200512/Bearer	A bearer token is requested. This key type can be used by requestors to indicate that they want a security token to be issued that does not require proof of possession.

1939 */wst:RequestSecurityToken/wst:KeySize*

1940 This optional integer element indicates the size of the key required specified in number of bits.
1941 This is a request, and, as such, the requested security token is not obligated to use the requested

Copyright © OASIS Open 2006. All Rights Reserved.

1942 key size. That said, the recipient SHOULD try to use a key at least as strong as the specified
1943 value if possible. The information is provided as an indication of the desired strength of the
1944 security.

1945 */wst:RequestSecurityToken/wst:SignatureAlgorithm*

1946 This optional URI element indicates the desired signature algorithm used within the returned
1947 token. This is specified as a URI indicating the algorithm (see [XML-Signature] for typical signing
1948 algorithms).

1949 */wst:RequestSecurityToken/wst:EncryptionAlgorithm*

1950 This optional URI element indicates the desired encryption algorithm used within the returned
1951 token. This is specified as a URI indicating the algorithm (see [XML-Encrypt] for typical
1952 encryption algorithms).

1953 */wst:RequestSecurityToken/wst:CanonicalizationAlgorithm*

1954 This optional URI element indicates the desired canonicalization method used within the returned
1955 token. This is specified as a URI indicating the method (see [XML-Signature] for typical
1956 canonicalization methods).

1957 */wst:RequestSecurityToken/wst:ComputedKeyAlgorithm*

1958 This optional URI element indicates the desired algorithm to use when computed keys are used
1959 for issued tokens.

1960 */wst:RequestSecurityToken/wst:Encryption*

1961 This optional element indicates that the requestor desires any returned secrets in issued security
1962 tokens to be encrypted for the specified token. That is, so that the owner of the specified token
1963 can decrypt the secret. Normally the security token is the contents of this element but a security
1964 token reference MAY be used instead. If this element isn't specified, the token used as the basis
1965 of the request (or specialized knowledge) is used to determine how to encrypt the key.

1966 */wst:RequestSecurityToken/wst:ProofEncryption*

1967 This optional element indicates that the requestor desires any returned secrets in proof-of-
1968 possession tokens to be encrypted for the specified token. That is, so that the owner of the
1969 specified token can decrypt the secret. Normally the security token is the contents of this element
1970 but a security token reference MAY be used instead. If this element isn't specified, the token
1971 used as the basis of the request (or specialized knowledge) is used to determine how to encrypt
1972 the key.

1973 */wst:RequestSecurityToken/wst:UseKey*

1974 If the requestor wishes to use an existing key rather than create a new one, then this optional
1975 element can be used to reference the security token containing the desired key. This element
1976 either contains a security token or a `<wsse:SecurityTokenReference>` element that
1977 references the security token containing the key that should be used in the returned token. If
1978 `<wst:KeyType>` is not defined and a key type is not implicitly known to the service, it MAY be
1979 determined from the token (if possible). Otherwise this parameter is meaningless and is ignored.
1980 Requestors SHOULD demonstrate authorized use of the public key provided.

1981 */wst:RequestSecurityToken/wst:KeyWrapAlgorithm*

1982 This optional URI element indicates the desired algorithm to use for key wrapping when STS
1983 encrypts the issued token for the relying party using an asymmetric key.

1984 */wst:RequestSecurityToken/wst:UseKey/@Sig*

1985 In order to *authenticate* the key referenced, a signature MAY be used to prove the referenced
1986 token/key. If specified, this optional attribute indicates the ID of the corresponding signature (by
1987 URI reference). When this attribute is present, a key need not be specified inside the element
1988 since the referenced signature will indicate the corresponding token (and key).

Copyright © OASIS Open 2006. All Rights Reserved.

1989 */wst:RequestSecurityToken/wst:SignWith*

1990 This optional URI element indicates the desired signature algorithm to be used with the issued
1991 security token (typically from the policy of the target site for which the token is being requested.
1992 While any of these optional elements MAY be included in RSTRs, this one is a likely candidate if
1993 there is some doubt (e.g., an X.509 cert that can only use DSS).

