XML Signature Transform Simplification: Requirements and Design

W3C Working Draft 26 February 2009

This version:
http://www.w3.org/TR/2009/WD-xmldsig-simplify-20090226/
Latest version:
http://www.w3.org/TR/xmldsig-simplify/

Editors:
Frederick Hirsch, Nokia
Pratik Datta, Oracle

Abstract

This document outlines a proposed simplification of the XML Signature Transform mechanism, intended to enhance security, performance, streamability and to ease adoption.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This is a First Public Working Draft of "XML Signature Transform Simplification: Requirements and Design."

This document is expected to be further updated based on both Working Group input and public comments. The Working Group anticipates to eventually publish a stabilized version of this document as a W3C Working Group Note.

The design outlined in this document is part of the XML Security Working Group's development of a revised version 2 of XML Signature; this Working Draft is published to solicit early community review of the direction that the Working Group expects to take with version 2. Early feed-back is welcome.

This document was developed by the XML Security Working Group.

Please send comments about this document to public-xmlsec-comments@w3.org (with public archive).

Table of Contents

1 Introduction

The Reference processing model and associated transforms currently defined by XML Signature [XMLDSIG2nd] are very general and open-ended, which complicates implementation and allows for misuse, leading to performance and security difficulties. Support for arbitrary canonicalization algorithms, and the complexity of the existing algorithms in order to meet various generic requirements is also a source of problems.

Current experience with the use of XML Signature suggests that a simplified reference, transform, and canonicalization processing model would address the most common use cases while improving performance and reducing complexity and security risks [XMLSecNextSteps] [BradHill]. This document outlines a proposed change to the XML Signature processing model to achieve these goals. It also outlines use cases and the new requirements associated with the suggested changes.

It should be noted that this proposal is not for an additional constrained processing model, but for an actual replacement of the existing generically extensible model that exists now. Thus, the changes proposed in this document would be a breaking change to XML Signature, necessitating new implementations and possibly precluding the ability to support some use cases currently supported.

Thus, before making such a change in a proposed new version of XML Signature, the XML Security Working Group would like to obtain additional feedback on this proposal. The purpose of this document is to solicit early feedback.

1.1 A note on namespaces

This document uses the XML namespace http://www.w3.org/2008/xmlsec/experimental# in a number of places. The use of this namespace is for illustrative purposes; should material from this document become normative in the future, a "real" namespace will be allocated.

2 Usage scenarios

One use of an XML Signature is for integrity protection, to determine if content has been changed. Content is identified by one or more ds:Reference elements, causing that content to be located and hashed. In the current XML Signature Second Edition processing model each ds:Reference may include a transform chain to apply one or more transforms before hashing the content for inclusion in a signature.

Obviously a signature operation may occur in a workflow after various transformations have been performed on content, as long as the content can be identified by a ds:Reference at the appropriate point. In this sense, XML Signature could be viewed as a step in a processing model, for example in XProc [XProc]. What is referred to here is not such application processing steps, but only the limited case of transforms defined and processed as part of the XML Signature processing.

There are cases however where transformations must occur as part of signature processing itself.. The reasons for these are more limited, however, so we propose in this document to simplify such processing. Reasons include the following:

1. Signing only pertains to a portion of the content, but the entire content has meaning outside of signing. Thus the signing operation should be able to sign a selected portion of content (and this may be also specified by signing all apart from a portion to be excluded).

2. A signature XML element may be included with the content, yet upon verification the signature element itself is excluded from the content that is verified.

3. Some content within a signature element might be included in signing and verification (e.g. signature properties) even though the signature is not itself.

4. Sometimes it may be necessary to sign, not the raw data, but the data that a user actually sees. This is called "sign what you see" requirement in Section 8.1.2 of the XML Signature specification. This might require, for example, using XSLT to transform the raw data into an

HTML form, and signing this HTML data.

Well-defined signature processing is necessary to handle needs specific to signing, but should not be expected to handle arbitrary processing that could he handled as well as part of a workflow outside of signing.

