Web Services Secure Conversation Language (WS-SecureConversation)

Version 1.1
May 2004

Authors

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

Copyright Notice

Abstract

This specification defines extensions that build on [WS-Security] and [WS-Trust] to provide secure communication across one or more messages. Specifically, this specification defines mechanisms for establishing and sharing security contexts, and deriving keys from established security contexts (or any shared secret).

Modular Architecture

By using the XML, SOAP, and WSDL extensibility models, the WS* specifications are designed to be composed with each other to provide a rich Web services environment. WS-SecureConversation by itself does not provide a complete security solution for Web services. WS-SecureConversation is a building block that is used in conjunction with other Web service and application-specific protocols to accommodate a wide variety of security models.

Status

This WS-SecureConversation Specification is a revised public draft release and is provided for review and evaluation only. BEA, Computer Associates, IBM, Layer7, Microsoft, Netegrity, Oblix, OpenNetwork, Ping Identity, Reactivity, RSA Security, VeriSign, and Westbridge hope to solicit your contributions and suggestions in the near future. BEA, Computer Associates, IBM, Layer7, Microsoft, Netegrity, Oblix,

OpenNetwork, Ping Identity, Reactivity, RSA Security, VeriSign, and Westbridge make no warrantees or representations regarding the specifications in any manner whatsoever.

Table of Contents

1. Overview

The mechanisms defined in [WS-Security] provide the basic mechanisms on top of which secure messaging semantics can be defined for multiple message exchanges. This specification defines extensions to allow security context establishment and sharing, and session key derivation. This allows contexts to be established and potentially more efficient keys or new key material to be exchanged, thereby increasing the overall performance and security of the subsequent exchanges.

The [WS-Security] specification focuses on the message authentication model. This approach, while useful in many situations, is subject to several forms of attack (see Security Considerations section of [WS-Security] specification).

Accordingly, this specification introduces a security context and its usage. The context authentication model authenticates a series of messages thereby addressing these shortcomings, but requires additional communications if authentication happens prior to normal application exchanges.

The security context is defined as a new [WS-Security] token type that is obtained using a binding of [WS-Trust].

Compliant services are NOT REQUIRED to implement everything defined in this specification. However, if a service implements an aspect of the specification, it MUST comply with the requirements specified (e.g. related "MUST" statements).

Sections 1, 8, 9, and 10 are non-normative.

1.1 Goals and Non-Goals

The primary goals of this specification are:

- Define how security contexts are established
- Describe how security contexts are amended
- Specify how derived keys are computed and passed

It is not a goal of this specification to define how trust is established or determined.

This specification is intended to provide a flexible set of mechanisms that can be used to support a range of security protocols. Some protocols may require separate mechanisms or restricted profiles of this specification.

1.2 Requirements

The following list identifies the key driving requirements:

- Derived keys and per-message keys
- Extensible security contexts

2. Notations and Terminology

This section specifies the notations, namespaces, and terminology used in this specification.

2.1 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 [RFC2396].

2.2 Namespace

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

```
http://schemas.xmlsoap.org/ws/2004/04/sc
```

The following namespaces are used in this document:

Prefix	Namespace

S11	http://schemas.xmlsoap.org/soap/envelope/
S12	http://www.w3.org/2003/05/soap-envelope
wsu	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd
wsse	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd
wst	http://schemas.xmlsoap.org/ws/2004/04/trust
wsc	http://schemas.xmlsoap.org/ws/2004/04/sc
wsp	http://schemas.xmlsoap.org/ws/2002/12/policy
ds	http://www.w3.org/2000/09/xmldsig#
xenc	http://www.w3.org/2001/04/xmlenc#

2.3 Schema File

The schema for this specification can be located at:

```
http://schemas.xmlsoap.org/ws/2004/04/sc
```

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).

2.4 Terminology

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

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.

Security Context – A *security context* is an abstract concept that refers to an established authentication state and negotiated key(s) that may have additional security-related properties.

Security Context Token – A *security context token (SCT)* is a wire representation of that security context abstract concept, which allows a context to be named by a URI and used with [WS-Security].

Signed Security Token – A *signed security token* is a security token that is asserted and 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).

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 token. 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.

Request Security Token (RST) – A *RST* is a message sent to a security token service to request a security token.

Request Security Token Response (RSTR) – A *RSTR* is a response to a request for a security token. In many cases this is a direct response from a security token service to a requestor after receiving an RST message. However, in multi-exchange scenarios the requestor and security token service may exchange multiple RSTR messages before the security token service issues a final RSTR message.

3. Security Context Token (SCT)

While message authentication is useful for simple or one-way messages, parties that wish to exchange multiple messages typically establish a secure security context in which to exchange multiple messages. A security context is shared among the communicating parties for the lifetime of a communications session.