1994 */wst:RequestSecurityToken/wst:EncryptWith*

1995 This optional URI element indicates the desired encryption algorithm to be used with the issued
1996 security token (typically from the policy of the target site for which the token is being requested.)
1997 While any of these optional elements MAY be included in RSTRs, this one is a likely candidate if
1998 there is some doubt.

1999 The following summarizes the various algorithm parameters defined above. T is the issued token, P is the
2000 proof key.
2001

2002 **SignatureAlgorithm** - The signature algorithm to use to sign T

2003 **EncryptionAlgorithm** - The encryption algorithm to use to encrypt T

2004 **CanonicalizationAlgorithm** - The canonicalization algorithm to use when signing T

2005 **ComputedKeyAlgorithm** - The key derivation algorithm to use if using a symmetric key for P
2006 where P is computed using client, server, or combined entropy.

2007 **Encryption** - The token/key to use when encrypting T

2008 **ProofEncryption** - The token/key to use when encrypting P

2009 **UseKey** - This is P. This is generally used when the client supplies a public-key that it wishes to
2010 be embedded in T as the proof key.

2011 **SignWith** - The signature algorithm the client intends to employ when using P to

2012 sign.

2013 The encryption algorithms further differ based on whether the issued token contains asymmetric key or
2014 symmetric key. Furthermore, they differ based on what type of key is used to protect the issued token
2015 from the STS to the relying party. The following cases can occur:

2016 T contains symmetric key/STS uses symmetric key to encrypt T for RP

2017 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP
2018 when using the proof key (e.g. AES256).

2019 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS should use to
2020 encrypt the T (e.g. AES256)

2021

2022 T contains symmetric key/STS uses asymmetric key to encrypt T for RP

2023 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP
2024 when using the proof key (e.g. AES256)

2025 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS should use to
2026 encrypt T for RP (e.g. AES256)

2027 **KeyWrapAlgorithm** – used to indicate the KeyWrap algorithm that the STS should use to wrap
2028 the generated key that is used to encrypt the T for RP.

2029

2030 T contains asymmetric key/STS uses symmetric key to encrypt T for RP

2031 **EncryptWith** – used to indicate the KeyWrap algorithm that the client will use to

2032 protect symmetric key that is used to protect message to RP when using the proof

2033 key (e.g. RSA-OAEP-MGF1P)

Copyright © OASIS Open 2006. All Rights Reserved.

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

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

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

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

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

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

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

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:wst="..." xmlns:ds="..." xmlns:xenc="...">
    <S11:Header>
        ...
        <wsse:Security>
            <xenc:ReferenceList>...</xenc:ReferenceList>
            <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>
            <wsse:BinarySecurityToken wsu:Id="requestEncryptionToken"
                    ValueType="...SomeTokenType" xmlns:x="...">
                MIIEZzCCA9CgAwIBAgIQEmtJZc0...
            </wsse:BinarySecurityToken>
            <wsse:BinarySecurityToken wsu:Id="requestProofToken"
                    ValueType="...SomeTokenType" xmlns:x="...">
                MIIEZzCCA9CgAwIBAgIQEmtJZc0...
           </wsse:BinarySecurityToken>
           <ds:Signature Id="proofSignature">
               ... signature proving requested key ...
               ... key info points to the "requestedProofToken" token ...
           </ds:Signature>
       </wsse:Security>
       ...
    </S11:Header>
    <S11:Body wsu:Id="req">
        <wst:RequestSecurityToken>
            <wst:TokenType>
                http://example.org/mySpecialToken
            </wst:TokenType>
            <wst:RequestType>
                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
            </wst:RequestType>
           <wst:KeyType>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
2088                http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey
2089            </wst:KeyType>
2090            <wst:KeySize>1024</wst:KeySize>
2091            <wst:SignatureAlgorithm>
2092                http://www.w3.org/2000/09/xmldsig#rsa-sha1
2093            </wst:SignatureAlgorithm>
2094            <wst:Encryption>
2095                <Reference URI="#requestEncryptionToken"/>
2096            </wst:Encryption>
2097            <wst:ProofEncryption>
2098                <wsse:Reference URI="#requestProofToken"/>
2099            </wst:ProofEncryption>
2100            <wst:UseKey Sig="#proofSignature"/>
2101        </wst:RequestSecurityToken>
2102    </S11:Body>
2103 </S11:Envelope>
```

2104 9.3 Delegation and Forwarding Requirements

2105 This section defines extensions to the `<wst:RequestSecurityToken>` element for indicating
2106 delegation and forwarding requirements on the requested security token(s).

2107 The syntax for these extension elements is as follows (note that the base elements described above are
2108 included here italicized for completeness):

