Web Services Security Kerberos Binding

December 2003

Authors

Giovanni Della-Libera, Microsoft
Brendan Dixon, Microsoft
Praerit Garg, Microsoft
Maryann Hondo, IBM
Chris Kaler, Microsoft
Hiroshi Maruyama, IBM
Anthony Nadalin, IBM
Nataraj Nagaratnam, IBM

Copyright Notice

Abstract

This document describes how to use Web Services security specifications with Kerberos.

Status

This Web Services Security Kerberos Binding Specification is an initial public draft release and is provided for review and evaluation only. IBM and Microsoft hope to solicit your contributions and suggestions in the near future. IBM and Microsoft make no warrantees or representations regarding the specifications in any manner whatsoever.

Table of Contents

1. Introduction

Kerberos is an established authentication and security infrastructure in use in many environments today. Consequently, as applications integrate with and are developed for Web services, there is a need to leverage existing security infrastructure. This specification describes how to integrate Kerberos security environments with the Web service security architecture.

Integration with Web services security requires the following aspects:

- Requesting and issuing security tokens
- Attaching security token to messages
- Establishing a secure context
- Signing and encrypting the message using the security context

This specification describes two models of Web service usage and interoperability: GSS-API and Raw Kerberos.

1.1. 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 RFC 2119.

When describing abstract data models, this specification uses the notational convention used by the XML Infoset [7]. Specifically, abstract property names always appear in square brackets (e.g., [some property]).

When describing concrete XML schemas, this specification uses the notational convention of WS-Security [13]. Specifically, each member of an element's [children] or [attributes] property is described using an XPath-like notation (e.g., /x:MyHeader/x:SomeProperty/@value1). The use of {any} indicates the presence of an element wildcard (<xs:any/>). The use of @{any} indicates the presence of an attribute wildcard (<xs:anyAttribute/>).

1.2. Namespaces

The following namespaces are used in this document:

Prefix	**Namespace**
S	http://www.w3.org/2002/06/soap-envelope
S11	http://schemas.xmlsoap.org/soap/
ds	http://www.w3.org/2000/09/xmldsig#
xenc	http://www.w3.org/2001/04/xmlenc#
wsse	http://schemas.xmlsoap.org/ws/2003/07/secext
wsu	http://schemas.xmlsoap.org/ws/2002/07/utility
wsk	http://schemas.xmlsoap.org/ws/2003/12/kerberos

This specification uses the SOAP 1.2 specification for its examples; however, the mechanisms defined here are also applicable to SOAP 1.1.

1.3. Schema File

The schema for this specification can be located at:

```
        http://schemas.xmlsoap.org/ws/2003/12/kerberos
```

2. Web Service Security Usage

This specification builds on the WS-Security [WS-Security], WS-Trust [WS-Trust], and WS-SecureConversation [WS-SecureConversation] specifications to integrate Kerberos functionality. The following sub-sections describe basic usage patterns that are common to both Kerberos scenarios defined in this specification.

2.1. Security Tokens

The security tokens used by both models are binary and not based on XML. Consequently, the `<wsse:BinarySecurityToken>` element from WS-Security is used to pass security tokens inside SOAP messages. The `wsse:ValueType` and `wsse:EncodingType` attributes describe the security token's type and encoding. Applications integrating Kerberos with WS-Security MUST include their tokens as instances of `<wsse:BinarySecurityToken>`. They SHOULD encode these in base64 (i.e., `wsse:EncodingType="wsse:Base64Binary"`).

Each of the models describes the valid security token formats and how these tokens are obtained.

2.2. Policies

Every security infrastructure has configuration and runtime options. For example, some allow mutual authentication and others allow negotiating the cryptographic algorithms and key properties. Interoperation requires all parties selecting compatible options, if not identical. Applications SHOULD note the options they expect in their policy which SHOULD be expressed using WS-Policy and WS-SecurityPolicy.

Each model definition describes its specific algorithms for establishing mutual policies.

2.3. Signing and Encrypting

The Web services security model uses the mechanisms described in WS-Security to sign and encrypt messages. Both mechanisms use URIs to indicate the type of signature or encryption.

XML Signature [XML Signature] enables multiple signatures, over potentially different parts of the message, and includes a number of pre-defined signature algorithms (e.g. `http://www.w3.org/2000/09/xmldsig#rsa-sha1`). XML Signatures apply indirectly to the signed content: That is, the signature is formed over the digest of the describing element (the `<ds:SignedInfo>` sub-element of the `<ds:Signature>` element) rather than over the content directly. The `<ds:SignedInfo>` element contains, among other details, the content's digest and an expression of the signature algorithm (using the `<ds:SignatureMethod>` element).

XML Encryption [XML Encryption], and its use by WS-Security, carries the encrypted content within an `<xenc:EncryptedData>` element, wholly replacing the encrypted element. Among the children of the `<xenc:EncryptedData>` element is `<xenc:EncryptionMethod>`, which expresses what encryption algorithm the application used.

Each application and security infrastructure must define the acceptable algorithms, security context, and keys. The exact algorithms for signing and encrypting vary between the different Kerberos models and are described in detail in those sections of this specification.

Within this context, the mechanisms described in WS-Security represent the *authenticator* that is used to secure messages. That is, typical Kerberos authenticators are not used verbatim. Instead, those aspects are included in the message and are referenced as part of a signature. For this reason it is STRONGLY RECOMMENDED that both message timestamps and nonces be included in signatures and that message freshness checks be applied. Refer to the security considerations section of the WS-Security specification for additional information. As well, it is RECOMMENDED by this specification that signatures be used instead of encryptions that include digests.

3. GSS-API for Kerberos Interoperability

GSS-API presents a common approach and feature set over a number of different and popular security protocols. It is frequently used when two Web services, both existing within Kerberos environments leveraging GSS-API, want to securely interoperate across the Internet.

