

## Chapter 1. XML Security©

The eXtensible Markup Language (XML) allows organizations to agree on a common, interoperable markup for document formatting (vocabulary), and use it to exchange business documents, taking advantage of many automated tools that create and handle XML documents. Therefore, XML is widely used for electronic business and commerce protocols over insecure networks such as the Internet and mobile (wireless) networks. XML is applicable both for business to business (B2B) and for business to consumer (B2C) applications. Some examples include XML-EDI and eBX (for B2B) and OFX, IFX (for e-banking). Security is very important for many e-commerce applications.

Recently, there have been multiple efforts to provide security mechanisms for XML. In this chapter, we describe two joint IETF/W3C efforts, which are useful to many applications: XML Digital Signature [DSIG] and XML Encryption [XML-Enc]. Implementations of both can be downloaded from [XML-Sec]. I describe a simple, secure XML transport protocol (SeXTP), built using XML Encryption and DSIG. SeXTP may be useful to secure client-server applications.

### 1.1. Secure Hashing of XML Objects

Before we describe XML signatures, we discuss how the specification for the cryptographic hash of (fragments of) XML documents. The DSIG Reference element specifies the hash (digest) value of a resource, and the hash algorithm (with SHA-1 as mandatory to implement). The Reference object has an optional URI<sup>1</sup> attribute, identifying the hashed resource. A Reference binds between a specific resource (identified by URI) and its content. See

```
<Reference URI="#ToBeSigned">
  <Transforms>
    <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315/">
  </Transforms>
  <DigestValue> slj87s...</DigestValue>
  <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1/">
</Reference>
```

Figure 1-1.

Committing to a value by a hash requires the use of a specific octet (byte) stream in evaluating and verification. The optional <Transforms> element specifies one or more transformations that we apply, in sequence, to the resource, to create the exact octet stream. The URI defines input to the first transform, and output of each transform is input to the next. DSIG defines several standard transforms: base64 decoding, Xpath filtering, Enveloped Signature (see later), XSLT mapping and canonicalization.

---

© COPYRIGHT NOTICE. Copyright © 2001-2002 by author (Amir Herzberg). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission from the author.

<sup>1</sup> URI stands for Universal Resource Identifier.

The *canonicalization* transforms, such as the mandatory to implement [Canonical XML \[XML-C14N\]](#), produce the *canonical form* of an XML object. The canonical form remains the same following any syntactic (but not semantic) modifications to the object. By canonicalizing an XML object before hashing, the hash (and any signature applied to it) remains valid even if we change the document syntax, e.g. for namespace resolution or to use a different line ending convention.

```
<Reference URI="#ToBeSigned">
  <Transforms>
    <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315/">
  </Transforms>
  <DigestValue> sljj87s...</DigestValue>
  <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1/">
</Reference>
```

Figure 1-1: **Example of Reference and Hash (Digest) Elements**

## 1.2. XML Digital Signatures (DSIG)

The XML Digital Signature working group [DSIG] is a joint effort of the Internet Engineering Task Force and the World Wide Web Consortium. Its goal is to develop an XML compliant syntax for representing signatures over Web resources and portions of protocol messages, and procedures for computing and verifying such signatures. Such signatures will be able to provide the security services of *data integrity*, *authentication*, and optionally *non-repudiation*.

Public key digital signatures such as RSA, providing non-repudiation, are computationally intensive operations. Therefore, DSIG uses *collision-resistant hashing* of the content before signing it, to limit the length to which the public key algorithm is applied. DSIG also allows shared-key authentication (Message Authentication Code – MAC), providing authentication but not non-repudiation, which is much more efficient than public key signatures. DSIG uses the <Signature> element also<sup>2</sup> when providing only shared key authentication (MAC).

The signed information may be an arbitrary document (handled as an opaque sequence of bits), but more often, it will be an XML object – a complete document or fragments of a document. The ability to sign only specific elements of an XML documents allows the unsigned parts of the XML document to be enhanced, modified or removed (for privacy or efficiency), keeping the signature valid.

DSIG signatures may contain the signed XML object (*enveloping*), be contained in the XML object (*enveloped*), or be *detached* from the signed object or document. When the signed XML object envelops the signature, the enveloped signature value itself is not included in signature calculation and validation computation. For this, use the *enveloped-signature transform*, removing the whole Signature element containing it from the digest calculation.

---

<sup>2</sup> Most cryptographic literature refer as digital signature only to asymmetric, public-key mechanisms, providing non-repudiation, i.e. where the ability to verify a signature uses public information (key) and does not imply the ability to sign (using private information or key).

### 1.2.1. Structure of DSIG Signatures

Figure 1-2 shows, in a simplified, `shorthand` notation, the structure of DSIG signatures. Elements appear zero or more times if followed by “\*”, zero or once if followed by “?” and once or more if followed by “+”. When not followed by a symbol, elements appear exactly once. Attributes were removed as well as the contents and end tag of some elements.