As an example of the need to sign or verify a portion of the content, suppose you have a document with the familiar "office use only" section. When a user signs the document, the document subset should be the entire document less the "office use only" section. This way, any change made to the document in any place except the "office use only" section would invalidate the signature. The purpose of a digital signature is to become invalid when any change is made, except those anticipated by the system. Thus, subtraction filtering is the best fit for a document subset signature.

By comparison, if a document subset signature merely selects the portion of the document to be signed, then additions can be made not only to the "office use only" section but also to any other location in the document that is outside of the selected portions of the document. It is entirely too easy to exploit the document semantics and inject unintended side effects. That is why exclusion is necessary. All is signed apart from the excluded portion, thus eliminating possibility of unwanted undetected additions.

3 Requirements

There are specific requirements associated with Signature transform processing:

1. Enable applications to determine what is signed.

   Support "see what you sign" by allowing applications to determine what was included for signing and possibly confirm that with users. The current unrestricted transform model makes it very difficult to inspect the signature to determine what was really signed, without actually executing all the transforms.

2. Enable higher performance and streamability

   Signing XML data should be almost as fast as serializing the XML to bytes (using an identity transformer) and then signing the bytes. Currently transforms are defined in terms of a "nodeset" and a nodeset implies using a DOM parser, which is very slow. It should be possible to sign documents using a streaming XML parser, in which the whole document is never loaded in memory at once.

3. Avoid performance penalties and security risks associated with arbitrary transformations by restricting the possible transformation technologies.

   Such generality may still be applied in a workflow outside of signature processing with this restriction.

4. Define a more robust canonicalization

   There are many problems with the current canonicalization algorithms. For example people are really taken aback when they are told that canonicalization does not remove whitespace in between tags. Whitespaces in base64 encoded content causes problems too. Prefix names being significant is yet another source of issues. Schema aware canonicalization is another possibility, but this may have issues related to requiring a schema.

3.1 Enable applications to determine what is signed

The current Transform chain mode is very procedural; it can have XPath, C14N, EnvelopedSign, Base64, XSLT etc transforms any number of times in any order. While this gives a lot of flexibility to the signer, it makes it extremely hard for the verifier to determine what was actually signed.

3.1.1 Current mechanisms to determine what is signed

Applications usually follow one of these mechanisms to determine what is signed

- *Trust the signer completely*

  Some applications do not inspect the transform chain at all. They expect that signer has sent a meaningful and safe transform chain, and since the transform chain is also signed it assures that the chain has not changed in transit.

  This does not work for scenarios where the verifier has little trust in the signer. As an example, suppose there is a application that expects requests to signed with the user's password, and there are tens of thousands of users. This application will of course not trust all of its users, and given the possibility of DoS attacks, and that some transforms can change which is really signed, it will not want to run a chain of transforms that it doesn't understand.

- *Check predigested data*

  Some XML signature libraries have a provision to return the predigested data back to the application, i.e. the octet stream that results from running all the transforms, including an implicit canonicalization at the end.

  The predigested data however cannot be easily compared with the expected data. Suppose the application expects XML elements A, B and C to be signed, it cannot just convert A, B, C to octet streams and search for them inside the predigested data octet stream. The predigested data is canonicalized, and so the search might fail. Also this mechanism is subject to wrapping attacks, as there is no information as to which part of the original document produced this predigested data.

- *Check nodeset just before canonicalization*

  If the transform chain only has nodeset->nodeset transforms (i.e. XPath or EnvelopedSig) in the beginning, followed by one final nodeset->binary transform (i.e. a C14n transform), then an implementation can return the nodeset just before the canonicalization. Unlike the predigested data, this is much easier to compare - DOM specifically has a method to compare nodes for equality, so this method could be used to compare expected nodeset with nodeset just before canonicalization.

  Unfortunately this mechanism does not work if there is any transform that causes an internal conversion from nodeset->binary->nodeset, because in such case the nodes cannot be compared any more. An XSLT transform does this kind of conversion as does the DecryptTransform.