The GSS-API model revolves around a *security context* – a set of state shared between a client (created by *GSS_Init_sec_context*) and server (accepted by *GSS_Accept_sec_context*). The context carries attributes such as replay detection elements, sequencing elements, and shared session keys. GSS-API includes this context on top of the underlying protocol used. For example, when using X.509 Certificates, GSS-API adds sequencing facilities and a session-level symmetric key. GSS-API abstracts existing protocols and adds additional features.

It should be noted that there are two approaches for GSS-API integration: a Web services security mechanism or layering Web services security on top of existing mechanisms. This specification is agnostic of approach as it defines the wire protocol and the primary difference between the two approaches is the API calls and sequencing.

3.1. Acquiring a Credentials Handle

GSS-API designates the abstracted security protocols as mechanisms. When an application first acquires its credentials handle, which scopes all of its operations within GSS-API, it specifies either a specific mechanism type (i.e. mech_type) or NULL to request the default system mechanism.

Since GSS-API shields an application from the vagaries of the different underlying protocols, applications are free to select their mechanism type (or use the default). WS-Security adds no new requirements since it can carry GSS-API security tokens as binary tokens. However, the sender and receiver must be in agreement as to the type and contents of the security token.

3.2. Establishing a Security Context

GSS-API requires the exchange of at least one, and perhaps multiple, messages to establish the shared context. When requiring an application-level exchange, *GSS_Init_sec_context* or *GSS_Accept_sec_context* returns with either *GSS_S_CONTINUE_NEEDED* or *GSS_S_COMPLETE*. The context does not exist until *GSS_S_COMPLETE* is returned.

Applications perform this message exchange using the mechanisms described in WS-Trust for handling challenge/responses and WS-SecureConversation for requesting a security context token. Essentially, a "binary negotiation" occurs using the SPNEGO blobs resulting in a Security Context Token (SCT) which is associated with the SPNEGO results. This means that the SCT can be used to indicate the context (and keys) to the recipient.

Specifically, the object returned from GSS-API MUST be attached using a `<wsse:BinaryNegotiation>` element inside of the `<wsse:RequestSecurityToken>` and `<wsse:RequestSecurityTokenResponse>` elements containing the GSS-API object. The *ValueType* attribute indicates the type of negotiation. It is RECOMMENDED that the *ValueType* attribute use `wsk:SPNEGO`. In this scenario, the `<wsse:BinaryNegotiation>` element is specified along with the security context token request element `<wsse:TokenType>` equal to `wsse:SecurityContextToken` inside of the `<wsse:RequestSecurityToken>` element. It is STRONGLY RECOMMENDED that the `<wsse:BinaryNegotiation>` element be signed (or in the case that signatures cannot be initially applied that it be signed as soon as possible).

While the exact format of the binary object is not relevant to this process, it is specific to the type specific on the `<wsse:BinaryNegotiation>` element. Negotiations of type `wsk:SPNEGO` MUST conform to RFC 2478.

The result is that the security context token issued is bound to the GSS-API context negotiated.

This "binding" is implicit, but in some scenarios, the issuer MAY include information in the issued security context token which it uses on subsequent messages to correlate the contexts. This is done by including a `<wsk:Correlation>` element in the `<wsse:SecurityContextToken>`. The contents of this element MUST be preserved verbatim and MUST be passed in all uses of the token. The syntax overview is as follows:

```
<wsse:SecurityContextToken>
    ...
    <wsk:Correlation Target="...">
        ...
    </wsk:Correlation>
    ...
</wsse:SecurityContextToken>
```

/wsse:SecurityContextToken/wsk:Correlation
This element is used to pass correlation information which MUST be returned *verbatim*. Note that multiple correlation elements MAY be present.

/wsse:SecurityContextToken/wsk:Correlation/@Target
This optional attribute allows a specific SOAP actor/role to be identified with the correlation element.

/wsse:SecurityContextToken/wsk:Correlation/{any}
The contents of this element are up to the creator and are not interpreted by anyone except the sender.

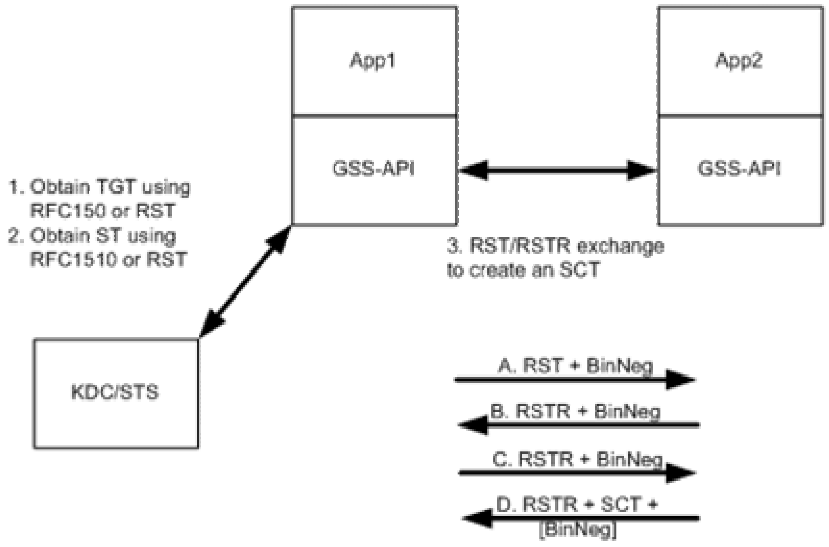
/wsse:SecurityContextToken/wsk:Correlation/@{any}

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

Applications MUST NOT sign or encrypt content using GSS-API until the mechanism is ready. Normally, that is not until shared context establishment. However, some mechanisms may support signing and encryption before fully establishing the context. These indicate readiness by returning *true* for the *prot_ready_state* boolean from *GSS_Init_sec_context* or *GSS_Accept_sec_context*.

It is RECOMMENDED that mutual authentication through GSS-API (i.e. *mutual_req_flag* is true) be requested although applications MAY choose not to do so.