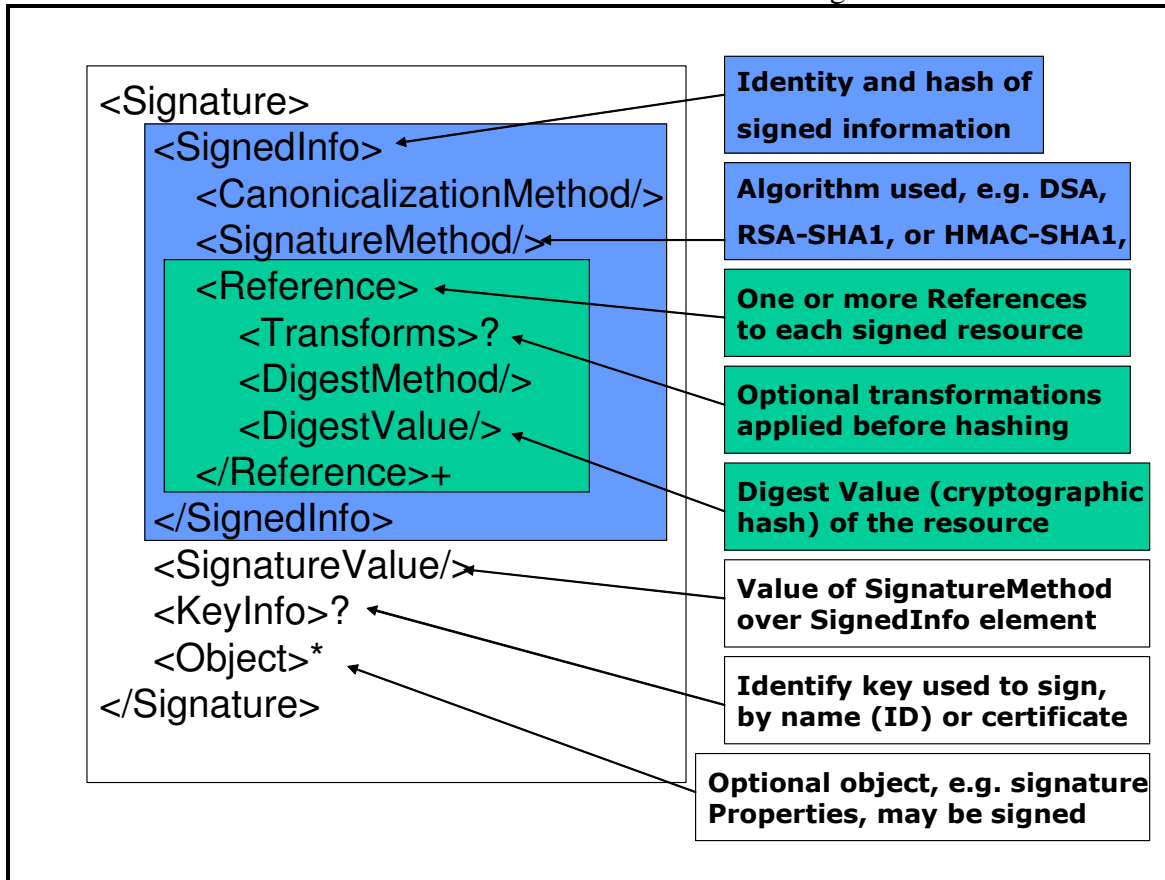


Figure 1-2: Structure of DSIG Signature Object

A simplified<sup>3</sup> example of a signature object appears in Figure 1-3. The signature object contains three elements in this example<sup>4</sup>: the signed information (in <SignedInfo> object), the signature value itself (in <SignatureValue>), and <KeyInfo> - information defining the keys used to compute and validate the signature.

<sup>3</sup> This simple example is valid, except that it does not contain <CanonicalizationMethod>, <DigestMethod> and <SignatureMethod> tags identifying the canonicalization, hashing and signature algorithms, respectively.

<sup>4</sup> There is one more, optional element of the Signature aggregate, <Object>, that can contain arbitrary content, e.g. for enveloping signature; see below.

```

<Signature>
  <SignedInfo>
    <Reference URI="#ToBeSigned">
      <DigestValue> sljlj87s...</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>gahpih...</SignatureValue>
  <KeyInfo>
    <KeyName>My Key Name</KeyName>
  </KeyInfo>
</Signature>

```

Figure 1-3: Simplified example of DSIG Signature

### 1.2.2. Signed Information (<SignedInfo> element)

The <SignedInfo> element contains one or more <Reference> elements, each identifying a signed resource and providing its secure hash value (see `Secure Hashing of XML Objects` above). It also identifies two algorithms (methods), which are applied to the entire <SignedInfo> object to generate the signature: first canonicalization and then a signature algorithm. Specify the algorithm for each of these goals, with the <CanonicalizationMethod> and <SignatureMethod> tags, respectively, each of them having a mandatory `Algorithm` attribute, identifying the algorithm (as a URI). The mandatory signature algorithms are [hmac-sha1](#) (for MAC) and [dsa-sha1](#) (for public key signature using the Digital Signature Standard). It is also recommended to implement [rsa-sha1](#), using RSA. Both public key method (DSA and RSA) first hash their inputs, to reduce its length. Since we also hash to compute each <Reference>, this implies that with DSIG, we must apply hashing at least twice.

### 1.2.3. Signature Value (<SignatureValue> element)

The <SignatureValue> element contains the actual value of the signature, encoded using base64.

### 1.2.4. Key Information (<KeyInfo> element)

One special property of a signature is the key used to validate it. DSIG identifies the key using the KeyInfo aggregate, which is an optional element of the Signature object. If KeyInfo is omitted, the recipient is expected to be able to identify the key based on application context. KeyInfo enables the verifier to identify, and obtain (if unknown), the key needed to validate the signature. For example, when using DSIG signature for shared secret key Message Authentication Code (MAC), KeyInfo identifies the key by the KeyName element of KeyInfo, containing a unique key identifier.

DSIG can also identify a public key signature key using KeyName. However, in this case, it is also possible to include the actual value of the public key in KeyInfo, inside the

KeyValue element. KeyInfo may also include one or more public key certificates<sup>5</sup> and/or certificate revocation list(s). Alternatively, KeyInfo may include a reference, using the optional RetrievalMethod element of KeyInfo, to `separate` certificates and/or revocation lists. Separating the certificates and/or revocation lists from the Signature Object allows sharing them among multiple Signature objects.

