WS-Trust 1.4

OASIS Committee Specification 01

29 November 2008

Specification URIs:
This Version:
http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cs-01.doc (Authoritative)
http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cs-01.pdf
http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cs-01.html

Previous Version:
http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cd-03.doc (Authoritative)
http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cd-03.pdf
http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/cd/ws-trust-1.4-spec-cd-03.html

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

Technical Committee:
OASIS Web Services Secure Exchange TC

Chair(s):
Kelvin Lawrence, IBM
Chris Kaler, Microsoft

Editor(s):
Anthony Nadalin, IBM
Marc Goodner, Microsoft
Martin Gudgin, Microsoft
Abbie Barbir, Nortel
Hans Granqvist, VeriSign

Related work:
N/A

Declared XML namespace(s):
http://docs.oasis-open.org/ws-sx/ws-trust/200512
http://docs.oasis-open.org/ws-sx/ws-trust/200802

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

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

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

Copyright © OASIS® 1993–2008. All Rights Reserved.

“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.

Copyright © OASIS® 1993–2008. All Rights Reserved.

Notices

Copyright © OASIS® 1993–2008. 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.

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

Table of Contents

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

Copyright © OASIS® 1993–2008. All Rights Reserved.

8.3 User Interaction Challenge 44
8.3.1 Challenge Format 45
8.3.2 PIN and OTP Challenges 48
8.4 Binary Exchanges and Negotiations 49
8.5 Key Exchange Tokens 49
8.6 Custom Exchanges 50
8.7 Signature Challenge Example 50
8.8 Challenge Examples 52
8.8.1 Text and choice challenge 52
8.8.2 PIN challenge 54
8.8.3 PIN challenge with optimized response 56
8.9 Custom Exchange Example 57
8.10 Protecting Exchanges 58
8.11 Authenticating Exchanges 58
9 Key and Token Parameter Extensions 60
9.1 On-Behalf-Of Parameters 60
9.2 Key and Encryption Requirements 60
9.3 Delegation and Forwarding Requirements 65
9.4 Policies 66
9.5 Authorized Token Participants 67
10 Key Exchange Token Binding 68
11 Error Handling 70
12 Security Considerations 71
13 Conformance 73
A. Key Exchange 74
A.1 Ephemeral Encryption Keys 74
A.2 Requestor-Provided Keys 74
A.3 Issuer-Provided Keys 75
A.4 Composite Keys 75
A.5 Key Transfer and Distribution 76
A.5.1 Direct Key Transfer 76
A.5.2 Brokered Key Distribution 76
A.5.3 Delegated Key Transfer 77
A.5.4 Authenticated Request/Reply Key Transfer 78
A.6 Perfect Forward Secrecy 79
B. WSDL 80
C. Acknowledgements 82

Copyright © OASIS® 1993–2008. All Rights Reserved.

1 Introduction

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

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

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

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

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

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

Section 12 is non-normative.

1.1 Goals and Non-Goals

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

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

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

The following are explicit non-goals for this document:

- Password authentication
- Token revocation
- Management of trust policies

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

- Establishing a security context token

Copyright © OASIS® 1993–2008. All Rights Reserved.

- Key derivation

1.2 Requirements

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

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

1.3 Namespace

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

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

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

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

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

Prefix	Namespace	Specification(s)
S11	http://schemas.xmlsoap.org/soap/envelope/	[SOAP]
S12	http://www.w3.org/2003/05/soap-envelope	[SOAP12]
wsu	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd	[WS-Security]
wsse	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd	[WS-Security]
wsse11	http://docs.oasis-open.org/wss/oasis-wss-wsecurity-secext-1.1.xsd	[WS-Security]
wst	http://docs.oasis-open.org/ws-sx/ws-trust/200512	This specification
wst14	http://docs.oasis-open.org/ws-sx/ws-trust/200802	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 or http://www.w3.org/ns/ws-policy	[WS-Policy]

Copyright © OASIS® 1993–2008. All Rights Reserved.

wsa	http://www.w3.org/2005/08/addressing	[WS-Addressing]
xs	http://www.w3.org/2001/XMLSchema	[XML-Schema1] [XML-Schema2]

1.4 Schema and WSDL Files

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

```
http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust.xsd
http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.xsd
```

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

```
http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.wsdl
```

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

1.5 Terminology

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

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

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

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

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

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

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

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

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

Copyright © OASIS® 1993–2008. All Rights Reserved.

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

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

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

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

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

1.5.1 Notational Conventions

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

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

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

- The syntax appears as an XML instance, but values in italics indicate data types instead of literal values.
- Characters are appended to elements and attributes to indicate cardinality:
    - "?" (0 or 1)
    - "*" (0 or more)
    - "+" (1 or more)
- The character "|" is used to indicate a choice between alternatives.
- The characters "(" and ")" are used to indicate that contained items are to be treated as a group with respect to cardinality or choice.
- The characters "[" and "]" are used to call out references and property names.
- Ellipses (i.e., "...") indicate points of extensibility. Additional children and/or attributes MAY be added at the indicated extension points but MUST NOT contradict the semantics of the parent and/or owner, respectively. By default, if a receiver does not recognize an extension, the receiver SHOULD ignore the extension; exceptions to this processing rule, if any, are clearly indicated below.
- XML namespace prefixes (see Table 1) are used to indicate the namespace of the element being defined.

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

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

Copyright © OASIS® 1993–2008. All Rights Reserved.

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

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

1.6 Normative References

[RFC2119] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, Harvard University, March 1997.
http://www.ietf.org/rfc/rfc2119.txt

[RFC2246] IETF Standard, "The TLS Protocol", January 1999.
http://www.ietf.org/rfc/rfc2246.txt

[SOAP] W3C Note, "SOAP: Simple Object Access Protocol 1.1", 08 May 2000.
http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[SOAP12] W3C Recommendation, "SOAP 1.2 Part 1: Messaging Framework", 24 June 2003.
http://www.w3.org/TR/2003/REC-soap12-part1-20030624/

[URI] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 3986, MIT/LCS, Day Software, Adobe Systems, January 2005.
http://www.ietf.org/rfc/rfc3986.txt

[WS-Addressing] W3C Recommendation, "Web Services Addressing (WS-Addressing)", 9 May 2006.
http://www.w3.org/TR/2006/REC-ws-addr-core-20060509

[WS-Policy] W3C Recommendation, "Web Services Policy 1.5 - Framework", 04 September 2007.
http://www.w3.org/TR/2007/REC-ws-policy-20070904/
W3C Member Submission, "Web Services Policy 1.2 - Framework", 25 April 2006.
http://www.w3.org/Submission/2006/SUBM-WS-Policy-20060425/

[WS-PolicyAttachment] W3C Recommendation, "Web Services Policy 1.5 - Attachment", 04 September 2007.
http://www.w3.org/TR/2007/REC-ws-policy-attach-20070904/
W3C Member Submission, "Web Services Policy 1.2 - Attachment", 25 April 2006.
http://www.w3.org/Submission/2006/SUBM-WS-PolicyAttachment-20060425/

[WS-Security] OASIS Standard, "OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)", March 2004.
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf
OASIS Standard, "OASIS Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)", February 2006.
http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

Copyright © OASIS® 1993–2008. All Rights Reserved.

[XML-C14N] W3C Recommendation, "Canonical XML Version 1.0", 15 March 2001.
http://www.w3.org/TR/2001/REC-xml-c14n-20010315
W3C Recommendation, "Canonical XML Version 1.1", 2 May 2008.
http://www.w3.org/TR/2008/REC-xml-c14n11-20080502/

[XML-Encrypt] W3C Recommendation, "XML Encryption Syntax and Processing", 10 December 2002.
http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/

[XML-Schema1] W3C Recommendation, "XML Schema Part 1: Structures Second Edition", 28 October 2004.
http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[XML-Schema2] W3C Recommendation, "XML Schema Part 2: Datatypes Second Edition", 28 October 2004.
http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[XML-Signature] W3C Recommendation, "XML-Signature Syntax and Processing", 12 February 2002.
http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/
[W3C Recommendation, D. Eastlake et al. XML Signature Syntax and Processing (Second Edition). 10 June 2008.
http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/

1.7 Non-Normative References

[Kerberos] J. Kohl and C. Neuman, "The Kerberos Network 149 Authentication Service (V5)," RFC 1510, September 1993.
http://www.ietf.org/rfc/rfc1510.txt

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

[WS-SecurityPolicy] OASIS Committee Specification 01, "WS-SecurityPolicy 1.3", November 2008
http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/cd/ws-securitypolicy-1.3-spec-cs-01.doc

[X509] S. Santesson, et al,"Internet X.509 Public Key Infrastructure Qualified Certificates Profile."
http://www.itu.int/rec/recommendation.asp?type=items&lang=e&parent=T-REC-X.509-200003-I

Copyright © OASIS® 1993–2008. All Rights Reserved.

2 Web Services Trust Model

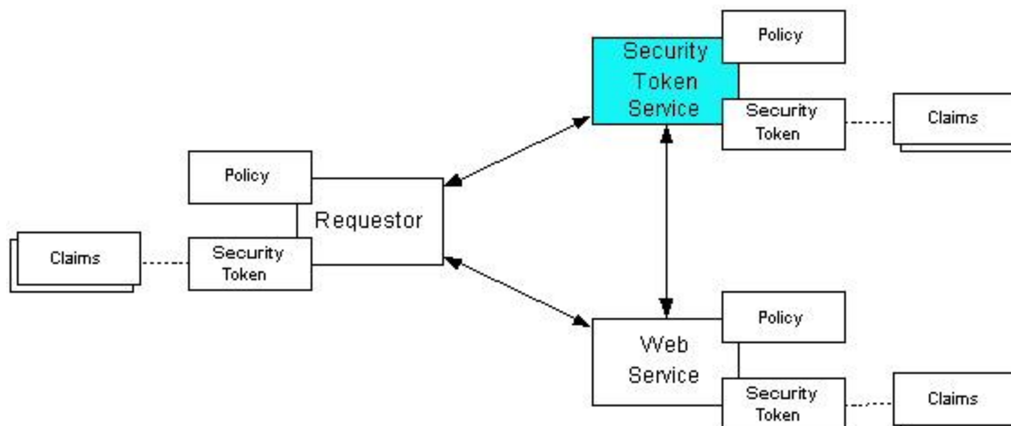
The Web service security model defined in WS-Trust is based on a process in which a Web service can require that an incoming message prove a set of claims (e.g., name, key, permission, capability, etc.). If a message arrives without having the required proof of claims, the service SHOULD ignore or reject 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® 1993–2008. 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® 1993–2008. All Rights Reserved.

296 or by another party on the requestor's behalf. If the security token service trusts the provided security
297 token (for example, because it trusts the issuing authority of the provided security token), and the request
298 can prove possession of that security token, then the exchange is processed by the security token
299 service.
300
301 The previous paragraph illustrates an example of token acquisition in a direct trust relationship. In the
302 case of a delegated request (one in which another party provides the request on behalf of the requestor
303 rather than the requestor presenting it themselves), the security token service generating the new token
304 MAY NOT need to trust the authority that issued the original token provided by the original requestor
305 since it does trust the security token service that is engaging in the exchange for a new security token.
306 The basis of the trust is the relationship between the two security token services.

307 2.3 Out-of-Band Token Acquisition

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

313 2.4 Trust Bootstrap

314 An administrator or other trusted authority MAY designate that all tokens of a certain type are trusted (e.g.
315 all Kerberos tokens from a specific realm or all X.509 tokens from a specific CA). The security token
316 service maintains this as a trust axiom and can communicate this to trust engines to make their own trust
317 decisions (or revoke it later), or the security token service MAY provide this function as a service to
318 trusting services.
319 There are several different mechanisms that can be used to bootstrap trust for a service. These
320 mechanisms are non-normative and are NOT REQUIRED in any way. That is, services are free to
321 bootstrap trust and establish trust among a domain of services or extend this trust to other domains using
322 any mechanism.
323
324 **Fixed trust roots** – The simplest mechanism is where the recipient has a fixed set of trust relationships.
325 It will then evaluate all requests to determine if they contain security tokens from one of the trusted roots.
326
327 **Trust hierarchies** – Building on the trust roots mechanism, a service MAY choose to allow hierarchies of
328 trust so long as the trust chain eventually leads to one of the known trust roots. In some cases the
329 recipient MAY require the sender to provide the full hierarchy. In other cases, the recipient MAY be able
330 to dynamically fetch the tokens for the hierarchy from a token store.
331
332 **Authentication service** – Another approach is to use an authentication service. This can essentially be
333 thought of as a fixed trust root where the recipient only trusts the authentication service. Consequently,
334 the recipient forwards tokens to the authentication service, which replies with an authoritative statement
335 (perhaps a separate token or a signed document) attesting to the authentication.

Copyright © OASIS® 1993–2008. 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.

3.1 Requesting a Security Token

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

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

The syntax for this element is as follows:

Copyright © OASIS® 1993–2008. All Rights Reserved.

```
378     <wst:RequestSecurityToken Context="..." xmlns:wst="...">
379         <wst:TokenType>...</wst:TokenType>
380         <wst:RequestType>...</wst:RequestType>
381         <wst:SecondaryParameters>...</wst:SecondaryParameters>
382         ...
383     </wst:RequestSecurityToken>
```

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

385 */wst:RequestSecurityToken*

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

387 */wst:RequestSecurityToken/@Context*

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

392 */wst:RequestSecurityToken/wst:TokenType*

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

397 */wst:RequestSecurityToken/wst:RequestType*

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

404 */wst:RequestSecurityToken/wst:SecondaryParameters*

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

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

412 */wst:RequestSecurityToken/{any}*

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

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

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

420 3.2 Returning a Security Token

421 The `<wst:RequestSecurityTokenResponse>` element (RSTR) is used to return a security token or
422 response to a security token request. The `<wst:RequestSecurityTokenResponseCollection>`
423 element (RSTRC) MUST be used to return a security token or response to a security token request on the
424 final response.