The figure below illustrates the model for GSS-API interoperation with the Web service security specifications:


App1
App2
GSS-API
GSS-API
1. Obtain TGT using RFC150 or RST
2. Obtain ST using RFC1510 or RST
3. RST/RSTR exchange to create an SCT
KDC/STS
A. RST + BinNeg
B. RSTR + BinNeg
C. RSTR + BinNeg
D. RSTR + SCT + [BinNeg]

1. A client service (App1) obtains a TGT to allow it to later obtain tickets for services. This can be done using the traditional Keberos mechanisms or using the mechanisms described in WS-Trust.
2. A client service (App1) obtains an ST for another service (App2). This can be done using the traditional Kerberos mechanisms or using the mechanisms described in WS-Trust.
3. The client uses the mechanisms described in WS-Trust to initialize the GSS-API context:

The client service issues a `<RequestSecurityToken>` message to the target service. As well, a `<BinaryNegotiation>` is included to encapsulate the GSS API initialization context.

The service MAY respond with a `<RequestSecurityTokenResponse>` message.

The client MAY respond with a `<RequestSecurityTokenResponse>` message.

Steps (b) and (c) continue until both sides are happy (and/or one fails). On success, the `<SecurityContextToken>` is returned. Note that steps (b) and (c) MAY be omitted.

The following sample message illustrates creating a security context based on a SPNEGO exchange between two parties. In this example, one round trip is required for SPNEGO.

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope">
   <S:Header>
      ...
      <wsse:Security
           xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/07/secext">
         <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <!-- Signature over the request ->
         </ds:Signature>
      </wsse:Security>
   </S:Header>
   <S:Body wsu:Id="body">
      <wsse:RequestSecurityToken
            xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/07/secext"
            xmlns:wsk="http://schemas.xmlsoap.org/ws/2003/12/kerberos">
        <wsse:TokenType>wsse:SecurityContextToken</wsse:TokenType>
        <wsse:BinaryNegotiation ValueType="wsk:SPNEGO"
                                Encoding="wsse:Base64Binary">
            jhkaJKHKJHD...
        </wsse:BinaryNegotiation>
      </wsse:RequestSecurityToken>
   </S:Body>
</S:Envelope>
```

The following is a sample response for the scenario described above:

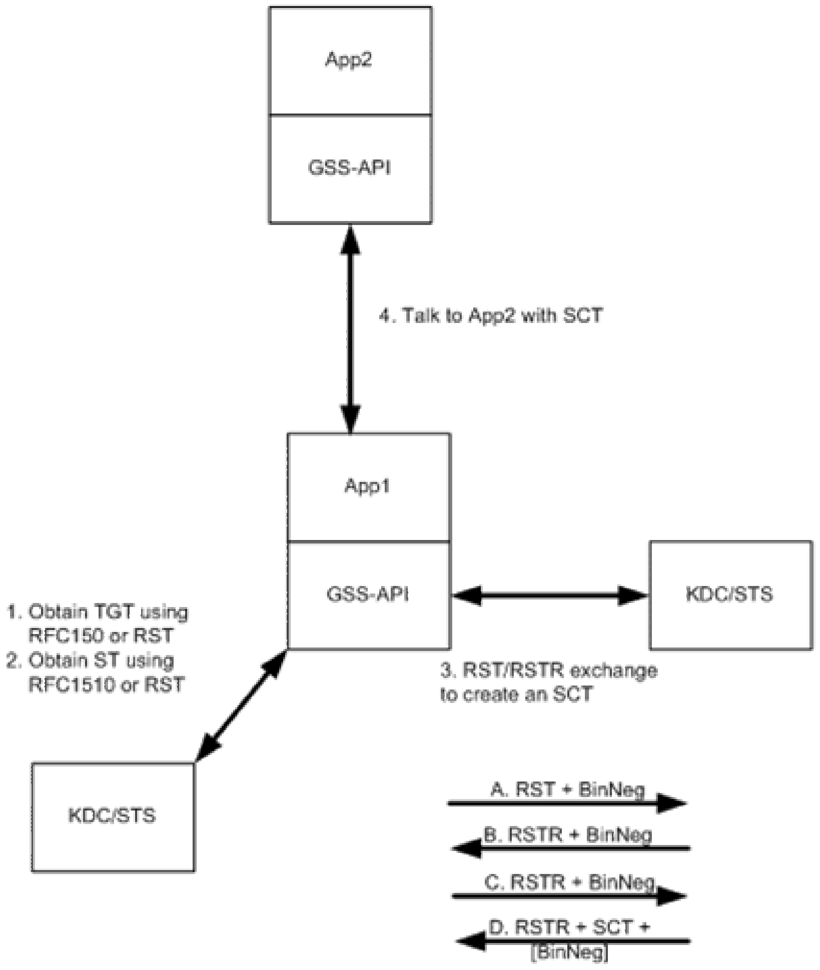
```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope">
   <S:Header>
      ...
      <wsse:Security
           xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/07/secext"
            xmlns:wsk="http://schemas.xmlsoap.org/ws/2003/12/kerberos">
         <wsse:SecurityContextToken wsu:Id="sct">
```

```
            <wsu:Identifier>
                 http://fabrikam123.com/ctx/124</wsu:Identifier>
            <wsk:Correlation>
                <wsu:Identifier>uuid:...</wsu:Identifier>
            </wsk:Correlation>
         </wsse:SecurityContextToken>
         <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <!-- Signature over the response ->
         </ds:Signature>
      </wsse:Security>
   </S:Header>
   <S:Body wsu:Id="body">
      <wsse:RequestSecurityTokenResponse
            "xmlns:wsse=http://schemas.xmlsoap.org/ws/2003/07/secext"
            "xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
            xmlns:wsk="http://schemas.xmlsoap.org/ws/2003/12/kerberos">
        <wsse:TokenType>wsse:SecurityContextToken</wsse:TokenType>
        <wsse:BinaryNegotiation ValueType="wsk:SPNEGO"
                                Encoding="wsse:Base64Binary">
            HorueHNUEE...
        </wsse:BinaryNegotiation>
        <wsse:RequestedSecurityToken>
          <wsse:SecurityContextToken>
            <wsee:Reference URI="#sct"/>
          </wsse:SecurityContextToken>
        </wsse:RequestedSecurityToken>
      </wsse:RequestSecurityTokenResponse>
   </S:Body>
</S:Envelope>
```

The figure below illustrates an alternate model which MAY also be used. In this case steps (1) and (2) are the same as above except that the communication is not with the target service, but instead with a broker (such as a KDC).

For this scenario, the `<RequestSecurityToken>` message MUST include an `<wsp:AppliesTo>` scoping element indicating the target service (App2). This allows the broker to know the intended target. This is critical because the `<SecurityContextToken>` returned MUST include the session key encrypted for the target (App2).

The client can then communicate with the target (App2) using the security context token and the mechanisms in WS-Security and this document to secure the message.



3.3. Signing with XML Signature

GSS-API supplies two methods for creating and verifying signatures: GSS_GetMIC and GSS_VerifyMIC. These operate within the context the client and target service pair. For example, they utilize any session keys established and include, when requested, sequencing components that prevent replay attacks.

Applications MAY use the GSS-API signature methods within the framework XML Signature describes. However, a new signature algorithm is required for GSS-API:

```
http://schemas.xmlsoap.org/ws/2003/12/IETF-RFC2743#GSS_MIC
```

When establishing the GSS-API shared context, the application MUST include an XML Signature `<ds:KeyInfo>` containing a `<wsse:SecurityTokenReference>` identifying the security context token that was established (as described in WS-SecureConversation).

The application MUST use the GSS-API methods to generate and verify the digital signatures when the above algorithm is specified.

Specifically, an application uses GSS-API to create an XML Signature as follows:

1. Compute the digest over the reference data as specified in XML Signature.
2. Specify the digest algorithm used to hash the referenced data as the `<ds:DigestMethod>`.
3. Build the `<ds:SignedInfo>` block, expressing the GSS-API signature algorithm as the `<ds:SignatureMethod>`.
4. Canonicalize the `<ds:SignedInfo>` block as defined in XML Signature.
5. Apply GSS_GetMIC output to the canonical form of the `<ds:SignedInfo>` block, generating an octet sequence.
6. Optionally (but RECOMMENDED) include a `<ds:KeyInfo>` containing a `<wsse:SecurityTokenReference>` identifying the security context associated with the GSS-API context.

Note that there MAY be a `<wsse:SecurityTokenReference>` element (Step 6) with a reference to the security context for the associated GSS-API context key. If a key isn't specified explicitly, and a context is associated with the message, the key from the context SHOULD be used.

The following sample message illustrates the use of this algorithm in conjunction with WS-Security.

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope">
   <S:Header>
      ...
      <wsse:Security
           xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/07/secext">
         <wsse:SecurityContextToken wsu:Id="sct">
            <wsu:Identifier>uuid:...</wsu:Identifier>
         </wsse:SecurityContextToken>
         <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <ds:SignedInfo>
               <ds:CanonicalizationMethod
                   Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#"/>
               <ds:SignatureMethod

Algorithm="http://schemas.xmlsoap.org/ws/2003/12/IETF-
RFC2743#GSS_MIC"/>
               <ds:Reference URI="#body">
                  <ds:Transforms>
                     <ds:Transform>
```

```
                         Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#"/>
                  </ds:Transforms>
                  <ds:DigestMethod
                   Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
                  <ds:DigestValue>EULddytSol...</ds:DigestValue>
               </ds:Reference>
            </ds:SignedInfo>

<ds:SignatureValue>BL8jdfToEb11/vXcMZNNjPOV...</ds:SignatureValue>
            <ds:KeyInfo>
               <wsse:SecurityTokenReference>
                  <wsse:Reference URI="#sct"/>
               </wsse:SecurityTokenReference>
            </ds:KeyInfo>
         </ds:Signature>
      </wsse:Security>
   </S:Header>
   <S:Body wsu:Id="body">
      <tru:StockSymbol xmlns:tru="http://fabrikam123.com/payloads">
         QQQ
      </tru:StockSymbol>
   </S:Body>
</S:Envelope>
```

The process of verifying an XML Signature made with GSS-API is analogous. The verifier checks the hashes of the referenced content as specified in XML Signature: The <ds:SignedInfo> element is first made canonical. It then passes the resulting byte stream and the decoded token (extracted from the <ds:SignatureValue> element) to GSS_VerifyMIC. The signature is cryptographically valid if and only if the call to GSS_VerifyMIC returns true.

3.4. Replay Prevention

GSS-API provides elements for replay detection and sequencing. GSS-API embeds and seals these elements directly into the GSS-API security token. A GSS-API Security token is a caller-opaque type that GSS-API uses to maintain synchronization between the context data structures at each end of a GSS-API security context. An appropriately initialized GSS-API security token, appropriately initialized, protects against different classes of attack.

The protection accrued from the GSS-API security token extends to the application specific protocol, in this case, the SOAP messages, through signatures and encryption. Using GSS-API to encrypt a message (through GSS_Wrap) binds it to the security token; that is, because GSS-API embeds digests, the intended recipient can decrypt and validate the message (through GSS_Unwrap). Similarly, only the intended recipient can verify a GSS-API generated signature (created through GSS_GetMIC and validated through GSS_VerifyMIC). This protection includes, within appropriately initialized GSS-API contexts, replay detection and sequencing from the security token and the shared GSS-API context. Applications SHOULD NOT create additional signatures to prevent tampering and replay unless they are part of higher-level protocols. Additional signatures are useful for other purposes such as enabling long-term auditing, non-repudiation, and some forms of SOAP intermediary processing.

The protections accrued by a GSS-API security token and shared context result from values supplied to the GSS_Init_sec_context function. To ensure interoperation and to prevent redundancy, applications SHOULD request both replay detection (i.e. replay_det_req_flag is true) and sequencing (i.e. sequence_req_flag is true) when creating the GSS-API context.

3.5. Encryption with XML Encryption

GSS-API has two encryption-related methods: GSS_Wrap and GSS_Unwrap. As with all GSS-API methods, they operate within the context the client and target service share, utilizing established sessions keys and replay detection elements.