- *Put restrictions on transforms*

  Many higher level protocols put restrictions on the transforms. For example, ebXML specifies that there should be exactly two transforms, namely XPath and then the EnvelopedSig transform. SAML specifies there should be only one transform, the EnvelopedSig transform. This is not a generic solution, but it works well for these specific cases.

3.1.2 Problems with the XPath Transforms

The XPath transform is a very useful transform to specify what is to be signed. Id based mechanisms are simpler, but they have many problems:

1. An Id identifies a complete subtree, if some parts of the subtree have to be excluded an XPath has to be used.
2. An Id attribute has to be of type ID. If there is no schema/DTD information it is not possible to determine the type. Some implementations get around this by having certain reserved names, e.g. `xml:id` or `wsu:id`. These attributes are allowed everywhere and assumed to be of type ID even if there is no schema available.
3. Ids require schema changes usually, i.e. the schema has to identify which elements can have id attributes.
4. Ids can also lead to wrapping attacks.

These problems are solved with XPath, but XPath has even more problems of its own:

1. A regular XPath Filter specifies XPaths "inside out". Anything more difficult than the simplest XPath requires using the "count" and other special functions. The XPath is often so complex it almost impossible to determine what is being signed by looking at the XPath expression.
2. An XPath 2.0 filter solves this problem and lets people write regular XPath, but it hasn't gained wide acceptance because it is optional. Also it offers too much unneeded flexibility allowing any number of union, intersect and subtract operations in any order. This flexibility again makes it harder for the verifier.
3. Unlike the ID which can only be once per reference, an XPath transform can be anywhere in the transform chain. For example, a transform chain can have XPath->C14N->XPath. A verifier getting this kind of transform chain would be clueless about the intent of the transform.

3.1.3 Required "declarative selection"

What would be preferable if instead of transforms the signature were more declarative and clearly separated selection from canonicalization. For example it could list out all the URIs, ids, or included XPaths, excluded XPaths of the the elements that are signed. Then it could apply canonicalization. This would make it easier for the verifier to first inspect the signature to determine what is signed and compare against a policy. To give one example, there might be a WS-SecurityPolicy with an expected list of XPaths. Only if this matches, will the verifier do the canonicalization to compute the digests.

3.2 Enable higher performance and streamability

XML Signature should not require DOM. There are existing streaming XML Signature implementations but they make various assumptions. It would be better to formalize these assumptions and requirements at the standardization level, rather than leave it up to each implementation.

3.2.1 Overheads of DOM

DOM parsers have a large overhead. Suppose there is a 1MB XML document. If this loaded into memory as a byte array it remains as a 1MB byte array. But if it is parsed into a DOM it explodes to 5-10x in size. This is because in DOM, each XML node has to become an object. Objects have overheads of memory book keeping, virtual function tables etc. Also each XML node needs parent, next sibling, previous sibling pointers, and it also needs prefix, namespaceURI etc, which could be objects themselves. All these eat up memory and it is a popular misconception that memory is very cheap. Even if this memory were temporary allocation only it would still be expensive - in garbage collected languages allocating and freeing too much of memory triggers then garbage collector too often which drastically slows down the system. Also this 10x DOM explosion can result in physical memory getting exhausted and requiring more pages to be swapped from disk. That is why web services often use streaming XML parsers on the server side. DOM parsers will croak and groan if asked to process multiple large XML documents simultaneously, whereas streaming XML parsers will happily chug along because of their low memory consumption.

3.2.2 One Pass

It is important to distinguish between one-pass and streamability. Streamability means not requiring to have the whole document in a parsed form available for random access, i.e. not requiring a DOM. While one pass is desirable, two pass doesn't take away all the merits of streaming. Suppose the signature value is before the data to be signed. This means that the signature value cannot be updated in the first pass, but only in the second pass - this is not really bad from the performance point of view. Let us the say the document is being streamed out into 1MB byte array, then in the first pass write some dummy bytes for this signature value and remember the location, and in the 2nd pass just update this location with the actual signature bytes, so the 2nd pass is very quick.