DSIG allows substantial flexibility in KeyInfo, allowing several certificate formats by optional elements in KeyInfo (for X.509, PGP, and SPKI) as well as the addition of arbitrary elements to KeyInfo (e.g. for yet additional certificate formats). I describe later how this flexibility allowed XML Encryption to reuse the DSIG KeyInfo element.

### 1.2.5. Optional Objects (<Object> element) and Enveloping Signature

The Object (optional) element of the Signature object can be used to include arbitrary objects within the Signature element. Its basic use is to create enveloping signature, where the signature object envelops (contains) actual signed content (referenced from SignedInfo). A special case of an enveloping signature is when placing a <Manifest> element within <Object>, to allow signing of multiple resources in one signature, with partial validation; see `Signing multiple references` and Figure 1-6 below.

The <Object> may include other elements, e.g. the SignatureProperties object [DSIG], defining properties of the signature, such as timestamp (time of signing).

### 1.2.6. Signature Creation and Validation Processes

Creation of DSIG signatures follows the simple process illustrated in Figure 1-4.

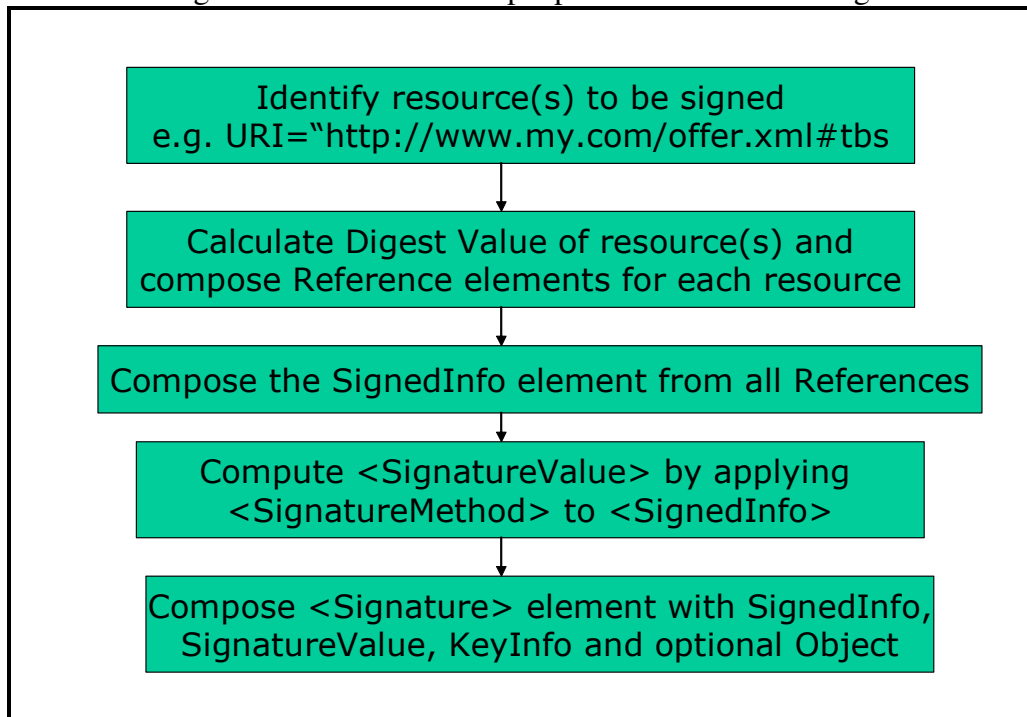


Figure 1-4: DSIG Signature Generation Process

<sup>5</sup> A certificate is a public key signature by a trusted third party called certificate authority, vouching for the validity of the public key, e.g. identifying the owner of the private key validated by this public key.

Validation of DSIG signatures involves validating each reference (against its hash - <DigestValue>), and then validation of the signature over the entire <SignedInfo> element. Figure 1-5 illustrates this process. Validating the hash of the references is an efficient process, which detects any unintentionally invalidated resources. Therefore it is preferable to perform it before the computationally-intensive validation of the public key signature (if used), identifying intentional tampering with the signature and preventing clogging attacks.

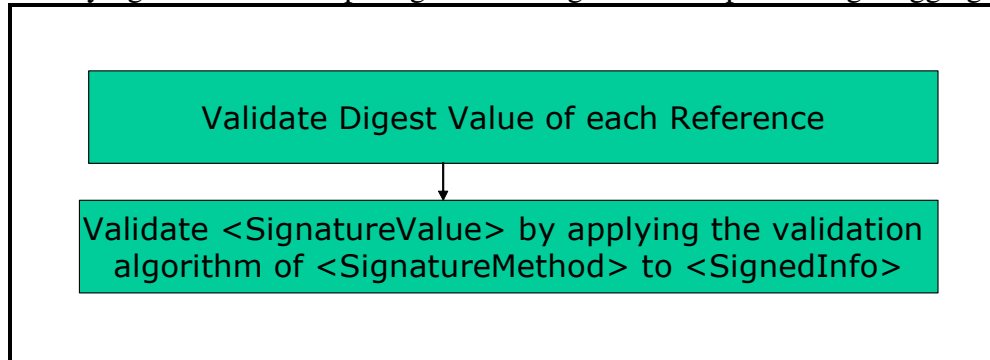


Figure 1-5: DSIG Signature Validation Process

#### 1.2.7. Signing multiple references and <Manifest>

An XML DSIG SignedInfo aggregate may contain multiple Reference elements. This allows the same signature to sign (authenticate) multiple resources. The main motivation for this is efficiency, especially when using public key signatures, which are computationally intensive. A single public key signature (over the SignedInfo element) provides authentication, integrity and non-repudiation for all of the Reference elements in the SignedInfo.

