Digital Signature Service Core Protocols, Elements, and Bindings

4th Committee Draft, 24 April 2006 (WD 42)

Document identifier:

oasis-dss-1.0-core-spec-cd-r04.doc

Location:

http://docs.oasis-open.org/dss/v1.0/

Editor:

Stefan Drees, *individual* <stefan@drees.name>

Contributors:

Dimitri Andivahis, Surety
Glenn Benson, JPMorganChase
Juan Carlos Cruellas, *individual*
Frederick Hirsch, Nokia
Pieter Kasselman, Cybertrust
Andreas Kuehne, *individual*
Konrad Lanz, Austria Federal Chancellery <Konrad.Lanz@iaik.tugraz.at>
Tommy Lindberg, *individual*
Paul Madsen, Entrust
John Messing, American Bar Association
Tim Moses, Entrust
Trevor Perrin, *individual*
Nick Pope, *individual*
Rich Salz, DataPower
Ed Shallow, Universal Postal Union

Abstract:

This document defines XML request/response protocols for signing and verifying XML documents and other data. It also defines an XML timestamp format, and an XML signature property for use with these protocols. Finally, it defines transport and security bindings for the protocols.

Status:

This is a **Committee Draft** produced by the OASIS Digital Signature Service Technical Committee. Committee members should send comments on this draft to dss@lists.oasis-open.org.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Digital Signature Service TC web page at http://www.oasis-open.org/committees/dss/ipr.php.

Table of Contents

1 Introduction

This specification defines the XML syntax and semantics for the Digital Signature Service core protocols, and for some associated core elements. The core protocols support the server-based creation and verification of different types of signatures and timestamps. The core elements include an XML timestamp format, and an XML signature property to contain a representation of a client's identity.

The core protocols are typically *bound* into other protocols for transport and security, such as HTTP and TLS. This document provides an initial set of bindings. The core protocols are also typically *profiled* to constrain optional features and add additional features. Other specifications are being produced which profile the core for particular application scenarios.

The following sections describe how to understand the rest of this specification.

1.1 Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD",

"SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this specification are to be interpreted as described in IETF RFC 2119 **[RFC 2119]**. These keywords are capitalized when used to unambiguously specify requirements over protocol features and behavior that affect the interoperability and security of implementations. When these words are not capitalized, they are meant in their natural-language sense.

This specification uses the following typographical conventions in text: `<DSSElement>`, `<ns:ForeignElement>`, `Attribute`, **Datatype**, `OtherCode`.

```
Listings of DSS schemas appear like this.
```

1.2 Schema Organization and Namespaces

The structures described in this specification are contained in the schema file **[Core-XSD]**. All schema listings in the current document are excerpts from the schema file. In the case of a disagreement between the schema file and this document, the schema file takes precedence.

This schema is associated with the following XML namespace:

```
urn:oasis:names:tc:dss:1.0:core:schema
```

If a future version of this specification is needed, it will use a different namespace.

Conventional XML namespace prefixes are used in the schema:

- The prefix `dss:` stands for the DSS core namespace **[Core-XSD]**.
- The prefix `ds:` stands for the W3C XML Signature namespace **[XMLSig]**.
- The prefix `xs:` stands for the W3C XML Schema namespace **[Schema1]**.
- The prefix `saml:` stands for the OASIS SAML Schema namespace **[SAMLCore1.1]**.

Applications MAY use different namespace prefixes, and MAY use whatever namespace defaulting/scoping conventions they desire, as long as they are compliant with the Namespaces in XML specification **[XML-ns]**.

The following schema fragment defines the XML namespaces and other header information for the DSS core schema:

```
<xs:schema xmlns:dss="urn:oasis:names:tc:dss:1.0:core:schema"
```

```
              xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
              xmlns:xs="http://www.w3.org/2001/XMLSchema"
              xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
              targetNamespace="urn:oasis:names:tc:dss:1.0:core:schema"
              elementFormDefault="qualified"
              attributeFormDefault="unqualified">
```

1.3 DSS Overview (Non-normative)

This specification describes two XML-based request/response protocols – a signing protocol and a verifying protocol. Through these protocols a client can send documents (or document hashes) to a server and receive back a signature on the documents; or send documents (or document hashes) and a signature to a server, and receive back an answer on whether the signature verifies the documents.

These operations could be useful in a variety of contexts – for example, they could allow clients to access a single corporate key for signing press releases, with centralized access control, auditing, and archiving of signature requests. They could also allow clients to create and verify signatures without needing complex client software and configuration.

The signing and verifying protocols are chiefly designed to support the creation and verification of XML signatures **[XMLSig]**, XML timestamps (see section 5.1), binary timestamps **[RFC 3161]** and CMS signatures **[RFC3369]**. These protocols may also be extensible to other types of signatures and timestamps, such as PGP signatures **[RFC 2440]**.

It is expected that the signing and verifying protocols will be *profiled* to meet many different application scenarios. In anticipation of this, these protocols have only a minimal set of required elements, which deal with transferring "input documents" and signatures back and forth between client and server. The input documents to be signed or verified can be transferred in their entirety, or the client can hash the documents themselves and only send the hash values, to save bandwidth and protect the confidentiality of the document content.

All functionality besides transferring input documents and signatures is relegated to a framework of "optional inputs" and "optional outputs". This document defines a number of optional inputs and outputs. Profiles of these protocols can pick and choose which optional inputs and outputs to support, and can introduce their own optional inputs and outputs when they need functionality not anticipated by this specification.

Examples of optional inputs to the signing protocol include: what type of signature to produce, which key to sign with, who the signature is intended for, and what signed and unsigned properties to place in the signature. Examples of optional inputs to the verifying protocol include: the time for which the client would like to know the signature's validity status, additional validation data necessary to verify the signature (such as certificates and CRLs), and requests for the server to return information such as the signer's name or the signing time.

The signing and verifying protocol messages must be transferred over some underlying protocol(s) which provide message transport and security. A *binding* specifies how to use the signing and verifying protocols with some underlying protocol, such as HTTP POST or TLS. Section 6 provides an initial set of bindings.

In addition to defining the signing and verifying protocols, this specification defines two XML elements that are related to these protocols. First, an XML timestamp element is defined in section 5.1. The signing and verifying protocols can be used to create and verify both XML and binary timestamps; a profile for doing so is defined in **[XML-TSP]**. Second, a RequesterIdentity element is defined in section 5.2. This element can be used as a signature property in an XML signature, to give the name of the end-user who requested the signature.

2 Common Protocol Structures

The following sections describe XML structures and types that are used in multiple places.

2.1 Type AnyType

The **AnyType** complex type allows arbitrary XML element content within an element of this type (see section 3.2.1 Element Content **[XML]**).

```
<xs:complexType name="AnyType">
  <xs:sequence>
    <xs:any processContents="lax"
            minOccurs="0"
            maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

2.2 Type InternationalStringType

The **InternationalStringType** complex type attaches an `xml:lang` attribute to a human-readable string to specify the string's language.

```
<xs:complexType name="InternationalStringType">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute ref="xml:lang" use="required">
    </xs:extension base="xs:string">
  </xs:simpleContent>
</xs:complexType>
```

2.3 Type saml:NameIdentifierType

The **saml:NameIdentifierType** complex type is used where different types of names are needed (such as email addresses, Distinguished Names, etc.). This type is borrowed from **[SAMLCore1.1]** section 2.4.2.2. It consists of a string with the following attributes:

`NameQualifier` [Optional]

The security or administrative domain that qualifies the name of the subject. This attribute provides a means to federate names from disparate user stores without collision.

`Format` [Optional]

A URI reference representing the format in which the string is provided. See section 7.3 of **[SAMLCore1.1]** for some URI references that may be used as the value of the `Format` attribute.

2.4 Element <InputDocuments>

The `<InputDocuments>` element is used to send input documents to a DSS server, whether for signing or verifying. An input document can be any piece of data that can be used as input to a signature or timestamp calculation. An input document can even *be* a signature or timestamp (for example, a pre-existing signature can be counter-signed or timestamped). An input document could also be a `<ds:Manifest>`, allowing the client to handle manifest creation while using the server to create the rest of the signature. Manifest validation is supported by the DSS Core.

The <InputDocuments> element consists of any number of the following elements:

<Document> [Any Number]

It contains an XML document as specified in section 2.4.2 of this document.

<TransformedData> [Any Number]

This contains the binary output of a chain of transforms applied by a client as specified in section 2.4.3 of this document.

<DocumentHash> [Any Number]

This contains the hash value of an XML document or some other data after a client has applied a sequence of transforms and also computed a hash value as specified in section 2.4.4 of this document.

<Other>

Other may contain arbitrary content that may be specified in a profile and can also be used to extend the Protocol for details see section 2.1.

```
<xs:element name="InputDocuments">
  <xs:complexType>
    <xs:sequence>
      <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:element ref="dss:Document"/>
        <xs:element ref="dss:TransformedData"/>
        <xs:element ref="dss:DocumentHash"/>
        <xs:element name="Other" type="dss:AnyType"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

When using DSS to create or verify XML signatures, each input document will usually correspond to a single <ds:Reference> element. Thus, in the descriptions below of the <Document>, <TransformedData> and <DocumentHash> elements, it is explained how certain elements and attributes of a <Document>, <TransformedData> and <DocumentHash> correspond to components of a <ds:Reference>.

2.4.1 Type DocumentBaseType

The **DocumentBaseType** complex type is subclassed by <Document>, <TransformedData> and <DocumentHash> elements. It contains the basic information shared by subclasses and remaining persistent during the process from input document retrieval until digest calculation for the relevant document. It contains the following elements and attributes:

ID [Optional]

This identifier gives the input document a unique label within a particular request message. Through this identifier, an optional input (see sections 2.7, 3.5.6 and 3.5.8) can refer to a particular input document.

RefURI [Optional]

This specifies the value for a <ds:Reference> element's URI attribute when referring to this input document. The RefURI attribute SHOULD be specified; no more than one RefURI attribute may be omitted in a single signing request.

RefType [Optional]

This specifies the value for a <ds:Reference> element's Type attribute when referring to this input document.

`SchemaRefs` [Optional]:

The identified schemas are to be used to identify `ID` attributes during parsing in sections 2.5.2, 3.3.1 1.a and 4.3 and for XPath evaluation in sections 2.6, 3.5.7, 4.3.1. If anything else but `<Schema>` are referred to, the server MUST report an error. If a referred to `<Schema>` is not used by the XML document instance this MAY be ignored or reported to the client in the `<Result>`/`<ResultMessage>`.

The Document is assumed to be valid against the first `<Schema>` referred to by `SchemaRefs`.

If a `<Schemas>` element is referred to first by `SchemaRefs` the document is assumed to be valid against the first `<Schema>` inside `<Schemas>`. In both cases, the remaining schemas may occur in any order and are used either directly or indirectly by the first schema.

The server MUST use the schemas to identify the ID attributes and MAY also perform complete validation against the schemas.

```
<xs:complexType name="DocumentBaseType" abstract="true">
  <xs:attribute name="ID" type="xs:ID" use="optional"/>
  <xs:attribute name="RefURI" type="xs:ID" use="optional"/>
  <xs:attribute name="RefType" type="xs:ID" use="optional"/>
  <xs:attribute name="SchemaRefs" type="xs:IDREFS" use="optional"/>
</xs:complexType>
```

2.4.2 Element <Document>

The `<Document>` element may contain the following elements (in addition to the common ones listed in section 2.4.1):

If the content inside one of the following mutually exclusive elements `<InlineXML>`, `<EscapedXML>` or `<Base64XML>` is not parseable XML data, after appropriate decoding, then the server MUST return a `<Result>` (section 2.6) issuing a `<ResultMajor>` `RequesterError` qualified by a `<ResultMinor>` `NotParseableXMLDocument`.

The server MUST use the `<Schema>` referred by `<SchemaRefs>` for validation if specified.

`<Base64XML>` [Optional] [Default]

This contains a base64 string obtained after base64 encoding of a XML data. The server MUST decode it to obtain the XML data.

`<InlineXML>` [Optional]

The InlineXMLType clearly expresses the fact, that content of `<InlineXML>` is inline XML that should be equivalent to a complete XML Document. I.e. having only one DocumentElement (see section 2.1 Well-Formed XML Documents **[XML]**) and not allowing anything but PI's and Comments before and after this one element.

It may contain the `ignorePIs` and `ignoreComments` attributes. These attributes indicate respectively, if processing instructions or comments MAY be ignored.

If one or both of these attributes are not present, their values MUST be considered to be "true".

`InlineXML` will work with PIs and/or Comments if `ignorePIs` and `ignoreComments` are false respectively and if the server supports such behavior.

`<EscapedXML>` [Optional]

This contains an escaped string. The server MUST unescape (escape sequences are processed to produce original XML sequence) it for obtaining XML data.