Also streamability does not require the ordering between the subelements of signature element. It can be assumed that the entire Signature element (assuming it is detached or enveloped signature) will be loaded up into a java/c++ object, so the order of the elements inside the Signature element does not affect streamability.

Verification in particular cannot be 1 pass - let us say you have a signed 1GB incoming message, which you need to verify first and then upload to a database. So you have to make two passes on this data - a first pass to verify and second pass to upload to the database. One cannot combine these two into 1 pass because verification result is determined only after reading the last byte.

3.2.3 Nodeset

The main impediment to streamability is the transform chain, because many of the transforms are defined on nodesets and nodeset requires a DOM. An XPath transform is the biggest culprit as there are many XPath expressions which cannot be streamed. It is necessary to define a streamable subset of XPath.

Nodesets have another big problem. This nodeset concept was borrowed from XPath 1.0, and an XPath nodeset introduces a new kind of XML node - the namespace node. Namespace nodes are different from namespace declarations in an important way - they are not inherited. This means they need to be repeated for every node for which they are applicable. To give an example, if there is a document with 100 namespace declarations at the top element and with 99 child elements of the top element, a regular DOM will only have 200 (1 top element node + 99 child element nodes + 100 attribute nodes), whereas a nodeset will have 10,100 nodes (1 top element + 99 child element + 100*100 namespace nodes).

A naive implementation which uses the nodeset as defined will therefore be very slow, and be also be subject to various denial of service attacks. A smart implementation can try to not expand the nodeset fully and use inheritance, but they it won't be fully compliant with the XML Signature spec. This is because an XPAth filter can address each of namespace nodes individually and filter them out, even though it is meaningless in XML. The Y4 test vector in the first interop has example of this. Because of these performance problems some implementations do not support this Y4 test vector or support it conditionally.

3.3 Avoid Security risks

The Best practices document points out many potential security risks in XML Signatures.

1. *Order of operations*

   Reference validation before signature validation is extremely susceptible to denial of service attacks in some scenarios.

2. *Insecurities in XSLT transforms*

   XSLT is a complete programming language. An untrusted XSLT can use deeply nested loops to launch DoS attacks, or use "user defined extensions" like "os.exec" to execute system commands.

3. *Full expansion of Nodesets*

   As mentioned above a full expansion of an XPath nodesets results in a huge amount of memory usage, and this can be exploited for DoS attacks.

4. *Complex XPaths*

   XPath Filter 1.0 requires very complex looking XPaths, these are very hard to understand, and an application can be potentially fooled into believing something is signed, whereas is is actually not. Also complex XPaths can use too many resources.

5. *Wrapping attacks*

   ID based references and lack of a mechanism to determine what was really signed can enable to wrapping attacks.

6. *Problems with `RetrievalMethod`*

   RetrievalMethod can lead to infinite loops. Also transforms in retrieval method can lead to many attacks, and these cannot be solved by changing the order of operations.

These security risks need to be addressed in the new specification.

3.4 Canonicalization

Besides the explicit design principles and requirements in [C14N-REQS], the Canonical XML and Exclusive Canonicalization specifications are guided by a number of design decisions that we present and discuss in this section.

3.4.1 Historical requirements

The basic idea of a canonical XML is to have a representation of an XML document (the output being a concrete string of bytes) that captures some kind of "essence" of the document, while disregarding certain properties that are considered artifacts of the input document (thought of, again, as an octet stream), and deemed to be safely ignorable.

The historic Canonical XML Requirements [C14N-REQS] include:

- The specification for Canonical XML shall describe how to derive the canonical form of any XML document. Every XML document shall have a unique canonical form.
- The canonical form of an XML document shall be a well formed XML document with the following invariant property:
  - Any XML document, say X, processed by a canonicalizer, will produce an XML Document X'.
  - X' passed through the same canonicalizer must produce X'.
  - X' passed through any other conforming canonicalizer should produce X', or else one of them in not conformant.