In this specification, a security context is represented by the `<wsc:SecurityContextToken>` security token. In the [WS-Security] and [WS-Trust] framework, the following URI is used to represent the token type:

```
http://schemas.xmlsoap.org/ws/2004/04/security/sc/sct
```

The SCT token does not support references to it using key identifiers or key names. All references MUST either use an ID (to a `wsu:Id` attribute) or a `<wsse:Reference>` to the `<wsc:Identifier>` element.

Once the context and secret have been established (authenticated), the mechanisms described in Derived Keys can be used to compute derived keys for each key usage in the secure context.

The following represents an overview of the syntax of the `<wsc:SecurityContextToken>` element. It should be noted that this token supports an open content model to allow context-specific data to be passed.

```
    <SecurityContextToken wsu:Id="...">
        <wsc:Identifier>...</wsc:Identifier>
    </SecurityContextToken>
```

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

/SecurityContextToken
: This element is a security token that describes a security context.

/SecurityContextToken/Identifier
: This required element identifies the security context using a URI. Each security context URI MUST be unique to both the sender and recipient. It is RECOMMENDED that the value be globally unique in time and space.

/SecurityContextToken/@wsu:Id
: This optional attribute specifies a string label for this element.

/SecurityContextToken/@{any}
: This is an extensibility mechanism to allow additional attributes, based on schemas, to be added to the element.

/SecurityContextToken/{any}
: This is an extensibility mechanism to allow additional elements (arbitrary content) to be used.

The <wsc:SecurityContextToken> token elements MUST be preserved. That is, whatever elements contained within the tag on creation MUST be preserved wherever the token is used. A consumer of a <wsc:SecurityContextToken> token MAY extend the token by appending information. Consequently producers of <wsc:SecurityContextToken> tokens should consider this fact when processing previously generated tokens. A service consuming (processing) a <wsc:SecurityContextToken> token MAY fault if it discovers an element inside the token that it doesn't understand, or it MAY ignore it. The fault code wsc:UnsupportedContextToken is RECOMMENDED if a fault is raised. The behavior is specified by the services policy. Care should be taken when adding information to tokens to ensure that relying parties can ensure the information has not been altered since the SCT definition does not require a specific way to secure its contents (which as noted above can be appended to).

Security contexts, like all security tokens, can be referenced using the mechanisms described in [WS-Security] (the <wsse:SecurityTokenReference> element referencing the wsu:Id attribute relative to the XML base document or referencing using the <wsc:Identifier> element's URI). When a token is referenced, the associated key is used. If a token provides multiple keys then specific bindings and profiles must describe how to reference the separate keys.

The following sample message illustrates the use of a security context token. In this example a context has been established and the secret is known to both parties. This secret is used to sign the message body.

```
(001) <?xml version="1.0" encoding="utf-8"?>
(002) <S11:Envelope xmlns:S11="..." xmlns:ds="..." xmlns:wsse="..."
              xmlns:wst="..." xmlns:wsc="...">
(003)     <S11:Header>
```

```
(004)         ...
(005)         <wsse:Security>
(006)             <wsc:SecurityContextToken wsu:Id="MyID">
(007)                 <wsc:Identifier>uuid:...</wsc:Identifier>
(008)             </wsc:SecurityContextToken>
(009)             <ds:Signature>
(010)                 ...
(011)                 <ds:KeyInfo>
(012)                     <wsse:SecurityTokenReference>
(013)                         <wsse:Reference URI="#MyID"/>
(014)                     </wsse:SecurityTokenReference>
(015)                 </ds:KeyInfo>
(016)             </ds:Signature>
(017)         </wsse:Security>
(018)     </S11:Header>
(019)     <S11:Body wsu:Id="MsgBody">
(020)         <tru:StockSymbol
                  xmlns:tru="http://fabrikam123.com/payloads">
              QQQ
           </tru:StockSymbol>
(021)     </S11:Body>
(022) </S11:Envelope>
```

Let's review some of the key sections of this example:

Lines (003)-(018) contain the SOAP message headers.

Lines (005)-(017) represent the <wsse:Security> header block. This contains the security-related information for the message.

Lines (006)-(008) specify a security token that is associated with the message. In this case it is a security context token. Line (007) specifies the unique ID of the context.

Lines (009)-(016) specify the digital signature. In this example, the signature is based on the security context (specifically the secret/key associated with the context). Line (010) represents the typical contents of an XML Digital Signature which, in this case, references the body and potentially some of the other headers expressed by line (004).

Lines (012)-(014) indicate the key that was used for the signature. In this case, it is the security context token included in the message. Line (013) provides a URI link to the security context token specified in Lines (006)-(008).

The body of the message is represented by lines (019)-(021).

4. Establishing Security Contexts

A security context needs to be created and shared by the communicating parties before being used. This specification defines three different ways of establishing a security context among the parties of a secure communication.