`<Base64Data>` [Optional]

This contains a base64 encoding of data that are not XML. The type of data is specified by its MimeType attribute, that may be required when using DSS with other signature types.

```
<xs:element name="Document" type="dss:DocumentType"/>

<xs:complexType name="DocumentType">
  <xs:complexContent>
    <xs:extension base="dss:DocumentBaseType">
      <xs:choice>
        <xs:element name="InlineXML" type="dss:InlineXMLType"/>
        <xs:element name="Base64XML" type="xs:base64Binary"/>
        <xs:element name="EscapedXML" type="xs:string"/>
        <xs:element ref="dss:Base64Data"/>
      </xs:choice>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="Base64Data">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:base64Binary">
        <xs:attribute name="MimeType" type="xs:string"
                      use="optional">
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:complexType name="InlineXMLType">
  <xs:sequence>
    <xs:any processContents="lax"/>
  </xs:sequence>
  <xs:attribute name="ignorePIs” type="xs:boolean"
                use="optional" default="true"/>
  <xs:attribute name="ignoreComments" type="xs:boolean"
                use="optional" default="true"/>
</xs:complexType>
```

2.4.3 Element <TransformedData>

The `<TransformedData>` element contains the following elements (in addition to the common ones listed in section 2.4.1):

`<ds:Transforms>` [Optional]

This is the sequence of transforms applied by the client and specifies the value for a `<ds:Reference>` element's `<ds:Transforms>` child element. In other words, this specifies transforms that the client has already applied to the input document before the server will hash it.

`<Base64Data>` [Required]

This gives the binary output of a sequence of transforms to be hashed at the server side.

```
<xs:element name="TransformedData">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="dss:DocumentBaseType">
        <xs:sequence>
         <xs:element ref="ds:Transforms" minOccurs="0"/>
```

```
          <xs:element ref="dss:Base64Data"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

2.4.4 Element <DocumentHash>

The `<DocumentHash>` element contains the following elements (in addition to the common ones listed in section 2.4.1):

`<ds:Transforms>` [Optional]

This specifies the value for a `<ds:Reference>` element's `<ds:Transforms>` child element when referring to this document hash. In other words, this specifies transforms that the client has already applied to the input document before hashing it.

`<ds:DigestMethod>` [Required]

This identifies the digest algorithm used to hash the document at the client side. This specifies the value for a `<ds:Reference>` element's `<ds:DigestMethod>` child element when referring to this input document.

`<ds:DigestValue>` [Required]

This gives the document's hash value. This specifies the value for a `<ds:Reference>` element's `<ds:DigestValue>` child element when referring to this input document.

```
<xs:element name="DocumentHash">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="dss:DocumentBaseType">
        <xs:sequence>
          <xs:element ref="ds:Transforms" minOccurs="0"/>
          <xs:element ref="ds:DigestMethod"/>
          <xs:element ref="ds:DigestValue"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

2.5 Element <SignatureObject>

The `<SignatureObject>` element contains a signature or timestamp of some sort. This element is returned in a sign response message, and sent in a verify request message. It may contain one of the following child elements:

`<ds:Signature>` [Optional]

An XML signature **[XMLSig]**.

`<Timestamp>` [Optional]

An XML, RFC 3161 or other timestamp (see section 5.1).

`<Base64Signature>` [Optional]

A base64 encoding of some non-XML signature, such as a PGP **[RFC 2440]** or CMS **[RFC 3369]** signature. The type of signature is specified by its `Type` attribute (see section 7.1).

`<SignaturePtr>` [Optional]

This is used to point to an XML signature in an input (for a verify request) or output (for a sign response) document in which a signature is enveloped.

`SchemaRefs` [Optional]

As described above in 2.4.1

A `<SignaturePtr>` contains the following attributes:

`WhichDocument` [Required]

This identifies the input document as in section 2.4.2 being pointed at (see also ID attribute in section 2.4.1).

`XPath` [Optional]

a) This identifies the signature element being pointed at.

b) The XPath expression is evaluated from the root node (see section 5.1 **[XPATH]**) of the document identified by `WhichDocument` after the XML data was extracted and parsed if necessary. The context node for the XPath evaluation is the document's DocumentElement (see section 2.1 Well-Formed XML Documents **[XML]**).

c) About namespace declarations for the expression necessary for evaluation see section 1 **[XPATH]**. Namespace prefixes used in XPath expressions MUST be declared within the element containing the XPath expression. E.g.: `<SignaturePtr xmlns:ds="http://www.w3.org/2000/09/xmldsig#" XPath="//ds:Signature">`. See also the following example below. A piece of a XML signature of a `<ds:Reference>` containing a `<ds:Transforms>` with a XPath filtering element that includes inline namespace prefixes declaration. This piece of text comes from one of the signatures that were generated in the course of the interoperability experimentation. As one can see they are added to the `<ds:XPath>` element:

```
<Reference URI="">
  <Transforms>
    <ds:Transform xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
                  Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
      <ds:XPath xmlns:upc1="http://www.ac.upc.edu/namespaces/ns1"
                xmlns:upc2="http://www.ac.upc.edu/namespaces/ns2">ancestor-
or-self::upc1:Root</ds:XPath>
    </ds:Transform>
  </Transforms>
  <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  <DigestValue>24xf8vfP3xJ40akfFAnEVM/zxXY=</DigestValue>
</Reference>
```

If the XPath does not evaluate to one element the server MUST return a `<Result>` (section 2.6) issuing a `<ResultMajor>` `RequesterError` qualified by a `<ResultMinor>` `XPathEvaluationError`.

`<Other>`

Other may contain arbitrary content that may be specified in a profile and can also be used to extend the Protocol.

The following schema fragment defines the `<SignatureObject>`, `<Base64Signature>`, and `<SignaturePtr>` elements:

```
<xs:element name="SignatureObject">
  <xs:complexType>
    <xs:sequence>
      <xs:choice>
        <xs:element ref="ds:Signature"/>
        <xs:element ref="dss:Timestamp"/>
        <xs:element ref="dss:Base64Signature"/>
        <xs:element ref="dss:SignaturePtr"/>
        <xs:element name="Other" ref="dss:AnyType"/>
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="SchemaRefs" type="xs:IDREFS" use="optional"/>
  </xs:complexType>
</xs:element>
<xs:element name="Base64Signature">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:base64Binary">
        <xs:attribute name="Type" type="xs:anyURI"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
<xs:element name="SignaturePtr">
  <xs:complexType>
    <xs:attribute name="WhichDocument" type="xs:IDREF"/>
    <xs:attribute name="XPath" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>
```

2.6 Element <Result>

The `<Result>` element is returned with every response message. It contains the following child elements:

`<ResultMajor>` [Required]

The most significant component of the result code.

`<ResultMinor>` [Optional]

The least significant component of the result code.

`<ResultMessage>` [Optional]

A message which MAY be returned to an operator, logged, used for debugging, etc.

```
<xs:element name="Result">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ResultMajor" type="xs:anyURI"/>
      <xs:element name="ResultMinor" type="xs:anyURI"
                  minOccurs="0"/>
      <xs:element name="ResultMessage"
                  type="InternationalStringType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The `<ResultMajor>` URIs MUST be values defined by this specification or by some profile of this specification. The `<ResultMajor>` values defined by this specification are:

`urn:oasis:names:tc:dss:1.0:resultmajor:Success`

The protocol executed successfully.

`urn:oasis:names:tc:dss:1.0:resultmajor:RequesterError`

The request could not be satisfied due to an error on the part of the requester.

`urn:oasis:names:tc:dss:1.0:resultmajor:ResponderError`

The request could not be satisfied due to an error on the part of the responder.

`urn:oasis:names:tc:dss:1.0:resultmajor:InsufficientInformation`

The request could not be satisfied due to insufficient information.

In case of doubt of who is responsible a `urn:oasis:names:tc:dss:1.0:resultmajor:ResponderError` is assumed.

This specification defines the following `<ResultMinor>` values, that are listed below, grouped by the respective associated `<ResultMajor>` code.

One of the following `<ResultMinor>` values MUST be returned when the `<ResultMajor>` code is Success.

`urn:oasis:names:tc:dss:1.0:resultminor:valid:signature:onAllDocuments`

The signature or timestamp is valid. Furthermore, the signature or timestamp covers all of the input documents just as they were passed in by the client.

`urn:oasis:names:tc:dss:1.0:resultminor:valid:signature:notAllDocumentsReferenced`

The signature or timestamp is valid. However, the signature or timestamp does not cover all of the input documents that were passed in by the client.

`urn:oasis:names:tc:dss:1.0:resultminor:invalid:incorrectSignature`

The signature fails to verify, for example due to the signed document being modified or the incorrect key being used.

`urn:oasis:names:tc:dss:1.0:resultminor:valid:signature:HasManifestResults`

The signature is valid with respect to XML Signature core validation. In addition, the message also contains `VerifyManifestResults`.
Note: In the case that the core signature validation failed no attempt is made to verify the manifest.

The following `<ResultMinor>` values is suggest MAY be returned when the `<ResultMajor>` code is `RequesterError`.

`urn:oasis:names:tc:dss:1.0:resultminor:referencedDocumentNotPresent`

A ds:Reference element is present in the ds:Signature containing a full URI, but the corresponding input document is not present in the request.

`urn:oasis:names:tc:dss:1.0:resultminor:KeyInfoNotProvided`

The required key information was not supplied by the client, but the server expected it to do so.

`urn:oasis:names:tc:dss:1.0:resultminor:MoreThanOneRefUriOmitted`

The server was not able to create a signature because more than one `RefUri` was omitted.

`urn:oasis:names:tc:dss:1.0:resultminor:NotParseableXMLDocument`

The server was not able to parse a Document.

`urn:oasis:names:tc:dss:1.0:resultminor:NotSupported`

The server doesn’t recognize or can’t handle any optional input.

`urn:oasis:names:tc:dss:1.0:resultminor:inappropriate:signature`

The signature or its contents are not appropriate in the current context.
For example, the signature may be associated with a signature policy and semantics which the DSS server considers unsatisfactory.

Further values for `<ResultMinor>` associated with `<ResultMajor>` code `urn:oasis:names:tc:dss:1.0:resultmajor:RequesterError` are left open to the implementer or profile to be defined with in their namespaces.

The following <ResultMinor> values MAY be returned when the <ResultMajor> code is ResponderError.

`urn:oasis:names:tc:dss:1.0:resultminor:GeneralError`

The processing of the request failed due to an error not covered by the existing error codes. Further details should be given in the result message for the user which may be passed on to the relevant administrator.

`urn:oasis:names:tc:dss:1.0:resultminor:invalid:KeyLookupFailed`

Locating the identified key failed (e.g. look up failed in directory or in local key file).

Further values for `<ResultMinor>` associated with `<ResultMajor>` code `urn:oasis:names:tc:dss:1.0:resultmajor:ResponderError` are left open to the implementer or profile to be defined within their namespaces.

The following `<ResultMinor>` values MAY be returned when the `<ResultMajor>` code is `InsufficientInformation`.

`urn:oasis:names:tc:dss:1.0:resultminor:CrlNotAvailiable`

The relevant certificate revocation list was not available for checking.

`urn:oasis:names:tc:dss:1.0:resultminor:OcspNotAvailiable`

The relevant revocation information was not available via the online certificate status protocol.

`urn:oasis:names:tc:dss:1.0:resultminor:CertificateChainNotComplete`

The chain of trust could not be established binding the public key used for validation to a trusted root certification authority via potential intermediate certification authorities.

2.7 Elements <OptionalInputs> and <OptionalOutputs>

All request messages can contain an `<OptionalInputs>` element, and all response messages can contain an `<OptionalOutputs>` element. Several optional inputs and outputs are defined in this document, and profiles can define additional ones.

The `<OptionalInputs>` contains additional inputs associated with the processing of the request. Profiles will specify the allowed optional inputs and their default values. The definition of an optional input MAY include a default value, so that a client may omit the `<OptionalInputs>` yet still get service from any profile-compliant DSS server.

If a server doesn't recognize or can't handle any optional input, it MUST reject the request with a `<ResultMajor>` code of `RequesterError` and a `<ResultMinor>` code of `NotSupported` (see section 2.6).

The `<OptionalOutputs>` element contains additional protocol outputs. The client MAY request the server to respond with certain optional outputs by sending certain optional inputs. The server MAY also respond with outputs the client didn't request, depending on the server's profile and policy.

The `<OptionalInputs>` and `<OptionalOutputs>` elements contain unordered inputs and outputs. Applications MUST be able to handle optional inputs or outputs appearing in any order within these elements. Normally, there will only be at most one occurrence of any particular optional input or output within a protocol message. Where multiple occurrences of an optional input (e.g. `<IncludeObject>` in section 3.5.6) or optional output are allowed, it will be explicitly specified (see section 4.6.8 for an example).

The following schema fragment defines the <OptionalInputs> and <OptionalOutputs> elements:

```
<xs:element name="OptionalInputs" type="dss:AnyType"/>

<xs:element name="OptionalOutputs" type="dss:AnyType"/>
```

2.8 Common Optional Inputs

These optional inputs can be used with both the signing protocol and the verifying protocol.

2.8.1 Optional Input <ServicePolicy>

The <ServicePolicy> element indicates a particular policy associated with the DSS service. The policy may include information on the characteristics of the server that are not covered by the Profile attribute (see sections 3.1 and 4.1). The <ServicePolicy> element may be used to select a specific policy if a service supports multiple policies for a specific profile, or as a sanity-check to make sure the server implements the policy the client expects.

```
<xs:element name="ServicePolicy" type="xs:anyURI"/>
```

2.8.2 Optional Input <ClaimedIdentity>

The <ClaimedIdentity> element indicates the identity of the client who is making a request. The server may use this to parameterize any aspect of its processing. Profiles that make use of this element MUST define its semantics.

The <SupportingInfo> child element can be used by profiles to carry information related to the claimed identity. One possible use of <SupportingInfo> is to carry authentication data that authenticates the request as originating from the claimed identity (examples of authentication data include a password or SAML Assertion **[SAMLCore1.1]**, or a signature or MAC calculated over the request using a client key).

The claimed identity may be authenticated using the security binding, according to section 6, or using authentication data provided in the <SupportingInfo> element. The server MUST check that the asserted <Name> is authenticated before relying upon the <Name>.