```
2109    <wst:RequestSecurityToken xmlns:wst="...">
2110        <wst:TokenType>...</wst:TokenType>
2111        <wst:RequestType>...</wst:RequestType>
2112        ...
2113        <wst:DelegateTo>...</wst:DelegateTo>
2114        <wst:Forwardable>...</wst:Forwardable>
2115        <wst:Delegatable>...</wst:Delegatable>
2116    </wst:RequestSecurityToken>
```

2117 */wst:RequestSecurityToken/wst:DelegateTo*

2118 This optional element indicates that the requested or issued token be delegated to another
2119 identity. The identity receiving the delegation is specified by placing a security token or
2120 `<wsse:SecurityTokenReference>` element within the `<wst:DelegateTo>` element.

2121 */wst:RequestSecurityToken/wst:Forwardable*

2122 This optional element, of type xs:boolean, specifies whether the requested security token should
2123 be marked as "Forwardable". In general, this flag is used when a token is normally bound to the
2124 requestor's machine or service. Using this flag, the returned token MAY be used from any source
2125 machine so long as the key is correctly proven. The default value of this flag is true.

2126 */wst:RequestSecurityToken/wst:Delegatable*

2127 This optional element, of type xs:boolean, specifies whether the requested security token should
2128 be marked as "Delegatable". Using this flag, the returned token MAY be delegated to another
2129 party. This parameter SHOULD be used in conjunction with `<wst:DelegateTo>`. The default
2130 value of this flag is false.

2131

2132 The following illustrates the syntax of a request for a custom token that can be delegated to the indicated
2133 recipient (specified in the binary security token) and used in the specified interval.

```
2134        <wst:RequestSecurityToken xmlns:wst="...">
2135        <wst:TokenType>
2136            http://example.org/mySpecialToken
2137        </wst:TokenType>
2138        <wst:RequestType>
```


Copyright © OASIS Open 2006. All Rights Reserved.

```
2139             http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
2140         </wst:RequestType>
2141         <wst:DelegateTo>
2142             <wsse:BinarySecurityToken
2143 xmlns:wsse="...">...</wsse:BinarySecurityToken>
2144         </wst:DelegateTo>
2145         <wst:Delegatable>true</wst:Delegatable>
2146     </wst:RequestSecurityToken>
```

2147 9.4 Policies

2148 This section defines extensions to the `<wst:RequestSecurityToken>` element for passing policies.

2149

2150 The syntax for these extension elements is as follows (note that the base elements described above are
2151 included here italicized for completeness):

```
2152     <wst:RequestSecurityToken xmlns:wst="...">
2153         <wst:TokenType>...</wst:TokenType>
2154         <wst:RequestType>...</wst:RequestType>
2155         ...
2156         <wsp:Policy xmlns:wsp="...">...</wsp:Policy>
2157         <wsp:PolicyReference xmlns:wsp="...">...</wsp:PolicyReference>
2158     </wst:RequestSecurityToken>
```

2159

2160 The following describes the attributes and elements listed in the schema overview above:

2161 */wst:RequestSecurityToken/wsp:Policy*

2162 This optional element specifies a policy (as defined in [WS-Policy]) that indicates desired settings
2163 for the requested token. The policy specifies defaults that can be overridden by the elements
2164 defined in the previous sections.

2165 */wst:RequestSecurityToken/wsp:PolicyReference*

2166 This optional element specifies a reference to a policy (as defined in [WS-Policy]) that indicates
2167 desired settings for the requested token. The policy specifies defaults that can be overridden by
2168 the elements defined in the previous sections.

2169

2170 The following illustrates the syntax of a request for a custom token that provides a set of policy
2171 statements about the token or its usage requirements.