Security context token created by a security token service – The context initiator asks a security token service to create a new security context token. The newly created security context token is distributed to the parties through the mechanisms defined here and in [WS-Trust]. For this scenario the initiating party sends a `<wst:RequestSecurityToken>` request to the token service and a `<wst:RequestSecurityTokenResponse>` is returned. The response contains a `<wst:RequestedSecurityToken>` containing (or pointing to) the new security context token and a `<wst:RequestedProofToken>` pointing to the "secret" for the returned context. The requestor then uses the security context token (with [WS-Security]) when securing messages to applicable services.

Security context token created by one of the communicating parties and propagated with a message – The initiator creates a security context token and sends it to the other parties on a message using the mechanisms described in this specification and in [WS-Trust]. This model works when the sender is trusted to always create a new security context token. For this scenario the initiating party creates a security context token and issues a signed unsolicited `<wst:RequestSecurityTokenResponse>` to the other party. The message contains a `<wst:RequestedSecurityToken>` containing (or pointing to) the new security context token and a `<wst:RequestedProofToken>` pointing to the "secret" for the security context token. The recipient can then choose whether or not to accept the security context token. As described in [WS-Trust], the `<wst:RequestSecurityTokenResponse>` element MAY be in the body or inside a header block. It should be noted that unless delegation tokens are used, this scenario requires that parties trust each other to share a secret key (and non-repudiation is probably not possible). As receipt of these messages may be expensive, and because a recipient may receive multiple messages, the *…*/RequestSecurityTokenResponse/*@Context* attribute in [WS-Trust] allows the initiator to specify a URI to indicate the intended usage (allowing processing to be optimized).

Security context token created through negotiation/exchanges – When there is a need to negotiate or participate in a sequence of message exchanges among the participants on the contents of the security context token, such as the shared secret, this specification allows the parties to exchange data to establish a security context. For this scenario the initiating party sends a `<wst:RequestSecurityToken>` request to the other party and a `<wst:RequestSecurityTokenResponse>` is returned. It is RECOMMENDED that the framework described in [WS-Trust] be used; however, the type of exchange will likely vary. If appropriate, the basic challenge-response definition in [WS-Trust] is RECOMMENDED. Ultimately (if successful), a final response contains a `<wst:RequestedSecurityToken>` containing (or pointing to) the new security context and a `<wst:RequestedProofToken>` pointing to the "secret" for the context.

If an SCT is received, but the key sizes are not supported, then a fault SHOULD be generated using the `wsc:UnsupportedContextToken` fault code unless another more specific fault code is available.

4.1 SCT Binding of WS-Trust

This binding describes how to use [WS-Trust] to request and return SCTs. This binding builds on the issuance binding for [WS-Trust] (note that other sections of this specification define new separate bindings of [WS-Trust]). Consequently, aspects of the issuance binding apply to this binding unless otherwise stated. For example, the token request type is the same as in the issuance binding.

When requesting and returning security context tokens the following Action URIs are used (note that a specialized action is used here because of the specialized semantics of SCTs):

```
    http://schemas.xmlsoap.org/ws/2004/04/security/trust/RST/SCT
    http://schemas.xmlsoap.org/ws/2004/04/security/trust/RSTR/SCT
```

As with all token services, the options supported may be limited. This is especially true of SCTs because the issuer may only be able to issue tokens for itself and quite often will only support a specific set of algorithms and parameters as expressed in its policy.

SCTs are not required to have lifetime semantics. That is, some SCTs may have specific lifetimes and others may be bound to other resources rather than have their own lifetimes.

Since the SCT binding builds on the issuance binding, it allows the optional extensions defined for the issuance binding including the use of exchanges. Subsequent profiles MAY restrict the extensions and types and usage of exchanges.

4.2 SCT Request Example

The following illustrates a request for a security context token from a security token service. In this example the key is encrypted for the recipient (security token service) using the token service's X.509 certificate as per XML Encryption. The encrypted data (using the encrypted key) contains a <wsse:UsernameToken> token that the recipient uses to authorize the request. The request is secured (integrity) using the X.509 certificate of the requestor. The response encrypts the proof information using the requestor's X.509 certificate and secures the message (integrity) using the token service's X.509 certificate. Note that the details of XML Signature and XML Encryption have been omitted; refer to [WS-Security] for additional details. It should be noted that if the requestor doesn't have an X.509 this scenario could be achieved using a TLS connection or by creating an ephemeral key.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:wsc="..." xmlns:wst="..." xmlns:xenc="...">>
    <S11:Header>
        ...
        <wsse:Security>
            <xenc:EncryptedKey>
                ...
            </xenc:EncryptedKey>
            <xenc:EncryptedData Id="encUsernameToken">
                ... encrypted username token (whose id is myToken) ...
```