Applications MAY use the GSS-API encryption methods within the framework XML Encryption describes. However a new encryption algorithm is required for GSS-API:

```
http://schemas.xmlsoap.org/2003/03/IETF-RFC2743#GSS_Wrap
```

The application MUST use the GSS-API methods to encrypt and decrypt the content when the above algorithm is specified.

Specifically, an application uses GSS-API to encrypt message content as follows:

Encrypt the content, using GSS_Wrap, and replace it in the message with an `<xenc:EncryptedData>` element as XML Encryption describes.

If desired, include a `<xenc:ReferenceList>` element under the Security header referencing the encrypted data.

Note that there SHOULD be a `<wsse:SecurityTokenReference>` element with a reference to the security context for the key, but applications MAY choose not to include it. If a key isn't specified explicitly, and a context is associated with the message, the key from the context SHOULD be used.

The following sample message illustrates the use of this algorithm in conjunction with WS-Security.

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S=" http://www.w3.org/2001/12/soap-envelope">
   <S:Header>
      ...
      <wsse:Security
```

```
           xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/07/secext"
           xmlns:xenc="http://www.w3.org/2001/04/xmlend#">
         <wsse:SecurityContextToken wsu:Id="sct">
            <wsu:Identifier>uuid:...</wsu:Identifier>
         </wsse:SecurityContextToken>
         <xenc:ReferenceList>
            <xenc:DataReference URI="#bodyID"/>
         </xenc:ReferenceList>
      </wsse:Security>
   </S:Header>
   <S:Body>
      <xenc:EncryptedData
          Id="bodyID"
          Type="http://www.w3.org/2001/04/xmlenc#Element"
          xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
         <xenc:EncryptionMethod
             Algorithm="http://schemas.xmlsoap.org/2003/12/IETF-
RFC2743#GSS_Wrap"/>
         <xenc:CipherData>
            <xenc:CipherValue>
                d2FpbmdvbGRfE0lm4byV0...
            </xenc:CipherValue>
         </xenc:CipherData>
         <ds:KeyInfo>
            <wsse:SecurityTokenReference>
               <wsse:Reference URI="#sct"/>
            </wsse:SecurityTokenReference>
         </ds:KeyInfo>
      </xenc:EncryptedData>
   </S:Body>
</S:Envelope>
```

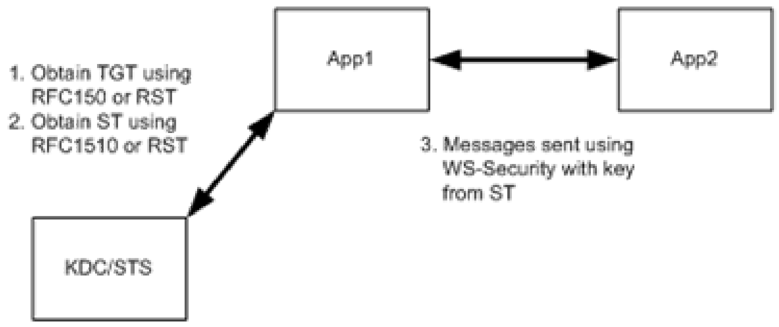
4. General Kerberos Interoperability

Alternatively instead of using GSS-API, interoperability can be achieved at the Kerberos level. That is, using raw Kerberos security tokens and cryptographic functions. The model is straightforward: tickets are obtained and the keys are extracted for use in signing and encrypting messages.

Kerberos is an IETF standard third-party mediated protocol as described in [RFC1510]. Clients interact with one or more Kerberos Key Distribution Centers (KDC) through Authentication and Ticket-Granting Services. Kerberos defines a request/response packet pair for each exchange. A key associated with each Kerberos packet is used to sign and encrypt different portions. Some of the keys, such as those contained in tickets, are ephemeral and have defined lifetimes.

Conceptually, a Kerberos KDC implements what WS-Trust calls a Security Token Service: It generates security tokens (e.g., Kerberos TGT) in exchange for other tokens (e.g., a user-name and password).

The figure below illustrates the model for general Kerberos interoperability using the Web service security specifications.


App1
App2
1. Obtain TGT using RFC150 or RST
2. Obtain ST using RFC1510 or RST
3. Messages sent using WS-Security with key from ST
KDC/STS

1. A client service (App1) obtains a TGT to allow it to later obtain tickets for services. This can be done using the traditional Kerberos mechanisms or using the mechanisms described in WS-Trust.
2. A Kerberos ST is obtained from the KDC or from a security token service using WS-Trust.
3. The result of step (1) is a Kerberos ST for App2 and a proof-of-possession token which contains the session key encrypted for App1. Using the shared secret, messages between App1 and App2 can be signed and encrypted as described in WS-Security.

4.1. Kerberos Security Token

The Kerberos v5 specification (RFC1510) stipulates an ASN.1 encoding for all Kerberos packets. While the Kerberos packets are self-describing (that is, the ASN.1 format contains the packet type), compliant applications SHOULD use the most specific token type (e.g., `wsse:ValueType="Kerberosv5TGT"`).

The WS-Security specification defines value types for Kerberos TGTs and STs. These types represent the raw ticket rather than the more common ticket + authenticator found in the TGS-REQ and AS-REQ packets.

4.2. Using XML Signature

The session key described in the ST or TGT is used to sign the message as described in WS-Security (unless some other form of proof is provided). Because applications have direct access to the session key, the XML Signature mechanisms described in

WS-Security can be used directly. This signature becomes the authenticator found in traditional Kerberos applications.

Implementations determine which signature algorithm to use by comparing and policies (see WS-Policy [WSPolicy] and WS-SecurityPolicy [WSSecPolicy]). The first preferred algorithm SHOULD be used.

Kerberos traditionally uses algorithms not described in XML Signature. For compatibility reasons, we define new algorithm URIs for use with XML Signature. However, we RECOMMEND that implementations use, at a minimum, the RECOMMENDED algorithms found in XML Signature. The following table specifies algorithms that MAY be used in addition to those specified in XML Signature.

URI	**Description**
http://schemas.xmlsoap.org/ws/2003/12/RFC150#DES-MD5	RSA MD5 Cryptographic Checksum
http://schemas.xmlsoap.org/ws/2003/12/RFC150#DES-MD4	RSA MD4 Cryptographic Checksum
http://schemas.xmlsoap.org/ws/2003/12/RFC150#DES-CHK	DES CBC Checksum

Note that additional algorithms MAY be specified by defining a well-known URI for the algorithm.

The application SHOULD use the same URI for both the <ds:DigestMethod> and <ds:SignatureMethod> when using an algorithm defined in RFC1510.

Applications MUST include an XML Signature <ds:KeyInfo> containing a `<wsse:SecurityTokenReference>` identifying the attached Kerberos token as described in WS-Security. As previously stated, the signature represents the message authenticator.

The following sample message illustrates signing using a Kerberos token.

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope">
   <S:Header>
      ...
      <wsse:Security
           xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/07/secext">
         <wsse:BinarySecurityToken id="Kerberosv5ST"
                                    wsse:ValueType="wsse:Kerberosv5ST"
                                  wsse:EncodingType="wsse:Base64Binary">
            MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
         </wsse:BinarySecurityToken>
         <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <ds:SignedInfo>
               <ds:CanonicalizationMethod
                   Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#"/>
               <ds:SignatureMethod
                         Algorithm="...#DES-MD5"/>
               <ds:Reference>
                  <ds:Transforms>
                     <ds:Transform>
                         Algorithm="http://...#MyTranform"/>
                     <ds:Transform>
                         Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#"/>
                  </ds:Transforms>
                  <ds:DigestMethod
                         Algorithm="...#DES-MD5"/>
                  <ds:DigestValue>EULddytSol...</ds:DigestValue>
               </ds:Reference>
            </ds:SignedInfo>
            <ds:SignatureValue>
               BL8jdfToEb11vXcMZNNjPOV...
            </ds:SignatureValue>
            <ds:KeyInfo>
               <wsse:SecurityTokenReference>
                  <wsse:Reference URI="#Kerberosv5ST"/>
               </wsse:SecurityTokenReference>
            </ds:KeyInfo>
         </ds:Signature>
      </wsse:Security>
   </S:Header>
   <S:Body>
      <tru:StockSymbol xmlns:tru="http://fabrikam123.com/payloads">
         QQQ
      </tru:StockSymbol>
   </S:Body>
</S:Envelope>
```