Copyright © OASIS® 1993–2008. All Rights Reserved.

425
426 It should be noted that any type of parameter specified as input to a token request MAY be present on
427 response in order to specify the exact parameters used by the issuer. Specific bindings describe
428 appropriate restrictions on the contents of the RST and RSTR elements.

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

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

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

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

448 The syntax for this element is as follows:

```
449     <wst:RequestSecurityTokenResponse Context="..." xmlns:wst="...">
450         <wst:TokenType>...</wst:TokenType>
451         <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
452         ...
453     </wst:RequestSecurityTokenResponse>
```

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

455 */wst:RequestSecurityTokenResponse*

456 This is the response to a security token request.

457 */wst:RequestSecurityTokenResponse/@Context*

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

463 */wst:RequestSecurityTokenResponse/wst:TokenType*

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

465 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

466 This OPTIONAL element is used to return the requested security token. Normally the requested
467 security token is the contents of this element but a security token reference MAY be used instead.
468 For example, if the requested security token is used in securing the message, then the security
469 token is placed into the `<wsse:Security>` header (as described in [WS-Security]) and a
470 `<wsse:SecurityTokenReference>` element is placed inside of the
471 `<wst:RequestedSecurityToken>` element to reference the token in the `<wsse:Security>`
472 header. The response MAY contain a token reference where the token is located at a URI

Copyright © OASIS® 1993–2008. All Rights Reserved.

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

477 */wst:RequestSecurityTokenResponse/{any}*

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

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

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

483 3.3 Binary Secrets

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

491 *.../wst:BinarySecret*

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

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

495 This OPTIONAL attribute indicates the type of secret being encoded. The pre-defined values are
496 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)

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

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

500 3.4 Composition

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

Copyright © OASIS® 1993–2008. 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® 1993–2008. All Rights Reserved.

```
549             <wsu:Created>...</wsu:Created>
550             <wsu:Expires>...</wsu:Expires>
551         </wst:Lifetime>
552     </wst:RequestSecurityToken>
```

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

554 */wst:RequestSecurityToken/wst:TokenType*

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

563 */wst:RequestSecurityToken/wsp:AppliesTo*

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

571 */wst:RequestSecurityToken/wst:Claims*

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

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

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

579 */wst:RequestSecurityToken/wst:Entropy*

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

584 */wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret*

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

591 */wst:RequestSecurityToken/wst:Lifetime*

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

Copyright © OASIS® 1993–2008. All Rights Reserved.

598 */wst:RequestSecurityToken/wst:Lifetime/wsu:Created*

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

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

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

611

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

```
618 <S11:Envelope xmlns:S11="..." xmlns:wsu="..." xmlns:wsse="..."
619         xmlns:xenc="..." xmlns:wst="...">
620     <S11:Header>
621         ...
622         <wsse:Security>
623             <xenc:ReferenceList>...</xenc:ReferenceList>
624             <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>
625             <ds:Signature xmlns:ds="...">
626                 ...
627              <ds:KeyInfo>
628                 <wsse:SecurityTokenReference>
629                     <wsse:Reference URI="#myToken"/>
630                 </wsse:SecurityTokenReference>
631              </ds:KeyInfo>
632             </ds:Signature>
633         </wsse:Security>
634         ...
635     </S11:Header>
636     <S11:Body wsu:Id="req">
637         <wst:RequestSecurityToken>
638             <wst:TokenType>
639                 http://example.org/mySpecialToken
640             </wst:TokenType>
641             <wst:RequestType>
642                 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
643             </wst:RequestType>
644         </wst:RequestSecurityToken>
645     </S11:Body>
646 </S11:Envelope>
```

647 4.2 Request Security Token Collection

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

Copyright © OASIS® 1993–2008. 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>
        <wst:Claims xmlns:wsp="...">
          http://manufacturer.example.com/certification
        </wst: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® 1993–2008. All Rights Reserved.

705 ### 4.2.1 Processing Rules

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

708

709 1. The single `RequestSecurityTokenResponseCollection` response MUST contain at least
710 one RSTR element corresponding to each RST element in the request. A RSTR element
711 corresponds to an RST element if it has the same Context attribute value as the RST element.
712 **Note:** Each request MAY generate more than one RSTR sharing the same Context attribute
713 value
714 a. Specifically there is no notion of a deferred response
715 b. If any RST request results in an error, then no RSTRs will be returned and a SOAP Fault
716 will be generated as the entire response.
717 2. Every RST in the request MUST use an action URI value in the RequestType element that is a
718 batch version corresponding to the non-batch version, in particular one of the following:
719 - `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchIssue`
720 - `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchValidate`
721 - `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchRenew`
722 - `http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchCancel`

723

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

726

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

729

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

737

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

740

741 ## 4.3 Returning a Security Token Collection

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

746 The syntax for this element is as follows:

Copyright © OASIS® 1993–2008. All Rights Reserved.

```
747    <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
748      <wst:RequestSecurityTokenResponse>...</wst:RequestSecurityTokenResponse> +
749    </wst:RequestSecurityTokenResponseCollection>
```

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

751 */wst:RequestSecurityTokenResponseCollection*

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

754 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

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

756 4.4 Returning a Security Token

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

```
760      <wst:RequestSecurityTokenResponse xmlns:wst="...">
761          <wst:TokenType>...</wst:TokenType>
762          <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
763          ...
764          <wsp:AppliesTo xmlns:wsp="...”>...</wsp:AppliesTo>
765          <wst:RequestedAttachedReference>
766          ...
767          </wst:RequestedAttachedReference>
768          <wst:RequestedUnattachedReference>
769          ...
770         </wst:RequestedUnattachedReference>
771          <wst:RequestedProofToken>...</wst:RequestedProofToken>
772          <wst:Entropy>
773              <wst:BinarySecret>...</wst:BinarySecret>
774          </wst:Entropy>
775          <wst:Lifetime>...</wst:Lifetime>
776      </wst:RequestSecurityTokenResponse>
```

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

778 */wst:RequestSecurityTokenResponse/wsp:AppliesTo*

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

783 */wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken*

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

789 */wst:RequestSecurityTokenResponse/wst:RequestedAttachedReference*

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

Copyright © OASIS® 1993–2008. All Rights Reserved.

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

800 */wst:RequestSecurityTokenResponse/wst:RequestedUnattachedReference*

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

807 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken*

808 This OPTIONAL element is used to return the proof-of-possession token associated with the
809 requested security token. Normally the proof-of-possession token is the contents of this element
810 but a security token reference MAY be used instead. The token (or reference) is specified as the
811 contents of this element. For example, if the proof-of-possession token is used as part of the
812 securing of the message, then it is placed in the `<wsse:Security>` header and a
813 `<wsse:SecurityTokenReference>` element is used inside of the
814 `<wst:RequestedProofToken>` element to reference the token in the `<wsse:Security>`
815 header. This element is OPTIONAL, but it is REQUIRED that at least one of
816 `<wst:RequestedSecurityToken>` or `<wst:RequestedProofToken>` be returned unless
817 there is an error.

818 */wst:RequestSecurityTokenResponse/wst:Entropy*

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

823 */wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret*

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

826 */wst:RequestSecurityTokenResponse/wst:Lifetime*

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

830 4.4.1 wsp:AppliesTo in RST and RSTR

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

835 - If neither the requestor nor the issuer specifies a scope then the scope of the issued token is
836 implied.
837 - If the requestor specifies a scope and the issuer does not then the scope of the token is assumed
838 to be that specified by the requestor.
839 - If the requestor does not specify a scope and the issuer does specify a scope then the scope of
840 the token is as defined by the issuers scope
841 - If both requestor and issuer specify a scope then there are two possible outcomes:
842   - If both the issuer and requestor specify the same scope then the issued token has that
843 scope.

Copyright © OASIS® 1993–2008. All Rights Reserved.

844 - If the issuer specifies a wider scope than the requestor then the issued token has the
845 scope specified by the issuer.

846 - The requestor and issuer MUST agree on the version of [WS-Policy] used to specify the scope of
847 the issued token. The Trust13 assertion in [WS-SecurityPolicy] provides a mechanism to
848 communicate which version of [WS-Policy] is to be used.

849

850 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.

851 ## 4.4.2 Requested References

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

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

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

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

867 ## 4.4.3 Keys and Entropy

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

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

872 - In the case of partial, the `<wst:Entropy>` element is included in the response, which indicates
873 partial key material from the issuer (not the full key) that is combined (by each party) with the
874 requestor's entropy to determine the resulting key(s). In this case a `<wst:ComputedKey>`

Copyright © OASIS® 1993–2008. All Rights Reserved.

875 element is returned inside the <wst:RequestedProofToken> to indicate how the key is
876 computed.

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

879

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

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

890

891 The following table illustrates the rules described above:

Requestor	**Issuer**	**Results**
Provide Entropy	Uses requestor entropy as key	No proof-of-possession token is returned.
	Provides entropy	No keys returned, key(s) derived using entropy from both sides according to method identified in response
	Issues own key (rejects requestor's entropy)	Proof-of-possession token contains issuer's key(s)
No Entropy provided	Issues own key	Proof-of-possession token contains issuer's key(s)
	Does not issue key	No proof-of-possession token

892 4.4.4 Returning Computed Keys

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

```
898 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...">
899   <wst:RequestSecurityTokenResponse>
900       <wst:RequestedProofToken>
901           <wst:ComputedKey>...</wst:ComputedKey>
902       </wst:RequestedProofToken>
903   </wst:RequestSecurityTokenResponse>
904 </wst:RequestSecurityTokenResponseCollection>
```

905

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

Copyright © OASIS® 1993–2008. All Rights Reserved.

907 */wst:RequestSecurityTokenResponse/wst:RequestedProofToken/wst:ComputedKey*

908 The value of this element is a URI describing how to compute the key. While this can be
909 extended by defining new URIs in other bindings and profiles, the following URI pre-defines one
910 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.

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

912 4.4.5 Sample Response with Encrypted Secret

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

```
917     <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
918       <wst:RequestSecurityTokenResponse>
919           <wst:RequestedSecurityToken>
920               <xyz:CustomToken xmlns:xyz="...">
921                   ...
922               </xyz:CustomToken>
923           </wst:RequestedSecurityToken>
924           <wst:RequestedProofToken>
925               <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
926                   ...
927               </xenc:EncryptedKey>
928           </wst:RequestedProofToken>
929       </wst:RequestSecurityTokenResponse>
930     </wst:RequestSecurityTokenResponseCollection>
```

931 4.4.6 Sample Response with Unencrypted Secret

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

```
934     <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
935       <wst:RequestSecurityTokenResponse>
936           <wst:RequestedSecurityToken>
937               <xyz:CustomToken xmlns:xyz="...">
938                   ...
939               </xyz:CustomToken>
940           </wst:RequestedSecurityToken>
941           <wst:RequestedProofToken>
942               <wst:BinarySecret>...</wst:BinarySecret>
943           </wst:RequestedProofToken>
944       </wst:RequestSecurityTokenResponse>
945     </wst:RequestSecurityTokenResponseCollection>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

4.4.7 Sample Response with Token Reference

If the returned token doesn't allow the use of the *wsu:Id* attribute, then a `<wst:RequestedAttachedReference>` 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:RequestedAttachedReference>
          <wsse:SecurityTokenReference xmlns:wsse="...”>
             <wsse:Reference URI="urn:fabrikam123:5445"/>
          </wsse:SecurityTokenReference>
      </wst:RequestedAttachedReference>
      ...
  </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.

4.4.8 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>
```

4.4.9 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>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

```
996              <xyz:CustomToken xmlns:xyz="...">
997                  ...
998              </xyz:CustomToken>
999          </wst:RequestedSecurityToken>
1000         <wst:RequestedProofToken>
1001             <xenc:EncryptedKey Id="newProof" xmlns:xenc="...”>
1002                 ...
1003             </xenc:EncryptedKey>
1004         </wst:RequestedProofToken>
1005     </wst:RequestSecurityTokenResponse>
1006   </wst:RequestSecurityTokenResponseCollection>
```

1007
4.4.10 More Than One Proof-of-Possession Tokens Case

1008 The second case is where multiple security tokens are returned that have separate proof-of-possession
1009 tokens. As a result, the proof-of-possession tokens, and possibly lifetime and other key parameters
1010 elements, MAY be different. To address this scenario, the body MAY be specified using the syntax
1011 illustrated below:

```
1012     <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
1013         <wst:RequestSecurityTokenResponse>
1014             ...
1015         </wst:RequestSecurityTokenResponse>
1016         <wst:RequestSecurityTokenResponse>
1017             ...
1018         </wst:RequestSecurityTokenResponse>
1019         ...
1020     </wst:RequestSecurityTokenResponseCollection>
```

1021 The following describes the attributes and elements listed in the schema overview above:

1022 */wst:RequestSecurityTokenResponseCollection*

1023 This element is used to provide multiple RSTR responses, each of which has separate key
1024 information. One or more RSTR elements are returned in the collection. This MUST always be
1025 used on the final response to the RST.