```
<xs:element name=”ClaimedIdentity”>
  <xs:complexType>
    <xs:sequence>
      <xs:element name=”Name” type=”saml:NameIdentifierType”/>
      <xs:element name=”SupportingInfo” type=”dss:AnyType”
                  minOccurs=”0”/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

2.8.3 Optional Input <Language>

The <Language> element indicates which language the client would like to receive **InternationalStringType** values in. The server should return appropriately localized strings, if possible.

```
<xs:element name="Language" type="xs:language"/>
```

2.8.4 Optional Input <AdditionalProfile>

The `<AdditionalProfile>` element can appear multiple times in a request. It indicates additional profiles which modify the main profile specified by the `Profile` attribute (thus the `Profile` attribute MUST be present; see sections 3.1 and 4.1 for details of this attribute). The interpretation of additional profiles is determined by the main profile.

```
<xs:element name="AdditionalProfile" type="xs:anyURI"/>
```

2.8.5 Optional Input <Schemas>

The `<Schemas>` element provides an in band mechanism for communicating XML schemas required for validating an XML document.

```
<xs:element name="Schemas" type="dss:SchemasType"/>
<xs:complexType name="SchemasType">
  <xs:sequence>
     <xs:element ref="dss:Schema" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="Schema" type="dss:DocumentType"/>
```

An XML schema is itself an XML document, however, only the following attributes, defined in `dss:DocumentType`, are meaningful for the `<Schema>` element:

`ID`

Used by relying XML document to identify a schema.

`RefURI`

The target namespace of the schema (i.e. the value of the `targetNamespace` attribute).

`RefType`

MUST NOT be used.

`SchemaRefs`

MUST NOT be used.

It is recommended to use `xml:id` as defined in **[xml:id]** as id in the payload being referenced by a `<ds:Reference>`, because the schema then does not have to be supplied for identifying the ID attributes.

2.9 Common Optional Outputs

These optional outputs can be used with both the signing protocol and the verifying protocol.

2.9.1 Optional Output <Schemas>

The `<Schemas>` element provides an in band mechanism for communicating XML schemas required for validating an XML document.

For a description of its syntax see above in section 2.8.5.

2.10 Type <RequestBaseType>

The `<RequestBaseType>` complex type is the base structure for request elements defined by the core protocol or profiles. It defines the following attributes and elements:

`RequestID` [Optional]

This attribute is used to correlate requests with responses. When present in a request, the server MUST return it in the response.

`Profile` [Optional]

This attribute indicates a particular DSS profile. It may be used to select a profile if a server supports multiple profiles, or as a sanity-check to make sure the server implements the profile the client expects.

`<OptionalInputs>` [Optional]

Any additional inputs to the request.

`<InputDocuments>` [Optional]

The input documents which the processing will be applied to.

```
<xs:complexType name="RequestBaseType">
  <xs:sequence>
    <xs:element ref="dss:OptionalInputs" minOccurs="0"/>
    <xs:element ref="dss:InputDocuments" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="RequestID" type="xs:string"
              use="optional"/>
  <xs:attribute name="Profile" type="xs:anyURI" use="optional"/>
</xs:element>
```

2.11 Type <ResponseBaseType>

The `<ResponseBaseType>` complex type is the base structure for response elements defined by the core protocol or profiles. It defines the following attributes and elements:

`RequestID` [Optional]

This attribute is used to correlate requests with responses. When present in a request, the server MUST return it in the response.

`Profile` [Required]

This attribute indicates the particular DSS profile used by the server. It may be used by the client for logging purposes or to make sure the server implements a profile the client expects.

`<Result>` [Required]

A code representing the status of the request.

`<OptionalOutputs>` [Optional]

Any additional outputs returned by the server.

```
<xs:complexType name="ResponseBaseType">
  <xs:sequence>
    <xs:element ref="dss:Result"/>
    <xs:element ref="dss:OptionalOutputs" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="RequestID" type="xs:string"
              use="optional"/>
  <xs:attribute name="Profile" type="xs:anyURI" use="required"/>
</xs:element>
```

2.12 Element <Response>

The `<Response>` element is an instance of the `<ResponseBaseType>` type. This element is useful in cases where the DSS server is not able to respond with a special response type. It is a general purpose response element for exceptional circumstances.

E.g.: "The server only supports verification requests.", "The server is currently under maintenance" or "The service operates from 8:00 to 17:00".

Other use cases for this type are expected to be described in special profiles (e.g. the Asynchronous profile).

```
<xs:element name=”Response” type=”ResponseBaseType”/>
```

3 The DSS Signing Protocol

3.1 Element <SignRequest>

The `<SignRequest>` element is sent by the client to request a signature or timestamp on some input documents. It contains the following attributes and elements inherited from `<RequestBaseType>`:

`RequestID` [Optional]

This attribute is used to correlate requests with responses. When present in a request, the server MUST return it in the response.

`Profile` [Optional]

This attribute indicates a particular DSS profile. It may be used to select a profile if a server supports multiple profiles, or as a sanity-check to make sure the server implements the profile the client expects.

`<OptionalInputs>` [Optional]

Any additional inputs to the request.

`<InputDocuments>` [Optional]

The input documents, which the signature will be calculated over. This element, while optional in `RequestBaseType`, is REQUIRED for the `<SignRequest>` element.

```
<xs:element name="SignRequest">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="dss:RequestBaseType"/>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

3.2 Element <SignResponse>

The `<SignResponse>` element contains the following attributes and elements inherited from `<ResponseBaseType>`:

`RequestID` [Optional]

This attribute is used to correlate requests with responses. When present in a request, the server MUST return it in the response.

`Profile` [Optional]

This attribute indicates the particular DSS profile used by the server. It may be used by the client for logging purposes or to make sure the server implements a profile the client expects.

`<Result>` [Required]

A code representing the status of the request.

`<OptionalOutputs>` [Optional]

Any additional outputs returned by the server.

In addition to `<ResponseBaseType>` the `<SignResponse>` element defines the following `<SignatureObject>` element:

`<SignatureObject>` [Optional]

The result signature or timestamp or, in the case of a signature being enveloped in an output document (see section 3.5.8), a pointer to the signature.

In the case of `<SignaturePlacement>` being used this MUST contain a `<SignaturePtr>`, having the same XPath expression as in `<SignaturePlacement>` and pointing to a `<DocumentWithSignature>` using it's `WhichDocument` attribute.

```
<xs:element name="SignResponse">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="dss:ResponseBaseType">
        <xs:sequence>
          <xs:element ref="dss:SignatureObject" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

3.3 Processing for XML Signatures

3.3.1 Basic Process for <Base64XML>

A DSS server that produces XML signatures SHOULD perform the following steps, upon receiving a `<SignRequest>`.

These steps may be changed or overridden by procedures defined for the optional inputs (for example, see section 3.5.6), or by the profile or policy the server is operating under.

The ordering of the `<Document>` elements inside the `<InputDocuments>` MAY be ignored by the server.

1. For each `<Document>` in `<InputDocuments>` the server MUST perform the following steps:
    a. In the case of `<Base64XML>` (see later sub-sections for other cases), the server base64-decodes the data contained within `<Document>` into an octet stream. This data MUST be a well formed XML Document as defined in **[XML]** section 2.1. If the `RefURI` attribute references within the same input document then the server parses the octet stream to NodeSetData (see **[XMLSig]** section 4.3.3.3) before proceeding to the next step.
    b. The data is processed and transforms applied by the server to produce a canonicalized octet string as required in **[XMLSig]** section 4.3.3.2.
    Note: Transforms can be applied as a server implementation MAY choose to increase robustness of the Signatures created. These Transforms may reflect idiosyncrasies of different parsers or solve encoding issues or the like. Servers MAY choose not to apply transforms in basic processing and extract the binary data for direct hashing or canonicalize the data directly if certain optional inputs (see sections 3.5.8 point 2 and 1.c.v, 3.5.9) are not to be implemented.
    Note: As required in **[XMLSig]** if the end result is an XML node set, the server MUST attempt to convert the node set back into an octet stream using Canonical XML **[XML-C14N]**.
    c. The hash of the resulting octet stream is calculated.
    d. The server forms a `<ds:Reference>` with the elements and attributes set as follows:

i. If the `<Document>` has a `RefURI` attribute, the `<ds:Reference>` element's `URI` attribute is set to the value of the `RefURI` attribute, else this attribute is omitted.
   A signature MUST NOT be created if more than one `RefURI` is omitted in the set of input documents and the server MUST report a RequesterError by setting `<ResultMajor>` `RequesterError` qualified by a `<ResultMinor>`.

ii. If the `<Document>` has a `RefType` attribute, the `<ds:Reference>` element's `Type` attribute is set to the value of the `RefType` attribute, else this attribute is omitted.

iii. The `<ds:DigestMethod>` element is set to the hash method used.

iv. The `<ds:DigestValue>` element is set to the hash value that is to be calculated as per **[XMLSig]**.

v. The `<ds:Transforms>` element is set to the sequence of transforms applied by the server in step b. This sequence MUST describe the effective transform as a reproducible procedure from parsing until hash.

2. References resulting from processing of optional inputs MUST be included. In doing so, the server MAY reflect the ordering of the `<Document>` elements.
3. The server creates an XML signature using the `<ds:Reference>` elements created in Step 1.d, according to the processing rules in **[XMLSig]**.

3.3.2 Process Variant for <InlineXML>

In the case of an input document which contains `<InlineXML>` Step 3.3.1 1.a is replaced with the following step:

1.
   a. The XML document is extracted from the DSS protocol envelope, without taking inherited namespaces and attributes. Exclusive Canonical XML **[XML-xcl-c14n]** MUST be applied to extract data AND assure context free extraction.
      If signed data is to be echoed back to the client and hence details could get lost refer to Appendix A.

In Step 3.3.1 step 1.d.v, the `<ds:Transforms>` element MUST begin with the canonicalization transform applied under revised step 3.3.2 1.a above.

3.3.3 Process Variant for <EscapedXML>

In the case of an input document which contains `<EscapedXML>` Step 3.3.1 1.a is replaced with the following:

1.
   a. In the case of `<EscapedXML>` the server unescapes the data contained within `<Document>` into a character string. If the RefURI references within the same input document the server parses the unescaped character content to NodeSetData if necessary. If the RefURI does not reference within the same input document then the server canonicalizes the characters or parsed NodeSetData (see **[XMLSig]** section 4.3.3.3) to octet stream if necessary before proceeding to the next step.

      Note: If the characters are converted to an octet stream directly a consistent encoding including ByteOrderMark has to be ensured.

In Step 3.3.1 1.d.v, the `<ds:Transforms>` element MUST begin with the canonicalization transform applied under revised step 3.3.3 1.a above.

3.3.4 Process Variant for <Base64Data>

In the case of an input document which contains `<Base64data>` Step 1 a and Step 1 b are replaced with the following:

1.
    a. The server base64-decodes the data contained within `<Document>` into an octet string.
    b. No transforms or other changes are made to the octet string before hashing.

       Note: If the RefURI references within the same input document the Document MUST also be referenced by `<IncludeObject>` in section 3.5.6 to include the object as base64 data inside a `<ds:Object>` otherwise a `<Result>` (section 2.6) issuing a `<ResultMajor>` `RequesterError` qualified by a `<ResultMinor>` `NotParseableXMLDocument`.

3.3.5 Process Variant for <TransformedData>

In the case of an input document which contains `<TransformedData>` Step 3.3.1 1 is replaced with the following:

1. For each `<TransformedData>` in `<InputDocuments>` the server MUST perform the following steps:
    a. The server base64-decodes the data contained within `<Base64Data>` of `<TransformedData>` into an octet string.
    b. Omitted.
    c. The hash over of the octet stream extracted in step a is calculated.
    d. as in 3.3.1 step 1d updated as follows
        i. replace the word "`<Document>`" by `<TransformedData>` otherwise as in as 3.3.1 step 1d.i.
        ii. replace the word "`<Document>`" by `<TransformedData>` otherwise as in as 3.3.1 step 1d.ii.
        iii. same as 3.3.1 step 1d.iii.
        iv. The `<ds:Transforms>` element is set to the sequence of transforms indicated by the client in the `<ds:Transforms>` element within the `<TransformedData>`. This sequence MUST describe the effective transform as a reproducible procedure from parsing until digest input.

3.3.6 Process Variant for <DocumentHash>

In the case of an input document which is provided in the form of a hash value in `<DocumentHash>` Step 3.3.1 1 is replaced with the following:

1. For each `<DocumentHash>` in `<InputDocuments>` the server MUST perform the following steps:
    a. Omitted.
    b. Omitted.

c. Omitted.

d. as in 3.3.1 step 1d updated as follows

i. replace the word "`<Document>`" by `<DocumentHash>` otherwise as in as 3.3.1 step 1d.i.

ii. replace the word "`<Document>`" by `<DocumentHash>` otherwise as in as 3.3.1 step 1d.ii.

iii. The `<ds:DigestMethod>` element is set to the value of `<ds:DigestMethod>` in `<DocumentHash>`

iv. The `<ds:DigestValue>` element is set to the value of `<ds:DigestValue>` in `<DocumentHash>`.

v. The `<ds:Transforms>` element is set to the sequence of transforms indicated by the client in the `<ds:Transforms>` element within `<DocumentHash>`, if any such transforms are indicated by the client. This sequence MUST describe the effective transform as a reproducible procedure from parsing until hash.