In other words, Canonicalization is historically thought of as a well-defined, idempotent mapping from the set of XML documents into itself.

In its main use case, XML Signature, Canonical XML [C14N] (and its cousin, Exclusive Canonicalization) is actually used to fulfill a number of distinct functions:

- Canonical XML is used as the canonical mapping from a node-set to an octet stream whenever such a mapping is required to connect distinct transforms to each other.
- Canonical XML is used to serialize the `ds:SignedInfo` element before it is hashed as part of the signing process; note that this element does not necessarily exist as a serialization.
- Canonical XML is used to discard artifacts of a specific representation before that representation is hashed in the course of either signature generation or validation.

3.4.2 Modified Requirements

This section summarizes a number of design options that arise when some of the requirements listed above are relaxed.

3.4.2.1 Only use Canonicalization for pre-hashing

It is not required to have canonicalization as general purpose transform to be used anywhere in a transform chain. Its only use would be to produce an octet stream that will be hashed.

Currently canonicalization is used whenever there is an impedance mismatch with one transform emitting binary, and next transform requiring nodeset. This is not required any more.

Also Canonicalization is picked up some other specs e.g. DSS to do some cleanup of the XML. This is not required either

3.4.2.2 Canonical output need not be valid XML

Assuming that a canonicalization step is necessary to be performed as the last step of reference processing before hashing of the resulting octet-stream, the requirement that XML canonicalization *produce* valid XML could be relaxed. Some interesting things can be done with this relaxation - namespace prefixes can be expanded out, tag names in closing tags can be omitted, and EXI serialization format can be used. A possible design is described in [Thompson].

3.4.2.3 Define a well-defined (and limited) serialization for `ds:SignedInfo`

For every application of XML Signature, a `ds:SignedInfo` element needs to be hashed and signed. This step *always* involves canonicalization of a document subset. While some parts of `ds:SignedInfo` include an open content model (`ds:Object`, in particular), there is a large class of signatures for which the content model of `ds:SignedInfo` is well-understood. A special-purpose canonicalization algorithm might be cost-effective if it can reduce the computational cost for canonicalizing `ds:SignedInfo` in a suitably large portion of use cases.

3.4.2.4 Limit the acceptable inputs for Canonicalization

This design option could manifest itself in several ways.

Constrain the classes of node-sets that are acceptable.

There is no need to be able to canonicalize a fully generic nodeset. Nodeset is an XPath concept and a generic nodeset can have many strange things - like attribute nodes without the containing element, removal of namespace nodes without removal of the corresponding namespace declarations - these kinds of things only increase the complexity of the Canonicalization algorithm without adding any value.

Instead of a generic nodeset, canonicalization needs to work on a different data model :

- Start with a subtree or a set of subtrees. These subtrees must be rooted at element nodes. For example, these subtrees can't be a single text node or a single attribute node.
- Optionally from this set, exclude some subtrees (of element nodes) or exclude some attribute nodes. Can only exclude regular attributes, not attributes that are namespace declarations. TBD if xml: attributes can be excluded.
- Optionally to this set, reinclude some subtrees (of element nodes)

This data model avoids namespace nodes completely. It only deals with namespace declarations. It also prohibits attribute nodes without parent element nodes. Another simplification with this model is if an element node is present, all its namespace declarations and all its child text nodes have to be present.

Constrain the classes of XML documents that are acceptable.

Canonical XML currently expends much complexity on merging relative URI references appearing in `xml:base` parameters. A revised version of Canonical XML could be defined to fail on documents in which the `xml:base` URI reference cannot be successfully absolutized.

3.4.2.5 Relax certain guarantees

Handling of namespaces is a known major source of complexity in Canonical XML (and, to a lesser extent, in Exclusive Canonicalization). At least part of this complexity is due to a design decision to preserve namespace prefixes, which in turn is necessary to protect the meaning of QNames.