The validation of a signature containing multiple Reference elements involves validating each of the resources referred. This, in turn, requires each of these resources to be available for validation. Some applications need the efficiency of using a single public key signature to sign multiple resources, but require validation without all resources (for efficiency or confidentiality). DSIG supports this by including in the SignedInfo a Reference to the (optional to implement) Manifest object, which is simply a list of one or more Reference elements. Validity of the references in the Manifest is the responsibility of the application. The Manifest element allows application-controlled validation of signed resources, allowing the application to decide which resources must be available and valid. The Manifest element may be included in the Signature element by placing it in an Object element.

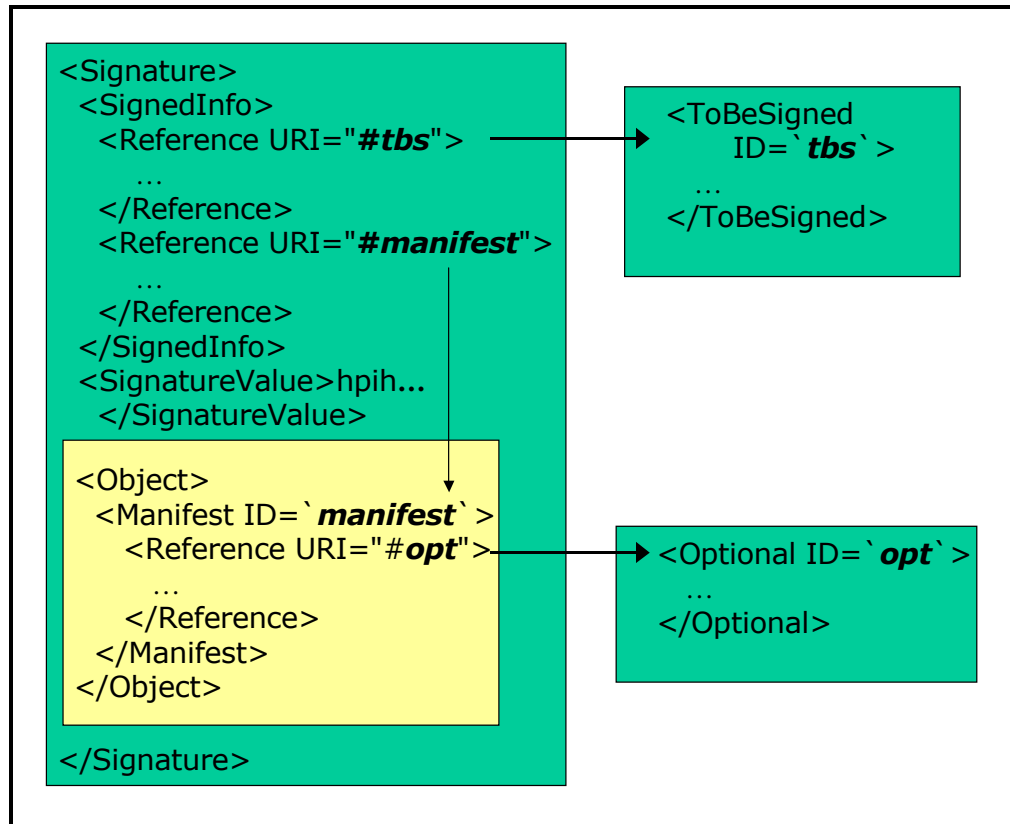


Figure 1-6: Signing Multiple References (all in same document)

### 1.3. XML Encryption

The XML Encryption working group [XML-Enc] is another joint effort of the Internet Engineering Task Force and the World Wide Web Consortium. At time of writing, this effort is still a draft, but I expect it to become a candidate recommendation soon (without major changes).

XML Encryption defines a process for encrypting *plaintext* data, producing *ciphertext*, and decrypting the ciphertext to retrieve the plaintext data. The draft also defines XML syntax to represent the ciphertext and information needed for the intended recipient to decrypt the ciphertext, such as identification or secure exchange of the decryption key. The plaintext may be an arbitrary, entire file, an entire XML document, or a fragment of an XML document. Only two forms of fragments of XML documents are allowed – an XML element or XML element content.

#### 1.3.1. Structure of XML Encryption Object (<EncryptedData>)

We place or reference the ciphertext with an XML <EncryptedData> element. When encrypting a fragment of an XML document, the <EncryptedData> element may replace the fragment. This may invalidate the XML document, unless the definition (DTD or schema) of the document allows this replacement.

Figure 1-7 shows, in simplified `shorthand` notation, the structure of the <EncryptedData> element. It contains three elements: <EncryptionMethod>, defining the encryption algorithm; <KeyInfo>, providing or identifying the encryption key; and <CipherData>,

providing the ciphertext and optional integrity protection for the plaintext. Both `<EncryptionMethod>` and `<KeyInfo>` are optional, as the sender and receiver may agree on the encryption method and key in advance (rather than specifying them with each encryption). Several elements use the definitions from DSIG, as indicated by use of the ``ds:`` namespace, e.g. `<ds:KeyInfo>`. We used the wildcard character `"*"` in `<ds:*>` to identify any one of several possible additional elements which DSIG allows to appear in `KeyInfo`, such as `<X509Data>`, mostly used to provide public key.