```
            </xenc:EncryptedData>
            <ds:Signature xmlns:ds="...">
                ...
            </ds:Signature>
        </wsse:Security>
        ...
    </S11:Header>
    <S11:Body wsu:Id="req">
        <wst:RequestSecurityToken>
            <wst:TokenType>
                http://schemas.xmlsoap.org/ws/2004/04/security/sc/sct
            </wst:TokenType>
            <wst:RequestType>
                http://schemas.xmlsoap.org/ws/2004/04/security/trust/Issue
            </wst:RequestType>
            <wst:Base>
                <wsse:Reference URI="#myToken"/>
            </wst:Base>
        </wst:RequestSecurityToken>
    </S11:Body>
</S11:Envelope>

<S11:Envelope xmlns:S11="..." xmlns:wsse="... " xmlns:wsu="..."
        xmlns:wst="..." xmlns:wsc="..." xmlns:xenc="...">>
    <S11:Header>
        ...
    </S11:Header>
    <S11:Body>
        <wst:RequestSecurityTokenResponse>
            <wst:RequestedSecurityToken>
                <wsc:SecurityContextToken>
                    <wsc:Identifier>uuid:...</wsc:Identifier>
                </wsc:SecurityContextToken>
            </wst:RequestedSecurityToken>
            <wst:RequestedProofToken>
                <xenc:EncryptedKey Id="newProof">
```

```
                    ...
                </xenc:EncryptedKey>
            </wst:RequestedProofToken>
        </wst:RequestSecurityTokenResponse>
    </S11:Body>
</S11:Envelope>
```

4.3 SCT Propagation Example

The following illustrates propagating a context to another party.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:wst="..." xmlns:wsc="..." xmlns:xenc="...">>
    <S11:Header>
        ...
    </S11:Header>
    <S11:Body>
        <wst:RequestSecurityTokenResponse>
            <wst:RequestedSecurityToken>
                <wsc:SecurityContextToken>
                    <wsc:Identifier>uuid:...</wsc:Identifier>
                </wsc:SecurityContextToken>
            </wst:RequestedSecurityToken>
            <wst:RequestedProofToken>
                <xenc:EncryptedKey Id="newProof">
                    ...
                </xenc:EncryptedKey>
            </wst:RequestedProofToken>
        </wst:RequestSecurityTokenResponse>
    </S11:Body>
</S11:Envelope>
```

5. Amending Contexts

When an SCT is created, a set of claims is associated with it. There are times when an existing SCT needs to be amended to carry additional claims (note that the decision as to who is authorized to amend a context is a service-specific decision). This is done using the SCT Amend binding. In such cases an explicit request is made to amend the claims associated with an SCT. It should be noted that using the mechanisms described in [WS-Trust], an issuer MAY, at any time, return an amended SCT by issuing an unsolicited (not explicitly requested) SCT inside an RSTR (either as a separate message or in a header).

The following Action URIs are used with this binding:

```
http://schemas.xmlsoap.org/ws/2004/04/security/trust/RST/SCT-Amend
http://schemas.xmlsoap.org/ws/2004/04/security/trust/RSTR/SCT-Amend
```

This binding allows optional extensions but DOES NOT allow key semantics to be altered.

Additional claims are indicated by providing signatures over the SCT token within the message proving additional security tokens that carry the claims to augment the context.

This binding uses the request type from the issuance binding.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:xenc="..." xmlns:wst="..." xmlns:wsc="...">
    <S11:Header>
        ...
        <wsa:Action xmlns:wsa="...">
        http://schemas.xmlsoap.org/ws/2004/04/security/trust/RST/SCT-Amend
        </wsa:Action>
            ...
        <wsse:Security>
            <xx:CustomToken wsu:Id="cust" xmlns:xx="...">
                ...
            </xx:CustomToken>
            <ds:Signature xmlns:ds="...">
                ...signature over #sig1 using #cust...
            </ds:Signature>
            <wsc:SecurityContextToken wsu:Id="sct">
                <wsc:Identifier>uuid:...UUID1...</wsc:Identifier>
            </wsc:SecurityContextToken>
           <ds:Signature xmlns:ds="..." ID="sig1">
               ...signature over body and key headers using #sct...
           </ds:Signature>
       </wsse:Security>
       ...
    </S11:Header>
    <S11:Body wsu:Id="req">
        <wst:RequestSecurityToken>
            <wst:RequestType>
                http://schemas.xmlsoap.org/ws/2004/04/security/trust/Issue
            </wst:RequestType>
```

```
            <wst:Base>
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#sct"/>
                </wsse:SecurityTokenReference>
            </wst:Base>
            <wst:Supporting>
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#cust"/>
                </wsse:SecurityTokenReference>
            </wst:Supporting>
        </wst:RequestSecurityToken>
    </S11:Body>
</S11:Envelope>