```
2172     <wst:RequestSecurityToken xmlns:wst="...">
2173         <wst:TokenType>
2174             http://example.org/mySpecialToken
2175         </wst:TokenType>
2176         <wst:RequestType>
2177             http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
2178         </wst:RequestType>
2179         <wsp:Policy xmlns:wsp="...">
2180             ...
2181         </wsp:Policy>
2182     </wst:RequestSecurityToken>
```

2183 9.5 Authorized Token Participants

2184 This section defines extensions to the `<wst:RequestSecurityToken>` element for passing information
2185 about which parties are authorized to participate in the use of the token. This parameter is typically used

Copyright © OASIS Open 2006. All Rights Reserved.

2186 when there are additional parties using the token or if the requestor needs to clarify the actual parties
2187 involved (for some profile-specific reason).

2188 It should be noted that additional participants will need to prove their identity to recipients in addition to
2189 proving their authorization to use the returned token. This typically takes the form of a second signature
2190 or use of transport security.

2191

2192 The syntax for these extension elements is as follows (note that the base elements described above are
2193 included here italicized for completeness):

```
2194     <wst:RequestSecurityToken xmlns:wst="...">
2195         <wst:TokenType>...</wst:TokenType>
2196         <wst:RequestType>...</wst:RequestType>
2197         ...
2198         <wst:Participants>
2199             <wst:Primary>...</wst:Primary>
2200             <wst:Participant>...</wst:Participant>
2201         </wst:Participants>
2202     </wst:RequestSecurityToken>
```

2203

2204 The following describes elements and attributes used in a `<wsc:SecurityContextToken>` element.

2205 */wst:RequestSecurityToken/wst:Participants/*

2206 This optional element specifies the participants sharing the security token. Arbitrary types may be
2207 used to specify participants, but a typical case is a security token or an endpoint reference (see
2208 [WS-Addressing]).

2209 */wst:RequestSecurityToken/wst:Participants/wst:Primary*

2210 This optional element specifies the primary user of the token (if one exists).

2211 */wst:RequestSecurityToken/wst:Participants/wst:Participant*

2212 This optional element specifies participant (or multiple participants by repeating the element) that
2213 play a (profile-dependent) role in the use of the token or who are allowed to use the token.

2214 */wst:RequestSecurityToken/wst:Participants/{any}*

2215 This is an extensibility option to allow other types of participants and profile-specific elements to
2216 be specified.

Copyright © OASIS Open 2006. All Rights Reserved.

10 Key Exchange Token Binding

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

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

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

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

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

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

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

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

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

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

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

```
<wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
  <wst:RequestSecurityTokenResponse>
      <wst:TokenType>...</wst:TokenType>
      <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
      ...
  </wst:RequestSecurityTokenResponse>
</wst:RequestSecurityTokenResponseCollection>
```

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

Copyright © OASIS Open 2006. All Rights Reserved.

2261 request could be made using transport security (e.g. TLS) and the key could be returned directly using
2262 `<wst:BinarySecret>`.