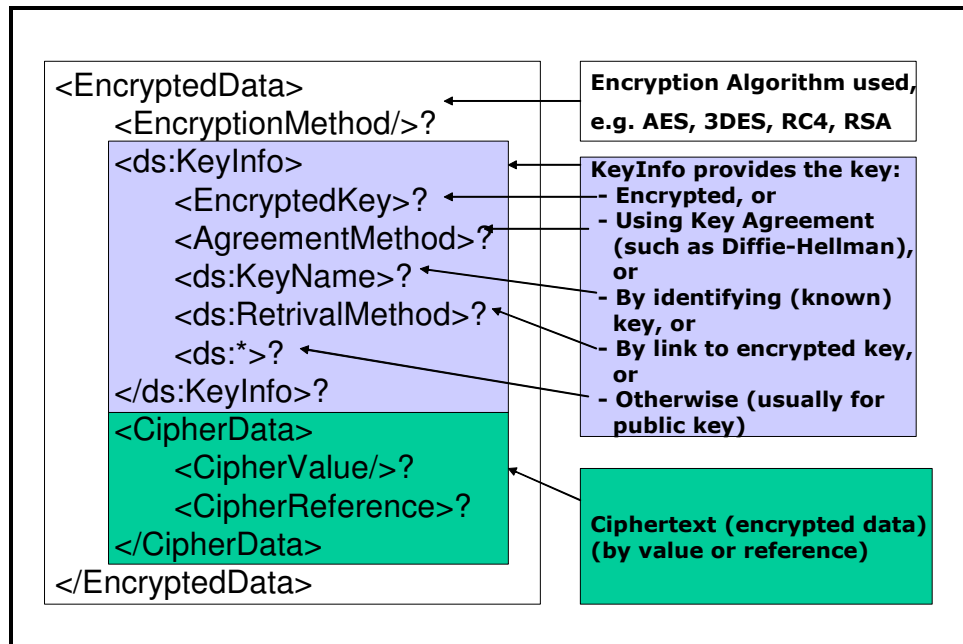


Figure 1-7: Structure of XML Encryption Object

A simple example of an XML Encryption object appears in **Figure 1-8**. This is a simple encryption of an XML element (and its contents), using pre-agreed encryption method (algorithm) and a shared key, identified by its name (“Encrypt Key 1”). The ciphertext is provided in the `<CipherValue>` element, as base64 encoded octet sequence.

```
<EncryptedData Type='Element'/>
  <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
    <ds:KeyName>Encrypt Key 1</ds:KeyName>
  </ds:KeyInfo>
  <CipherData>
    <CipherValue>Des353ADBEEF</CipherValue>
  </CipherData>
</EncryptedData>
```

Figure 1-8: Simple example of XML Encrypted Data element

A slightly more complex example of XML Encryption appears in appears in Figure 1-9 below. This example includes identification of the `<EncryptionMethod>` (for readability I removed the beginning of the URL, <http://www.w3.org/2001/04/xmlenc#aes128-cbc>). I refer to the ciphertext rather than provide its value directly, using the `<CipherReference>`

element; this element may include transforms to extract the ciphertext, e.g. using XPATH to retrieve particular components from an XML document.

```
<EncryptedData Id='ED'>
  <EncryptionMethod Algorithm='... /xmlenc#aes128-cbc'/>
  <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
    <ds:KeyName>Encrypt Key 2</ds:KeyName>
    <ds:RetrievalMethod URI='#EK' Type=" EncryptedKey">
  </ds:KeyInfo>
  <CipherData>
    <CipherReference URI=http://www.my.com/Ciphertext.xml>
      <Transforms>
        ...
      </Transforms>
    </CipherReference>
  </CipherData>
</EncryptedData>
```

**Figure 1-9:** Example of XML Encrypted Data with references to key and data

In this example, I provided the key in two ways: first, by identifying it by `<KeyName>`, as before; and second, by a reference (URI) to another XML object, in this case an object with identifier “EK” in the same document. I next discuss the different ways of providing keys in XML Encrypt.

### 1.3.2. Encryption key information (using `<ds:KeyInfo>`)

XML Encryption allows several ways to provide and/or identify the decryption key:

1. The key may be fixed in advanced, in which no identification is needed.
2. The parties may share the key in advance and identify it by name (identifier). This identifier is placed in the `<ds:KeyName>` element included in `<ds:KeyInfo>`, as shown in both examples above.
3. The key may be encrypted in the `<EncryptedKey>` object, also defined by XML Encryption, and placed in `<KeyInfo>`.
4. Use a key agreement algorithm such as Diffie-Hellman [DH76] to exchange the key. The `<AgreementMethod>` `Algorithm` attribute identify the algorithm, and its contents are the parameters providing the key to the recipient.
5. The key information may be retrieved from outside `<KeyInfo>`, using the `RetrievalMethod` element, as shown in Figure 1-9 above.

The `<EncryptedKey>` method is used when using `hybrid encryption`, i.e. encrypting the data using a symmetric (shared) key algorithm, by sending the shared key encrypted using the recipient’s public key. Figure 1-10 shows an `<EncryptedKey>` object, which belongs to the same XML document as Figure 1-9, which referred to the `EncryptedKey` object using its identifier (`Id='EK'`). The `CarriedKeyName` provides the name of the encrypted key,

allowing future use by referring to the KeyName rather than including the EncryptedKey again.

The EncryptionMethod for EncryptedKey is often a public key encryption algorithm such as RSA (two variants of RSA encryption formats, RSA-1\_5 and RSA-OAEP, are mandatory to implement). public key is known to the receiver, hence <KeyInfo> simply identifies it using <KeyName>.