<S11:Envelope xmlns:S11="..." xmlns:wst="..." xmlns:wsc="...">
    <S11:Header>
        ...
        <wsa:Action xmlns:wsa="...">
       http://schemas.xmlsoap.org/ws/2004/04/security/trust/RSTR/SCT-Amend
        </wsa:Action>
        ...
    </S11:Header>
    <S11:Body>
        <wst:RequestSecurityTokenResponse>
            <wst:RequestedSecurityToken>
                <wsc:SecurityContextToken>
                    <wsc:Identifier>uuid:...UUID1...</wsc:Identifier>
                </wsc:SecurityContextToken>
            </wst:RequestedSecurityToken>
        </wst:RequestSecurityTokenResponse>
    </S11:Body>
</S11:Envelope>
```

6. Deriving Keys

A security context token implies or contains a shared secret. This secret MAY be used for signing and/or encrypting messages, but it is RECOMMENDED that derived keys be used for signing and encrypting messages associated only with the security context.

Using a common secret, parties may define different key derivations to use. For example, four keys may be derived so that two parties can sign and encrypt using separate keys. In order to keep the keys fresh (prevent providing too much data for analysis), subsequent derivations may be used. We introduce the <wsc:DerivedKeyToken> token as a mechanism for indicating which derivation is being used within a given message.

The derived key mechanism can use different algorithms for deriving keys. The algorithm is expressed using a URI. This specification defines one such algorithm.

As well, while presented here using security context tokens, the <wsc:DerivedKeyToken> token can be used to derive keys from any security token that has a shared secret, key, or key material.

We use a subset of the mechanism defined for TLS in RFC 2246. Specifically, we use the P_SHA-1 function to generate a sequence of bytes that can be used to generate security keys. We refer to this algorithm as:

```
http://schemas.xmlsoap.org/ws/2004/04/security/sc/dk/p_sha1
```

This function is used with three values – *secret*, *label*, and *seed*. The secret is the shared secret that is exchanged (note that if two secrets were securely exchanged, possible as part of an initial exchange, they are concatenated in the order they were sent/received). Secrets are processed as octets representing their binary value (value prior to encoding). The label is the concatenation of the client's label and the service's label. These labels can be discovered in each party's policy (or specifically within a <wsc:DerivedKeyToken> token). Labels are processed as UTF-8 encoded octets. If either isn't specified in the policy, then a default value of "WS-SecureConversation" (represented as UTF-8 octets) is used. The seed is the concatenation of nonce values (if multiple were exchanged) that were exchanged (initiator + receiver). The nonce is processed as a binary octet sequence (the value prior to base64 encoding). The nonce seed is required, and MUST be generated by one or more of the communicating parties. The P_SHA-1 function has two parameters – *secret* and *value*. We concatenate the *label* and the *seed* to create the *value*. That is:

```
P_SHA1 (secret, label + seed)
```

At this point, both parties can use the P_SHA-1 function to generate shared keys as needed. For this protocol, we don't define explicit derivation uses.

The <wsc:DerivedKeyToken> element is used to indicate that the key for a specific reference is generated from the function. This is so that explicit security tokens, secrets, or key material need not be exchanged as often thereby increasing efficiency and overall scalability. However, parties MUST mutually agree on specific derivations (e.g. the first 128 bits is the client's signature key, the next 128 bits in the client's encryption key, and so on). The policy presents a method for specifying this information. The RECOMMENDED approach is to use separate nonces and have independently generated keys for signing and encrypting in each direction. Furthermore, it is RECOMMENDED that new keys be derived for each message (i.e., previous nonces are not re-used).

Once the parties determine a shared secret to use as the basis of a key generation sequence, an initial key is generated using this sequence. When a new key is required, a new <wsc:DerivedKeyToken> may be passed referencing the previously generated key. The recipient then knows to use the sequence to generate a new key, which will

match that specified in the security token. If both parties pre-agree on key sequencing, then additional token exchanges are not required.

For keys derived using a shared secret from a security context, the <wsse:SecurityTokenReference> element SHOULD be used to reference the <wsc:SecurityContextToken>. Basically, a signature or encryption references a <wsc:DerivedKeyToken> in the <wsse:Security> header that, in turn, references the <wsc:SecurityContextToken>.

Derived keys are expressed as security tokens. The following URI is used to represent the token type:

```
http://schemas.xmlsoap.org/ws/2004/04/security/sc/dk
```

The derived key token does not support references using key identifiers or key names. All references MUST use an ID (to a *wsu:Id* attribute) or a URI reference to the <wsc:Identifier> element in the SCT.

6.1 Syntax

The syntax for <wsc:DerivedKeyToken> is as follows:

```
<DerivedKeyToken wsu:Id="..." Algorithm="...">
    <wsse:SecurityTokenReference>...</wsse:SecurityTokenReference>
    <Properties>...</Properties>
    <Generation>...</Generation>
    <Offset>...</Offset>
    <Length>...</Length>
    <Label>...</Label>
    <Nonce>...</Nonce>
</DerivedKeyToken>
```

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

/DerivedKeyToken
: This specifies a key that is derived from a shared secret.

/DerivedKeyToken/@wsu:Id
: This optional attribute specifies an XML ID that can be used locally to reference this element.

/DerivedKeyToken/@Algorithm
: This optional URI attribute specifies key derivation algorithm to use. This specification predefines the P_SHA1 algorithm described above. If this attribute isn't specified, this algorithm is assumed.

/DerivedKeyToken/wsse:SecurityTokenReference
: This optional element is used to specify security context token, security token, or shared key/secret used for the derivation. If not specified, it is assumed that the recipient can determine the shared key from the message context. If the context cannot be determined, then a fault such as wsc:UnknownDerivationSource should be raised.

/DerivedKeyToken/Properties

This optional element allows metadata to be associated with this derived key. For example, if the <wsc:Name> property is defined, this derived key is given a URI name that can then be used as the source for other derived keys. The <wsc:Nonce> and <wsc:Label> elements can be specified as properties and indicate the nonce and label to use (defaults) for all keys derived from this key.

/DerivedKeyToken/Generation

If fixed-size keys (generations) are being generated, then this element can be used to specify which generation of the key to use. The value of this element is an unsigned long value indicating the generation number to use (beginning with zero). This element MUST NOT be used if the <wsc:Offset> element is specified. Specifying this element is equivalent to specifying the <wsc:Offset> and <wsc:Length> elements having multiplied out the values. That is, offset = (generation) * fixed_size and length = fixed_size.

/DerivedKeyToken/Offset

If fixed-size keys are not being generated, then the <wsc:Offset> and <wsc:Length> elements indicate where in the byte stream to find the generated key. This specifies the ordering (in bytes) of the generated output. The value of this element is an unsigned long value indicating the byte position (starting at 0). For example, 0 indicates the first byte of output and 16 indicates the 17th byte of generated output. This element MUST NOT be used if the <wsc:Generation> element is specified. It should be noted that not all algorithms will support the <wsc:Offset> and <wsc:Length> elements.

/DerivedKeyToken/Length

This element specifies the length (in bytes) of the derived key. This element can be specified in conjunction with <wsc:Offset> or <wsc:Generation>. If this isn't specified, it is assumed that the recipient knows the key size to use. The value of this element is an unsigned long value indicating the size of the key in bytes (e.g., 16).

/DerivedKeyToken/Label

If specified, this element defines a label that is used in the key derivation function for this derived key. If this isn't specified, it is assumed that the recipient knows the label to use. The string content of this element is UTF-8 encoded to obtain the label used in key derivation. Note that once a label is used for a derivation sequence, the same label SHOULD be used for all subsequent derivations.

/DerivedKeyToken/Nonce

If specified, this element specified a nonce that is used in the key derivation function for this derived key. If this isn't specified, it is assumed that the recipient knows the nonce to use. Note that once a nonce is used for a derivation sequence, the same nonce SHOULD be used for all subsequent derivations.

If additional information is not specified (such as explicit elements or policy), then the following defaults apply:

- The offset is 0
- The length is 32 bytes (256 bits)

It is RECOMMENDED that separate derived keys be used to strengthen the cryptography. If multiple keys are used, then care should be taken not to derive too many times and risk key attacks.

6.2 Examples

The following example illustrates a message sent using two derived keys, one for signing and one for encrypting:

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
        xmlns:xenc="..." xmlns:wsc="..." xmlns:ds="...">
    <S11:Header>
        <wsse:Security>
            <wsc:SecurityContextToken wsu:Id="ctx2">
                <wsc:Identifier>uuid:...UUID2...</wsc:Identifier>
            </wsc:SecurityContextToken>
            <wsc:DerivedKeyToken wsu:Id="dk2">
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#ctx2"/>
                </wsse:SecurityTokenReference>
            <wsc:Nonce>KJHFRE...</wsc:Nonce>
            </wsc:DerivedKeyToken>
            <xenc:ReferenceList>
                ...
                <ds:KeyInfo>
                    <wsse:SecurityTokenReference>
                        <wsse:Reference URI="#dk2"/>
                    </wsse:SecurityTokenReference>
                </ds:KeyInfo>
                    ...
            </xenc:ReferenceList>
            <wsc:SecurityContextToken wsu:Id="ctx1">
                <wsc:Identifier>uuid:...UUID1...</wsc:Identifier>
            </wsc:SecurityContextToken>
            <wsc:DerivedKeyToken wsu:Id="dk1">
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#ctx1"/>
                </wsse:SecurityTokenReference>
                <wsc:Nonce>KJHFRE...</wsc:Nonce>
            </wsc:DerivedKeyToken>
            <xenc:ReferenceList>
                ...
```

```
                <ds:KeyInfo>
                    <wsse:SecurityTokenReference>
                        <wsse:Reference URI="#dk1"/>
                    </wsse:SecurityTokenReference>
                </ds:KeyInfo>
                ...
            </xenc:ReferenceList>
        </wsse:Security>
    ...
    </S11:Header>
    <S11:Body>
        ...
    </S11:Body>
</S11:Envelope>
```