3.4 Basic Processing for CMS Signatures

A DSS server that produces CMS signatures **[RFC 3852]** SHOULD perform the following steps, upon receiving a `<SignRequest>`. These steps may be changed or overridden by the optional inputs, or by the profile or policy the server is operating under. With regard to the compatibility issues in validation / integration of PKCS#7 signatures and CMS implementations please refer to **[RFC 3852]** section 1.1.1 “Changes Since PKCS #7 Version 1.5”.

The `<SignRequest>` MUST contain either a single `<Document>` not having `RefURI`, `RefType` set or a single `<DocumentHash>` not having `RefURI`, `RefType`, `<ds:Transforms>` set:

1. If a <Document> is present, the server hashes its contents as follows:

   a. If the `<Document>` contains `<Base64XML>`, the server extracts the ancestry context free text content of the `<Base64XML>` as an octet stream by base64 decoding it’s contents.

   b. If the `<Document>` contains `<InlineXML>`, the server extracts the ancestry context free text content of the `<InlineXML>` as an octet stream as explained in (section 3.3.2 1.a). This octet stream has to be returned as `<TransformedDocument>`/`<Base64XML>`. For CMS signatures this only has to be returned in the case of CMS signatures that are external/detached/"without eContent", as these return the signed Data anyway.

   c. If the <Document> contains `<EscapedXML>`, the server unescapes the content of the `<EscapedXML>` as a character stream and converts the character stream to an octet stream using an encoding as explained in (section 3.3.3).

   d. If the `<Document>` contains `<Base64Data>`, the server base64-decodes the text content of the `<Base64Data>` into an octet stream.

   e. The server hashes the resultant octet stream.

2. The server forms a `SignerInfo` structure based on the input document. The components of the `SignerInfo` are set as follows:

   a. The `digestAlgorithm` field is set to the OID value for the hash method that was used in step 1.c (for a `<Document>`), or to the OID value that is equivalent to the input document's `<ds:DigestMethod>` (for a `<DocumentHash>`).
   b. The signedAttributes field's message-digest attribute contains the hash value that was calculated in step 1.e (for a `<Document>`), or that was sent in the input document's `<ds:DigestValue>` (for a `<DocumentHash>`). Other `signedAttributes` may be added by the server, according to its profile or policy, or according to the `<Properties>` optional input (see section 3.5.5).
   c. The remaining fields (`sid`, `signatureAlgorithm`, and `signature`) are filled in as per a normal CMS signature.
3. The server creates a CMS signature (i.e. a `SignedData` structure) containing the `SignerInfo` that was created in Step 2. The resulting `SignedData` should be detached (i.e. external or "without eContent") unless the client sends the `<IncludeEContent>` optional input (see section 3.5.9).

3.4.1 Process Variant for <DocumentHash>

In the case of a `<DocumentHash>` the processing by the server is as follows:

1. Omitted.
   a. Omitted.
   b. Omitted.
   c. Omitted.
   d. Omitted.
   e. Omitted.
2. Same as in 3.4 step 2
   a. Unchanged.
   b. Unchanged.
   c. Unchanged.
3. Essentially the same as in 3.4 step 3, with the requirement that the signature has to be external/detached/"without eContent", since `<DocumentHash>` is incompatible with `<IncludeEContent>`.

3.5 Optional Inputs and Outputs

This section defines some optional inputs and outputs that profiles of the DSS signing protocol might find useful. Section 2.8 defines some common optional inputs that can also be used with the signing protocol. Profiles of the signing protocol can define their own optional inputs and outputs, as well. General handling of optional inputs and outputs is discussed in section 2.7.

3.5.1 Optional Input <SignatureType>

The `<SignatureType>` element indicates the type of signature or timestamp to produce (such as a XML signature, a XML timestamp, a RFC 3161 timestamp, a CMS signature, etc.). See section 7.1 for some URI references that MAY be used as the value of this element.

```
<xs:element name="SignatureType" type="xs:anyURI"/>
```

3.5.2 Optional Input <AddTimestamp>

The `<AddTimestamp>` element indicates that the client wishes the server to embed a timestamp token as a property or attribute of the resultant or the supplied signature. The timestamp token will be applied to the signature value in the case of CMS/PKCS7 signatures or the `<ds:SignatureValue>` element in the case of XML signatures.

Note: Procedures for handling other forms of timestamp may be defined in profiles of the Core. In particular, the DSS XAdES profile **[DSS-XAdES-P]** defines procedures for generating timestamps over the content which is about to be signed (sometimes called content timestamps), and the DSS Timestamp profile **[DSS-TS-P]** defines procedures for handling standalone timestamps.

The schema definition of this optional input is as follows:

```
<xs:element name=”AddTimestamp”>
  <xs:complexType>
    <xs:attribute name=”Type” type=”xs:anyURI” use=”optional”/>
  </xs:complexType>
</xs:element>
```

The `Type` attribute, if present, indicates what type of timestamp to apply. Profiles that use this optional input MUST define the allowed values, and the default value, for the `Type` attribute (unless only a single type of timestamp is supported, in which case the `Type` attribute can be omitted).

Two scenarios for the timestamping of CMS signatures are supported by this Optional Input. They are as follows:

a) Create and embed a timestamp token into the signature being created as part of this `SignRequest.`

b) Create and embed a timestamp token into an existing signature which is passed in the `<InputDocuments>` element of this `SignRequest`.

3.5.2.1 Processing for CMS signatures time-stamping

In both scenarios, the timestamp token created by the server SHALL be created according to **[RFC 3161]**. The `MessageImprint` field within the `TstInfo` structure of the timestamp token will be derived from the signature value of the just-created or incoming signature depending on the scenario. The timestamp SHALL be embedded in the CMS signature as an unsigned attribute with the object identifier (see Appendix A of **[RFC 3161]**):

{ iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 14}

The signature and its embedded timestamp is returned in the `<SignatureObject>` of the `<SignResponse>`.

In scenario b) the incoming signature is passed in a `<Base64Data>` element, with the `MimeType` attribute set to application/pkcs7-signature.

The `Type` attribute of the `<AddTimestamp>` optional input SHALL be set to:

"urn:ietf:rfc:3161".

Note: In scenario b) the server SHOULD not verify the signature before adding the timestamp. If a client wishes that its signatures be verified as a condition of time stamping, the client SHOULD use the `<AddTimestamp>` optional input of the Verify protocol.

3.5.2.2 Processing for XML signatures time-stamping

In both scenarios, the timestamp token created by the server shall be a timestamp as defined in section 5.1 of the present document. It SHALL be a `<ds:Signature>` with at least two `<ds:Reference>` elements:

- One with the `Type` attribute set to "`urn:oasis:names:tc:dss:1.0:core:schema:XMLTimeStampToken`". and referencing a `<ds:Object>` element whose content is a `<TSTInfo>` element.
- The other referencing the `<ds:SignatureValue>` being timestamped.

[XMLSig] defines mechanisms for adding any kind of unsigned property to XML signatures. In addition XAdES defines a specific mechanism for incorporating signature timestamps for XML signatures. The present document does not establish a unique solution. It is a matter of server policy to decide how to incorporate signature timestamps in the XML signatures rendered to its clients, as long as the selected mechanism does not break the rules defined in **[XMLSig]**.

The signature and its embedded timestamp SHALL be returned in the `<SignatureObject>` of the `<SignResponse>`.

In scenario b) the incoming signature MUST be passed in on one of the following three elements `<EscapedXML>`, `<InlineXML>` or `<Base64XML>`.

The Type attribute of the `<AddTimestamp>` optional input SHALL be set to:

" urn:ietf:rfc:3275.

Note: In scenario b) the server SHOULD not verify the signature before adding the timestamp. If a client wishes that its signatures be verified as a condition of time stamping, the client SHOULD use the `<AddTimestamp>` optional input of the Verify protocol.

3.5.3 Optional Input <IntendedAudience>

The `<IntendedAudience>` element tells the server who the target audience of this signature is. The server MAY use this to parameterize any aspect of its processing (for example, the server MAY choose to sign with a key that it knows a particular recipient trusts).

```
<xs:element name="IntendedAudience">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Recipient" type="saml:NameIdentifierType"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

3.5.4 Optional Input <KeySelector>

The `<KeySelector>` element tells the server which key to use.

```
<xs:element name="KeySelector">
  <xs:complexType>
    <xs:choice>
      <xs:element ref="ds:KeyInfo"/>
      <xs:element name="Other" ref="dss:AnyType"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

3.5.5 Optional Input <Properties>

The `<Properties>` element is used to request that the server add certain signed or unsigned properties (aka “signature attributes”) into the signature. The client can send the server a particular value to use for each property, or leave the value up to the server to determine. The server can add additional properties, even if these aren’t requested by the client.

The `<Properties>` element contains:

`<SignedProperties>` [Optional]

These properties will be covered by the signature.

`<UnsignedProperties>` [Optional]

These properties will not be covered by the signature.

Each `<Property>` element contains:

`<Identifier>` [Required]

A URI reference identifying the property.

`<Value>` [Optional]

If present, the value the server should use for the property.

This specification does not define any properties. Profiles that make use of this element MUST define the allowed property URIs and their allowed values.

```
<xs:element name=”Properties”>
  <xs:complexType>
    <xs:sequence>
      <xs:element name=”SignedProperties”
                  type=”dss:PropertiesType” minOccurs=”0”/>
      <xs:element name=”UnsignedProperties”
                  type=”dss: PropertiesType” minOccurs=”0”/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name=”PropertiesType”>
  <xs:sequence>
    <xs:element ref=”dss:Property” maxOccurs=”unbounded”/>
  </xs:sequence>
</xs:complexType>

<xs:element name=”Property”>
  <xs:complexType>
    <xs:sequence>
      <xs:element name=”Identifier” type=”xs:anyURI”/>
      <xs:element name=”Value” type=”dss:AnyType”
                  minOccurs=”0”/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

3.5.6 Optional Input <IncludeObject>

Optional input `<IncludeObject>` is used to request the creation of an XMLSig enveloping signature as follows.

The attributes of `<IncludeObject>` are:

`WhichDocument` [Required]

Identifies the input document which will be inserted into the returned signature (see the `ID` attribute in section 2.4.1).

`hasObjectTagsAndAttributesSet`

If True indicates that the `<Document>` contains a `<ds:Object>` element which has been prepared ready for direct inclusion in the `<ds:Signature>`.

`ObjId` [optional]

Sets the `Id` attribute on the returned `<ds:Object>`.

`createReference`

This attribute set to true causes the `<ds:Object>` to be referenced by a `<ds:Reference>` and hence to be actually digested and signed. Otherwise it has to be referenced by another reference or it is just included but not signed.

```
<xs:element name="IncludeObject">
  <xs:complexType>
    <xs:attribute name="WhichDocument" type="xs:IDREF"/>
    <xs:attribute name="hasObjectTagsAndAttributesSet"
                  type="xs:boolean" default="false"/>
    <xs:attribute name="ObjId" type="xs:string"
                  use="optional"/>
    <xs:attribute name="createReference" type="xs:boolean"
                  use="optional" default="true"/>
  </xs:complexType>
</xs:element>
```

3.5.6.1 XML DSig Variant Optional Input <IncludeObject>

An enveloping signature is a signature having `<ds:Object>`s which are referenced by `<ds:Reference>`s having a same-document URI.

For each `<IncludeObject>` the server creates a new `<ds:Object>` element containing the document, as identified using the `WhichDocument` attribute, as its child. This object is carried within the enveloping signature. This `<Document>` (or documents) MUST include a “same-document” `RefURI` attribute (having a value starting with “#”) which references the data to be signed.

The URI in the `RefURI` attribute of this `<Document>` should at least reference the relevant parts of the Object to be included in the calculation for the corresponding reference. Clients MUST generate requests in a way that some `<ds:Reference>`’s URI values actually will reference the `<ds:Object>` generated by the server once this element will have been included in the `<ds:Signature>` produced by the server.

1. For each <IncludeObject> the server MUST carry out the following steps:
    a. The server identifies the <Document> that is to be placed into a <ds:Object> as indicated by the WhichDocument attribute.
    b. The data to be carried in the enveloping signature is extracted and decoded as described in 3.3.1 Step 1 a (or equivalent step in variants of the basic process as defined in 3.3.2 onwards depending of the form of the input document).
    c. if the hasObjectTagsAndAttributesSet attribute is false or not present the server builds the <ds:Object> as follows:
        i. The server generates the new <ds:Object> and sets its Id attribute to the value indicated in ObjId attribute of the optional input if present.

ii. In the case of the Document pointed at by WhichDocument having Base64Data, <ds:Object>('s) MIME Type is to be set to the value of <dss:Base64Data>('s) MIME Type value and the Encoding is to be set to http://www.w3.org/TR/xmlschema-2/#base64Binary

d. The server splices the to-be-enveloped documents as <ds:Object>(s) into the <ds:Signature>, which is to be returned.

The server then continues with processing as specified in section 3.3.1 if create reference is true otherwise this `<Document>` is excluded from further processing and basic processing is applied for the rest of the `<Document>s` as specified in section 3.3.1.

3.5.7 Optional Input <IncludeEContent>

In the case of the optional input `<IncludeEContent>` (that stands for include enveloped or encapsulated content) section 3.4 step 3 is overridden as follows.

3. The server creates a CMS signature (i.e. a `SignedData` structure) containing the `SignerInfo` that was created in Step 3. The resulting `SignedData` is now internal, as the document is enveloped in the signature.

For CMS details in this context please refer to **[RFC 3852]** sections 5.1 “SignedData Type” and 5.2 “EncapsulatedContentInfo Type”.

3.5.8 Enveloped Signatures, Optional Input <SignaturePlacement> and Output <DocumentWithSignature>

Optional input `<SignaturePlacement>` is used to request the creation of an XMLSig enveloped signature placed within an input document. The resulting document with the enveloped signature is placed in the optional output `<DocumentWithSignature>`.

The server places the signature in the document identified using the `WhichDocument` attribute. This `<Document>` MUST include a “same-document” RefURI attribute which references the data to be signed of the form RefURI=“”.

In the case of an XML input document, the client may instruct the server precisely where to place the signature with the optional `<XpathAfter>` and `<XpathFirstChildOf>` child elements. In the case of a non-XML input document, or when these child elements are omitted, then the server places the signature in the input document in accordance with procedures defined in a profile or as part of the server policy.

The `<SignaturePlacement>` element contains the following attributes and elements:

`WhichDocument` [Required]

Identifies the input document which the signature will be inserted into (see the `ID` attribute in section 2.4.1).

`CreateEnvelopedSignature`

If this is set to true a reference having an enveloped signature transform is created.

`<XpathAfter>` [Optional]

Identifies an element, inside the XML input document, after which the signature will be inserted. (The rules for XPath evaluation are those stated in section 2.5 SignatureObject)

`<XpathFirstChildOf>` [Optional]

Identifies an element, in the XML input document, which the signature will be inserted as the first child of. For details on the evaluation of The XPath expression see above

(<XpathAfter>). The signature is placed immediately after the start tag of the specified element.