The EncryptedKey object may also include an optional <ReferenceList> element, listing elements which used this key to encrypt data (<DataReference>) or other keys (<KeyReference>).

```
<EncryptedKey Id='EK' CarriedKeyName="Encrypt Key 2">
  <EncryptionMethod Algorithm='... /xmlenc#rsa-1_5'/>
  <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
    <ds:KeyName>Joe's public key</ds:KeyName>
  </ds:KeyInfo>
  <CipherData>
    <CipherValue>Des353ADBEEF</CipherValue>
  </CipherData>
  <ReferenceList>
    <DataReference URI='#ED'>
  </DataReference>
  </ReferenceList>
</EncryptedData>
```

Figure 1-10: Encrypted Key with reference to data encrypted with it

### 1.3.3. Ciphertext (<CipherData> Object)

The CipherData is a mandatory element that provides the encrypted data, by either containing it (in <CipherValue>, as base64 encoded string), or providing a reference to it (via <CipherReference>).

There is not much value in signing the ciphertext (encrypted data)<sup>6</sup>, as a way to sign a confidential document. Instead, we can sign a hash of the plaintext, by placing it in <DigestValue> (of course the plaintext must have sufficient redundancy to prevent exposure of any information from the secure hash of it). At time of writing, there is still some debate in the group whether to allow the DigestValue element; an alternate method to sign the plaintext is to place a hash of the plaintext in a <Manifest> tag (see `Signing multiple references and <Manifest>`).

### 1.3.4. Encryption Process

XML Encryption follows the process illustrated (slightly simplified) in Figure 1-11. If the recipient does not know the decryption key in advance, then the sender generates and sends it. We protect the key in transit by encryption, in EncryptedKey object, or using key agreement, via <AgreementMethod>. Either are sent inside the EncryptedData's <KeyInfo> element or referenced from <KeyInfo> using <RetrievalMethod>.

<sup>6</sup> Providing the decryption key does not help to validate the signature, as there may be two keys that result in two seemingly-valid decrypted documents (esp. with one-time pad).

If the plaintext data (to encrypt) is XML element or content, then we encode it using UTF-8 and perform any necessary transforms to it; otherwise, if an external resource, we simply consider it as an octet sequence. In any case, we calculate a hash (digest) value of it, if we may want to ensure integrity after decryption. We then encrypt the data, creating CipherValue, which we place in, or reference from, EncryptedData.

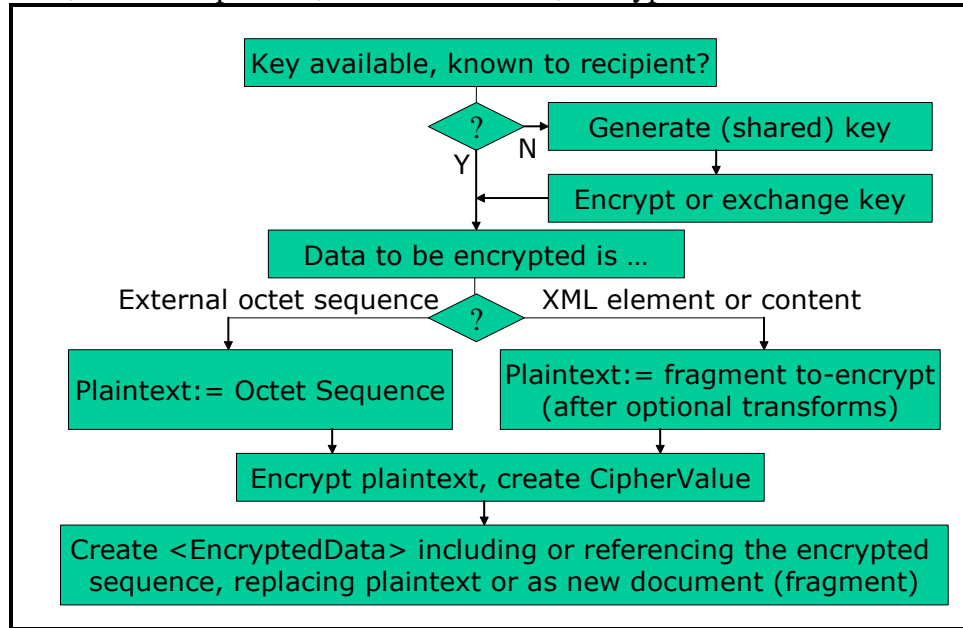


Figure 1-11: XML Encryption Process

#### 1.4. Secure XML Transport Protocol (SeXTP)

I now illustrate XML Encryption and DSIG by showing how we use them in SeXTP, a simple, efficient and highly secure transport protocol for client-server applications. SeXTP allows stateless servers and has minimal session setup overhead. It may be useful for many client-server e-commerce applications, esp. where the TLS/SSL protocol [R00] may be too heavy (requiring state in server, substantial session setup overhead, and complex implementation). See Table 1 comparing SeXTP and TLS/SSL. Most services are common to both. SeXTP, being stateless, does not provide freshness and no-replay for requests<sup>7</sup>.

<sup>7</sup> Prevention of replays when the server maintains state is easy. In a session protocol like TLS/SSL, the server sends a challenge with each response, which the client has to return with the request. In a non-interactive protocol, the client sends a request-counter with each request.

Table 1 TLS/SSL vs. (basic level of) SeXTP

Property\Protocol	TLS/SSL	SeXTP
Protocol type	Session	Client/server
Performance		
State-less server (more efficient)	No	Yes
Session setup overhead	High	Low
Security Services		
Confidentiality	Yes	Yes
Authentication of message source and integrity	Yes	Yes
Prevention of Replay and Re-ordering		
Of request	Yes	No
Of response	Yes	Yes
Non-repudiation of origin (identity of sender)	No	Yes
Clogging-prevention	No	Yes
Resilience to Penetrations and Cryptoanalysis		
Periodical key refresh	Yes	Yes
Key separation	Yes	Yes
Forward security	Yes	Yes

SeXTP provides secure transport of application request-response messages, with authentication, confidentiality, clogging prevention and non-repudiation. The secure transport uses a secret key shared between the parties, used for efficient encryption and authentication (MAC).