4.3. Using XML Encryption

The Session key described in the ST or TGT is used to encrypt the message as described in WS-Security (unless another confidentiality mechanism is used). Because applications have direct access to the session key, the XML Encryption mechanisms described in WS-Security can be used directly. All of the recommendations in WS-Security SHOULD be followed.

Implementations determine which encryption algorithm to use by comparing and intersecting policies. The first preferred algorithm SHOULD be used.

Kerberos traditionally uses algorithms not described in XML Encryption. For compatibility reasons, we define a new algorithm URI for use with XML Encryption. However, we RECOMMEND that implementations use the RECOMMENDED algorithms found in XML Encryption. The following table specifies algorithms that MAY be used in addition to those specified in XML Encryption.

URI	**Description**
http://schemas.xmlsoap.org/ws/2003/12/RFC150#DES-CBC	DES CBC
http://schemas.xmlsoap.org/ws/2003/12/RFC150#RC4	RSA RC4

Note that additional algorithms MAY be specified by defining a well-known URI for the algorithm.

Kerberos v5 packet encryption, as defined in RFC1510, generally uses DES with Cipher Block Chaining (CBC). RFC1510 specifies multiple forms since Kerberos also includes a checksum in the encrypted content. XML Encryption, on the other hand, does not automatically include a signature or checksum. Implementations SHOULD use the integrity mechanisms defined in WS-Security, but MAY include a signature or checksum in the XML data being encrypted if a specialized algorithm is used (to indicate the presence of the data) or if both parties have agree on the data format.

The following example illustrates encryption using a Kerberos token.

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope">
   <S:Header>
      ...
      <wsse:Security
           xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/07/secext"
           xmlns:xenc="http://www.w3.org/2001/04/xmlend#">
         <wsse:BinarySecurityToken id="Kerberosv5ST"
                                   wsse:ValueType="Kerberosv5ST"
```