1026 */wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse*

1027 Each RequestSecurityTokenResponse element is an individual RSTR.

1028 */wst:RequestSecurityTokenResponseCollection/{any}*

1029 This is an extensibility mechanism to allow additional elements, based on schemas, to be added.

1030 */wst:RequestSecurityTokenResponseCollection/@{any}*

1031 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added.

1032 The following illustrates the syntax of a response that includes multiple tokens each, in a separate RSTR,
1033 each with their own proof-of-possession token.

```
1034     <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
1035         <wst:RequestSecurityTokenResponse>
1036             <wst:RequestedSecurityToken>
1037                 <xyz:CustomToken xmlns:xyz="...">
1038                     ...
1039                 </xyz:CustomToken>
1040             </wst:RequestedSecurityToken>
1041             <wst:RequestedProofToken>
1042                 <xenc:EncryptedKey Id="newProofA">
1043                     ...
1044                 </xenc:EncryptedKey>
1045             </wst:RequestedProofToken>
1046         </wst:RequestSecurityTokenResponse>
1047         <wst:RequestSecurityTokenResponse>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

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

4.5 Returning Security Tokens in Headers

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

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

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

/wst:IssuedTokens

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

/wst:IssuedTokens/wst:RequestSecurityTokenResponse

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

/wst:IssuedTokens/{any}

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

/wst:IssuedTokens/@{any}

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

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

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

Copyright © OASIS® 1993–2008. All Rights Reserved.

1095 The following example illustrates a response that includes multiple `<wst:IssuedTokens>` headers.

```
1096 <?xml version="1.0" encoding="utf-8"?>
1097 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsp="..." xmlns:ds="..."
1098 xmlns:x="...">
1099    <S11:Header>
1100     <wst:IssuedTokens>
1101       <wst:RequestSecurityTokenResponse>
1102         <wsp:AppliesTo>
1103           <x:SomeContext1 />
1104         </wsp:AppliesTo>
1105         <wst:RequestedSecurityToken>
1106         ...
1107         </wst:RequestedSecurityToken>
1108         ...
1109       </wst:RequestSecurityTokenResponse>
1110       <wst:RequestSecurityTokenResponse>
1111         <wsp:AppliesTo>
1112           <x:SomeContext1 />
1113         </wsp:AppliesTo>
1114         <wst:RequestedSecurityToken>
1115         ...
1116         </wst:RequestedSecurityToken>
1117         ...
1118       </wst:RequestSecurityTokenResponse>
1119     </wst:IssuedTokens>
1120     <wst:IssuedTokens S11:role="http://example.org/somerole" >
1121       <wst:RequestSecurityTokenResponse>
1122         <wsp:AppliesTo>
1123             <x:SomeContext2 />
1124         </wsp:AppliesTo>
1125         <wst:RequestedSecurityToken>
1126         ...
1127         </wst:RequestedSecurityToken>
1128         ...
1129       </wst:RequestSecurityTokenResponse>
1130     </wst:IssuedTokens>
1131   </S11:Header>
1132   <S11:Body>
1133   ...
1134   </S11:Body>
1135 </S11:Envelope>
```


Copyright © OASIS® 1993–2008. 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® 1993–2008. All Rights Reserved.

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

/wst:RequestSecurityToken/wst:RenewTarget

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

/wst:RequestSecurityToken/wst:AllowPostdating

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

/wst:RequestSecurityToken/wst:Renewing

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

/wst:RequestSecurityToken/wst:Renewing/@Allow

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

/wst:RequestSecurityToken/wst:Renewing/@OK

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

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

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

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

```
    <wst:RequestSecurityToken xmlns:wst="...”>
        <wst:TokenType>
            http://example.org/mySpecialToken
        </wst:TokenType>
        <wst:RequestType>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/Renew
        </wst:RequestType>
        <wst:RenewTarget>
            ... reference to previously issued token ...
        </wst:RenewTarget>
    </wst:RequestSecurityToken>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

```
1230 <wst:RequestSecurityTokenResponse xmlns:wst="...”>
1231     <wst:TokenType>
1232         http://example.org/mySpecialToken
1233     </wst:TokenType>
1234     <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1235     <wst:Lifetime>...</wst:Lifetime>
1236     ...
1237 </wst:RequestSecurityTokenResponse>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

6 Cancel Binding

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

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

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

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

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

Extensions MAY be specified in the request (and the response), 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. Signature confirmation is RECOMMENDED on the closure response.

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

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

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.

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


Copyright © OASIS® 1993–2008. 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® 1993–2008. All Rights Reserved.

1331 A cancelled token is no longer valid for authentication and authorization usages.

1332

1333 The mechanism to determine the availability of STS-initiated Cancel binding on the STS is out of scope of
1334 this specification. Similarly, how the client communicates its endpoint address to the STS so that it can
1335 send the STSCancel messages to the client is out of scope of this specification. This functionality is
1336 implementation specific and can be solved by different mechanisms that are not in scope for this
1337 specification.

1338

1339 This is a one-way operation, no response is returned from the recipient of the message.

1340

1341 The syntax of the request is as follows:

```
1342     <wst:RequestSecurityToken xmlns:wst="...”>
1343         <wst:RequestType>...</wst:RequestType>
1344         ...
1345         <wst:CancelTarget>...</wst:CancelTarget>
1346     </wst:RequestSecurityToken>
```

1347 */wst:RequestSecurityToken/wst:CancelTarget*

1348 This REQUIRED element identifies the token being cancelled. Typically this contains a
1349 `<wsse:SecurityTokenReference>` pointing at the token, but it could also carry the token
1350 directly.

1351 The following example illustrates a request to cancel a custom token.

```
1352 <?xml version="1.0" encoding="utf-8"?>
1353 <S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsse="...">
1354   <S11:Header>
1355     <wsse:Security>
1356       ...
1357     </wsse:Security>
1358   </S11:Header>
1359   <S11:Body>
1360     <wst:RequestSecurityToken>
1361         <wst:RequestType>
1362             http://docs.oasis-open.org/ws-sx/ws-trust/200512/STSCancel
1363         </wst:RequestType>
1364         <wst:CancelTarget>
1365             ...
1366         </wst:CancelTarget>
1367     </wst:RequestSecurityToken>
1368   </S11:Body>
1369 </S11:Envelope>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

7 Validation Binding

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

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

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

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

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

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

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

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

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

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

The validation binding does not allow the use of exchanges.

The RSTR for this binding carries the following 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:ValidateTarget>... </wst:ValidateTarget>
    ...
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

1413
```
 </wst:RequestSecurityToken>
```
1414

1415–1423
```
<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>
```
1424

1425 */wst:RequestSecurityToken/wst:ValidateTarget*

1426 This REQUIRED element identifies the token being validated. Typically this contains a
1427 <wsse:SecurityTokenReference> pointing at the token, but could also carry the token
1428 directly.

1429 */wst:RequestSecurityTokenResponse/wst:Status*

1430 When a validation request is made, this element MUST be in the response. The code value
1431 indicates the results of the validation in a machine-readable form. The accompanying text
1432 element allows for human textual display.

1433 */wst:RequestSecurityTokenResponse/wst:Status/wst:Code*

1434 This REQUIRED URI value provides a machine-readable status code. The following URIs are
1435 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

1436 */wst:RequestSecurityTokenResponse/wst:Status/wst:Reason*

1437 This OPTIONAL string provides human-readable text relating to the status code.

1438

1439 The following illustrates the syntax of a validation request and response. In this example no token is
1440 requested, just a status.

1441–1448
```
 <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>
```
1449

1450–1453
```
<wst:RequestSecurityTokenResponse xmlns:wst="...”>
     <wst:TokenType>
       http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Status
     </wst:TokenType>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

```
1454            <wst:Status>
1455                <wst:Code>
1456             http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
1457                </wst:Code>
1458            </wst:Status>
1459            ...
1460        </wst:RequestSecurityTokenResponse>
```

1461 The following illustrates the syntax of a validation request and response. In this example a custom token
1462 is requested indicating authorized rights in addition to the status.

```
1463        <wst:RequestSecurityToken xmlns:wst="...”>
1464            <wst:TokenType>
1465                http://example.org/mySpecialToken
1466            </wst:TokenType>
1467            <wst:RequestType>
1468                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate
1469            </wst:RequestType>
1470        </wst:RequestSecurityToken>
```

1471

```
1472       <wst:RequestSecurityTokenResponse xmlns:wst="...”>
1473            <wst:TokenType>
1474                http://example.org/mySpecialToken
1475            </wst:TokenType>
1476            <wst:Status>
1477                <wst:Code>
1478             http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
1479                </wst:Code>
1480            </wst:Status>
1481            <wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
1482            ...
1483        </wst:RequestSecurityTokenResponse>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

1484 # 8 Negotiation and Challenge Extensions

1485 The general security token service framework defined above allows for a simple request and response for
1486 security tokens (possibly asynchronous). However, there are many scenarios where a set of exchanges
1487 between the parties is REQUIRED prior to returning (e.g., issuing) a security token. This section
1488 describes the extensions to the base WS-Trust mechanisms to enable exchanges for negotiation and
1489 challenges.

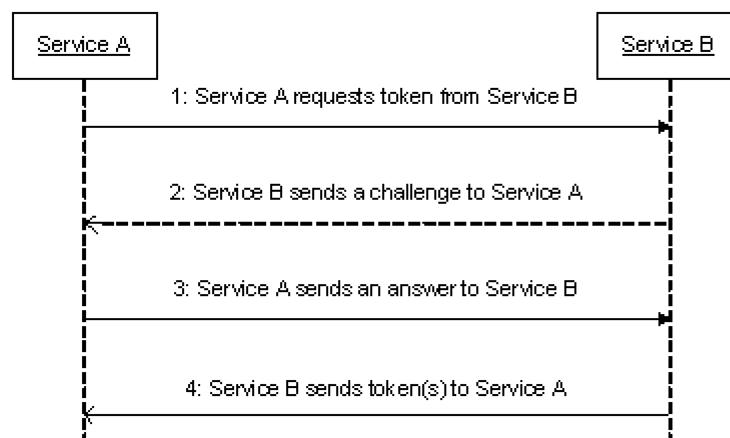
1490

1491 There are potentially different forms of exchanges, but one specific form, called "challenges", provides
1492 mechanisms in addition to those described in [WS-Security] for authentication. This section describes
1493 how general exchanges are issued and responded to within this framework. Other types of exchanges
1494 include, but are not limited to, negotiation, tunneling of hardware-based processing, and tunneling of
1495 legacy protocols.

1496

1497 The process is straightforward (illustrated here using a challenge):

1498



1499

1500 1. A requestor sends, for example, a `<wst:RequestSecurityToken>` message with a
1501 timestamp.

1502 2. The recipient does not trust the timestamp and issues a
1503 `<wst:RequestSecurityTokenResponse>` message with an embedded challenge.

1504 3. The requestor sends a `<wst:RequestSecurityTokenReponse>` message with an answer to
1505 the challenge.

1506 4. The recipient issues a `<wst:RequestSecurityTokenResponseCollection>` message with
1507 the issued security token and OPTIONAL proof-of-possession token.

1508

1509 It should be noted that the requestor might challenge the recipient in either step 1 or step 3. In which
1510 case, step 2 or step 4 contains an answer to the initiator's challenge. Similarly, it is possible that steps 2
1511 and 3 could iterate multiple times before the process completes (step 4).

1512

1513 The two services can use [WS-SecurityPolicy] to state their requirements and preferences for security
1514 tokens and encryption and signing algorithms (general policy intersection). This section defines
1515 mechanisms for legacy and more sophisticated types of negotiations.

Copyright © OASIS® 1993–2008. 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® 1993–2008. All Rights Reserved.

1560 The following describes the attributes and tags listed in the schema above:

1561 *.../wst:SignChallenge*

1562 This OPTIONAL element describes a challenge that requires the other party to sign a specified
1563 set of information.

1564 *.../wst:SignChallenge/wst:Challenge*

1565 This REQUIRED string element describes the value to be signed. In order to prevent certain
1566 types of attacks (such as man-in-the-middle), it is strongly RECOMMENDED that the challenge
1567 be bound to the negotiation. For example, the challenge SHOULD track (such as using a digest
1568 of) any relevant data exchanged such as policies, tokens, replay protection, etc. As well, if the
1569 challenge is happening over a secured channel, a reference to the channel SHOULD also be
1570 included. Furthermore, the recipient of a challenge SHOULD verify that the data tracked
1571 (digested) matches their view of the data exchanged. The exact algorithm MAY be defined in
1572 profiles or agreed to by the parties.

1573 *.../SignChallenge/{any}*

1574 This is an extensibility mechanism to allow additional negotiation types to be used.

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

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

1578 *.../wst:SignChallengeResponse*

1579 This OPTIONAL element describes a response to a challenge that requires the signing of a
1580 specified set of information.

1581 *.../wst:SignChallengeResponse/wst:Challenge*

1582 If a challenge was issued, the response MUST contain the challenge element exactly as
1583 received. As well, while the RSTR response SHOULD always be signed, if a challenge was
1584 issued, the RSTR MUST be signed (and the signature coupled with the message to prevent
1585 replay).

1586 *.../wst:SignChallengeResponse/{any}*

1587 This is an extensibility mechanism to allow additional negotiation types to be used.

1588 *.../wst:SignChallengeResponse/@{any}*

1589 This is an extensibility mechanism to allow additional attributes, based on schemas, to be added
1590 to the element.