#### 1.4.1. SeXTP simplified use of XML Encryption and DSIG

SeXTP is using the XML Encryption and DSIG recommendations for providing confidentiality and authentication (including non-repudiation and integrity), respectively.

To encrypt, replace the SeXTP <Protocol> element with XML Encryption <EncryptedData> element, and identify the key with a simple <KeyName> identifier. To protect the integrity of the plaintext, namely ensure that the authentication and signature refer to the plaintext resulting from decryption, we include the <DigestValue> element in the <CipherData> object of <EncryptedData>. Before computing the digest value for authentication over the SeXTPtbs element, we first use a transform to remove the ciphertext (CipherValue) from the hashed sequence; this allows the signature to be used even if the encryption key is changed. Apart from these minor issues, this is simple instance of the XML Encryption specification, as described above.

In contract, for DSIG, and in order to simplify implementations and deployments, SeXTP makes and allows few simplifications, compared to the DSIG specification. Most simplifications are virtue of SeXTP's simple message structure, as illustrated in **Figure 1-12**.

The <Authenticator> object contains the <MAC> and (optionally) <PKSignature> elements authenticating (and optionally signing) the SeXTPtba object. In the `basic`

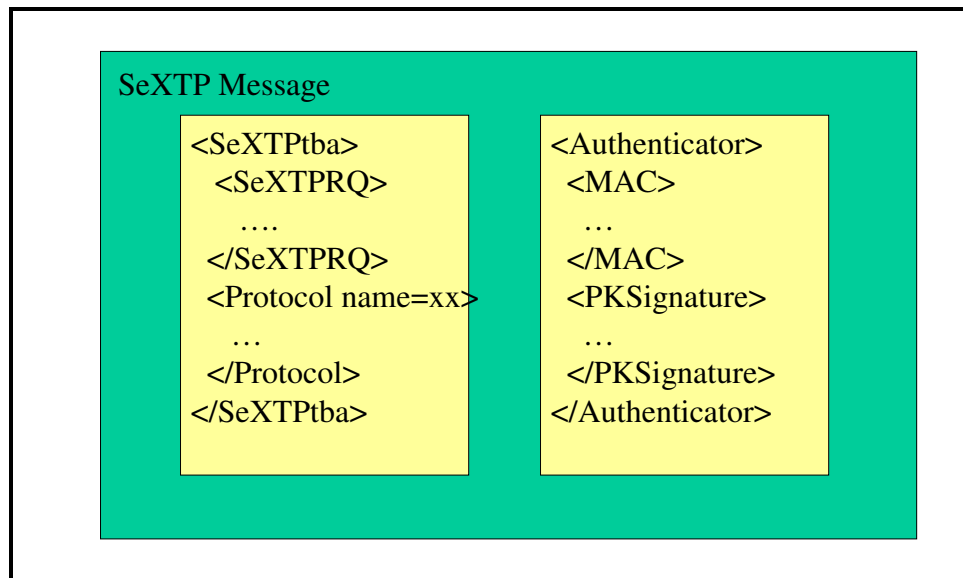
SeXTP implementation, authentication (and signature) is always on the complete SeXTPtba object. This reduces substantially the requirements from DSIG implementations. In particular:

1. Implementations can ignore the fact that the SeXTPtba object is in XML and sign it as an octet stream, before input into XML parser. In this case, canonicalization is not necessary.
2. DSIG implementations need only the (simple) `detached` mode and not the (more complex) enveloped/enveloping modes.
3. DSIG implementations sign always a single reference (the SeXTPtba part). In particular, there is no need to support the `Manifest` element.

Another simplification is syntactic. To properly use SeXTP, the server and client must agree in advance on the algorithms (methods) they use. There is no need to send this information with each encryption and signature (or MAC), as mandatory in DSIG (but not in XML Encryption). Therefore, we make the `Method` elements (e.g. DigestMethod) optional. To distinguish between public key signature and MAC, we define distinct tags, <PKSignature> and <MAC>, both derived from DSIG's <Signature> by making the `Method` elements optional.

#### 1.4.2. SeXTP messages

**Figure 1-12** shows SeXTP's message structure. SeXTP messages consist of two distinct XML objects, the SeXTPtba (`to be authenticated`) object and the SeXTP Authenticator object.



**Figure 1-12: SeXTP message structure**

The <SeXTPtba> object consists of two elements: SeXTP header (SeXTPRQ for requests, SeXTPRS for responses) and <Protocol>. The <Protocol> element contains arbitrary, protocol-specific elements; for confidentiality, we encrypt these elements and <Protocol> contains the <EncryptedData> element of XML Encryption.

The <SeXTPRQ> and <SeXTPRS> headers contain SeXTP management information, such as required services (encryption, signature, receipt), as attributes in requests. Both headers contain elements identifying client and server, for non-repudiation and to allow using the same key by multiple parties. In addition, both contain a unique request ID, randomly selected by the client, ensuring freshness.

Response headers also contain a unique response ID, and the client returns the last response ID in requests. However, the server does not reject a request whose response ID is not the last one sent, since requests are not necessarily in synchronized sequence. We log the response ID and use it for auditing and monitoring. To provide a limited assurance of freshness for requests, <SeXTPRQ> includes the time per the client's clock as well as the last time received from the server (from <SeXTPRS>).