```
<xs:element name="SignaturePlacement">
  <xs:complexType>
    <xs:choice>
      <xs:element name="XPathAfter" type="xs:string"/>
      <xs:element name="XPathFirstChildOf"
                  type="xs:string"/>
    </xs:choice>
    <xs:attribute name="WhichDocument" type="xs:IDREF"/>
    <xs:attribute name="CreateEnvelopedSignature"
                  type="xs:boolean" default="true"/>
  </xs:complexType>
</xs:element>
```

The <DocumentWithSignature> optional output contains the input document with the signature inserted. It has one child element:

<Document> [Required]

This contains the input document with a signature inserted in some fashion.

```
<xs:element name="DocumentWithSignature">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="dss:Document"/>
    <xs:sequence>
  </xs:complexType>
</xs:element>
```

For an XMLSig enveloped signature the client produces a request including elements set as follows:

1. The WhichDocument attribute is set to identify the <Document> to envelope the signature.
2. The RefURI attribute for the relevant <Document> is set to reference the relevant parts of the Document to be included in the calculation for the corresponding reference. This MUST be a relative reference within the same document. (e.g. URI="", URI="#xpointer(/)", URI="#xpointer(/DocumentElement/ToBeSignedElement)", URI="#xpointer(//ToBeSignedElements)", …).
3. The createEnvelopedSignature is set to true (or simply omitted).

If the <SignaturePlacement> element is present the server processes it as follows:

1. The server identifies the <Document> in which the signature is to be enveloped as indicated by the WhichDocument attribute.
2. This document is extracted and decoded as described in 3.3.1 Step 1.a (or equivalent step in variants of the basic process as defined in 3.3.2 onwards depending of the form of the input document).
3. The server splices the <ds:Signature> to-be-enveloped into the document.
4. If createEnvelopedSignature equals true create a <ds:Reference> for the document in question by performing Basic processing as in section 3.3.1 and Step 1.b to 1.d is performed with the following amendments:

   1.

      a. [No 1.a]

[replaced] Include an EnvelopedSignatureTransform as the first transform for calculation (even preceding transforms used for extraction) and continue as in 3.3.1 Step 1.b applied on the previously extracted document bearing the incomplete signature.

   b. (same as in 3.3.1 Step 1.c)

   c. (same as in 3.3.1 Step 1.d.i to 1.d.iv) plus 1.d.v amended as follows:

      v. The EnvelopedSignatureTransform is included as the first Transform (even before excl-c14n if it was used for extraction) in the `<ds:Transforms>` element. The sequence MUST describe the effective transform as a reproducible procedure from parsing until hash.

         Note: This is necessary because the EnvelopedSignatureTransform would not work if there was a Canonicalization before it. Similar problems apply to transforms using the here() function, if such are to be supported the use of Base64XML or Escaped may be required.

5. Add the returned `<ds:Reference>` as required in 3.3.1 Step 2 of Basic processing.
6. The server continues with processing as specified in section 3.3.1 for the rest of the documents.
7. The `<SignatureObject>` element of the result MUST include a `<SignaturePtr>` element set to point to the document with the same `WhichDocument` as in the request and an XPath expression pointing to the newly created signature.

3.5.9 Optional Input <SignedReferences>

The `<SignedReferences>` element gives the client greater control over how the `<ds:Reference>` elements are formed. When this element is present, step 1 of Basic Processing (section 3.3.1) is overridden. Instead of there being a one-to-one correspondence between input documents and `<ds:Reference>` elements, now each `<SignedReference>` element controls the creation of a corresponding `<ds:Reference>`.

Since each `<SignedReference>` refers to an input document, this allows multiple `<ds:Reference>` elements to be based on a single input document. Furthermore, the client can request additional transforms to be applied to each `<ds:Reference>`, and can set each `<ds:Reference>` element's `Id` or `URI` attribute. These aspects of the `<ds:Reference>` can only be set through the `<SignedReferences>` optional input; they cannot be set through the input documents, since they are aspects of the reference to the input document, not the input document itself.

Each `<SignedReference>` element contains:

`WhichDocument` [Required]

Which input document this reference refers to (see the `ID` attribute in section 2.4.1).

`RefId` [Optional]

Sets the `Id` attribute of the corresponding `<ds:Reference>`.

`RefURI` [Optional]

overrides the `RefURI` of `<dss:Document>` and if present from the `<SignedReferences>` creates an additional `<ds:Reference>`

`RefType` [Optional]

overrides the `RefType` of `<dss:Document>`

`<ds:Transforms>` [Optional]

Requests the server to perform additional transforms on this reference.

When the `<SignedReferences>` optional input is present, basic processing 3.3.1 step 1 is performed for each `<SignedReference>` overriding steps a., b., c. and d.:

If the `<SignaturePlacement>` element is present the server processes it as follows:

For each `<SignedReference>` in `<SignedReferences>`

1. The server identifies the `<Document>` referenced as indicated by the `WhichDocument` attribute.
2. If `RefURI` is present create an additional `<ds:Reference>` for the document in question by performing basic processing as in section 3.3.1 Step 1 amended as follows:
    1.
        a. Unchanged.
        b. Applies the transforms indicated in `<ds:Transforms>`. Afterwards, the server may apply any other transform it considers appropriate as per its policy and then generates a canonicalized octet string as required in step b. of basic Processing before hashing.
        c. Unchanged.
        d. The server forms a `<ds:Reference>` with the elements and attributes set as follows:
            i. Use this `RefURI` attribute from the `<SignedReference>` if present instead of `RefURI` from `<dss:Document>` in step i. of Basic Processing.
            The `Id` attribute is set to the `<SignedReference>` element's `RefId` attribute. If the `<SignedReference>` has no `RefId` attribute, the `<ds:Reference>` element's `Id` attribute is omitted.
            ii. Unchanged.
            iii. Unchanged.
            iv. Unchanged.
            v. The `<ds:Transforms>` used here will have to be added to `<ds:Transforms>` of step v. of basic processing so that this element describes the sequence of transforms applied by the server and describing the effective transform as a reproducible procedure from parsing until hash.
    2. Add the returned `<ds:Reference>` as required in 3.3.1 Step 2 of Basic processing.
3. If `RefURI` is not present perform basic processing for the input document not creating an additional `<ds:Reference>` amending Step 1 as follows:
    1.
        a. Unchanged.
        b. Applies the transforms indicated in `<ds:Transforms>`. Afterwards, the server may apply any other transform it considers as appropriate as per its policy and then generates generating a canonicalized octet string as required in step b. of basic Processing before hashing.
        c. Unchanged.
        d. The server forms a `<ds:Reference>` with the elements and attributes set as follows:
            i. Perform step i. of Basic Processing and the `Id` attribute is set to the `<SignedReference>` element's `RefId` attribute. If the

`<SignedReference>` has no `RefId` attribute, the `<ds:Reference>` element's `Id` attribute is omitted.

   ii. Unchanged

   iii. Unchanged

   iv. Unchanged

   v. The `<ds:Transforms>` used here will have to be added to `<ds:Transforms>` of step v. of basic processing so that this element describes the sequence of transforms applied by the server and describing the effective transform as a reproducible procedure from parsing until hash.

4. The server continues with processing as specified in section 3.3.1 for the rest of the documents.

```
<xs:element name=”SignedReferences”>
  <xs:complexType>
    <xs:sequence>
      <xs:element ref=”dss:SignedReference”
                  maxOccurs=”unbounded”/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="SignedReference">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="ds:Transforms" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="WhichDocument" type="xs:IDREF" use="required"/>
    <xs:attribute name="RefURI" type="xs:anyURI" use="optional"/>
    <xs:attribute name="RefId" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>
```

4 The DSS Verifying Protocol

4.1 Element <VerifyRequest>

The `<VerifyRequest>` inherits from `<RequestBaseType>`. This element is sent by the client to verify a signature or timestamp on some input documents. It contains the following additional elements:

`<SignatureObject>` [Optional]

This element contains a signature or timestamp, or else contains a `<SignaturePtr>` that points to an XML signature in one of the input documents. If this element is omitted, there must be only a single `<InputDocument>` which the server will search to find the to-be-verified signature(s). A `<SignaturePtr>` or omitted `<SignatureObject>` MUST be used whenever the to-be-verified signature is an XML signature which uses an Enveloped Signature Transform; otherwise the server would have difficulty locating the signature and applying the Enveloped Signature Transform.

```
<xs:element name="VerifyRequest">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="dss:RequestBaseType">
        <xs:sequence>
          <xs:element ref="dss:SignatureObject" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

4.2 Element <VerifyResponse>

The `<VerifyResponse>` inherits from <Response>. This element defines no additional attributes and elements

4.3 Basic Processing for XML Signatures

A DSS server that verifies XML signatures SHOULD perform the following steps, upon receiving a `<VerifyRequest>`. These steps may be changed or overridden by the optional inputs, or by the profile or policy the server is operating under. For more details on multi-signature verification, see section 4.3.1.

1. The server retrieves one or more `<ds:Signature>` objects, as follows: If the `<SignatureObject>` is present, the server retrieves either the `<ds:Signature>` that is a child element of the `<SignatureObject>` (see: Note at the end of this section), or those `<ds:Signature>` objects which are pointed to by the `<SignaturePtr>` in the `<SignatureObject>`.
    a. If the <SignaturePtr> points to an input document but not a specific element in that document, the pointed-to input document must be a `<Document>` element containing XML either in an `<Base64XML>`, `<EscapedXML>` or `<InlineXML>` element.
    If the document is inside <Base64XML> or <EscapedXML> it is decoded and parsed as described in 3.3.1 Step 1.a or 3.3.3 Step 1a respectively.

If the document is inside `<InlineXML>` the document is extracted using exclusive canonicalization. The `<ds:Reference>` corresponding to the document MUST have a chain of transforms (at least one `ds:Transform` inside `ds:Transforms`) that anticipates and reflects this. If this is not the case the server MUST throw an Error (`urn:oasis:names:tc:dss:1.0:resultminor:inappropriate:signature`). Note: Otherwise false negatives due to namespace conflicts may appear.

   b. If the `<SignatureObject>` is omitted, there MUST be only a single <Document> element. This case is handled as if a `<SignaturePtr>` pointing to the single `<Document>` was present: the server will search and find every `<ds:Signature>` element in this input document, and verify each `<ds:Signature>` according to the steps below.

2. For each `<ds:Reference>` in the `<ds:Signature>`, the server finds the input document with matching RefURI and RefType values. If the `<ds:Reference>` uses a same-document URI, the XPointer should be evaluated against the input document the `<ds:Signature>` is contained within, or against the `<ds:Signature>` itself if it is contained within the `<SignatureObject>` element. The `<SchemaRef>` element or optional input <Schema> of the input document or `<SignatureObject>` will be used, if present, to identify ID attributes when evaluating the XPointer expression. If the `<ds:Reference>` uses an external URI and the corresponding input document is not present, the server will skip the `<ds:Reference>`, and later return a result code such as `ReferencedDocumentNotPresent` to indicate this. The `RefURI` MAY be omitted in at most one of the set of Input documents.
   a. If the input document is a <Document>, the server extracts and decodes as described in 3.3.1 Step 1.a (or equivalent step in variants of the basic process as defined in 3.3.2 onwards depending of the form of the input document).
   b. If the input document is a `<TransformedData>`, the server checks that the `<ds:Transforms>` match between the `<TransformedData>` and the `<ds:Reference>` and then hashes the resultant data object according to `<ds:DigestMethod>`, and checks that the result matches `<ds:DigestValue>`.
   c. If the input document is a `<DocumentHash>`, the server checks that the `<ds:Transforms>`, `<ds:DigestMethod>`, and `<ds:DigestValue>` elements match between the `<DocumentHash>` and the `<ds:Reference>`.
   d. If such an input document isn't present, and the `<ds:Reference>` uses a same-document URI without a barename XPointer (URI=""), then the relevant input document is the input document the `<ds:Signature>` is contained within, or the `<ds:Signature>` itself if it is contained within the `<SignatureObject>` element and processed according to a. above.
3. The server then validates the signature according to section 3.2.2 in [XMLSig].
4. If the signature validates correctly, the server returns one of the first three `<ResultMinor>` codes listed in section 4.4, depending on the relationship of the signature to the input documents (not including the relationship of the signature to those XML elements that were resolved through XPointer evaluation; the client will have to inspect those relationships manually). If the signature fails to validate correctly, the server returns some other code; either one defined in section 4.4 of this specification, or one defined by some profile of this specification.