1591 8.3 User Interaction Challenge

1592 User interaction challenge requests are issued by including the <InteractiveChallenge> element. The
1593 response is returned in a <InteractiveChallengeResponse> element. Both the challenge and response
1594 elements are specified within the <wst:RequestSecurityTokenResponse> element. In some instances, the
1595 requestor may issue a challenge to the recipient or provide a response to an anticipated challenge from
1596 the recipient in the initial request. Consequently, these elements are also allowed within a
1597 <wst:RequestSecurityToken> element. The challenge/response exchange between client and server
1598 MAY be iterated over multiple legs before a final response is issued.

1599 Implementations SHOULD take into account the possibility that messages in either direction may be lost
1600 or duplicated. In the absence of a lower level protocol guaranteeing delivery of every message in order
1601 and exactly once, which retains the ordering of requests and responses traveling in opposite directions,
1602 implementations SHOULD observe the following procedures:

1603 The STS SHOULD:

1604 1. Never send a new request while an existing request is pending,

Copyright © OASIS® 1993–2008. All Rights Reserved.

1605 2. Timeout requests and retransmit them.
1606 3. Silently discard responses when no request is pending.
1607
1608 The service consumer MAY:
1609 1. Respond to a repeated request with the same information
1610 2. Retain user input until the Challenge Interation is complete in case it is necessary to repeat the
1611 response.
1612 Note that the xml:lang attribute may be used where allowed via attribute extensibility to specify a
1613 language of localized elements and attributes using the language codes specified in [RFC 3066].

1614 8.3.1 Challenge Format

1615 The syntax of the user interaction challenge element is as follows:

```
1616 <wst14:InteractiveChallenge xmlns:wst14="..." ...>
1617   <wst14:Title ...> xs:string </wst14:Title> ?
1618   <wst14:TextChallenge RefId="xs:anyURI" Label="xs:string"?
1619                        MaxLen="xs:int"? HideText="xs:boolean"? ...>
1620     <wst14:Image MimeType="xs:string"> xs:base64Binary </wst14:Image> ?
1621   </wst14:TextChallenge> *
1622   <wst14:ChoiceChallenge RefId="xs:anyURI" Label="xs:string"?
1623                          ExactlyOne="xs:boolean"? ...>
1624     <wst14:Choice RefId="xs:anyURI" Label="xs:string"? ...>
1625       <wst14:Image MimeType="xs:string"> xs:base64Binary </wst14:Image> ?
1626     </wst14:Choice> +
1627   </wst14:ChoiceChallenge> *
1628   < wst14:ContextData RefId="xs:anyURI"> xs:any </wst14:ContextData> *
1629   ...
1630 </wst14:InteractiveChallenge>
```

1631 The following describes the attributes and elements listed in the schema outlined above:
1632
1633 *.../wst14:InteractiveChallenge*
1634 A container element for a challenge that requires interactive user input.
1635 *.../wst14:InteractiveChallenge/wst14:Title*
1636 An OPTIONAL element that specifies an overall title text to be displayed to the user (e.g. a title
1637 describing the purpose or nature of the challenge). How the preferred language of the requestor
1638 is communicated to the STS is left up to implementations.
1639 *.../wst14:InteractiveChallenge/wst14:TextChallenge*
1640 An OPTIONAL element that specifies a challenge that requires textual input from the user.
1641 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@RefId*
1642 A REQUIRED attribute that specifies a reference identifier for this challenge element which is
1643 used to correlate the corresponding element in the response to the challenge.
1644 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@MaxLen*
1645 An OPTIONAL attribute that specifies the maximum length of the text string that is sent as the
1646 response to this text challenge. This value serves as a hint for the user interface software at the
1647 requestor which manifests the end-user experience for this challenge.
1648 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@HideText*
1649 An OPTIONAL attribute that specifies that the response to this text challenge MUST receive
1650 treatment as hidden text in any user interface. For example, the text entry may be displayed as a

Copyright © OASIS® 1993–2008. All Rights Reserved.

1651 series of asterisks in the user interface. This attribute serves as a hint for the user interface
1652 software at the requestor which manifests the end-user experience for this challenge.

1653 *.../wst14:InteractiveChallenge/wst14:TextChallenge/@Label*

1654 An OPTIONAL attribute that specifies a label for the text challenge item (e.g. a label for a text
1655 entry field) which will be shown to the user. How the preferred language of the requestor is
1656 communicated to the STS is left up to implementations.

1657 *.../wst14:InteractiveChallenge/wst14:TextChallenge/Image*

1658 An OPTIONAL element that contains a base64 encoded inline image specific to the text
1659 challenge item to be shown to the user (e.g. an image that the user must see to respond
1660 successfully to the challenge). The image presented is intended as an additional label to a
1661 challenge element which could be CAPTCHA, selection of a previously established image secret
1662 or any other means by which images can be used to challenge a user to interact in a way to
1663 satisfy a challenge.

1664 *.../wst14:InteractiveChallenge/wst14:TextChallenge/Image/@MimeType*

1665 A REQUIRED attribute that specifies a MIME type (e.g., image/gif, image/jpg) indicating the
1666 format of the image.

1667 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge*

1668 An OPTIONAL element that specifies a challenge that requires a choice among multiple items by
1669 the user.

1670 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/@RefId*

1671 A REQUIRED attribute that specifies a reference identifier for this challenge element which is
1672 used to correlate the corresponding element in the response to the challenge.

1673 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/@Label*

1674 An OPTIONAL attribute that specifies a title label for the choice challenge item (e.g., a text
1675 header describing the list of choices as a whole) which will be shown to the user. How the
1676 preferred language of the requestor is communicated to the STS is left up to implementations.

1677 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/@ExactlyOne*

1678 An OPTIONAL attribute that specifies if exactly once choice must be selected by the user from
1679 among the child element choices. The absence of this attribute implies the value “false” which
1680 means multiple choices can be selected.

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

1682 A REQUIRED element that specifies a single choice item within the choice challenge.

1683 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/@RefId*

1684 A REQUIRED attribute that specifies a reference identifier for this specific choice item which is
1685 used to correlate the corresponding element in the response to the challenge.

1686 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/@Label*

1687 An OPTIONAL attribute that specifies a text label for the choice item (e.g., text describing the
1688 individual choice) which will be shown to the user. How the preferred language of the requestor is
1689 communicated to the STS is left up to implementations.

1690 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/wst14:Image*

1691 An OPTIONAL element that contains a base64 encoded inline image specific to the choice item
1692 to be shown to the user (e.g. an image that the user must see to respond successfully to the
1693 challenge). The image presented is intended as an additional label to a challenge element which
1694 could be CAPTCHA, selection of a previously established image secret or any other means by
1695 which images can be used to challenge a user to interact in a way to satisfy a challenge.

Copyright © OASIS® 1993–2008. All Rights Reserved.

1696 *.../wst14:InteractiveChallenge/wst14:ChoiceChallenge/wst14:Choice/wst14:Image/@MimeType*

1697 A REQUIRED attribute that specifies a MIME type (e.g., image/gif, image/jpg) indicating the
1698 format of the image.

1699 *.../wst14:InteractiveChallenge/wst14:ContextData*

1700 An OPTIONAL element that specifies a value that MUST be reflected back in the response to the
1701 challenge (e.g., cookie). The element may contain any value. The actual content is opaque to the
1702 requestor; it is not required to understand its structure or semantics. This can be used by an STS,
1703 for instance, to store information between the challenge/response exchanges that would
1704 otherwise be lost if the STS were to remain stateless.

1705 *.../wst14:InteractiveChallenge/wst14:ContextData/@RefId*

1706 A REQUIRED attribute that specifies a reference identifier for this context element which is used
1707 to correlate the corresponding element in the response to the challenge.

1708 *.../wst14:InteractiveChallenge/{any}*

1709 This is an extensibility mechanism to allow additional elements to be specified.

1710 *.../wst14:InteractiveChallenge/@{any}*

1711 This is an extensibility mechanism to allow additional attributes to be specified.

1712

1713 The syntax of the user interaction challenge response element is as follows:

```
1714 <wst14:InteractiveChallengeResponse xmlns:wst14="..." ...>
1715   <wst14:TextChallengeResponse RefId="xs:anyURI" ...>
1716     xs:string
1717   </wst14:TextChallengeResponse> *
1718   <wst14:ChoiceChallengeResponse RefId="xs:anyURI"> *
1719     <wst14:ChoiceSelected RefId="xs:anyURI" /> *
1720   </wst14:ChoiceChallengeResponse>
1721   <wst14:ContextData RefId="xs:anyURI"> xs:any </wst14:ContextData> *
1722   ...
1723 </wst14:InteractiveChallengeResponse>
```

1724 The following describes the attributes and elements listed in the schema outlined above:

1725

1726 *.../wst14:InteractiveChallengeResponse*

1727 A container element for the response to a challenge that requires interactive user input.

1728 *.../wst14:InteractiveChallengeResponse/wst14:TextChallengeResponse*

1729 This element value contains the user input as the response to the original text challenge issued.

1730 *.../wst14:InteractiveChallengeResponse/wst14:TextChallengeResponse/@RefId*

1731 A required attribute that specifies the identifier for the text challenge element in the original
1732 challenge which can be used for correlation.

1733 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse*

1734 A container element for the response to a choice challenge.

1735 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse/@RefId*

1736 A required attribute that specifies the reference identifier for the choice challenge element in the
1737 original challenge which can be used for correlation.

1738 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse/wst14:ChoiceSelected*

1739 A required element that specifies a choice item selected by the user from the choice challenge.

1740 *.../wst14:InteractiveChallengeResponse/wst14:ChoiceChallengeResponse/wst14:ChoiceSelected/@RefId*

Copyright © OASIS® 1993–2008. All Rights Reserved.

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

.../wst14:InteractiveChallengeResponse/wst14:ContextData

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

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

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

.../wst14:InteractiveChallengeResponse/{any}

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

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

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

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

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

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

8.3.2 PIN and OTP Challenges

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

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

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

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

```
http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN
http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/OTP
```

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

Copyright © OASIS® 1993–2008. All Rights Reserved.

8.4 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

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

.../wst:BinaryExchange/@ValueType

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

.../wst:BinaryExchange/@EncodingType

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

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

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

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

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

8.5 Key Exchange Tokens

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

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

The syntax of these elements is as follows (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:RequestKET xmlns:wst="..." />
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

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

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

.../wst:RequestKET

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

.../wst:KeyExchangeToken

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

8.6 Custom Exchanges

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

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

8.7 Signature Challenge Example

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

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

```
<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:KeyInfo>
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#reqToken"/>
                </wsse:SecurityTokenReference>
              </ds:KeyInfo>
            </ds:Signature>
        </wsse:Security>
        ...
    </S11:Header>
    <S11:Body>
        <wst:RequestSecurityToken>
            <wst:TokenType>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

```
1878                 http://example.org/mySpecialToken
1879             </wst:TokenType>
1880             <wst:RequestType>
1881                 http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
1882             </wst:RequestType>
1883         </wst:RequestSecurityToken>
1884     </S11:Body>
1885 </S11:Envelope>
```

1886

1887 The issuer (recipient) service doesn't trust the sender's timestamp (or one wasn't specified) and issues a
1888 challenge using the exchange framework defined in this specification. This message is signed using the
1889 private key associated with the issuer's X.509 certificate and contains a random challenge that the
1890 requestor must sign:

```
1891 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
1892         xmlns:wst="...">
1893     <S11:Header>
1894         ...
1895         <wsse:Security>
1896             <wsse:BinarySecurityToken
1897                     wsu:Id="issuerToken"
1898                     ValueType="...X509v3">
1899                 DFJHuedsujfnrnv45JZc0...
1900             </wsse:BinarySecurityToken>
1901             <ds:Signature xmlns:ds="...">
1902                 ...
1903             </ds:Signature>
1904        </wsse:Security>
1905        ...
1906     </S11:Header>
1907     <S11:Body>
1908         <wst:RequestSecurityTokenResponse>
1909             <wst:SignChallenge>
1910                 <wst:Challenge>Huehf...</wst:Challenge>
1911             </wst:SignChallenge>
1912         </wst:RequestSecurityTokenResponse>
1913     </S11:Body>
1914 </S11:Envelope>
```

1915

1916 The requestor receives the issuer's challenge and issues a response that is signed using the requestor's
1917 X.509 certificate and contains the challenge. The signature only covers the non-mutable elements of the
1918 message to prevent certain types of security attacks:

```
1919 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
1920         xmlns:wst="...">
1921     <S11:Header>
1922         ...
1923         <wsse:Security>
1924             <wsse:BinarySecurityToken
1925                     wsu:Id="reqToken"
1926                     ValueType="...X509v3">
1927                 MIIEZzCCA9CgAwIBAgIQEmtJZc0...
1928             </wsse:BinarySecurityToken>
1929             <ds:Signature xmlns:ds="...">
1930                 ...
1931             </ds:Signature>
1932         </wsse:Security>
1933         ...
1934     </S11:Header>
1935     <S11:Body>
1936         <wst:RequestSecurityTokenResponse>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

```
1937            <wst:SignChallengeResponse>
1938                <wst:Challenge>Huehf...</wst:Challenge>
1939            </wst:SignChallengeResponse>
1940        </wst:RequestSecurityTokenResponse>
1941    </S11:Body>
1942 </S11:Envelope>
```

1943

1944 The issuer validates the requestor's signature responding to the challenge and issues the requested
1945 token(s) and the associated proof-of-possession token. The proof-of-possession token is encrypted for
1946 the requestor using the requestor's public key.

```
1947 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
1948         xmlns:wst="..." xmlns:xenc="...">
1949     <S11:Header>
1950         ...
1951         <wsse:Security>
1952             <wsse:BinarySecurityToken
1953                     wsu:Id="issuerToken"
1954                     ValueType="...X509v3">
1955                 DFJHuedsujfnrnv45JZc0...
1956             </wsse:BinarySecurityToken>
1957             <ds:Signature xmlns:ds="...">
1958                 ...
1959             </ds:Signature>
1960         </wsse:Security>
1961         ...
1962     </S11:Header>
1963     <S11:Body>
1964       <wst:RequestSecurityTokenResponseCollection>
1965         <wst:RequestSecurityTokenResponse>
1966             <wst:RequestedSecurityToken>
1967                 <xyz:CustomToken xmlns:xyz="...">
1968                     ...
1969                 </xyz:CustomToken>
1970             </wst:RequestedSecurityToken>
1971             <wst:RequestedProofToken>
1972                 <xenc:EncryptedKey Id="newProof">
1973                     ...
1974                 </xenc:EncryptedKey>
1975             </wst:RequestedProofToken>
1976         </wst:RequestSecurityTokenResponse>
1977       </wst:RequestSecurityTokenResponseCollection>
1978     </S11:Body>
1979 </S11:Envelope>
```

1980 8.8 Challenge Examples

1981 8.8.1 Text and choice challenge

1982 Here is an example of a user interaction challenge using both text and choice challenges. In this example,
1983 a user requests a custom token using a username/password for authentication. The STS uses the
1984 challenge mechanism to challenge the user for additional information in the form of a secret question (i.e.,
1985 Mother's maiden name) and an age group choice. The challenge additionally includes one contextual
1986 data item that needs to be reflected back in the response. The user interactively provides the requested
1987 data and, once validated, the STS issues the requested token. All messages are sent over a protected
1988 transport using SSLv3.