### 1.4.3. SeXTP Shared Key Initialization and Refresh

SeXTP has a simple request-response, to initialize and refresh the shared secret *session key* ( $k$ ), and provide the public key of each party to the other. I explain these mechanisms briefly, as they are not specific to XML, and may be omitted for simple implementation using a fixed shared secret key.

SeXTP uses one of the following two `out-of-band` mechanisms to secure key initialization and refresh:

- Client and server share a secret initialization key, and/or
- Client knows or can validate the server's public key (e.g. via a certificate). In addition, the server may know or be able to validate the client's public key.

With a shared secret initialization code, this code authenticates the request and response (MAC). This efficient authentication allows us to identify bogus requests (clogging attempt), before performing any (computationally intensive) public key operations. When such a code does not exist a-priori, SeXTP begins with an unauthenticated `init code request`. If the server allows initialization without secret code, it replies with a hash of the current time and IP address of the client, used as initialization code. Thereby, the server can detect clogging attempts per IP address. This resembles the `cookie` mechanism to prevent clogging in [IKE]. For improved security against clogging attacks, and whenever practical, use a secret initialization key.

After the initialization key is available, SeXTP establishes a session key, as in Figure 1-13. Optionally (in green), the client knows initially the (hash of the) public key of the server and vice versa, or each can validate a certificate sent by the other, allowing validation of the exchanged public keys.

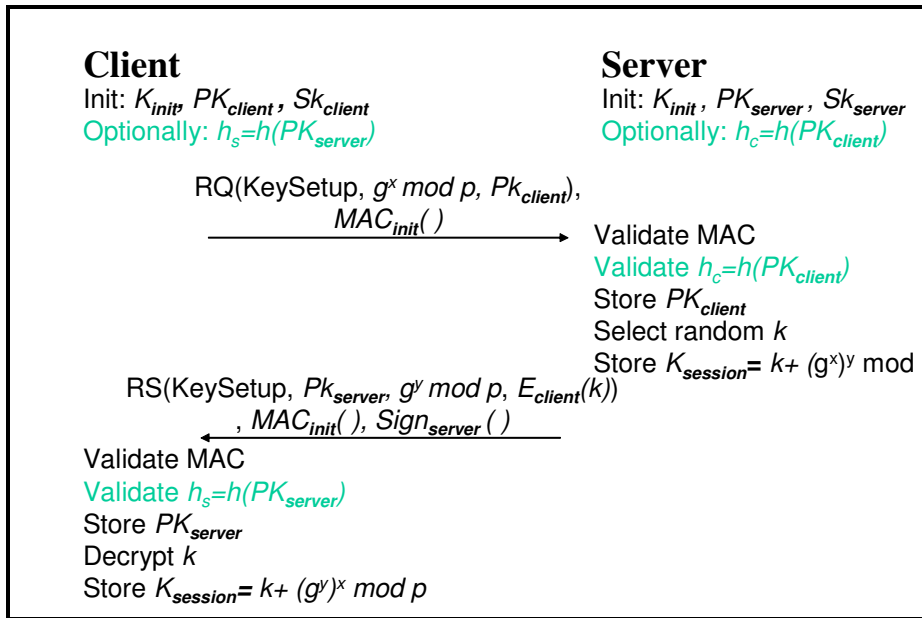


Figure 1-13: SeXTP key initialization and refresh request-response

SeXTP derives the session key from a key encrypted by the client’s public key, combine with a Diffie-Hellman key exchange [DH76]. For simpler implementations, one could use only one of these mechanisms (encrypted key or Diffie-Hellman exchange). However, each of them has its contribution in some cases. Specifically, Diffie-Hellman is subject to man-in-the-middle attack when the initialization key is weakly protected and may be exposed. On the other hand, by using Diffie-Hellman, we provide forward secrecy, namely even if the attacker exposes all keys at some time, she is not able to decrypt pre-recorded communication using previously used session keys.

The session key is not used directly. Instead, we derive from it two different keys, one for encryption and another for authentication, to prevent weaknesses due to using the same key for both purposes. Each of the keys should be ‘cryptographically independent’, i.e. exposure of the key use for one purpose should not help in exposing any of the other. We achieve this by deriving each of these keys by a pseudo-random permutation, i.e. a keyed function such that if the key is pseudo-random, then the attacker cannot tell it from a randomly chosen permutation. By default, SeXTP uses the HMAC function also for this purpose (this is common heuristic).

Table 2: Derivation of MAC and Encryption Keys from Session Key

Function	Key
Encryption	$f_k(\text{“Encrypt”})$
MAC	$f_k(\text{“MAC”})$

## 1.5. Bibliography

[DH76] W. Diffie and M.E. Hellman, “New Directions in Cryptography”, IEEE Tran. On Information Theory, 22 (1976), 644-654.

[DSIG] [XML-Signature Syntax and Processing](http://www.w3.org/TR/xmlsig-core/), <http://www.w3.org/TR/xmlsig-core/>, W3C Proposed Recommendation, 20 August 2001, Editors: [Donald Eastlake](#), [Joseph Reagle](#), [David Solo](#).

[IKE] D. Harkins and D. Carrel, [The Internet Key Exchange \(IKE\)](#), IETF Network Working Group, [RFC 2409](#), November 1998.

[TLS] Tim Dierks and Christopher Allen. [The TLS Protocol Version 1.0, RFC 2246](#), IETF Network Working Group, January 1999.

[XML-Enc] XML Encryption Syntax and Processing, WG Working Draft, 26 June 2001, Editors: [Donald Eastlake](#), [Joseph Reagle](#).

[XML-Sec] [XML Security Suite, IBM](#), available for download from <http://www.alphaworks.ibm.com/tech/xmlsecuritysuite>.