```
2263     <wst:RequestSecurityToken xmlns:wst="...”>
2264         <wst:RequestType>
2265             http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET
2266         </wst:RequestType>
2267     </wst:RequestSecurityToken>
```

2268

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


Copyright © OASIS Open 2006. All Rights Reserved.

2276 # 11 Error Handling

2277 There are many circumstances where an *error* can occur while processing security information. Errors
2278 use the SOAP Fault mechanism. Note that the reason text provided below is RECOMMENDED, but
2279 alternative text MAY be provided if more descriptive or preferred by the implementation. The tables
2280 below are defined in terms of SOAP 1.1. For SOAP 1.2, the Fault/Code/Value is env:Sender (as defined
2281 in SOAP 1.2) and the Fault/Code/Subcode/Value is the *faultcode* below and the Fault/Reason/Text is the
2282 *faultstring* below. It should be noted that profiles MAY provide second-level detail fields, but they should
2283 be careful not to introduce security vulnerabilities when doing so (e.g., by providing too detailed
2284 information).

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

Copyright © OASIS Open 2006. All Rights Reserved.

12 Security Considerations

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

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

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

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

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

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

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

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

For some forms of multi-message exchanges, the exchanges are susceptible to attacks whereby signatures are altered. To address this, it is suggested that a signature confirmation mechanism be used. In such cases, each leg should include the confirmation of the previous leg. That is, leg 2 includes confirmation for leg 1, leg 3 for leg 2, leg 4 for leg 3, and so on. In doing so, each side can confirm the correctness of the message outside of the message body.

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

Copyright © OASIS Open 2006. All Rights Reserved.

2328

2329 It should be noted that use of unsolicited RSTRs implies that the recipient is prepared to accept such
2330 issuances. Recipients should ensure that such issuances are properly authorized and recognize their
2331 use could be used in denial-of-service attacks.

2332 In addition to the consideration identified here, readers should also review the security considerations in
2333 [WS-Security].

2334

2335 Both token cancellation bindings defined in this specification require that the STS MUST NOT validate or
2336 renew the token after it has been successfully canceled. The STS must take care to ensure that the token
2337 is properly invalidated before confirming the cancel request or sending the cancel notification to the client.
2338 This can be more difficult if the token validation or renewal logic is physically separated from the issuance
2339 and cancellation logic. It is out of scope of this spec how the STS propagates the token cancellation to its
2340 other components. If STS cannot ensure that the token was properly invalidated it MUST NOT send the
2341 cancel notification or confirm the cancel request to the client.

Copyright © OASIS Open 2006. All Rights Reserved.

A. Key Exchange

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

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

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

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

A.1 Ephemeral Encryption Keys

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

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

```
<xenc:EncryptedKey xmlns:xenc="...">
    ...
    <ds:KeyInfo xmlns:ds="...">
        <wsse:SecurityTokenReference xmlns:wsse="...">
            <ds:X509IssuerSerial>
                <ds:X509IssuerName>
                    DC=ACMECorp, DC=com
                </ds:X509IssuerName>
            <ds:X509SerialNumber>12345678</ds:X509SerialNumber>
            </ds:X509IssuerSerial>
        </wsse:SecurityTokenReference>
    </ds:KeyInfo>
    ...
</xenc:EncryptedKey>
```

A.2 Requestor-Provided Keys

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

Copyright © OASIS Open 2006. All Rights Reserved.

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

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

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

A.3 Issuer-Provided Keys

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

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

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

A.4 Composite Keys

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

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

Copyright © OASIS Open 2006. All Rights Reserved.

```
2435 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
2436   <wst:RequestSecurityTokenResponse>
2437       <wst:RequestedSecurityToken>
2438           <xyz:CustomToken xmlns:xyz="...">
2439               ...
2440           </xyz:CustomToken>
2441       </wst:RequestedSecurityToken>
2442       <wst:Entropy>
2443           <wst:BinarySecret>UIH...</wst:BinarySecret>
2444       </wst:Entropy>
2445   </wst:RequestSecurityTokenResponse>
2446 </wst:RequestSecurityTokenResponseCollection>
```

2447 A.5 Key Transfer and Distribution

2448 There are also a few mechanisms where existing keys are transferred to other parties.

2449 A.5.1 Direct Key Transfer

2450 If one party has a token and key and wishes to share this with another party, the key can be directly
2451 transferred. This is accomplished by sending an RSTR (either in the body or header) to the other party.
2452 The RSTR contains the token and a proof-of-possession token that contains the key encrypted for the
2453 recipient.

2454

2455 In the following example a custom token and its associated proof-of-possession token are known to party
2456 A who wishes to share them with party B. In this example, A is a member in a secure on-line chat
2457 session and is inviting B to join the conversation. After authenticating B, A sends B an RSTR. The RSTR
2458 contains the token and the key is communicated as a proof-of-possession token that is encrypted for B:

```
2459 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
2460   <wst:RequestSecurityTokenResponse>
2461       <wst:RequestedSecurityToken>
2462           <xyz:CustomToken xmlns:xyz="...">
2463               ...
2464           </xyz:CustomToken>
2465       </wst:RequestedSecurityToken>
2466       <wst:RequestedProofToken>
2467           <xenc:EncryptedKey xmlns:xenc="..."  Id="newProof">
2468               ...
2469           </xenc:EncryptedKey>
2470       </wst:RequestedProofToken>
2471   </wst:RequestSecurityTokenResponse>
2472 </wst:RequestSecurityTokenResponseCollection>
```

2473 A.5.2 Brokered Key Distribution

2474 A third party may also act as a broker to transfer keys. For example, a requestor may obtain a token and
2475 proof-of-possession token from a third-party STS. The token contains a key encrypted for the target
2476 service (either using the service's public key or a key known to the STS and target service). The proof-of-
2477 possession token contains the same key encrypted for the requestor (similarly this can use public or
2478 symmetric keys).

2479

2480 In the following example a custom token and its associated proof-of-possession token are returned from a
2481 broker B to a requestor R for access to service S. The key for the session is contained within the custom
2482 token encrypted for S using either a secret known by B and S or using S's public key. The same secret is
2483 encrypted for R and returned as the proof-of-possession token:

Copyright © OASIS Open 2006. All Rights Reserved.

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

A.5.3 Delegated Key Transfer

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

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

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


Copyright © OASIS Open 2006. All Rights Reserved.

A.5.4 Authenticated Request/Reply Key Transfer

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

Specifically, the sender wishes the following:

Transfer a key to a recipient that they can use to secure a reply

Ensure that only the recipient can see the key

Provide proof that the sender issued the key

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

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

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

1. Sign the appropriate message parts using a random key (or ideally a key derived from a random key)
2. Encrypt the appropriate message parts using the random key (or ideally another key derived from the random key)
3. Encrypt the random key for the recipient
4. Sign just the encrypted key

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

```
       <wsse:Security xmlns:wsse="..." xmlns:wsu="..."
            xmlns:ds="..." xmlns:xenc="...">
        <wsse:BinarySecurityToken wsu:Id="myToken">
            ...
        </wsse:BinarySecurityToken>
        <ds:Signature>
            ...signature over #secret using token #myToken...
        </ds:Signature>
        <xenc:EncryptedKey Id="secret">
            ...
        </xenc:EncryptedKey>
        <xenc:RefrenceList>
            ...manifest of encrypted parts using token #secret...
        </xenc:RefrenceList>
        <ds:Signature>
            ...signature over key message parts using token #secret...
        </ds:Signature>
    </wsse:Security>
```

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

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


Copyright © OASIS Open 2006. All Rights Reserved.

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

A.6 Perfect Forward Secrecy

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

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

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

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

Copyright © OASIS Open 2006. All Rights Reserved.

B. WSDL

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

```
<?xml version="1.0"?>
<wsdl:definitions
        targetNamespace="http://docs.oasis-open.org/ws-sx/ws-
trust/200512/wsdl"
        xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/wsdl"
        xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
<!-- this is the WS-I BP-compliant way to import a schema -->
    <wsdl:types>
        <xs:schema>
            <xs:import
      namespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
      schemaLocation="http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-
trust.xsd"/>
        </xs:schema>
    </wsdl:types>

<!-- WS-Trust defines the following GEDs -->
    <wsdl:message name="RequestSecurityTokenMsg">
        <wsdl:part name="request" element="wst:RequestSecurityToken" />
    </wsdl:message>
    <wsdl:message name="RequestSecurityTokenResponseMsg">
        <wsdl:part name="response"
                element="wst:RequestSecurityTokenResponse" />
    </wsdl:message>
    <wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
        <wsdl:part name="responseCollection"
                element="wst:RequestSecurityTokenResponseCollection"/>
    </wsdl:message>

<!-- This portType models the full request/response the Security Token
Service: -->

    <wsdl:portType name="WSSecurityRequestor">
        <wsdl:operation name="SecurityTokenResponse">
            <wsdl:input
                    message="tns:RequestSecurityTokenResponseMsg"/>
        </wsdl:operation>
        <wsdl:operation name="SecurityTokenResponse2">
            <wsdl:input
                message="tns:RequestSecurityTokenResponseCollectionMsg"/>
        </wsdl:operation>
        <wsdl:operation name="Challenge">
            <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
            <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
        </wsdl:operation>
        <wsdl:operation name="Challenge2">
            <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
            <wsdl:output
                message="tns:RequestSecurityTokenResponseCollectionMsg"/>
        </wsdl:operation>
    </wsdl:portType>