```
                                    wsse:EncodingType="wsse:Base64Binary">
            MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
         </wsse:BinarySecurityToken>
         <xenc:ReferenceList>
            <xenc:DataReference URI="#bodyID"/>
         </xenc:ReferenceList>
      </wsse:Security>
   </S:Header>
   <S:Body>
      <xenc:EncryptedData
          Id="bodyID"
          Type="http://www.w3.org/2001/04/xmlenc#Element"
          xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
         <xenc:EncryptionMethod Algorithm="...#DES-CBC"/>
         <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <wsse:SecurityTokenReference>
               <wsse:Reference URI="#Kerberosv5ST"/>
            </wsse:SecurityTokenReference>
         </ds:KeyInfo>
         <xenc:CipherData>
          <xenc:CipherValue>d2FpbmdvbGRfE0lm4byV0...</xenc:CipherValue>
         </xenc:CipherData>
      </xenc:EncryptedData>
   </S:Body>
</S:Envelope>
```

4.4. PKI Interoperability

The sections above describe secure communication between Kerberos-based Web services. That is, allowing two services, one of which exists in a Kerberos-based realm and one of which exists in a PKI-based realm, to securely communicate. This involves token issuance at interoperability at the service to STS layer as well as core message security at the service to service layer. This can be achieved if Web services share common cryptographic algorithms at the message security layer even if different security infrastructures are used. Specifically, this section discusses how PKI-based clients can interoperate with Kerberos-based servers and vice versa.

4.4.1. PKI → Kerberos

Consider, first, a PKI-based requestor communicating with a Kerberos-based server. In this scenario, the server's policy would indicate the following:

- It only accepts security tokens of type *wsse:Kerberosv5ST* issued by its associated security token service (part of or closely tied to a known KDC).
- The signing and encrypting algorithms it supports.

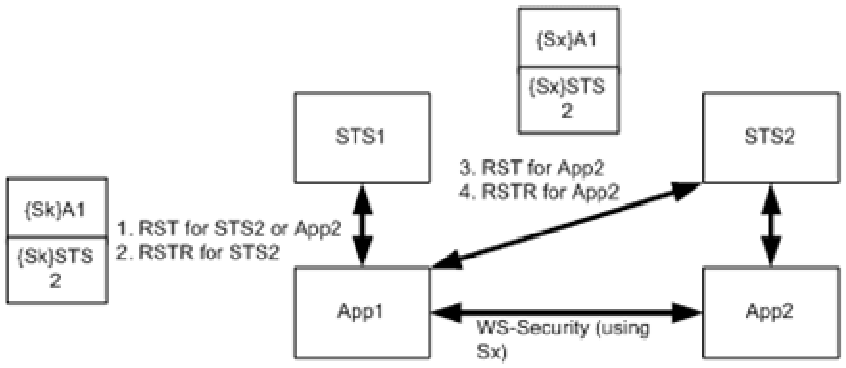
The requestor communicates with the specified security token service as follows:

- The requestor requests a *wsse:Kerberosv5ST* security token using the mechanisms described in WS-Trust from the security token service.
- The requestor provides a signature on the request using (and attaching) its PKI-based token and optionally provides its policy.
- The security token service compares the PKI-based token against its known trusts and determines that it trusts the token.
- The security token service returns a session key for the client encrypted using the client's public key and a Kerberos ST token containing the session key for the service (note that the key is protected by a shared secret between the token service and the service).

The client can now communicate with the service as follows:

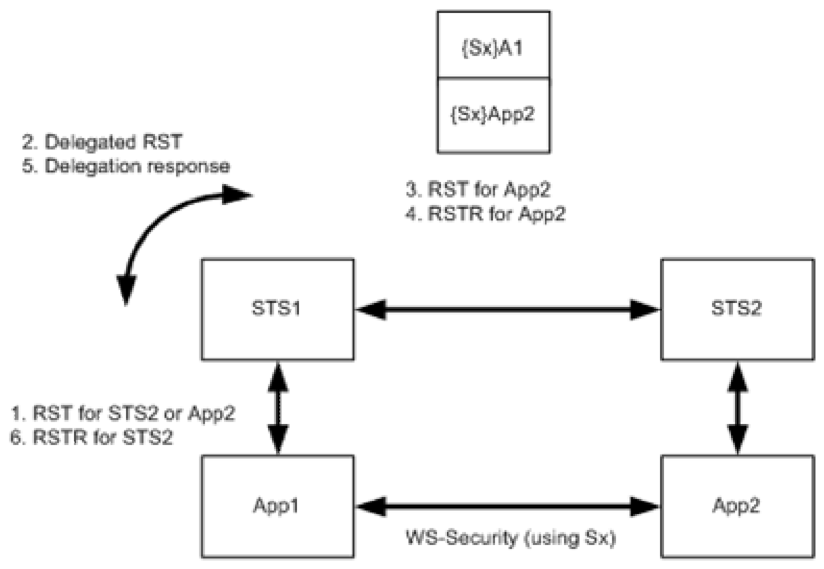
- The Kerberos ST token is attached to the message.
- A signature is added using the session key (or a key derived from the session key if a derived key token is also present).
- Optionally, the message is encrypted using the session key or a derived key.

The figure below illustrates this model:



1. The client (App1) requests a security token for either App2 or its token service STS2.
2. The security token service STS1, having a trust relationship, knows that STS2 must broker and returns a token for STS2.
3. The client requests a token from STS2 for App2.
4. The token is returned as a Kerberos format for App2 and an encrypted key using PKI for App1.

Alternatively, the first security token service STS1 could have obtained the security tokens on behalf of STS1 from STS2 and returned them directly as illustrated below:



4.4.2. Kerberos → PKI

Now consider a Kerberos-based requestor communicating with a PKI-based server. In this scenario, the server's policy would indicate the following:

- It only accepts tokens issued by a specified set of security token services.
- The signing and encrypting algorithms it supports.

The requestor identifies a security token service that it *trusts* and communicates as follows (in this case it is part of or closely tied to the client's KDC):

- It requests a security token using the mechanisms described in WS-Trust.
- It provides a signature on the request using the session key in its TGT token (and attaching the TGT token).
- The token service validates the request and returns a session key encrypted for the client using the session key from the TGT and a token containing the session key encrypted using the server's public key for the server.

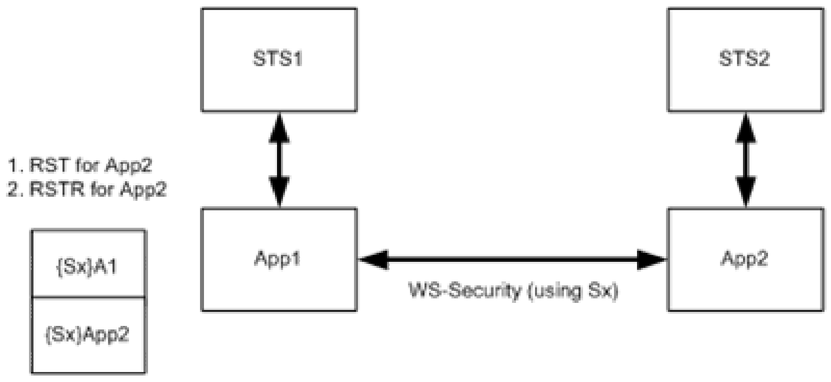
The client can now communicate with the server as follows:

- The new token is attached to the message.
- A signature is added using the session key (or a key derived from the session key if a derived key token is also present).
- Optionally, the message is encrypted using the session key or a derived key.

There are two common alternative scenarios:

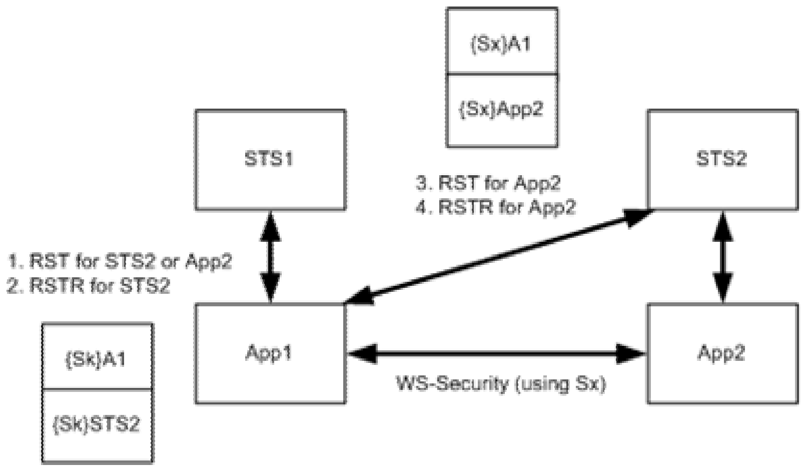
- The token service collaborates with a KDC that has an established trust with the client's KDC.
- The client uses traditional Kerberos requests to obtain an ST for the server. In this case the KDC recognizes the server and has a PKI-based trust relationship. The KDC returns an opaque "ST" which is really the session key encrypted with the server's public key.

The figure below illustrates this model:



1. The client (App1) requests a token for App2. Since it has a trust relationship with STS2 (or the domain for App2), and if it has the token for App2 available (or can obtain it), the STS can return the appropriate tokens.
2. The STS returns a Kerberos-format for the client and the session key encrypted for App2 using PKI.

Alternatively, if the STS doesn't have the public key for App2, it can direct the client to the security token service for App2 (STS2). It can do this by returning credentials "introducing" the client to STS2. This is illustrated below:



5. Derived Keys

The sections above describe using the secret key within the specified Kerberos token signing and encryption either in pure Kerberos exchanges or in exchanges between Kerberos and PKI. However, it is also possible to use this secret to derive multiple keys for signing and encryption. This is done using the `<wsse:DerivedKey>` element defined in WS-SecureConversation. To do this, the `<wsse:SecurityTokenReference>` within the signature or encryption refers to a derived key, which, in turn, refers to the Kerberos token. Specifically, the `<SecurityTokenReference>` element of the derived key token points to the ID (using relative URIs) of the Kerberos token or the security context token.

The following example illustrates a derived key based on a Kerberos token:

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