A limited revised version of Canonical XML might be one in which namespace prefixes are not guaranteed to be preserved, possibly breaking the meaning of QNames.

4 Design

As mentioned above, the term "transform" implies a processing step, so we propose a new syntax which is more "declarative" and less "procedural". The implementation can now choose the most efficient way to perform the signature. This new syntax can be mapped exactly to a subset of the old syntax, so an implementation can simply convert this 2.0 syntax to a 1.0 syntax and execute it using an 1.0 XML Signature implementation if needed. Note however that not all 1.0 transformations can be expressed in a 2.0 format.

4.1 Overview of new syntax

Here is an overview of the new syntax

```
<Reference>
  <Selection
    type="http://www.w3.org/2008/xmlsec/experimental#xml"
    URI=" ..."
    includedXPath=".."
    excludedXPath="..."
    reincludedXPath="..."
    envelopedSignature="true/false">
  ...
  </Selection>

  <Transforms>
    ...
  </Transforms>

  <Canonicalization
     inclusive="yes"
     ignoreComments="yes">
   ...
  </Canonicalization>
</Reference>
```

Notice how the single `Transforms` section has been split up into three sections `Selection`, `Transforms` and `Canonicalization`. Each of these elements can be present at most once and they have to be in that order.

- The `Selection` element identifies the data that is selected for signing. For XML data it is equivalent to a restricted XPath Filter 2 transform followed by an EnvelopedSignature transform. The included/excluded/reincluded XPath attributes are exactly equivalent to the Intersect/Subtract/Union filters of an XPath2 transform (in that order) and the envelopedSignature attribute is equivalent to an EnvelopedSignature transform. The `URI` attribute has been moved from the `Reference` element to the `Selection` element.

- The `Transforms` element is optional, it can only have a restricted XSLT transform and a Decrypt transform. This element is only to support the "sign what is seen" requirement, i.e. to convert the raw data to a form that the user sees.

- The `Canonicalization` element converts the data into a octet stream. This element is equivalent to a modified canonicalization transform.

Even though this new syntax can be mapped to the transform model, it is greatly restrictive. These restrictions can be explained in terms of the old syntax as follows.

- There is only one Canonicalization transform and that is always the last one. Canonicalization cannot be used when signing binary data.
- There can be only one XPath Filter 2 transform, and that should be the first one, XPath Filter 2 should have at most one each of Intersec t, Subtract, Union filters, in that order. XPath Filter 1 transform is not used.
- There can be only one Enveloped Signature transform, and that is always after the XPath Filter transform.
- Binary mode uses a different set of transforms - see details below. There are two subtypes for binary. BinaryFromExternalURI, BinaryFromBase64Nodes. BinaryFromExternalURI can only have the ByteRange transform. BinaryFromBase64Node can only have XPathFilter2, Base64Decode and ByteRange transforms in that order

These restriction achieve two important things. First it makes it easy to determine what is signed. The application does not need to execute the transforms to determine what is signed, it just needs to inspect the attributes/subelements of the `Selection` element.

Secondly these restrictions also enable higher performance. While a simple implementation can simply convert the new syntax to the old syntax and just rely on a existing XML signature implementation, a brand new implementation can be do things very differently as follows:

- In the new syntax the output of the canonicalization is only used for digesting so an implementation can do the canonicalization and digesting together, thereby avoiding allocating a large memory buffer to hold the canonicalized output.
- Because of these restrictions, the implementation can take many shortcuts, for example instead of doing the EnvelopedSignature as a whole new transform, it can just "mark" the signature node, and then while performing canonicalization, it can simply skip over this signature subtree. In the earlier syntax this was not always possible because there can be an XPath filter transform after an Enveloped Signature Transform, which reintroduces the Signature element or parts of it, so an implementation cannot assume that an EnvelopedSignatureTransform will definitely remove the Signature element.
- The new syntax doesn't use a "nodeset". Nodeset is inherently a DOM concept and not scalable. Also there is no implicit conversion from nodeset to binary, or binary to nodeset - which are very expensive operations. See below for a streaming algorithm, which does XPath filtering, enveloped signature and canonicalization all together, without using a nodeset.