<!-- These portTypes model the individual message exchanges -->
```


Copyright © OASIS Open 2006. All Rights Reserved.

2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703

```
    <wsdl:portType name="SecurityTokenRequestService">
        <wsdl:operation name="RequestSecurityToken">
            <wsdl:input message="tns:RequestSecurityTokenMsg"/>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:portType name="SecurityTokenService">
        <wsdl:operation name="RequestSecurityToken">
            <wsdl:input message="tns:RequestSecurityTokenMsg"/>
            <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
        </wsdl:operation>
        <wsdl:operation name="RequestSecurityToken2">
            <wsdl:input message="tns:RequestSecurityTokenMsg"/>
            <wsdl:output
                 message="tns:RequestSecurityTokenResponseCollectionMsg"/>
        </wsdl:operation>
    </wsdl:portType>
</wsdl:definitions>
```


Copyright © OASIS Open 2006. All Rights Reserved.

C. Acknowledgements

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

Original Authors of the initial contribution:
Steve Anderson, OpenNetwork
Jeff Bohren, OpenNetwork
Toufic Boubez, Layer 7
Marc Chanliau, Computer Associates
Giovanni Della-Libera, Microsoft
Brendan Dixon, Microsoft
Praerit Garg, Microsoft
Martin Gudgin (Editor), Microsoft
Phillip Hallam-Baker, VeriSign
Maryann Hondo, IBM
Chris Kaler, Microsoft
Hal Lockhart, BEA
Robin Martherus, Oblix
Hiroshi Maruyama, IBM
Anthony Nadalin (Editor), IBM
Nataraj Nagaratnam, IBM
Andrew Nash, Reactivity
Rob Philpott, RSA Security
Darren Platt, Ping Identity
Hemma Prafullchandra, VeriSign
Maneesh Sahu, Actional
John Shewchuk, Microsoft
Dan Simon, Microsoft
Davanum Srinivas, Computer Associates
Elliot Waingold, Microsoft
David Waite, Ping Identity
Doug Walter, Microsoft
Riaz Zolfonoon, RSA Security

Original Acknowledgments of the initial contribution:
Paula Austel, IBM
Keith Ballinger, Microsoft
Bob Blakley, IBM
John Brezak, Microsoft
Tony Cowan, IBM
Cédric Fournet, Microsoft
Vijay Gajjala, Microsoft
HongMei Ge, Microsoft
Satoshi Hada, IBM
Heather Hinton, IBM
Slava Kavsan, RSA Security
Scott Konersmann, Microsoft
Leo Laferriere, Computer Associates
Paul Leach, Microsoft
Richard Levinson, Computer Associates
John Linn, RSA Security
Michael McIntosh, IBM
Steve Millet, Microsoft

Copyright © OASIS Open 2006. All Rights Reserved.

2757 Birgit Pfitzmann, IBM
2758 Fumiko Satoh, IBM
2759 Keith Stobie, Microsoft
2760 T.R. Vishwanath, Microsoft
2761 Richard Ward, Microsoft
2762 Hervey Wilson, Microsoft

2763

2764 **TC Members during the development of this specification:**
2765 Don Adams, Tibco Software Inc.
2766 Jan Alexander, Microsoft Corporation
2767 Steve Anderson, BMC Software
2768 Donal Arundel, IONA Technologies
2769 Howard Bae, Oracle Corporation
2770 Abbie Barbir, Nortel Networks Limited
2771 Charlton Barreto, Adobe Systems
2772 Mighael Botha, Software AG, Inc.
2773 Toufic Boubez, Layer 7 Technologies Inc.
2774 Norman Brickman, Mitre Corporation
2775 Melissa Brumfield, Booz Allen Hamilton
2776 Lloyd Burch, Novell
2777 Scott Cantor, Internet2
2778 Greg Carpenter, Microsoft Corporation