1989

1990 The requestor sends the initial request that includes the username/password for authentication as follows.

1991

Copyright © OASIS® 1993–2008. All Rights Reserved.

```
<S11:Envelope ...>
  <S11:Header>
    ...
    <wsse:Security>
      <wsse:UsernameToken>
        <wsse:Username>Zoe</wsse:Username>
        <wsse:Password
           Type="http://...#PasswordText">ILoveDogs</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityToken>
      <wst:TokenType>http://example.org/customToken</wst:TokenType>
      <wst:RequestType>...</wst:RequestType>
    </wst:RequestSecurityToken>
  </S11:Body>
</S11:Envelope>
```

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

```
<S11:Envelope ...>
  <S11:Header>
    ...
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityTokenResponse>
      <wst14:InteractiveChallenge xmlns:wst14="..." >
        <wst14:Title>
          Please answer the following additional questions to login.
        </wst14:Title>
        <wst14:TextChallenge RefId=http://.../ref#text1
              Label="Mother's Maiden Name" MaxLen=80 />
        <wst14:ChoiceChallenge RefId="http://.../ref#choiceGroupA"
              Label="Your Age Group:" ExactlyOne="true">
          <wst14:Choice RefId="http://.../ref#choice1" Label="18-30" />
          <wst14:Choice RefId="http://.../ref#choice2" Label="31-40" />
          <wst14:Choice RefId="http://.../ref#choice3" Label="41-50" />
          <wst14:Choice RefId="http://.../ref#choice4" Label="50+" />
        </wst14:ChoiceChallenge>
        <wst14:ContextData RefId="http://.../ref#cookie1">
           ...
        </wst14:ContextData>
      </wst14:InteractiveChallenge>
    </wst:RequestSecurityTokenResponse>
  </S11:Body>
</S11:Envelope>
```

The requestor receives the challenge, provides the necessary user experience for soliciting the required inputs, and sends a response to the challenge back to the STS as follows.

```
<S11:Envelope ...>
  <S11:Header>
    ...
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityTokenResponse>
      <wst14:InteractiveChallengeResponse xmlns:wst14="..." >
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

```
        <wst14:TextChallengeResponse RefId="http://.../ref#text1">
          Goldstein
        </wst14:TextChallengeResponse>
        <wst14:ChoiceChallengeResponse RefId="http://.../ref#choiceGroupA">
          <wst14:ChoiceSelected RefId="http://.../ref#choice3" />
        </wst14:ChoiceChallengeResponse>
        <wst14:ContextData RefId="http://.../ref#cookie1">
          ...
        </wst14:ContextData>
      </wst14:InteractiveChallengeResponse>
    </wst:RequestSecurityTokenResponse>
  </S11:Body>
</S11:Envelope>
```

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

```
<S11:Envelope ...>
  <S11:Header>
    ...
  </S11:Header>
  <S11:Body>
   <wst:RequestSecurityTokenResponseCollection>
    <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
        <xyz:CustomToken xmlns:xyz="...">
          ...
        </xyz:CustomToken>
      </wst:RequestedSecurityToken>
      <wst:RequestedProofToken>
        ...
      </wst:RequestedProofToken>
    </wst:RequestSecurityTokenResponse>
  </wst:RequestSecurityTokenResponseCollection>
  </S11:Body>
</S11:Envelope>
```

8.8.2 PIN challenge

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

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

```
<S11:Envelope ...>
  <S11:Header>
    ...
    <wsse:Security>
      <wsse:UsernameToken>
        <wsse:Username>Zoe</wsse:Username>
        <wsse:Password Type="http://...#PasswordText">
          ILoveDogs
        </wsse:Password>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

2106
2107
2108
2109
2110
2111
2112
2113
2114
2115

```
      </wsse:UsernameToken>
    </wsse:Security>
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityToken>
      <wst:TokenType>http://example.org/customToken</wst:TokenType>
      <wst:RequestType>...</wst:RequestType>
    </wst:RequestSecurityToken>
  </S11:Body>
</S11:Envelope>
```

2116

2117 The STS issues a challenge for a secret PIN using the text challenge mechanism as follows.

2118

2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133

```
<S11:Envelope ...>
  <S11:Header>
    ...
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityTokenResponse>
      <wst14:InteractiveChallenge xmlns:wst14="..." >
       <wst14:TextChallenge
        RefId="http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN"
        Label="Please enter your PIN" />
       </wst14:TextChallenge>
      </wst14:InteractiveChallenge>
    </wst:RequestSecurityTokenResponse>
  </S11:Body>
</S11:Envelope>
```

2134

2135 The requestor receives the challenge, provides the necessary user experience for soliciting the PIN, and
2136 sends a response to the challenge back to the STS as follows.

2137

2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152

```
<S11:Envelope ...>
  <S11:Header>
    ...
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityTokenResponse>
     <wst14:InteractiveChallengeResponse xmlns:wst14="..." >
      <wst14:TextChallengeResponse
       RefId="http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN">
          9988
      </wst14:TextChallengeResponse>
     </wst14:InteractiveChallengeResponse>
    </wst:RequestSecurityTokenResponse>
  </S11:Body>
</S11:Envelope>
```

2153

2154 The STS validates the PIN response, and issues the requested token as follows.

2155

2156
2157
2158
2159
2160
2161

```
<S11:Envelope ...>
  <S11:Header>
    ...
  </S11:Header>
  <S11:Body>
   <wst:RequestSecurityTokenResponseCollection>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

```
    <wst:RequestSecurityTokenResponse>
      <wst:RequestedSecurityToken>
        <xyz:CustomToken xmlns:xyz="...">
          ...
        </xyz:CustomToken>
      </wst:RequestedSecurityToken>
      <wst:RequestedProofToken>
        ...
      </wst:RequestedProofToken>
    </wst:RequestSecurityTokenResponse>
   </wst:RequestSecurityTokenResponseCollection>
  </S11:Body>
</S11:Envelope>
```

8.8.3 PIN challenge with optimized response

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

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

```
<S11:Envelope ...>
  <S11:Header>
    ...
    <wsse:Security>
      <wsse:UsernameToken>
        <wsse:Username>Zoe</wsse:Username>
        <wsse:Password Type="http://...#PasswordText">
          ILoveDogs
        </wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </S11:Header>
  <S11:Body>
    <wst:RequestSecurityToken>
     <wst:TokenType>http://example.org/customToken</wst:TokenType>
     <wst:RequestType>...</wst:RequestType>
     <wst14:InteractiveChallengeResponse xmlns:wst14="..." >
      <wst14:TextChallengeResponse
       RefId="http://docs.oasis-open.org/ws-sx/ws-trust/200802/challenge/PIN">
          9988
      </wst14:TextChallengeResponse>
     </wst14:InteractiveChallengeResponse>
    </wst:RequestSecurityToken>
  </S11:Body>
</S11:Envelope>
```

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

```
<S11:Envelope ...>
  <S11:Header>
    ...
  </S11:Header>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

```
    <S11:Body>
     <wst:RequestSecurityTokenResponseCollection>
      <wst:RequestSecurityTokenResponse>
        <wst:RequestedSecurityToken>
          <xyz:CustomToken xmlns:xyz="...">
            ...
          </xyz:CustomToken>
        </wst:RequestedSecurityToken>
        <wst:RequestedProofToken>
          ...
        </wst:RequestedProofToken>
      </wst:RequestSecurityTokenResponse>
     </wst:RequestSecurityTokenResponseCollection>
    </S11:Body>
  </S11:Envelope>
```

8.9 Custom Exchange Example

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

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

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

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

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>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

2267 The issuer processes the exchange and determines that the exchange is complete and that a token
2268 should be issued. Consequently it issues the requested token(s) and the associated proof-of-possession
2269 token. The proof-of-possession token is encrypted for the requestor using the requestor's public key.

```
2270 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
2271   <wst:RequestSecurityTokenResponse>
2272       <wst:RequestedSecurityToken>
2273           <xyz:CustomToken xmlns:xyz="...">
2274               ...
2275           </xyz:CustomToken>
2276       </wst:RequestedSecurityToken>
2277       <wst:RequestedProofToken>
2278           <xenc:EncryptedKey Id="newProof" xmlns:xenc="...">
2279               ...
2280           </xenc:EncryptedKey>
2281       </wst:RequestedProofToken>
2282       <wst:RequestedProofToken>
2283          <xenc:EncryptedKey xmlns:xenc="...">...</xenc:EncryptedKey>
2284       </wst:RequestedProofToken>
2285   </wst:RequestSecurityTokenResponse>
2286 </wst:RequestSecurityTokenResponseCollection>
```

2287 It should be noted that other example exchanges include the issuer returning a final custom exchange
2288 element, and another example where a token isn't returned.

2289 8.10 Protecting Exchanges

2290 There are some attacks, such as forms of man-in-the-middle, that can be applied to token requests
2291 involving exchanges. It is RECOMMENDED that the exchange sequence be protected. This MAY be
2292 built into the exchange messages, but if metadata is provided in the RST or RSTR elements, then it is
2293 subject to attack.

2294

2295 Consequently, it is RECOMMENDED that keys derived from exchanges be linked cryptographically to the
2296 exchange. For example, a hash can be computed by computing the SHA1 of the exclusive
2297 canonicalization [XML-C14N] of all RST and RSTR elements in messages exchanged. This value can
2298 then be combined with the exchanged secret(s) to create a new master secret that is bound to the data
2299 both parties sent/received.

2300

2301 To this end, the following computed key algorithm is defined to be OPTIONALLY used in these scenarios:

URI	**Meaning**
http://docs.oasis-open.org/ws-sx/ws-trust/200512/CK/HASH	The key is computed using P_SHA1 as follows: H=SHA1(ExclC14N(RST...RSTRs)) X=encrypting H using negotiated key and mechanism Key=P_SHA1(X,H+"CK-HASH") The octets for the "CK-HASH" string are the UTF-8 octets.

2302 8.11 Authenticating Exchanges

2303 After an exchange both parties have a shared knowledge of a key (or keys) that can then be used to
2304 secure messages. However, in some cases it may be desired to have the issuer prove to the requestor

Copyright © OASIS® 1993–2008. All Rights Reserved.

2305 that it knows the key (and that the returned metadata is valid) prior to the requestor using the data.
2306 However, until the exchange is actually completed it MAY be (and is often) inappropriate to use the
2307 computed keys. As well, using a token that hasn't been returned to secure a message may complicate
2308 processing since it crosses the boundary of the exchange and the underlying message security. This
2309 means that it MAY NOT be appropriate to sign the final leg of the exchange using the key derived from
2310 the exchange.

2311

2312 For this reason an authenticator is defined that provides a way for the issuer to verify the hash as part of
2313 the token issuance. Specifically, when an authenticator is returned, the
2314 `<wst:RequestSecurityTokenResponseCollection>` element is returned. This contains one
2315 RSTR with the token being returned as a result of the exchange and a second RSTR that contains the
2316 authenticator (this order SHOULD be used). When an authenticator is used, RSTRs MUST use the
2317 *@Context* element so that the authenticator can be correlated to the token issuance. The authenticator is
2318 separated from the RSTR because otherwise computation of the RST/RSTR hash becomes more
2319 complex. The authenticator is represented using the `<wst:Authenticator>` element as illustrated
2320 below:

```
2321    <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
2322        <wst:RequestSecurityTokenResponse Context="...">
2323            ...
2324        </wst:RequestSecurityTokenResponse>
2325        <wst:RequestSecurityTokenResponse Context="...">
2326            <wst:Authenticator>
2327                <wst:CombinedHash>...</wst:CombinedHash>
2328                ...
2329            </wst:Authenticator>
2330        </wst:RequestSecurityTokenResponse>
2331    </wst:RequestSecurityTokenResponseCollection>
```

2332

2333 The following describes the attributes and elements listed in the schema overview above (the ... notation
2334 below represents the path RSTRC/RSTR and is used for brevity):

2335 *.../wst:Authenticator*

2336 This OPTIONAL element provides verification (authentication) of a computed hash.

2337 *.../wst:Authenticator/wst:CombinedHash*

2338 This OPTIONAL element proves the hash and knowledge of the computed key. This is done by
2339 providing the base64 encoding of the first 256 bits of the P_SHA1 digest of the computed key and
2340 the concatenation of the hash determined for the computed key and the string "AUTH-HASH".
2341 Specifically, P_SHA1(*computed-key*, H + "AUTH-HASH")$_{0\text{-}255}$. The octets for the "AUTH-HASH"
2342 string are the UTF-8 octets.