Note: The extraction of the `<ds:Signature>` from the `<SignatureObject>` should be performed without namespace inheritance. If the signature `<ds:Signature>` does not use exclusive canonicalization for it's `<ds:CanonicalizationMethod>` there can appear problems caused by namespace declarations moved by gateways or protocol processors of outer protocol bindings that alter the signature object and cause false negatives on validation. Problems appearing due to different behavior of xml parsers in schema validating parsing vs. non-validating parsing like data type normalizations would have to be healed by canonicalization only as no transforms are available for ds:SignedInfo. As currently available specifications of canonicalization are not aware of schema data types a solution to heal these defects is currently not possible. Beware, these problems can already occur on parsing the whole request including protocol bindings like SOAP. Implementors are encouraged to make use of `<dss:Base64XML>` or `<dss: EscapedXML>` instead.

4.3.1 Multi-Signature Verification

If a client requests verification of an entire input document, either using a `<SignaturePtr>` without an `<XPath>` or a missing `<SignaturePtr>` (see section 4.3 step 1), then the server MUST determine whether the input document contains zero, one, or more than one `<ds:Signature>` elements. If zero, the server should return a `<ResultMajor>` code of `RequesterError`.

If more than one `<ds:Signature>` elements are present, the server MUST either reject the request with a `<ResultMajor>` code of `RequesterError` and a `<ResultMinor>` code of `NotSupported`, or accept the request and try to verify all of the signatures.

If the server accepts the request in the multi-signature case (or if only a single signature is present) and one of the signatures fails to verify, the server should return one of the error codes in section 4.4, reflecting the first error encountered.

If all of the signatures verify correctly, the server should return the `Success` `<ResultMajor>` code and the following `<ResultMinor>` code:

`urn:oasis:names:tc:dss:1.0:resultminor:ValidMultiSignatures`

> Note: These procedures only define procedures for handling of multiple signature on one input document. Multiple signature on multiple documents is not supported.

Only certain optional inputs and outputs are allowed when performing multi-signature verification. See section 4.6 for details.

4.3.2 Signature Timestamp verification procedure

The following sub-sections will describe the processing rules for verifying:

- CMS RFC 3161 signature timestamp tokens
- XML signature timestamp tokens

This section describes signature timestamp processing when the timestamp is embedded in the incoming signature.

Note: procedures for handling other forms of timestamp may be defined in profiles of the Core. In particular, the DSS XAdES profile defines procedures for handling timestamps against the document being signed, and the DSS Timestamp profile defines procedures for handling standalone timestamps.

For a definition of the `<Timestamp>` element see section 5.1 Details of the XML timestamp token can be found in subsection 5.1.1.

4.3.2.1 Verification Processing of CMS RFC 3161 timestamp tokens.

The present section describes the processing rules for verifying a CMS RFC3161 timestamp token passed in on a Verify call within the `<SignatureObject>` of the `<VerifyRequest>`

element. In the CMS case, since the "signature timestamp" is embedded in the signature as an unsigned attribute, only the time stamped signature is required for verification processing. As such, no additional input is required.

The processing by the server is broken down into the following steps:

1. The signature timestamp is embedded in the incoming signature as an unsigned attribute whose object identifier is 1.2.840.11359.1.9.16.2.14. Extract and verify the timestamp token.
2. Verify that the token's public verification certificate is authorized for time stamping by examining the Extended Key Usage field for the presence of the time stamping OID "1.3.6.1.5.5.7.3.8".
3. Validate that the `TstInfo` structure has a valid layout as defined in **[RFC 3161]**.
4. Extract the `MessageImprint` hash value and associated algorithm from the `TstInfo` structure which will be compared against the hash value derived in the next step.
5. Recalculate the hash of the signature value field of the signature in which the timestamp is embedded.
6. Compare the hash values from the two previous steps, and if they are equivalent, then this timestamp is valid for the signature that was time stamped.
7. Verify that the public verification certificate conforms to all relevant aspects of the relying-party's policy including algorithm usage, policy OIDs, time accuracy tolerances, and the Nonce value.
8. Set the `dss:Result` element as defined in this specification.

4.3.2.2 Processing for XML timestamp tokens

The present section describes the processing rules for verifying an XML Signature timestamp token embedded within an XML signature as an unsigned property. This XML signature may be passed in on a Verify call within the `<SignatureObject>` or embedded within a `<Document>`'s child.

The server shall verify the timestamp token performing the steps detailed below. If any one of them results in failure, then the timestamp token SHOULD be rejected.

1. Extract the timestamp token embedded in the incoming signature.
2. Verify that the verification key and algorithms used conforms to all relevant aspects of the applicable policy. Should this key come within a public certificate, verify that the certificate conforms to all relevant aspects of the applicable policy including algorithm usage, policy OIDs, and time accuracy tolerances.
3. Verify that the aforementioned verification key is consistent with the `ds:SignedInfo/SignatureMethod/@Algorithm` attribute value.
4. Verify the timestamp token signature in accordance with the rules defined in **[XMLSig]**.
5. Verify that the `<ds:SignedInfo>` element contains at least two `<ds:Reference>` elements.
6. Verify that one of the `<ds:Reference>` element has its Type attribute set to "urn:oasis:names:tc:dss:1.0:core:schema:XMLTimeStampToken". Take this one and proceed as indicated below:
    a. Retrieve the referenced data object. Verify that it references a `<ds:Object>` element, which in turn envelopes a `<TSTInfo>` element.
    b. Verify that the `<TSTInfo>` element has a valid layout as per the present specification.

    c. Extract the digest value and associated algorithm from its `<ds:DigestValue>` and `<ds:DigestMethod>` elements respectively.
    d. Recalculate the digest of the retrieved data object as specified by **[XMLSig]** with the digest algorithm indicated in `<ds:DigestMethod>`, and compare this result with the contents of `<ds:DigestValue>`.

7. Take the other `<ds:Reference>` and proceed as indicated below:
    a. Retrieve the referenced data object. Verify that this is the `<ds:SignatureValue>` of the time-stamped XML signature.
    b. Extract the digest value and associated algorithm from its `<ds:DigestValue>` and `<ds:DigestMethod>` elements respectively.
    c. Recalculate the digest of the retrieved data as specified by **[XMLSig]** with the digest algorithm indicated in the `<ds:DigestMethod>`, and compare the computed digest value with the one in the `<ds:DigestValue>` element.
8. Set the `<dss:Result>` element as appropriate.

4.4 Result Codes

For Information with regard to Result Codes see above section 2.6 “Element <Result>”.

4.5 Basic Processing for CMS Signatures

A DSS server that verifies CMS signatures SHOULD perform the following steps, upon receiving a `<VerifyRequest>`. These steps may be changed or overridden by the optional inputs, or by the profile or policy the server is operating under.

1. The server retrieves the CMS signature by decoding the `<Base64Signature>` child of `<SignatureObject>`.
2. The server retrieves the input data. If the CMS signature is detached, there must be a single input document: i.e. a single `<Document>` or `<DocumentHash>` element. Otherwise, if the CMS signature is enveloping, it contains its own input data and there MUST NOT be any input documents present.
3. The CMS signature and input data are verified in the conventional way (see **[RFC 3369]** for details).
4. If the signature validates correctly, the server returns the first `<ResultMinor>` code listed in section 4.4. If the signature fails to validate correctly, the server returns some other code; either one defined in section 4.4 of this specification, or one defined by some profile of this specification.

4.6 Optional Inputs and Outputs

This section defines some optional inputs and outputs that profiles of the DSS verifying protocol might find useful. Section 2.8 defines some common optional inputs that can also be used with the verifying protocol. Profiles of the verifying protocol can define their own optional inputs and outputs, as well. General handling of optional inputs and outputs is discussed in section 2.7.

4.6.1 Optional Input <VerifyManifests> and Output <VerifyManifestResults>

The presence of this element instructs the server to validate manifests in an XML signature.

On encountering such a document in step 2 of basic processing, the server shall repeat step 2 for all the `<ds:Reference>` elements within the manifest. In accordance with **[XMLSIG]** section 5.1, DSS Manifest validation does not affect a signature's core validation. The results of verifying individual `<ds:Reference>`'s within a `<ds:Manifest>` are returned in the `<dss:VerifyManifestResults>` optional output.
For example, a client supplies the optional input `<VerifyManifests>`, then the returned `<ResultMinor>` is `urn:oasis:names:tc:dss:1.0:resultminor:valid:hasManifestResults` if XMLSig core validation succeeds and the optional output `<VerifyManifestResults>` is returned indicating the status of the manifest reference verification. In case of a negative XMLSig core validation no attempt is made to verify manifests.

The `<VerifyManifests>` optional input is allowed in multi-signature verification.

`<ReferenceXpath>` [Required]

Identifies the manifest reference, in the XML signature, to which this result pertains.

`<Status>` [Required]

Indicates the manifest validation result. It takes one of the values urn:oasis:names:tc:dss:1.0:manifeststatus:Valid or urn:oasis:names:tc:dss:1.0:manifeststatus:Invalid.

```
<xs:element name="VerifyManifestResults"
type="dss:VerifyManifestResultsType"/>

<xs:complexType name="VerifyManifestResultsType">
  <xs:sequence>
    <xs:element ref="dss:ManifestResult" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="ManifestResult">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ReferenceXpath" type="xs:string"/>
      <xs:element name="Status" type="xs:anyURI"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

4.6.2 Optional Input <VerificationTime>

This element instructs the server to attempt to determine the signature’s validity at the specified time, instead of the current time.

This optional input is allowed in multi-signature verification.

```
<xs:element name=”VerificationTime” type=”xs:dateTime”/>
```

4.6.3 Optional Input <AdditionalKeyInfo>

This element provides the server with additional data (such as certificates and CRLs) which it can use to validate the signing key.

This optional input is not allowed in multi-signature verification.

```
<xs:element name=”AdditionalKeyInfo”>
  <xs:complexType>
```

```
    <xs:sequence>
      <xs:element ref="ds:KeyInfo"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

4.6.4 Optional Input <ReturnProcessingDetails> and Output <ProcessingDetails>

The presence of the `<ReturnProcessingDetails>` optional input instructs the server to return a `<ProcessingDetails>` output.

These options are not allowed in multi-signature verification.

```
<xs:element name="ReturnProcessingDetails"/>
```

The `<ProcessingDetails>` optional output elaborates on what signature verification steps succeeded or failed. It may contain the following child elements:

`<ValidDetail>` [Any Number]

A verification detail that was evaluated and found to be valid.

`<IndeterminateDetail>` [Any Number]

A verification detail that could not be evaluated or was evaluated and returned an indeterminate result.

`<InvalidDetail>` [Any Number]

A verification detail that was evaluated and found to be invalid.

```
<xs:element name="ProcessingDetails">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ValidDetail" type="dss:DetailType"
                  minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="IndeterminateDetail"
                  type="dss:DetailType"
                  minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="InvalidDetail" type="xs:dss:DetailType"
                  minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Each detail element is of type `dss:DetailType`. A `dss:DetailType` contains the following child elements and attributes:

`Type` [Required]

A URI which identifies the detail. It may be a value defined by this specification, or a value defined by some other specification. For the values defined by this specification, see below.

Multiple detail elements of the same `Type` may appear in a single `<ProcessingDetails>`. For example, when a signature contains a certificate chain that certifies the signing key, there may be details of the same `Type` present for each certificate in the chain, describing how each certificate was processed.

`<Code>` [Optional]

A URI which more precisely specifies why this detail is valid, invalid, or indeterminate. It must be a value defined by some other specification, since this specification defines no values for this element.

`<Message>` [Optional]

A human-readable message which MAY be logged, used for debugging, etc.

```
<xs:complexType name="DetailType">
  <xs:sequence>
    <xs:element name="Code" type="xs:anyURI" minOccurs="0"/>
    <xs:element name="Message" type="InternationalStringType"
                minOccurs="0"/>
    <xs:any processContents="lax" minOccurs="0"
            maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="Type" type="xs:anyURI" use="required"/>
</xs:element>
```

The values for the `Type` attribute defined by this specification are the following:

`urn:oasis:names:tc:dss:1.0:detail:IssuerTrust`

Whether the issuer of trust information for the signing key (or one of the certifying keys) is considered to be trustworthy.

`urn:oasis:names:tc:dss:1.0:detail:RevocationStatus`

Whether the trust information for the signing key (or one of the certifying keys) is revoked.

`urn:oasis:names:tc:dss:1.0:detail:ValidityInterval`

Whether the trust information for the signing key (or one of the certifying keys) is within its validity interval.

`urn:oasis:names:tc:dss:1.0:detail:Signature`

Whether the document signature (or one of the certifying signatures) verifies correctly.

`urn:oasis:names:tc:dss:1.0:detail:ManifestReference`

Whether a manifest reference in the XML signature verified correctly.

4.6.5 Optional Input <ReturnSigningTime> and Output <SigningTime>

The presence of the `<ReturnSigningTime>` optional input instructs the server to return a `<SigningTime>` output. This output typically gives the client access to a time value carried within a signature attribute or a signature timestamp, or within a timestamp token if the signature itself is a timestamp (e.g. see section 5.1.1). If no such value is present, and the server has no other way of determining when the signature was performed, the server should omit the `<SigningTime>` output. If there are multiple such values present, behavior is profile-defined.

These options are not allowed in multi-signature verification.