4.2 The `Selection` element

The `Selection` element chooses what is to be signed. By clearly separating out this section from the rest of the transforms, it becomes much easier to determine what is signed. The `type` and `subtype` attributes specifies what kind of data is being signed. type can be `"http://www.w3.org/2008/xmlsec/experimental#xml"` or `"http://www.w3.org/2008/xmlsec/experimental#binary"` or any other user defined value. This attribute makes the intention of the signature very clear, so the implementation doesn't have to deduce it by looking at the transforms.

- `type = "...xml"`

  This indicates XML data has been signed - the subset is either indicated by the URI and the three optional XPath attributes. Examples:

1. `URI="#chapter1"` and all the XPath attributes be absent - indicates that complete subtree identified by the ID "chapter1", in current document is being signed.
2. `URI=""` and `includedXPath="/book/chapter"` indicates that all the subtrees indicated by "/book/chapter" in the current document are signed
3. `URI="#chapter1"` and `excludedXPath="price"` indicates the subtree identified by the ID "chapter1" minus any subtrees with "price" element are being signed.
4. `URI="http://example.com/bar.xml"` indicates that the entire external document bar.xml is signed
5. `URI="http://example.com/bar.xml"` and `includedXPath="/book/chapter"`indicates that the /book/chapter subtrees of the external document bar.xml are signed.

- `type = "...binary"` and `subtype = "...fromURI"`

  This indicates that binary data directly fetched from an external URI is signed. IDs cannot be used , nor can XPath attributes.

- `type = "...binary"` and `subtype = "...fromBase64Node"`

  This indicates that binary data which is present in the XML as a base64 text node is being signed. Just like the `type="...xml"` an combination of URI and includedXPath attributes can be used to identify an element have text node children. These text nodes will be coalesced and then base64 decoded, to get the binary data. This is subset of the Base64Decode transform, the Base64Decode transform works with nodeset containing multiple element nodes, but this one is only defined for a single element node.

Examples of binary signatures

```
<Selection
   type="http://www.w3.org/2008/xmlsec/experimental#binary"
   subtype="http://www.w3.org/2008/xmlsec/experimental#fromURI"
   URI="..."
   byteRange="0-20,220-270,320-"
 />

<Selection
   type=""http://www.w3.org/2008/xmlsec/experimental#binary"
   subtype=""http://www.w3.org/2008/xmlsec/experimental#fromBase64Node"
   URI="..."
   includedXPath=".."
   excludedXPath="..."
   reincludedXPath="..."
   byteRange="0-20,220-270,320-"
/>
```

The `byteRange` attribute represents the ByteRange Transform which was proposed by Chris Solc. It is always applied last.

Binary data is not canonicalized. So there should not be a canonicalization section for binary data.

4.2.1 The XPath attributes

In the current transform model, there are three ways of specifying XPath - 1) in XPath Filter which uses an "inside-out" XPath, 2) in XPath Filter 2 which uses regular XPath, 3) as XPointer URIs.

The new syntax settles on one model, a subset of the XPath Filter 2 transform. The reason for this choice is that the alternatives are problematical. XPath Filter 1 uses very complex and unreadable

XPaths. XPointer cannot do exclusions. XPath Filter 2 allows any number of Intersect, Subtract, Union filters in any order, but we restrict it down to just one of each in that order.(Refer email chain with John Boyer).

1. First evaluate the URI to get a subtree (or entire document if URI is not present).
2. Then evaluate the `includedXPath` and take the intersection of the previous subtree and the subtrees identified by this XPath.
3. Then evaluate the `excludedXPath` and subtract away these subtrees from the previous result.
4. Finally evaluate the `reincludedXPath` and do a union of these subtrees with the precious result.