2343

2344 This `<wst:CombinedHash>` element is OPTIONAL (and an open content model is used) to allow for
2345 different authenticators in the future.

Copyright © OASIS® 1993–2008. 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® 1993–2008. All Rights Reserved.

2389 values must be adhered to and faults generated for invalid inputs, or if the issuer will provide alterative
2390 values in the response.

2391

2392 Although illustrated using the `<wst:RequestSecurityToken>` element, these options can also be
2393 returned in a `<wst:RequestSecurityTokenResponse>` element.

2394 The syntax for these OPTIONAL elements is as follows (note that the base elements described above are
2395 included here italicized for completeness):

```
2396      <wst:RequestSecurityToken xmlns:wst="...”>
2397          <wst:TokenType>...</wst:TokenType>
2398          <wst:RequestType>...</wst:RequestType>
2399          ...
2400          <wst:AuthenticationType>...</wst:AuthenticationType>
2401          <wst:KeyType>...</wst:KeyType>
2402          <wst:KeySize>...</wst:KeySize>
2403          <wst:SignatureAlgorithm>...</wst:SignatureAlgorithm>
2404          <wst:EncryptionAlgorithm>...</wst:EncryptionAlgorithm>
2405          <wst:CanonicalizationAlgorithm>...</wst:CanonicalizationAlgorithm>
2406          <wst:ComputedKeyAlgorithm>...</wst:ComputedKeyAlgorithm>
2407          <wst:Encryption>...</wst:Encryption>
2408          <wst:ProofEncryption>...</wst:ProofEncryption>
2409          <wst:KeyWrapAlgorithm>...</wst:KeyWrapAlgorithm>
2410          <wst:UseKey Sig="..."> </wst:UseKey>
2411          <wst:SignWith>...</wst:SignWith>
2412          <wst:EncryptWith>...</wst:EncryptWith>
2413      </wst:RequestSecurityToken>
```

2414

2415 The following describes the attributes and elements listed in the schema overview above:

2416 */wst:RequestSecurityToken/wst:AuthenticationType*

2417 This OPTIONAL URI element indicates the type of authentication desired, specified as a URI.
2418 This specification does not predefine classifications; these are specific to token services as is the
2419 relative strength evaluations. The relative assessment of strength is up to the recipient to
2420 determine. That is, requestors SHOULD be familiar with the recipient policies. For example, this
2421 might be used to indicate which of the four U.S. government authentication levels is REQUIRED.

2422 */wst:RequestSecurityToken/wst:KeyType*

2423 This OPTIONAL URI element indicates the type of key desired in the security token. The
2424 predefined values are identified in the table below. Note that some security token formats have
2425 fixed key types. It should be noted that new algorithms can be inserted by defining URIs in other
2426 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.

2427 */wst:RequestSecurityToken/wst:KeySize*

Copyright © OASIS® 1993–2008. All Rights Reserved.

2428 This OPTIONAL integer element indicates the size of the key REQUIRED specified in number of
2429 bits. This is a request, and, as such, the requested security token is not obligated to use the
2430 requested key size. That said, the recipient SHOULD try to use a key at least as strong as the
2431 specified value if possible. The information is provided as an indication of the desired strength of
2432 the security.

2433 */wst:RequestSecurityToken/wst:SignatureAlgorithm*

2434 This OPTIONAL URI element indicates the desired signature algorithm used within the returned
2435 token. This is specified as a URI indicating the algorithm (see [XML-Signature] for typical signing
2436 algorithms).

2437 */wst:RequestSecurityToken/wst:EncryptionAlgorithm*

2438 This OPTIONAL URI element indicates the desired encryption algorithm used within the returned
2439 token. This is specified as a URI indicating the algorithm (see [XML-Encrypt] for typical
2440 encryption algorithms).

2441 */wst:RequestSecurityToken/wst:CanonicalizationAlgorithm*

2442 This OPTIONAL URI element indicates the desired canonicalization method used within the
2443 returned token. This is specified as a URI indicating the method (see [XML-Signature] for typical
2444 canonicalization methods).

2445 */wst:RequestSecurityToken/wst:ComputedKeyAlgorithm*

2446 This OPTIONAL URI element indicates the desired algorithm to use when computed keys are
2447 used for issued tokens.

2448 */wst:RequestSecurityToken/wst:Encryption*

2449 This OPTIONAL element indicates that the requestor desires any returned secrets in issued
2450 security tokens to be encrypted for the specified token. That is, so that the owner of the specified
2451 token can decrypt the secret. Normally the security token is the contents of this element but a
2452 security token reference MAY be used instead. If this element isn't specified, the token used as
2453 the basis of the request (or specialized knowledge) is used to determine how to encrypt the key.

2454 */wst:RequestSecurityToken/wst:ProofEncryption*

2455 This OPTIONAL element indicates that the requestor desires any returned secrets in proof-of-
2456 possession tokens to be encrypted for the specified token. That is, so that the owner of the
2457 specified token can decrypt the secret. Normally the security token is the contents of this element
2458 but a security token reference MAY be used instead. If this element isn't specified, the token
2459 used as the basis of the request (or specialized knowledge) is used to determine how to encrypt
2460 the key.

2461 */wst:RequestSecurityToken/wst:KeyWrapAlgorithm*

2462 This OPTIONAL URI element indicates the desired algorithm to use for key wrapping when STS
2463 encrypts the issued token for the relying party using an asymmetric key.

2464 */wst:RequestSecurityToken/wst:UseKey*

2465 If the requestor wishes to use an existing key rather than create a new one, then this OPTIONAL
2466 element can be used to reference the security token containing the desired key. This element
2467 either contains a security token or a `<wsse:SecurityTokenReference>` element that
2468 references the security token containing the key that SHOULD be used in the returned token. If
2469 `<wst:KeyType>` is not defined and a key type is not implicitly known to the service, it MAY be
2470 determined from the token (if possible). Otherwise this parameter is meaningless and is ignored.
2471 Requestors SHOULD demonstrate authorized use of the public key provided.

2472 */wst:RequestSecurityToken/wst:UseKey/@Sig*

2473 In order to *authenticate* the key referenced, a signature MAY be used to prove the referenced
2474 token/key. If specified, this OPTIONAL attribute indicates the ID of the corresponding signature

Copyright © OASIS® 1993–2008. All Rights Reserved.

2475 (by URI reference). When this attribute is present, a key need not be specified inside the element
2476 since the referenced signature will indicate the corresponding token (and key).

2477 */wst:RequestSecurityToken/wst:SignWith*

2478 This OPTIONAL URI element indicates the desired signature algorithm to be used with the issued
2479 security token (typically from the policy of the target site for which the token is being requested.
2480 While any of these OPTIONAL elements MAY be included in RSTRs, this one is a likely
2481 candidate if there is some doubt (e.g., an X.509 cert that can only use DSS).

2482 */wst:RequestSecurityToken/wst:EncryptWith*

2483 This OPTIONAL URI element indicates the desired encryption algorithm to be used with the
2484 issued security token (typically from the policy of the target site for which the token is being
2485 requested.) While any of these OPTIONAL elements MAY be included in RSTRs, this one is a
2486 likely candidate if there is some doubt.

2487 The following summarizes the various algorithm parameters defined above. T is the issued token, P is the
2488 proof key.
2489

2490 **SignatureAlgorithm** - The signature algorithm to use to sign T

2491 **EncryptionAlgorithm** - The encryption algorithm to use to encrypt T

2492 **CanonicalizationAlgorithm** - The canonicalization algorithm to use when signing T

2493 **ComputedKeyAlgorithm** - The key derivation algorithm to use if using a symmetric key for P
2494 where P is computed using client, server, or combined entropy

2495 **Encryption** - The token/key to use when encrypting T

2496 **ProofEncryption** - The token/key to use when encrypting P

2497 **UseKey** - This is P. This is generally used when the client supplies a public-key that it wishes to
2498 be embedded in T as the proof key

2499 **SignWith** - The signature algorithm the client intends to employ when using P to

2500 sign

2501 The encryption algorithms further differ based on whether the issued token contains asymmetric key or
2502 symmetric key. Furthermore, they differ based on what type of key is used to protect the issued token
2503 from the STS to the relying party. The following cases can occur:

2504 T contains symmetric key/STS uses symmetric key to encrypt T for RP

2505 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP
2506 when using the proof key (e.g. AES256)

2507 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to
2508 encrypt the T (e.g. AES256)

2509

2510 T contains symmetric key/STS uses asymmetric key to encrypt T for RP

2511 **EncryptWith** – used to indicate symmetric algorithm that client will use to protect message to RP
2512 when using the proof key (e.g. AES256)

2513 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to
2514 encrypt T for RP (e.g. AES256)

2515 **KeyWrapAlgorithm** – used to indicate the KeyWrap algorithm that the STS SHOULD use to
2516 wrap the generated key that is used to encrypt the T for RP

2517

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

2519 **EncryptWith** – used to indicate the KeyWrap algorithm that the client will use to

Copyright © OASIS® 1993–2008. All Rights Reserved.

2520 protect the symmetric key that is used to protect messages to RP when using the proof key (e.g.
2521 RSA-OAEP-MGF1P)

2522 **EncryptionAlgorithm** – used to indicate the symmetric algorithm that the STS SHOULD use to
2523 encrypt T for RP (e.g. AES256)

2524

2525 T contains asymmetric key/STS uses asymmetric key to encrypt T for RP

2526 **EncryptWith** - used to indicate the KeyWrap algorithm that the client will use to

2527 protect symmetric key that is used to protect message to RP when using the proof

2528 key (e.g. RSA-OAEP-MGF1P)

2529 **EncryptionAlgorithm** - used to indicate the symmetric algorithm that the STS SHOULD use to
2530 encrypt T for RP (e.g. AES256)

2531 **KeyWrapAlgorithm** – used to indicate the KeyWrap algorithm that the STS SHOULD use to
2532 wrap the generated key that is used to encrypt the T for RP

2533

2534 The example below illustrates a request that utilizes several of these parameters. A request is made for a
2535 custom token using a username and password as the basis of the request. For security, this token is
2536 encrypted (see "encUsername") for the recipient using the recipient's public key and referenced in the
2537 encryption manifest. The message is protected by a signature using a public key from the sender and
2538 authorized by the username and password.

2539

2540 The requestor would like the custom token to contain a 1024-bit public key whose value can be found in
2541 the key provided with the "proofSignature" signature (the key identified by "requestProofToken"). The
2542 token should be signed using RSA-SHA1 and encrypted for the token identified by
2543 "requestEncryptionToken". The proof should be encrypted using the token identified by
2544 "requestProofToken".

```
2545 <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
2546         xmlns:wst="..." xmlns:ds="..." xmlns:xenc="...">
2547     <S11:Header>
2548         ...
2549         <wsse:Security>
2550             <xenc:ReferenceList>...</xenc:ReferenceList>
2551             <xenc:EncryptedData Id="encUsername">...</xenc:EncryptedData>
2552             <wsse:BinarySecurityToken wsu:Id="requestEncryptionToken"
2553                     ValueType="...SomeTokenType" xmlns:x="...">
2554                 MIIEZzCCA9CgAwIBAgIQEmtJZc0...
2555             </wsse:BinarySecurityToken>
2556             <wsse:BinarySecurityToken wsu:Id="requestProofToken"
2557                     ValueType="...SomeTokenType" xmlns:x="...">
2558                 MIIEZzCCA9CgAwIBAgIQEmtJZc0...
2559            </wsse:BinarySecurityToken>
2560            <ds:Signature Id="proofSignature">
2561                ... signature proving requested key ...
2562                ... key info points to the "requestedProofToken" token ...
2563            </ds:Signature>
2564        </wsse:Security>
2565        ...
2566     </S11:Header>
2567     <S11:Body wsu:Id="req">
2568         <wst:RequestSecurityToken>
2569             <wst:TokenType>
2570                 http://example.org/mySpecialToken
2571             </wst:TokenType>
2572             <wst:RequestType>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

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

9.3 Delegation and Forwarding Requirements

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

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:DelegateTo>...</wst:DelegateTo>
        <wst:Forwardable>...</wst:Forwardable>
        <wst:Delegatable>...</wst:Delegatable>
        <wst:ActAs>...</wst:ActAs>
    </wst:RequestSecurityToken>
```

/wst:RequestSecurityToken/wst:DelegateTo

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

/wst:RequestSecurityToken/wst:Forwardable

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

/wst:RequestSecurityToken/wst:Delegatable

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

/wst:RequestSecurityToken/wst:ActAs

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

Copyright © OASIS® 1993–2008. All Rights Reserved.