```
<xs:element name="ReturnSigningTime"/>
```

The `<SigningTime>` optional output contains an indication of when the signature was performed, and a boolean attribute that indicates whether this value is attested to by a third-party timestamp authority (if true), or only by the signer (if false).

```
<xs:element name="SigningTime">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:dateTime">
        <xs:attribute name="ThirdPartyTimestamp"
```

```
                    type="xs:boolean" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

4.6.6 Optional Input <ReturnSignerIdentity> and Output <SignerIdentity>

The presence of the `<ReturnSignerIdentity>` optional input instructs the server to return a `<SignerIdentity>` output.

This optional input and output are not allowed in multi-signature verification.

```
<xs:element name="ReturnSignerIdentity"/>
```

The `<SignerIdentity>` optional output contains an indication of who performed the signature.

```
<xs:element name="SignerIdentity" type="saml:NameIdentifierType"/>
```

4.6.7 Optional Input <ReturnUpdatedSignature> and Output <UpdatedSignature>

The presence of the `<ReturnUpdatedSignature>` optional input instructs the server to return an `<UpdatedSignature>` output, containing a new or updated signature.

The `Type` attribute on `<ReturnUpdatedSignature>`, if present, defines exactly what it means to “update” a signature. For example, the updated signature may be the original signature with some additional unsigned signature properties added to it (such as timestamps, counter-signatures, or additional information for use in verification), or the updated signature could be an entirely new signature calculated on the same input documents as the input signature. Profiles that use this optional input MUST define the allowed values and their semantics, and the default value, for the `Type` attribute (unless only a single type of updated signature is supported, in which case the `Type` attribute can be omitted).

Multiple occurrences of this optional input can be present in a single verify request message. If multiple occurrences are present, each occurrence MUST have a different `Type` attribute. Each occurrence will generate a corresponding optional output. These optional outputs SHALL be distinguishable based on their `Type` attribute, which will match each output with an input.

These options are not allowed in multi-signature verification.

```
<xs:element name="ReturnUpdatedSignature">
  <xs:complexType>
    <xs:attribute name="Type" type="xs:anyURI" use="optional"/>
  </xs:complexType>
</xs:element>
```

The `<UpdatedSignature>` optional output contains the returned signature.

```
<xs:element name="UpdatedSignature">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="dss:SignatureObject">
    <xs:sequence>
    <xs:attribute name="Type" type="xs:anyURI" use="optional"/>
  </xs:complexType>
</xs:element>
```

4.6.8 Optional Input <ReturnTransformedDocument> and Output <TransformedDocument>

The <ReturnTransformedDocument> optional input instructs the server to return an input document to which the XML signature transforms specified by a particular <ds:Reference> have been applied. The <ds:Reference> is indicated by the zero-based WhichReference attribute (0 means the first <ds:Reference> in the signature, 1 means the second, and so on). Multiple occurrences of this optional input can be present in a single verify request message. Each occurrence will generate a corresponding optional output.

These options are not allowed in multi-signature verification.

```
<xs:element name="ReturnTransformedDocument">
  <xs:complexType>
    <xs:attribute name="WhichReference" type="xs:integer"
                  use="required"/>
  </xs:complexType>
</xs:element>
```

The <TransformedDocument> optional output contains a document corresponding to the specified <ds:Reference>, after all the transforms in the reference have been applied. In other words, the hash value of the returned document should equal the <ds:Reference> element's <ds:DigestValue>. To match outputs to inputs, each <TransformedDocument> will contain a WhichReference attribute which matches the corresponding optional input.

```
<xs:element name="TransformedDocument">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="dss:Document">
    </xs:sequence>
  </xs:complexType>
  <xs:attribute name="WhichReference" type="xs:integer"
                use="required"/>
</xs:element>
```

4.6.9 Optional Input <AddTimestamp>

The <AddTimestamp> element within a <VerifyRequest> message indicates that the client wishes the server to update the signature after its verification by embedding a signature timestamp token as an unauthenticated attribute (see "unauthAttrs" in section 9.1 on page 29 **[RFC 3369]**) or *unsigned* property (see section 6.2.5 "The UnsignedSignatureProperties element" or section 7.3 "The SignatureTimeStamp element" **[XAdES]**) of the supplied signature.

The timestamp token will be on the signature value in the case of CMS/PKCS7 signatures or the <ds:SignatureValue> element in the case of XML signatures.

Note: Procedures for handling other forms of timestamp may be defined in profiles of the Core. In particular, the DSS XAdES profile **[DSS-XAdES-P]**defines procedures for handling timestamps against the document being signed, and the DSS Timestamp profile **[DSS-TS-P]** defines procedures for handling standalone timestamps.

The Type attribute, if present, indicates what type of timestamp to apply. Profiles that use this optional input MUST define the allowed values, and the default value, for the Type attribute (unless only a single type of timestamp is supported, in which case the Type attribute can be omitted).

The schema for <AddTimestamp> is as defined in 3.5.2.

Procedures for handling RFC 3161 and XML timestamps are as defined in 3.5.2.1 and 3.5.2.2.

5 DSS Core Elements

This section defines two XML elements that may be used in conjunction with the DSS core protocols.

5.1 Element <Timestamp>

This section defines an XML timestamp. A `<Timestamp>` contains some type of timestamp token, such as an RFC 3161 `TimeStampToken` **[RFC 3161]** or a `<ds:Signature>` (aka an "XML timestamp token") (see section 5.1.1). Profiles may introduce additional types of timestamp tokens. Standalone XML timestamps can be produced and verified using the timestamping profile of the DSS core protocols **[XML-TSP]**.

An XML timestamp may contain:

`<ds:Signature>` [Optional]

This is an enveloping XML signature, as defined in section 5.1.1.

`<RFC3161TimeStampToken>` [Optional]

This is a base64-encoded `TimeStampToken` as defined in **[RFC3161]**.

```
<xs:element name="Timestamp">
  <xs:complexType>
    <xs:choice>
      <xs:element ref="ds:Signature"/>
      <xs:element name="RFC3161TimeStampToken"
                  type="xs:base64Binary"/>
      <xs:element name="Other" type="AnyType"/>
    <xs:choice>
  </xs:complexType>
</xs:element>
```

5.1.1 XML Timestamp Token

An XML timestamp token is similar to an RFC 3161 `TimeStampToken`, but is encoded as a `<TstInfo>` element (see section 5.1.2) inside an enveloping `<ds:Signature>`. This allows conventional XML signature implementations to validate the signature, though additional processing is still required to validate the timestamp properties (see section 4.3.2.2).

The following text describes how the child elements of the `<ds:Signature>` MUST be used:

`<ds:KeyInfo>` [Required]

The `<ds:KeyInfo>` element SHALL identify the issuer of the timestamp and MAY be used to locate, retrieve and validate the timestamp token signature-verification key. The exact details of this element may be specified further in a profile.

`<ds:SignedInfo>/<ds:Reference>` [Required]

There MUST be a single `<ds:Reference>` element whose URI attribute references the `<ds:Object>` containing the enveloped `<TstInfo>` element, and whose Type attribute is equal to `urn:oasis:names:tc:dss:1.0:core:schema:XMLTimeStampToken`.

`<ds:Object>` [Required]

A `<TstInfo>` element SHALL be contained in a `<ds:Object>` element.

Additional `<ds:Reference>` elements MUST appear for data objects **[XMLSig]** being time-stamped. For details on further use of time-stamps, please refer to appropriate profiles.

5.1.2 Element <TstInfo>

A `<TstInfo>` element is included in an XML timestamp token as a `<ds:Signature>` / `<ds:Object>` child element. A `<TstInfo>` element has the following children:

`<SerialNumber>` [Required]

This element SHALL contain a serial number produced by the timestamp authority (TSA). It MUST be unique across all the tokens issued by a particular TSA.

`<CreationTime>` [Required]

The time at which the token was issued.

`<Policy>` [Optional]

This element SHALL identify the policy under which the token was issued. The TSA's policy SHOULD identify the fundamental source of its time.

`<ErrorBound>` [Optional]

The TSA's estimate of the maximum error in its local clock.

`<Ordered>` [Default="false"]

This element SHALL indicate whether or not timestamps issued by this TSA, under this policy, are strictly ordered according to the value of the `CreationTime` element value.

TSA [Optional]

The name of the TSA.

```
<xs:element name="TstInfo">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="SerialNumber" type="xs:integer"/>
      <xs:element name="CreationTime" type="xs:dateTime"/>
      <xs:element name="Policy" type="xs:anyURI" minOccurs="0"/>
      <xs:element name="ErrorBound" type="xs:duration"
                  minOccurs="0"/>
      <xs:element name="Ordered" type="xs:boolean"
                  default="false" minOccurs="0"/>
      <xs:element name="TSA" type="saml:NameIdentifierType"
                  minOccurs="0"/>
    <xs:sequence>
  </xs:complexType>
</xs:element>
```

5.2 Element <RequesterIdentity>

This section contains the definition of an XML Requester Identity element. This element can be used as a signature property in an XML signature to identify the client who requested the signature.

This element has the following children:

`Name` [Required]

The name or role of the requester who requested the signature be performed.

`SupportingInfo` [Optional]

Information supporting the name (such as a SAML Assertion **[SAMLCore1.1]**, Liberty Alliance Authentication Context, or X.509 Certificate).