The following example illustrates a derived key based on the 3rd generation of the shared key identified in the specified security context:

```
    <wsc:DerivedKeyToken>
        <wsse:SecurityTokenReference>
            <wsse:Reference URI=".../ctx1"/>
        </wsse:SecurityTokenReference>
        <wsc:Generation>2</wsc:Generation>
    </wsc:DerivedKeyToken>
```

The following example illustrates a derived key based on the 1st generation of a key derived from an existing derived key (4th generation):

```
    <wsc:DerivedKeyToken>
        <wsc:Properties>
            <wsc:Name>.../derivedKeySource</wsc:Name>
            <wsc:Label>NewLabel</wsc:Label>
            <wsc:Nonce>FHFE...</wsc:Nonce>
        </wsc:Properties>
        <wsc:Generation>3</wsc:Generation>
    </wsc:DerivedKeyToken>

    <wsc:DerivedKeyToken wsu:Id="newKey">
        <wsse:SecurityTokenReference>
            <wsse:Reference URI=".../wsse:derivedKeySource"/>
```

```
        </wsc:SecurityTokenReference>
        <wsc:Generation>0</wsc:Generation>
    </wsc:DerivedKeyToken>
```

In the example above we have named a derived key so that other keys can be derived from it. To do this we use the <wsc:Properties> element name tag to assign a global name attribute. Note that in this example, the ID attribute could have been used to name the base derived key if we didn't want it to be a globally named resource. We have also included the <wsc:Label> and <wsc:Nonce> elements as metadata properties indicating how to derive sequences of this derivation.

6.3 Implied Derived Keys

This specification also defines a shortcut mechanism for referencing certain types of derived keys. Specifically, a *@wsc:Nonce* attribute can be added to the security token reference (STR) defined in the [WS-Security] specification. When present, it indicates that the key is not in the referenced token, but is a key derived from the referenced token's key/secret.

Consequently, the following two examples are functionally equivalent:

```
...
    <wsse:Security>
        <xx:MyToken wsu:Id="base">...</xx:MyToken>
        <wsc:DerivedKeyToken wsu:Id="newKey">
            <wsc:Nonce>...</wsc:Nonce>
            <wsse:SecurityTokenReference>
                <wsse:Reference URI="#base"/>
            </wsse:SecurityTokenReference>
        </wsc:DerivedKeyToken>
        <ds:Signature>
            ...
            <ds:KeyInfo>
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#newKey"/>
                </wsse:SecurityTokenReference>
            </ds:KeyInfo>
        </ds:Signature>
    </wsse:Security>
...
```

This is functionally equivalent to the following:

```
...
    <wsse:Security>
```

```
        <xx:MyToken wsu:Id="base">...</xx:MyToken>
        <ds:Signature>
            ...
            <ds:KeyInfo>
                <wsse:SecurityTokenReference wsc:Nonce="...">
                    <wsse:Reference URI="#base"/>
                </wsse:SecurityTokenReference>
            </ds:KeyInfo>
        </ds:Signature>
    </wsse:Security>
...
```

7. Error Handling

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

Error that occurred (faultstring)	**Fault code (faultcode)**
The requested context elements are insufficient or unsupported.	wsc:BadContextToken
Not all of the values associated with the SCT are supported.	wsc:UnsupportedContextToken
The specified source for the derivation is unknown.	wsc:UnknownDerivationSource

8. Security Considerations

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

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

It is critical that all relevant elements of a message be included in signatures. As well, the signatures for security context establishment must include a timestamp, nonce, or sequence number depending on the degree of replay prevention required. Security context establishment should include full policies to prevent possible attacks (e.g. downgrading attacks).

Authenticating services are susceptible to denial of service attacks. Care should be taken to mitigate such attacks as is warranted by the service.

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

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

9. Acknowledgements

This specification has been developed as a result of joint work with many individuals and teams, including:

Paula Austel, IBM
Keith Ballinger, Microsoft
John Brezak, Microsoft
Tony Cowan, IBM
John de Freitas, Netegrity
HongMei Ge, Microsoft
Slava Kavsan, RSA Security
Scott Konersmann, Microsoft
Leo Laferriere, Netegrity
Paul Leach, Microsoft
Richard Levinson, Netegrity
John Linn, RSA Security
Michael McIntosh, IBM
Steve Millet, Microsoft
Birgit Pfitzmann, IBM
Fumiko Satoh, IBM
Keith Stobie, Microsoft
T.R. Vishwanath, Microsoft
Doug Walter, Microsoft
Richard Ward, Microsoft
Hervey Wilson, Microsoft

10. References

[RFC2119]
S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels," RFC 2119, Harvard University, March 1997.

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

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

[URI]

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

[WS-Addressing]
"Web Services Addressing (WS-Addressing)," BEA, IBM, Microsoft, March 2004.

[WS-Policy]
"Web Services Policy Framework," BEA, IBM, Microsoft, SAP, December 2002.

[WS-PolicyAttachment]
"Web Services Policy Attachment Language," BEA, IBM, Microsoft, SAP, December 2002.