2779 Steve Carter, Novell
2780 Ching-Yun (C.Y.) Chao, IBM
2781 Martin Chapman, Oracle Corporation
2782 Kate Cherry, Lockheed Martin
2783 Henry (Hyenvui) Chung, IBM
2784 Luc Clement, Systinet Corp.
2785 Paul Cotton, Microsoft Corporation
2786 Glen Daniels, Sonic Software Corp.
2787 Peter Davis, Neustar, Inc.
2788 Martijn de Boer, SAP AG
2789 Werner Dittmann, Siemens AG
2790 Abdeslem DJAOUI, Associate Member
2791 Fred Dushin, IONA Technologies
2792 Petr Dvorak, Systinet Corp.
2793 Colleen Evans, Microsoft Corporation
2794 Ruchith Fernando, WSO2
2795 Mark Fussell, Microsoft Corporation
2796 Vijay Gajjala, Microsoft Corporation
2797 Marc Goodner, Microsoft Corporation
2798 Hans Granqvist, VeriSign
2799 Martin Gudgin, Microsoft Corporation
2800 Tony Gullotta, SOA Software Inc.
2801 Jiandong Guo, Sun Microsystems
2802 Phillip Hallam-Baker, VeriSign
2803 Patrick Harding, Ping Identity Corporation
2804 Heather Hinton, IBM
2805 Frederick Hirsch, Nokia Corporation
2806 Jeff Hodges, Neustar, Inc.
2807 Will Hopkins, BEA Systems, Inc.
2808 Alex Hristov, Otecia Incorporated
2809 John Hughes, Associate Member
2810 Diane Jordan, IBM
2811 Venugopal K, Sun Microsystems
2812 Chris Kaler, Microsoft Corporation

Copyright © OASIS Open 2006. All Rights Reserved.

2813 Dana Kaufman, Forum Systems, Inc.
2814 Paul Knight, Nortel Networks Limited
2815 Ramanathan Krishnamurthy, IONA Technologies
2816 Christopher Kurt, Microsoft Corporation
2817 Kelvin Lawrence, IBM
2818 Hubert Le Van Gong, Sun Microsystems
2819 Jong Lee, BEA Systems, Inc.
2820 Rich Levinson, Oracle Corporation
2821 Tommy Lindberg, Associate Member
2822 Mark Little, JBoss Inc.
2823 Hal Lockhart, BEA Systems, Inc.
2824 Mike Lyons, Layer 7 Technologies Inc.
2825 Eve Maler, Sun Microsystems
2826 Ashok Malhotra, Oracle Corporation
2827 Anand Mani, CrimsonLogic Pte Ltd
2828 Jonathan Marsh, Microsoft Corporation
2829 Robin Martherus, Oracle Corporation
2830 Miko Matsumura, Infravio, Inc.
2831 Gary McAfee, IBM
2832 Michael McIntosh, IBM
2833 John Merrells, Sxip Networks SRL
2834 Jeff Mischkinsky, Oracle Corporation
2835 Prateek Mishra, Oracle Corporation
2836 Bob Morgan, Internet2
2837 Vamsi Motukuru, Oracle Corporation
2838 Raajmohan Na, EDS
2839 Anthony Nadalin, IBM
2840 Andrew Nash, Reactivity, Inc.
2841 Eric Newcomer, IONA Technologies
2842 Duane Nickull, Adobe Systems
2843 Toshihiro Nishimura, Fujitsu Limited
2844 Rob Philpott, RSA Security
2845 Denis Pilipchuk, BEA Systems, Inc.
2846 Darren Platt, Ping Identity Corporation
2847 Martin Raepple, SAP AG
2848 Nick Ragouzis, Associate Member
2849 Prakash Reddy, CA
2850 Alain Regnier, Ricoh Company, Ltd.
2851 Irving Reid, Hewlett-Packard
2852 Bruce Rich, IBM
2853 Tom Rutt, Fujitsu Limited
2854 Maneesh Sahu, Actional Corporation
2855 Frank Siebenlist, Argonne National Laboratory
2856 Joe Smith, Apani Networks
2857 Davanum Srinivas, WSO2
2858 Yakov Sverdlov, CA
2859 Gene Thurston, AmberPoint
2860 Victor Valle, IBM
2861 Asir Vedamuthu, Microsoft Corporation
2862 Greg Whitehead, Hewlett-Packard
2863 Ron Williams, IBM
2864 Corinna Witt, BEA Systems, Inc.
2865 Kyle Young, Microsoft Corporation

Copyright © OASIS Open 2006. All Rights Reserved.