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

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

```
        <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:DelegateTo>
            <wsse:BinarySecurityToken xmlns:wsse="...">
              ...
            </wsse:BinarySecurityToken>
        </wst:DelegateTo>
        <wst:Delegatable>true</wst:Delegatable>
    </wst:RequestSecurityToken>
```

9.4 Policies

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

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>
        ...
        <wsp:Policy xmlns:wsp="...”>...</wsp:Policy>
        <wsp:PolicyReference xmlns:wsp="...”>...</wsp:PolicyReference>
    </wst:RequestSecurityToken>
```

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

/wst:RequestSecurityToken/wsp:Policy

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

/wst:RequestSecurityToken/wsp:PolicyReference

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

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

```
    <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>
        <wsp:Policy xmlns:wsp="...">
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

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

9.5 Authorized Token Participants

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

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

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:Participants>
        <wst:Primary>...</wst:Primary>
        <wst:Participant>...</wst:Participant>
    </wst:Participants>
</wst:RequestSecurityToken>
```

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

/wst:RequestSecurityToken/wst:Participants/

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

/wst:RequestSecurityToken/wst:Participants/wst:Primary

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

/wst:RequestSecurityToken/wst:Participants/wst:Participant

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

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

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

Copyright © OASIS® 1993–2008. 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® 1993–2008. All Rights Reserved.

2757 request could be made using transport security (e.g. TLS) and the key could be returned directly using
2758 <wst:BinarySecret>.

```
2759     <wst:RequestSecurityToken xmlns:wst="...”>
2760         <wst:RequestType>
2761             http://docs.oasis-open.org/ws-sx/ws-trust/200512/KET
2762         </wst:RequestType>
2763     </wst:RequestSecurityToken>
```

2764

```
2765   <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
2766     <wst:RequestSecurityTokenResponse>
2767         <wst:RequestedSecurityToken>
2768             <xenc:EncryptedKey xmlns:xenc="...”>...</xenc:EncryptedKey>
2769         </wst:RequestedSecurityToken>
2770     </wst:RequestSecurityTokenResponse>
2771   </wst:RequestSecurityTokenResponseCollection>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

2772 # 11 Error Handling

2773 There are many circumstances where an *error* can occur while processing security information. Errors
2774 use the SOAP Fault mechanism. Note that the reason text provided below is RECOMMENDED, but
2775 alternative text MAY be provided if more descriptive or preferred by the implementation. The tables
2776 below are defined in terms of SOAP 1.1. For SOAP 1.2, the Fault/Code/Value is env:Sender (as defined
2777 in SOAP 1.2) and the Fault/Code/Subcode/Value is the *faultcode* below and the Fault/Reason/Text is the
2778 *faultstring* below. It should be noted that profiles MAY provide second-level detail fields, but they should
2779 be careful not to introduce security vulnerabilities when doing so (e.g., by providing too detailed
2780 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® 1993–2008. All Rights Reserved.

2781 # 12 Security Considerations

2782 As stated in the Goals section of this document, this specification is meant to provide extensible
2783 framework and flexible syntax, with which one could implement various security mechanisms. This
2784 framework and syntax by itself does not provide any guarantee of security. When implementing and using
2785 this framework and syntax, one must make every effort to ensure that the result is not vulnerable to any
2786 one of a wide range of attacks.

2787

2788 It is not feasible to provide a comprehensive list of security considerations for such an extensible set of
2789 mechanisms. A complete security analysis must be conducted on specific solutions based on this
2790 specification. Below we illustrate some of the security concerns that often come up with protocols of this
2791 type, but we stress that this *is not an exhaustive list of concerns.*

2792

2793 The following statements about signatures and signing apply to messages sent on unsecured channels.

2794

2795 It is critical that all the security-sensitive message elements must be included in the scope of the
2796 message signature. As well, the signatures for conversation authentication must include a timestamp,
2797 nonce, or sequence number depending on the degree of replay prevention required as described in [WS-
2798 Security] and the UsernameToken Profile. Also, conversation establishment should include the policy so
2799 that supported algorithms and algorithm priorities can be validated.

2800

2801 It is required that security token issuance messages be signed to prevent tampering. If a public key is
2802 provided, the request should be signed by the corresponding private key to prove ownership. As well,
2803 additional steps should be taken to eliminate replay attacks (refer to [WS-Security] for additional
2804 information). Similarly, all token references should be signed to prevent any tampering.

2805

2806 Security token requests are susceptible to denial-of-service attacks. Care should be taken to mitigate
2807 such attacks as is warranted by the service.

2808

2809 For security, tokens containing a symmetric key or a password should only be sent to parties who have a
2810 need to know that key or password.

2811

2812 For privacy, tokens containing personal information (either in the claims, or indirectly by identifying who is
2813 currently communicating with whom) should only be sent according to the privacy policies governing
2814 these data at the respective organizations.

2815

2816 For some forms of multi-message exchanges, the exchanges are susceptible to attacks whereby
2817 signatures are altered. To address this, it is suggested that a signature confirmation mechanism be used.
2818 In such cases, each leg should include the confirmation of the previous leg. That is, leg 2 includes
2819 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
2820 correctness of the message outside of the message body.

2821

2822 There are many other security concerns that one may need to consider in security protocols. The list
2823 above should not be used as a "check list" instead of a comprehensive security analysis.

Copyright © OASIS® 1993–2008. All Rights Reserved.

2824

2825 It should be noted that use of unsolicited RSTRs implies that the recipient is prepared to accept such
2826 issuances. Recipients should ensure that such issuances are properly authorized and recognize their
2827 use could be used in denial-of-service attacks.

2828 In addition to the consideration identified here, readers should also review the security considerations in
2829 [WS-Security].

2830

2831 Both token cancellation bindings defined in this specification require that the STS MUST NOT validate or
2832 renew the token after it has been successfully canceled. The STS must take care to ensure that the token
2833 is properly invalidated before confirming the cancel request or sending the cancel notification to the client.
2834 This can be more difficult if the token validation or renewal logic is physically separated from the issuance
2835 and cancellation logic. It is out of scope of this spec how the STS propagates the token cancellation to its
2836 other components. If STS cannot ensure that the token was properly invalidated it MUST NOT send the
2837 cancel notification or confirm the cancel request to the client.

2838

Copyright © OASIS® 1993–2008. All Rights Reserved.

2839 # 13 Conformance

2840

2841 An implementation conforms to this specification if it satisfies all of the MUST or REQUIRED level
2842 requirements defined within this specification. A SOAP Node MUST NOT use the XML namespace
2843 identifier for this specification (listed in Section 1.3) within SOAP Envelopes unless it is compliant with this
2844 specification.

2845 This specification references a number of other specifications (see the table above). In order to comply
2846 with this specification, an implementation MUST implement the portions of referenced specifications
2847 necessary to comply with the required provisions of this specification. Additionally, the implementation of
2848 the portions of the referenced specifications that are specifically cited in this specification MUST comply
2849 with the rules for those portions as established in the referenced specification.

2850 Additionally normative text within this specification takes precedence over normative outlines (as
2851 described in section 1.5.1), which in turn take precedence over the XML Schema [XML Schema Part 1,
2852 Part 2] and WSDL [WSDL 1.1] descriptions. That is, the normative text in this specification further
2853 constrains the schemas and/or WSDL that are part of this specification; and this specification contains
2854 further constraints on the elements defined in referenced schemas.

2855 This specification defines a number of extensions; compliant services are NOT REQUIRED to implement
2856 OPTIONAL features defined in this specification. However, if a service implements an aspect of the
2857 specification, it MUST comply with the requirements specified (e.g. related "MUST" statements). If an
2858 OPTIONAL message is not supported, then the implementation SHOULD Fault just as it would for any
2859 other unrecognized/unsupported message. If an OPTIONAL message is supported, then the
2860 implementation MUST satisfy all of the MUST and REQUIRED sections of the message.

Copyright © OASIS® 1993–2008. 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® 1993–2008. All Rights Reserved.

2905 recipient doesn't accept, then the issuer SHUOLD reject the request. Note that the issuer need not return
2906 the key provided by the requestor.

2907

2908 The following illustrates the syntax of a request for a custom security token and includes a secret that is
2909 to be used for the key. In this example the entropy is encrypted for the issuer (if transport security was
2910 used for confidentiality then the `<wst:Entropy>` element would contain a `<wst:BinarySecret>`
2911 element):

```
2912         <wst:RequestSecurityToken xmlns:wst="...”>
2913         <wst:TokenType>
2914             http://example.org/mySpecialToken
2915         </wst:TokenType>
2916         <wst:RequestType>
2917             http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
2918         </wst:RequestType>
2919         <wst:Entropy>
2920             <xenc:EncryptedData xmlns:xenc="...">...</xenc:EncryptedData>
2921         </wst:Entropy>
2922     </wst:RequestSecurityToken>
```

2923 A.3 Issuer-Provided Keys

2924 If a requestor fails to provide key material, then issued proof-of-possession tokens contain an issuer-
2925 provided secret that is encrypted for the requestor (either using `<xenc:EncryptedKey>` or by using a
2926 transport security).

2927

2928 The following illustrates the syntax of a token being returned with an associated proof-of-possession
2929 token that is encrypted using the requestor's public key.

```
2930     <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
2931     <wst:RequestSecurityTokenResponse>
2932         <wst:RequestedSecurityToken>
2933             <xyz:CustomToken xmlns:xyz="...">
2934                 ...
2935             </xyz:CustomToken>
2936         </wst:RequestedSecurityToken>
2937         <wst:RequestedProofToken>
2938             <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">
2939                 ...
2940             </xenc:EncryptedKey>
2941         </wst:RequestedProofToken>
2942     </wst:RequestSecurityTokenResponse>
2943   </wst:RequestSecurityTokenResponseCollection>
```

2944 A.4 Composite Keys

2945 The safest form of key exchange/generation is when both the requestor and the issuer contribute to the
2946 key material. In this case, the request sends encrypted key material. The issuer then returns additional
2947 encrypted key material. The actual secret (key) is computed using a function of the two pieces of data.
2948 Ideally this secret is never used and, instead, keys derived are used for message protection.

2949

2950 The following example illustrates a server, having received a request with requestor entropy returning its
2951 own entropy, which is used in conjunction with the requestor's to generate a key. In this example the
2952 entropy is not encrypted because the transport is providing confidentiality (otherwise the
2953 `<wst:Entropy>` element would have an `<xenc:EncryptedData>` element).

Copyright © OASIS® 1993–2008. All Rights Reserved.

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

A.5 Key Transfer and Distribution

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

A.5.1 Direct Key Transfer

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

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

```
<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.5.2 Brokered Key Distribution

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

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

Copyright © OASIS® 1993–2008. All Rights Reserved.

```
3003 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
3004   <wst:RequestSecurityTokenResponse>
3005       <wst:RequestedSecurityToken>
3006           <xyz:CustomToken xmlns:xyz="...">
3007               ...
3008               <xenc:EncryptedKey xmlns:xenc="...">
3009                   ...
3010               </xenc:EncryptedKey>
3011               ...
3012           </xyz:CustomToken>
3013       </wst:RequestedSecurityToken>
3014       <wst:RequestedProofToken>
3015           <xenc:EncryptedKey Id="newProof">
3016               ...
3017           </xenc:EncryptedKey>
3018       </wst:RequestedProofToken>
3019   </wst:RequestSecurityTokenResponse>
3020 </wst:RequestSecurityTokenResponseCollection>
```

3021 ### A.5.3 Delegated Key Transfer

3022 Key transfer can also take the form of delegation. That is, one party transfers the right to use a key
3023 without actually transferring the key. In such cases, a delegation token, e.g. XrML, is created that
3024 identifies a set of rights and a delegation target and is secured by the delegating party. That is, one key
3025 indicates that another key can use a subset (or all) of its rights. The delegate can provide this token and
3026 prove itself (using its own key – the delegation target) to a service. The service, assuming the trust
3027 relationships have been established and that the delegator has the right to delegate, can then authorize
3028 requests sent subject to delegation rules and trust policies.

3029

3030 In this example a custom token is issued from party A to party B. The token indicates that B (specifically
3031 B's key) has the right to submit purchase orders. The token is signed using a secret key known to the
3032 target service T and party A (the key used to ultimately authorize the requests that B makes to T), and a
3033 new session key that is encrypted for T. A proof-of-possession token is included that contains the
3034 session key encrypted for B. As a result, B is *effectively* using A's key, but doesn't actually know the key.

```
3035 <wst:RequestSecurityTokenResponseCollection xmlns:wst="...”>
3036   <wst:RequestSecurityTokenResponse>
3037       <wst:RequestedSecurityToken>
3038           <xyz:CustomToken xmlns:xyz="...">
3039               ...
3040               <xyz:DelegateTo>B</xyz:DelegateTo>
3041               <xyz:DelegateRights>
3042                   SubmitPurchaseOrder
3043               </xyz:DelegateRights>
3044               <xenc:EncryptedKey xmlns:xenc="...">
3045                   ...
3046               </xenc:EncryptedKey>
3047               <ds:Signature xmlns:ds="...">...</ds:Signature>
3048               ...
3049           </xyz:CustomToken>
3050       </wst:RequestedSecurityToken>
3051       <wst:RequestedProofToken>
3052           <xenc:EncryptedKey xmlns:xenc="..." Id="newProof">
3053               ...
3054           </xenc:EncryptedKey>
3055       </wst:RequestedProofToken>
3056   </wst:RequestSecurityTokenResponse>
3057 </wst:RequestSecurityTokenResponseCollection>
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

3058

A.5.4 Authenticated Request/Reply Key Transfer

3059 In some cases the RST/RSTR mechanism is not used to transfer keys because it is part of a simple
3060 request/reply. However, there may be a desire to ensure mutual authentication as part of the key
3061 transfer. The mechanisms of [WS-Security] can be used to implement this scenario.