The following schema fragment defines the `<RequesterIdentity>` element:

```
<xs:element name="RequesterIdentity">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name" type="saml:NameIdentifierType"/>
        <xs:element name="SupportingInfo" type="dss:AnyType"
                    minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

6 DSS Core Bindings

Mappings from DSS messages into standard communications protocols are called DSS *bindings*. *Transport bindings* specify how DSS messages are encoded and carried over some lower-level transport protocol. *Security bindings* specify how confidentiality, authentication, and integrity can be achieved for DSS messages in the context of some transport binding.

Below we specify an initial set of bindings for DSS. Future bindings may be introduced by the OASIS DSS TC or by other parties.

6.1 HTTP POST Transport Binding

In this binding, the DSS request/response exchange occurs within an HTTP POST exchange **[RFC 2616]**. The following rules apply to the HTTP request:

The client may send an HTTP/1.0 or HTTP/1.1 request.

The Request URI may be used to indicate a particular service endpoint.

The `Content-Type` header MUST be set to “application/xml”.

The `Content-Length` header MUST be present and correct.

The DSS request message MUST be sent in the body of the HTTP Request.

The following rules apply to the HTTP Response:

The `Content-Type` header MUST be set to “text/xml”.

The `Content-Length` header MUST be present and correct.

The DSS response message MUST be sent in the body of the HTTP Response.

The HTTP status code MUST be set to 200 if a DSS response message is returned. Otherwise, the status code can be set to 3*xx* to indicate a redirection, 4*xx* to indicate a low-level client error (such as a malformed request), or 5*xx* to indicate a low-level server error.

6.2 SOAP 1.2 Transport Binding

In this binding, the DSS request/response exchange occurs using the SOAP 1.2 message protocol **[SOAP]**. The following rules apply to the SOAP request:

A single DSS `<SignRequest>` or `<VerifyRequest>` element will be transmitted within the body of the SOAP message.

The client MUST NOT include any additional XML elements in the SOAP body.

The UTF-8 character encoding must be used for the SOAP message.

Arbitrary SOAP headers may be present.

The following rules apply to the SOAP response:

The server MUST return either a single DSS `<SignResponse>` or `<VerifyResponse>` element within the body of the SOAP message, or a SOAP fault code.

The server MUST NOT include any additional XML elements in the SOAP body.

If a DSS server cannot parse a DSS request, or there is some error with the SOAP envelope, the server MUST return a SOAP fault code. Otherwise, a DSS result code should be used to signal errors.

The UTF-8 character encoding must be used for the SOAP message.

Arbitrary SOAP headers may be present.

On receiving a DSS response in a SOAP message, the client MUST NOT send a fault code to the DSS server.

6.3 TLS Security Bindings

TLS **[RFC 2246]** is a session-security protocol that can provide confidentiality, authentication, and integrity to the HTTP POST transport binding, the SOAP 1.2 transport binding, or others. TLS supports a variety of authentication methods, so we define several security bindings below. All of these bindings inherit the following rules:

TLS 1.0 MUST be supported. SSL 3.0 MAY be supported. Future versions of TLS MAY be supported.

RSA ciphersuites MUST be supported. Diffie-Hellman and DSS ciphersuites MAY be supported.

TripleDES ciphersuites MUST be supported. AES ciphersuites SHOULD be supported. Other ciphersuites MAY be supported, except for weak ciphersuites intended to meet export restrictions, which SHOULD NOT be supported.

6.3.1 TLS X.509 Server Authentication

The following ciphersuites defined in **[RFC 2246]** and **[RFC 3268]** are supported. The server MUST authenticate itself with an X.509 certificate chain **[RFC 3280]**. The server MUST NOT request client authentication.

MUST:

TLS_RSA_WITH_3DES_EDE_CBC_SHA

SHOULD:

TLS_RSA_WITH_AES_128_CBC_SHA

TLS_RSA_WITH_AES_256_CBC_SHA

6.3.2 TLS X.509 Mutual Authentication

The same ciphersuites mentioned in section 6.2.1 are supported. The server MUST authenticate itself with an X.509 certificate chain, and MUST request client authentication. The client MUST authenticate itself with an X.509 certificate chain.

6.3.3 TLS SRP Authentication

SRP is a way of using a username and password to accomplish mutual authentication. The following ciphersuites defined in **[draft-ietf-tls-srp-08]** are supported.

MUST:

TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA

SHOULD:

TLS_SRP_SHA_WITH_AES_128_CBC_SHA

TLS_SRP_SHA_WITH_AES_256_CBC_SHA

6.3.4 TLS SRP and X.509 Server Authentication

SRP can be combined with X.509 server authentication. The following ciphersuites defined in **[draft-ietf-tls-srp-08]** are supported.

MUST:

  TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA

SHOULD:

  TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA

  TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA

7 DSS-Defined Identifiers

The following sections define various URI-based identifiers. Where possible an existing URN is used to specify a protocol. In the case of IETF protocols the URN of the most current RFC that specifies the protocol is used (see **[RFC 2648]**). URI references created specifically for DSS have the following stem:

urn:oasis:names:tc:dss:1.0:

7.1 Signature Type Identifiers

The following identifiers MAY be used as the content of the `<SignatureType>` optional input (see section 3.5.1).

7.1.1 XML Signature

- **URI:** urn:ietf:rfc:3275
- This refers to an XML signature per **[XMLSig]**.

7.1.2 XML TimeStampToken

- **URI:** urn:oasis:names:tc:dss:1.0:core:schema:XMLTimeStampToken
- This refers to an XML timestamp containing an XML signature, per section 5.1.

7.1.3 RFC 3161 TimeStampToken

- **URI:** urn:ietf:rfc:3161
- This refers to an XML timestamp containing an ASN.1 TimeStampToken, per **[RFC 3161]**.

7.1.4 CMS Signature

- **URI:** urn:ietf:rfc:3369
- This refers to a CMS signature per **[RFC 3369]**.

7.1.5 PGP Signature

- **URI:** urn:ietf:rfc:2440
- This refers to a PGP signature per **[RFC 2440]**.

8 References

8.1 Normative

[Core-XSD] S. Drees, T. Perrin, JC Cruellas, N Pope, K Lanz, et al. *DSS Schema*. OASIS, November 2005.

[DSS-TS-P] T Perrin et al. *DSS Timestamp Profile*. OASIS, April 2006.

[DSS-XAdES-P] JC Cruellas et al. *DSS XAdES Profile*. OASIS, April 2006.

[RFC 2119] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. IETF RFC 2396, August 1998.

http://www.ietf.org/rfc/rfc2396.txt.

[RFC 2246] T Dierks, C. Allen. *The TLS Protocol Version 1.0.* IETF RFC 2246, January 1999.

http://www.ietf.org/rfc/rfc2246.txt.

[RFC 2396] T. Berners-Lee et al. *Uniform Resource Identifiers (URI): Generic Syntax.* IETF RFC 2396, August 1998.

http://www.ietf.org/rfc/rfc2396.txt.

[RFC 2440] J. Callas, L. Donnerhacke, H. Finney, R. Thayer. *OpenPGP Message Format*. IETF RFC 2440, November 1998.

http://www.ietf.org/rfc/rfc2440.txt.

[RFC 2616] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. IETF RFC 2616, June 1999.

http://www.ietf.org/rfc/rfc2616.txt.

[RFC 2648] R. Moats. *A URN Namespace for IETF Documents*. IETF RFC 2648, August 1999.

http://www.ietf.org/rfc/rfc2648.txt.

[RFC 2822] P. Resnick. *Internet Message Format*. IETF RFC 2822, April 2001. http://www.ietf.org/rfc/rfc2822.txt

[RFC 3161] C. Adams, P. Cain, D. Pinkas, R. Zuccherato. *Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)*. IETF RFC 3161, August 2001.

http://www.ietf.org/rfc/rfc3161.txt.

[RFC 3268] P. Chown. *AES Ciphersuites for TLS*. IETF RFC 3268, June 2002. http://www.ietf.org/rfc/rfc3268.txt.

[RFC 3280] R. Housley, W. Polk, W. Ford, D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. IETF RFC 3280, April 2002.

http://www.ietf.org/rfc/rfc3280.txt.

[RFC 3852] R. Housley. *Cryptographic Message Syntax*. IETF RFC 3852, July 2004.

http://www.ietf.org/rfc/rfc3852.txt.

[SAMLCore1.1] E. Maler et al. Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V 1.1. OASIS, November 2002.

http://www.oasis-open.org/committees/download.php/3406/oasis-sstc-saml-core-1.1.pdf

[Schema1] H. S. Thompson et al. *XML Schema Part 1: Structures.* W3C Recommendation, May 2001.

http://www.w3.org/TR/xmlschema-1/

[SOAP] M. Gudgin et al. *SOAP Version 1.2 Part 1: Messaging Framework.* W3C Recommendation, June 2003.

http://www.w3.org/TR/xmlschema-1/

[XML-C14N] J. Boyer. *Canonical XML Version 1.0*. W3C Recommendation, March 2001.

http://www.w3.org/TR/xml-c14n

[XML-ESCAPE] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, et al. *Predefined Entities* in *Extensible Markup Language (XML) 1.0 (Third Edition)*, W3C Recommendation, 04 February 2004, http://www.w3.org/TR/REC-xml/#dt-escape

[xml:id] xml:id, Version 1.0, W3C Recommendation, 9 September 2005, http://www.w3.org/TR/xml-id/

[XML-ns] T. Bray, D. Hollander, A. Layman. *Namespaces in XML.* W3C Recommendation, January 1999.

http://www.w3.org/TR/1999/REC-xml-names-19990114

[XML-NT-Document] http://www.w3.org/TR/2004/REC-xml-20040204/#NT-document

[XML-PROLOG] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, et al. *Prolog and Document Type Declaration* in *Extensible Markup Language (XML) 1.0 (Third Edition)*, W3C Recommendation, 04 February 2004, http://www.w3.org/TR/REC-xml/#sec-prolog-dtd

[XMLSig] *D. Eastlake et al. XML-Signature Syntax and Processing. W3C Recommendation, February 2002.* http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/

[XML-TSP] T. Perrin et al. *XML Timestamping Profile of the OASIS Digital Signature Services.* W3C Recommendation, February 2002. OASIS, **(MONTH/YEAR TBD)**

[XML] Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation 04 February 2004 http://www.w3.org/TR/REC-xml/#sec-element-content

[XPATH] XML Path Language (XPath) Version 1.0. W3C Recommendation 16 November 1999 http://www.w3.org/TR/xpath

[XML-xcl-c14n] Exclusive XML Canonicalization Version 1.0. W3C Recommendation 18 July 2002 http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/

Appendix A. Use of Exclusive Canonicalization

Exclusive Canonicalization of dereferenced and transformed data can be achieved by appending exclusive canonicalization as the last transform in the `<ds:Transforms>` element of `<TransformedData>` or `<DocumentHash>`.

In the case of `<Document>` being used this can be done by adding exclusive canonicalization as the last transform in the `<ds:Transforms>` of a `<SignedReference>` pointing to that `<Document>`.

By doing this the resulting data produced by the chain of transforms will always be octet stream data which will be hashed without further processing on a `<ds:Reference>` level by the server as indicated by basic processing section 3.3.1 step 1 b. and c.

Another possibility to apply exclusive canonicalization on `<ds:Reference>` level is the freedom given to servers to apply additional transforms to increase robustness. This however implies that only trustworthy transformations are appended by a server.

As in section 3.3.1 step 1 b an implementation can choose to use exclusive canonicalization: "... Transforms are applied as a server implementation MAY choose to increase robustness of the Signatures created. These Transforms may reflect idiosyncrasies of different parsers or solve encoding issues or the like. ..."

In such a case that the exclusive canonicalization is to be included in the `<ds:Transforms>` as well (cf. section 3.3.1 step 1.d.v.)

The standards default is however in line with [XMLSig] as indicated in the Note in section 3.3.1 step 1 b.

However after the server formed a `<ds:SignedInfo>` (section 3.3.1 step 3.) this information to be signed also needs to be canonicalized and digested, here [XMLSig] offers the necessary element `<ds:CanonicalizationMethod>` directly and can be used to specify exclusive canonicalization.

Appendix B. More Complex <Response> Example

To further explain the use of the `<Response>` element which is useful in cases where the DSS server is not able to respond with a special response type a more complex example is given in the following paragraph.

E.g. a client sends a `<SignRequest>` to a service that only supports `<VerifyRequest>`'s over plain HTTP (as opposed to protocols where some information could be derived from the header). As the service does not support `<SignRequest>`'s it has to either generate a `<VerifyResponse>` with a "bad message" result or fail at the HTTP layer. In the former case, the client will receive a response that does not correspond semantically to the request - it got a `<VerifyResponse>` to a `<SignRequest>`. This leaves both parties thinking that the other one is at fault.

2153 # Appendix C. Revision History

Rev	Date	By Whom	What
wd-01	2003-10-03	Trevor Perrin	Initial version
wd-02	2003-10-13	Trevor Perrin	Skeleton of verify as well
wd-03	2003-10-19	Trevor Perrin	Added TimeStampToken, References
wd-04	2003-10-29	Trevor Perrin	Fleshed things out
wd-05	2003-11-9	Trevor Perrin	Added Name, clarified options-handling
wd-06	2003-11-12	Trevor Perrin	Added more options/outputs
wd-07	2003-11-25	Trevor Perrin	URNs, <Timestamp>, other changes.
Wd-08	2003-12-6	Trevor Perrin	Many suggestions from Juan Carlos, Frederick, and Nick incorporated.
Wd-09	2004-1-6	Trevor Perrin	A few minor tweaks to fix a typo, add clarity, and change the order of SignResponse's children
wd-10	2004-1-20	Trevor Perrin	Organized references, updated processing details, touched up a few things.

Rev	Date	By Whom	What
Wd-11	2004-2-04	Trevor Perrin	Added transport and security bindings, and <Language> optional input
wd-12	2004-2-12	Trevor Perrin	Editorial suggestions from Frederick
wd-13	2004-2-29	Trevor Perrin	Added SOAP Transport binding, and made ‘Profile’ attribute part of the Request messages, instead of an option.
Wd-14	2004-3-07	Trevor Perrin	Fixes from Krishna
wd-15	2004-3-08	Trevor Perrin	Property URI -> QNames, added some Editorial issues
wd-16	2004-3-21	Trevor Perrin	Replaced dss:NameType with saml:NameIdentifierType, per Nick’s suggestion.
Wd-17	2004-4-02	Trevor Perrin	Schema URN -> URL, TryAgainLater
wd-18	2004-4-04	Trevor Perrin	Fixes from Karel Wouters
wd-19	2004-4-15	Trevor Perrin	ResultMajor URIs, AdditionalProfile
wd-20	2004-4-19	Trevor Perrin	Updated <Timestamp>, few tweaks
wd-21	2004-5-11	Trevor Perrin	CMS, special handling of enveloping/enveloped DSIG, multi-signature DSIG verification.
Wd-23	2004-6-08	Trevor Perrin	Added DTD example, added returned Profile attribute on SignResponse and VerifyResponse.
Wd-24	2004-6-20	Trevor Perrin	Removed xmlns:xml from schema.
Wd-25	2004-6-22	Trevor Perrin	Fixed a typo.
Wd-26	2004-6-28	Trevor Perrin	Mentioned as committee draft
wd-27	200410-04	Trevor Perrin	Gregor Karlinger’s feedback
wd-28	200410-18	Trevor Perrin	Added a little text to clarify manifests and <ReturnSigningTime>
wd-29	200411-01	Trevor Perrin	Added a little text to clarify <ReturnUpdatedSignature>, and added

Rev	Date	By Whom	What
			<SupportingInfo> to <ClaimedIdentity>

Rev	Date	By Whom	What
wd-30	20041113	Trevor Perrin	-
wd-31	20050627	Stefan Drees	Added all resolved issues from oasis-dss-1.0-comments-track-wd-03
wd-32	20050629	Stefan Drees	Synchronized with Schema, clarified ambiguity issues in Basic Processing for CMS Signatures and Transforms.
wd-33	20050715	Stefan Drees	Added Feedback from mailing list and telco 20050708. Introduced <InlineXMLType>. Simplified basic processing.
wd-34	20051021	Stefan Drees	Added Feedback from discussions of technical committee members from 20050808 through 20051020: Structural changes (optional inputs etc.), new basic processing, consistent handling of XPath and editorial changes/fixes. Preparation for cd-34 candidate: Schema element, Canonicalization and Manifest validation.
Wd-35	20051124	Stefan Drees	PreCD-Version (WD-35) adapting the CD-balloting comments and following e-mail discussions. Added basic time stamping support.
WD-36	20060109	Stefan Drees	Post-CD (WD-36) initial version, including Timestamping contribution in sections 3.5.2 and 5.1.3
WD-37	20060123	Stefan Drees	Embedded feedback upon placement of time stamping chapters
WD-38	20060220	Stefan Drees	Embedded Feedback on ResultMajor and ResultMinor.
WD-39	20060313	Stefan Drees	Embedded feedback upon Timestamps, InlineXML, VerifyRequest, XML Time-stamp with implied references, Trans-formedData, process references for DocumentHash and TranformedData and clarification of some backre-ferences.
WD-40	20060320	Stefan Drees	Minor corrections.
WD-41	20060403	Stefan Drees	Embedded feedback moved editorial issues into comments document and minor/stylish corrections.

Rev	Date	By Whom	What
WD-42	20060423	Stefan Drees	Final comments before CD.

Appendix D. Notices