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

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

[WS-Trust]
"Web Services Trust Language," BEA, Computer Associates, IBM, Layer7, Microsoft, Netegrity, Oblix, OpenNetwork, Ping Identity, Reactivity, RSA Security, VeriSign, Westbridge, March 2004.

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

[XML-Encrypt]
W3C Recommendation, "XML Encryption Syntax and Processing," 10 December 2002.

[XML-ns]
W3C Recommendation, "Namespaces in XML," 14 January 1999.

[XML-Schema1]
W3C Recommendation, "XML Schema Part 1: Structures," 2 May 2001.

[XML-Schema2]
W3C Recommendation, "XML Schema Part 2: Datatypes," 2 May 2001.

[XML-Signature]
W3C Recommendation, "XML-Signature Syntax and Processing," 12 February 2002.

Appendix I – Sample Usages

This non-normative appendix illustrates several sample usage patterns of [WS-Trust] and [WS-SecureConversation]. Specifically, it illustrates different patterns that could be used to parallel, at an end-to-end message level, the selected TLS/SSL scenarios. This is not intended to be the definitive method for the scenarios, nor is it fully inclusive. Its purpose is simply to illustrate, in a context familiar to readers, how this specification might be used.

The following sections are based on a scenario where the client wishes to authenticate the server prior to sharing any of its own credentials.

It should be noted that the following sample usages are illustrative; any implementation of the examples illustrated below should be carefully reviewed for potential security attacks. For example, multi-leg exchanges such as those below should be careful to prevent man-in-the-middle attacks or downgrade attacks. It may be desirable to use

running hashes as challenges that are signed or a similar mechanism to ensure continuity of the exchange.

The examples below assume that both parties understand the appropriate security policies in use and can correctly construct signatures and encryption that the other party can process.

I.1 Anonymous SCT

In this scenario the requestor wishes to remain anonymous while authenticating the recipient and establishing an SCT for secure communication.

This scenario assumes that the requestor has a key for the recipient. If this isn't the case, they can use [WS-MetadataExchange] or the mechanisms described in a later section or obtain one from another security token service.

There are two basic patterns that can apply, which only vary slightly. The first is as follows:

1. The requestor sends an RST to the recipient requesting an SCT. The request contains key material encrypted for the recipient. The request is not authenticated.
2. The recipient, if it accepts such requests, returns an RSTR with the SCT as the requested token and does not return any proof information indicating that the requestor's key is the proof.

A slight variation on this is as follows:

1. The requestor sends an RST to the recipient requesting an SCT. The request contains key material encrypted for the recipient. The request is not authenticated.
2. The recipient, if it accepts such requests, returns an RSTR with the SCT as the requested token and returns its own key material encrypted using the requestor's key.

Another slight variation is to return a new key encrypted using the requestor's provided key.

It should be noted that the variations that involve encrypting data using the requestor's key material might be subject to certain types of key attacks.

Yet another approach is to establish a secure channel (e.g. TLS/SSL IP/Sec) between the requestor and the recipient. Key material can then safely flow in either direction. In some circumstances, this provides greater protection than the approach above when returning key information to the requestor.

I.2 Mutual Authentication SCT

In this scenario the requestor is willing to authenticate, but wants the recipient to authenticate first. The following steps outline the message flow:

1. The requestor sends an RST requesting an SCT. The request contains key material encrypted for the recipient. The request is not authenticated.
2. The recipient returns an RSTR including a challenge for the requestor. The RSTR is secured by the recipient so that the requestor can authenticate it.
3. The requestor, after authenticating the recipient's RSTR, sends an RSTR responding to the challenge.

4. The recipient, after authenticating the requestor's RSTR, sends a secured RSTR containing the token and either proof information or partial key material (depending on whether or not the requestor provided key material).

Another variation exists where step 1 includes a specific challenge for the service. Depending on the type of challenge used this may not be necessary because the message may contain enough entropy to ensure a fresh response from the recipient.

In other variations the requestor doesn't include key information until step 3 so that it can first verify the signature of the recipient in step 2.

I.3 Token Discovery Using RST/RSTR

If the recipient's security token is not known, the RST/RSTR mechanism can still be used. The following example illustrates one possible sequence of messages:

1. The requestor sends an RST requesting an SCT. This request does not contain any key material, nor is the request authenticated.
2. The recipient sends an RSTR to the requestor with an embedded challenge. The RSTR is secured by the recipient so that the requestor can authenticate it.
3. The requestor sends an RSTR to the recipient and includes key information protected for the recipient. This request may or may not be secured depending on whether or not the request is anonymous.
4. The final issuance step depends on the exact scenario. Any of the final legs from above might be used.

Note that step 1 might include a challenge for the recipient. Please refer to the comment in the previous section on this scenario.

Also note that in response to step 1 the recipient might issue a fault secured with [WS-Security] providing the requestor with information about the recipient's security token.