3062

3063 Specifically, the sender wishes the following:

3064 - Transfer a key to a recipient that they can use to secure a reply
3065 - Ensure that only the recipient can see the key
3066 - Provide proof that the sender issued the key

3067

3068 This scenario could be supported by encrypting and then signing. This would result in roughly the
3069 following steps:

3070 1. Encrypt the message using a generated key
3071 2. Encrypt the key for the recipient
3072 3. Sign the encrypted form, any other relevant keys, and the encrypted key

3073

3074 However, if there is a desire to sign prior to encryption then the following general process is used:

3075 1. Sign the appropriate message parts using a random key (or ideally a key derived from a random
3076 key)
3077 2. Encrypt the appropriate message parts using the random key (or ideally another key derived from
3078 the random key)
3079 3. Encrypt the random key for the recipient
3080 4. Sign just the encrypted key

3081

3082 This would result in a <wsse:Security> header that looks roughly like the following:

```
3083        <wsse:Security xmlns:wsse="..." xmlns:wsu="..."
3084             xmlns:ds="..." xmlns:xenc="...">
3085         <wsse:BinarySecurityToken wsu:Id="myToken">
3086             ...
3087         </wsse:BinarySecurityToken>
3088         <ds:Signature>
3089             ...signature over #secret using token #myToken...
3090         </ds:Signature>
3091         <xenc:EncryptedKey Id="secret">
3092             ...
3093         </xenc:EncryptedKey>
3094         <xenc:RefrenceList>
3095             ...manifest of encrypted parts using token #secret...
3096         </xenc:RefrenceList>
3097         <ds:Signature>
3098             ...signature over key message parts using token #secret...
3099         </ds:Signature>
3100     </wsse:Security>
```

3101

3102 As well, instead of an <xenc:EncryptedKey> element, the actual token could be passed using
3103 <xenc:EncryptedData>. The result might look like the following:

```
3104     <wsse:Security xmlns:wsse="..." xmlns:wsu="..."
3105             xmlns:ds="..." xmlns:xenc="...">
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

```
3106        <wsse:BinarySecurityToken wsu:Id="myToken">
3107            ...
3108        </wsse:BinarySecurityToken>
3109        <ds:Signature>
3110            ...signature over #secret or #Esecret using token #myToken...
3111        </ds:Signature>
3112        <xenc:EncryptedData Id="Esecret”>
3113            ...Encrypted version of a token with Id="secret"...
3114        </xenc:EncryptedData>
3115        <xenc:RefrenceList>
3116            ...manifest of encrypted parts using token #secret...
3117        </xenc:RefrenceList>
3118        <ds:Signature>
3119            ...signature over key message parts using token #secret...
3120        </ds:Signature>
3121    </wsse:Security>
```

3122 ## A.6 Perfect Forward Secrecy

3123 In some situations it is desirable for a key exchange to have the property of perfect forward secrecy. This
3124 means that it is impossible to reconstruct the shared secret even if the private keys of the parties are
3125 disclosed.

3126

3127 The most straightforward way to attain perfect forward secrecy when using asymmetric key exchange is
3128 to dispose of one's key exchange key pair periodically (or even after every key exchange), replacing it
3129 with a fresh one. Of course, a freshly generated public key must still be authenticated (using any of the
3130 methods normally available to prove the identity of a public key's owner).

3131

3132 The perfect forward secrecy property MAY be achieved by specifying a `<wst:entropy>` element that
3133 contains an `<xenc:EncryptedKey>` that is encrypted under a public key pair created for use in a single
3134 key agreement. The public key does not require authentication since it is only used to provide additional
3135 entropy. If the public key is modified, the key agreement will fail. Care should be taken, when using this
3136 method, to ensure that the now-secret entropy exchanged via the `<wst:entropy>` element is not
3137 revealed elsewhere in the protocol (since such entropy is often assumed to be publicly revealed plaintext,
3138 and treated accordingly).

3139

3140 Although any public key scheme might be used to achieve perfect forward secrecy (in either of the above
3141 methods) it is generally desirable to use an algorithm that allows keys to be generated quickly. The Diffie-
3142 Hellman key exchange is often used for this purpose since generation of a key only requires the
3143 generation of a random integer and calculation of a single modular exponent.

Copyright © OASIS® 1993–2008. 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"
        xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
>
<!-- 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="RequestSecurityTokenCollectionMsg">
        <wsdl:part name="requestCollection"
                element="wst:RequestSecurityTokenCollection"/>
    </wsdl:message>
    <wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
        <wsdl:part name="responseCollection"
                element="wst:RequestSecurityTokenResponseCollection"/>
    </wsdl:message>

    <!-- This portType an example of a Requestor (or other) endpoint that
         Accepts SOAP-based challenges from a Security Token Service -->
    <wsdl:portType name="WSSecurityRequestor">
      <wsdl:operation name="Challenge">
       <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
       <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
      </wsdl:operation>
    </wsdl:portType>

    <!-- This portType is an example of an STS supporting full protocol -->
    <wsdl:portType name="SecurityTokenService">
      <wsdl:operation name="Cancel">
        <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
trust/200512/RST/Cancel" message="tns:RequestSecurityTokenMsg"/>
        <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
trust/200512/RSTR/CancelFinal" message="tns:RequestSecurityTokenResponseMsg"/>
      </wsdl:operation>
      <wsdl:operation name="Issue">
```


Copyright © OASIS® 1993–2008. All Rights Reserved.

```
3202          <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3203  trust/200512/RST/Issue" message="tns:RequestSecurityTokenMsg"/>
3204          <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3205  trust/200512/RSTRC/IssueFinal"
3206  message="tns:RequestSecurityTokenResponseCollectionMsg"/>
3207        </wsdl:operation>
3208        <wsdl:operation name="Renew">
3209          <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3210  trust/200512/RST/Renew" message="tns:RequestSecurityTokenMsg"/>
3211          <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3212  trust/200512/RSTR/RenewFinal" message="tns:RequestSecurityTokenResponseMsg"/>
3213        </wsdl:operation>
3214        <wsdl:operation name="Validate">
3215          <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3216  trust/200512/RST/Validate" message="tns:RequestSecurityTokenMsg"/>
3217          <wsdl:output wsam:Action=http://docs.oasis-open.org/ws-sx/ws-
3218  trust/200512/RSTR/ValidateFinal
3219  message="tns:RequestSecurityTokenResponseMsg"/>
3220        </wsdl:operation>
3221        <wsdl:operation name="KeyExchangeToken">
3222          <wsdl:input wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3223  trust/200512/RST/KET" message="tns:RequestSecurityTokenMsg"/>
3224          <wsdl:output wsam:Action="http://docs.oasis-open.org/ws-sx/ws-
3225  trust/200512/RSTR/KETFinal" message="tns:RequestSecurityTokenResponseMsg"/>
3226        </wsdl:operation>
3227        <wsdl:operation name="RequestCollection">
3228          <wsdl:input message="tns:RequestSecurityTokenCollectionMsg"/>
3229          <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg"/>
3230        </wsdl:operation>
3231      </wsdl:portType>
3232  
3233      <!-- This portType is an example of an endpoint that accepts
3234           Unsolicited RequestSecurityTokenResponse messages -->
3235      <wsdl:portType name="SecurityTokenResponseService">
3236        <wsdl:operation name="RequestSecurityTokenResponse">
3237          <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
3238        </wsdl:operation>
3239      </wsdl:portType>
3240  
3241  </wsdl:definitions>
3242  
```


Copyright © OASIS® 1993–2008. 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, Oracle Corporation
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

Copyright © OASIS® 1993–2008. All Rights Reserved.

3290 Paul Leach, Microsoft
3291 Richard Levinson, Computer Associates
3292 John Linn, RSA Security
3293 Michael McIntosh, IBM
3294 Steve Millet, Microsoft
3295 Birgit Pfitzmann, IBM
3296 Fumiko Satoh, IBM
3297 Keith Stobie, Microsoft
3298 T.R. Vishwanath, Microsoft
3299 Richard Ward, Microsoft
3300 Hervey Wilson, Microsoft
3301

3302 **TC Members during the development of this specification:**
3303 Don Adams, Tibco Software Inc.
3304 Jan Alexander, Microsoft Corporation
3305 Steve Anderson, BMC Software
3306 Donal Arundel, IONA Technologies
3307 Howard Bae, Oracle Corporation
3308 Abbie Barbir, Nortel Networks Limited
3309 Charlton Barreto, Adobe Systems
3310 Mighael Botha, Software AG, Inc.
3311 Toufic Boubez, Layer 7 Technologies Inc.
3312 Norman Brickman, Mitre Corporation
3313 Melissa Brumfield, Booz Allen Hamilton
3314 Lloyd Burch, Novell
3315 Geoff Bullen, Microsoft Corporation
3316 Scott Cantor, Internet2
3317 Greg Carpenter, Microsoft Corporation
3318 Steve Carter, Novell
3319 Ching-Yun (C.Y.) Chao, IBM
3320 Martin Chapman, Oracle Corporation
3321 Kate Cherry, Lockheed Martin
3322 Henry (Hyenvui) Chung, IBM
3323 Luc Clement, Systinet Corp.
3324 Paul Cotton, Microsoft Corporation
3325 Glen Daniels, Sonic Software Corp.
3326 Peter Davis, Neustar, Inc.
3327 Martijn de Boer, SAP AG
3328 Duane DeCouteau, Veterans Health Administration
3329 Werner Dittmann, Siemens AG
3330 Abdeslem DJAOUI, CCLRC-Rutherford Appleton Laboratory
3331 Fred Dushin, IONA Technologies
3332 Petr Dvorak, Systinet Corp.
3333 Colleen Evans, Microsoft Corporation

Copyright © OASIS® 1993–2008. All Rights Reserved.

3334 Ruchith Fernando, WSO2
3335 Mark Fussell, Microsoft Corporation
3336 Vijay Gajjala, Microsoft Corporation
3337 Marc Goodner, Microsoft Corporation
3338 Hans Granqvist, VeriSign
3339 Martin Gudgin, Microsoft Corporation
3340 Tony Gullotta, SOA Software Inc.
3341 Jiandong Guo, Sun Microsystems
3342 Phillip Hallam-Baker, VeriSign
3343 Patrick Harding, Ping Identity Corporation
3344 Heather Hinton, IBM
3345 Frederick Hirsch, Nokia Corporation
3346 Jeff Hodges, Neustar, Inc.
3347 Will Hopkins, Oracle Corporation
3348 Alex Hristov, Otecia Incorporated
3349 John Hughes, PA Consulting
3350 Diane Jordan, IBM
3351 Venugopal K, Sun Microsystems
3352 Chris Kaler, Microsoft Corporation
3353 Dana Kaufman, Forum Systems, Inc.
3354 Paul Knight, Nortel Networks Limited
3355 Ramanathan Krishnamurthy, IONA Technologies
3356 Christopher Kurt, Microsoft Corporation
3357 Kelvin Lawrence, IBM
3358 Hubert Le Van Gong, Sun Microsystems
3359 Jong Lee, Oracle Corporation
3360 Rich Levinson, Oracle Corporation
3361 Tommy Lindberg, Dajeil Ltd.
3362 Mark Little, JBoss Inc.
3363 Hal Lockhart, Oracle Corporation
3364 Mike Lyons, Layer 7 Technologies Inc.
3365 Eve Maler, Sun Microsystems
3366 Ashok Malhotra, Oracle Corporation
3367 Anand Mani, CrimsonLogic Pte Ltd
3368 Jonathan Marsh, Microsoft Corporation
3369 Robin Martherus, Oracle Corporation
3370 Miko Matsumura, Infravio, Inc.
3371 Gary McAfee, IBM
3372 Michael McIntosh, IBM
3373 John Merrells, Sxip Networks SRL
3374 Jeff Mischkinsky, Oracle Corporation
3375 Prateek Mishra, Oracle Corporation

Copyright © OASIS® 1993–2008. All Rights Reserved.

3376 Bob Morgan, Internet2
3377 Vamsi Motukuru, Oracle Corporation
3378 Raajmohan Na, EDS
3379 Anthony Nadalin, IBM
3380 Andrew Nash, Reactivity, Inc.
3381 Eric Newcomer, IONA Technologies
3382 Duane Nickull, Adobe Systems
3383 Toshihiro Nishimura, Fujitsu Limited
3384 Rob Philpott, RSA Security
3385 Denis Pilipchuk, Oracle Corporation
3386 Darren Platt, Ping Identity Corporation
3387 Martin Raepple, SAP AG
3388 Nick Ragouzis, Enosis Group LLC
3389 Prakash Reddy, CA
3390 Alain Regnier, Ricoh Company, Ltd.
3391 Irving Reid, Hewlett-Packard
3392 Bruce Rich, IBM
3393 Tom Rutt, Fujitsu Limited
3394 Maneesh Sahu, Actional Corporation
3395 Frank Siebenlist, Argonne National Laboratory
3396 Joe Smith, Apani Networks
3397 Davanum Srinivas, WSO2
3398 David Staggs, Veterans Health Administration
3399 Yakov Sverdlov, CA
3400 Gene Thurston, AmberPoint
3401 Victor Valle, IBM
3402 Asir Vedamuthu, Microsoft Corporation
3403 Greg Whitehead, Hewlett-Packard
3404 Ron Williams, IBM
3405 Corinna Witt, Oracle Corporation
3406 Kyle Young, Microsoft Corporation
3407

Copyright © OASIS® 1993–2008. All Rights Reserved.