```
        xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/07/secext"
        xmlns:xenc="http://www.w3.org/2001/04/xmlend#">
      <wsse:BinarySecurityToken id="Kerberosv5ST"
                                wsse:ValueType="Kerberosv5ST"
                              wsse:EncodingType="wsse:Base64Binary">
         MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
      </wsse:BinarySecurityToken>
      <wsse:DerivedKey wsu:Id="dk">
          <wsse:SecurityTokenReference>
              <wsse:Reference URI="#Kerberosv5ST"/>
          </wsse:SecurityTokenReference>
              ...
      </wsse:DerivedKey>
      ...
      <ds:Signature xmlns:ds="...">
          ...
          <ds:KeyInfo>
              <wsse:SecurityTokenReference>
                  <wsse:Reference URI="#dk"/>
              </wsse:SecurityTokenReference>
          </ds:KeyInfo>
       </ds:Signature>
   </wsse:Security>
 ...
```

The following example illustrates a derived key based on a security context token:

```
 ...
   <wsse:Security
        xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/07/secext"
        xmlns:xenc="http://www.w3.org/2001/04/xmlend#">
      <wsse:SecurityContextToken id="sct">
          <wsu:Identifier>uuid:...</wsu:Identifier>
      </wsse:SecurityContextToken>
      <wsse:DerivedKey wsu:Id="dk">
          <wsse:SecurityTokenReference>
              <wsse:Reference URI="#sct"/>
```

```
            </wsse:SecurityTokenReference>
            ...
        </wsse:DerivedKey>
        ...
        <ds:Signature xmlns:ds="...">
            ...
            <ds:KeyInfo>
                <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#dk"/>
                </wsse:SecurityTokenReference>
            </ds:KeyInfo>
         </ds:Signature>
     </wsse:Security>
   ...
```

6. Security Considerations

All messages SHOULD include an authenticator be it in a GSS-API token or using XML Signature.

All requests and responses for security tokens MUST be signed.

7. Acknowledgements

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

Messaoud Benantar, IBM
Tony Cowan, IBM
Richard Ward, Microsoft

8. References

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

[RFC1510]
J. Kohl, C. Neuman, "The Kerberos Network Authentication Service (V5)," RFC 1510, Digital Equipment Corporation, ISI, September 1993.

[RFC2478]
J. Kohl, C. Neuman, "The Simple and Protected GSS-API Negotiation Mechanism," RFC 2478, December 1998.

[RFC2743]
E. Baize and D. Pinkas, "Generic Security Service Application Program Interface Version 2 Update 1," RFC 2743, January 2000.

[SHA-1]

FIPS PUB 180-1. Secure Hash Standard. U.S. Department of Commerce / National Institute of Standards and Technology. http://csrc.nist.gov/publications/fips/fips180-1/fip180-1.txt

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

[SOAP12]
W3C Candidate Recommendation, "SOAP Version 1.2 Part 1: Messaging Framework", 19 December 2002

[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.

[WSSec]
"Web Services Security (WS-Security)," IBM, Microsoft & VeriSign, 5 April 2002

[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

[XML-Encrypt]
W3C Working Draft, "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.

[WSSecConv]
"Web Services Secure Conversation Language (WS-SecureConversation)," IBM, Microsoft, RSA Security & VeriSign, 18 December 2002

[WSTrust]
"Web Services Trust Language (WS-Trust)," IBM, Microsoft, RSA Security, VeriSign, 18 December 2002

[WSPolicy]
"Web Services Policy Framework (WS-Policy)," Microsoft, BEA Systems, IBM and SAP, 28 May 2003

[WSSecPolicy]
"Web Services Security Policy Language (WS-SecurityPolicy)," IBM, Microsoft, RSA Security Inc. & VeriSign, 18 December 2002