4.2.2 Subset of XPath for performance

Only a subset of XPath expressions are allowed. This is a subset that can be easily evaluated by a streaming XPath implementation. Here are the restrictions:

- The XPath expression can only select elements. It cannot select namespace nodes as they are extremely detrimental for performance. Text nodes are also disallowed because text nodes can be very long, and so expressions using text nodes values can be very slow. Attributes are disallowed because they can result in attributes without their parent element, which cannot be represented in a streaming parser like StAX. However we can choose to allow attributes in the excludedXPath expression, but not namespace attributes and not xml: attributes, as excluding these really complicates canonicalization.
- The XPath expression can only use self, child, descendant and attribute axis. That is because a streaming parser only knows the current node, its attributes and ancestors.
- The XPath expression can have a predicate only at the last step. There are many restrictions on this predicate - it can only have an expression using the element name or attribute values, it cannot use any functions, especially the `pos` function. Simple expressions are allowed, however.

Here is an algorithm for Streaming XPath. For simplicity this algorithm assumes that excludedXPath and reincludedXPath are not present:

For parsing:

1. Split up the union expression by `"|"`. i.e. break up the `locationPath | locationPath | ..` into individual location paths.
2. Split up each location paths to get individual steps and the final predicate. i.e. break up the `/ step / step / step .. / step [ predicate ]` to get the steps and optional predicate. Two slashes together indicates descendant axis.
3. The predicate will have an expression involving attribute names e.g. `@a = "foo"` and `@b > "bar"` You need to have an expression parsing and evaluating engine to do this.

For executing:

1. A streaming XML Parser (e.g. StAX), reads an XML document and produces "events" like StartElement, EndElement, TextNode etc. At any point this parser only remembers the current node. If the current node is an start element, then it also reads all the attributes for that element. To execute a streaming XPath you maintain a stack of ancestor element

names, i.e whenever you get a StartElement tag, you need to push the element QName onto this stack, and when you get an EndElement tag you need to pop it off.

2. As you stream through the nodes, you need to execute this XPath expression for every node. I.e. utilize the current element, the current element's attributes and the stack of ancestors to evaluate the XPath expression.

   For each `locationPath`, match up the `step`s to the ancestor stack, If they match, evaluate the predicate with the current element's expression. If that passes too, this element and all its descendants are included.

4.3 The `Transforms` element

4.3.1 Sign what is seen

While Web services do not have the requirement to sign what is seen, document signing often has this requirement. Suppose the document is derived from data residing in a database, instead of signing the raw data, the data is transformed to HTML using an XSLT and resulting HTML is signed. This is because the raw data is not something that the user sees, but the HTML and associated stylesheets.

The decrypt transform is kind of a similar, instead of signing the encrypted data, it is decrypted and the resulting plaintext data is signed.

From the security point of view, signing the raw data is probably as secure as signing the transformed data. Also doing these transforms is expensive - decryption and XSLT are expensive operations, and result is thrown after digest computation. So the document needs to be transformed again to be displayed to the user - (intermediate results from a transform chain are not available).

Apart from being expensive XSLT can also be very insecure. XSLT is a complete programming language and it can have infinite loops leading to denial of service attacks. It can also have use extension mechanisms to call into other code. However XSLT can be safely used in certain scenarios - it could be a well known XSLT, or maybe the signer and verifier are the same entity, so implicitly trust the XSLT.

So we propose a modified XSLT transform which only supports well known XSLT.

```
  <Transforms>
     <Transform Algorithm="...xslt.." xsltName="foo.xsl" />
   </Transforms>
```

Note: The XSLT is referred to by a well known name, not a file path.

4.4 The `Canonicalization` element

The canonicalization is always the last element, and it is optional for binary data.

```
   <Canonicalization
      inclusive="true"
      ignoreComments="true"
      trimTextNodes="true"
      serialization="EXI/XML"
      >
    <InclusiveNamespaces  PrefixList="..."/>
   </Canonicalization>
```

Canonicalization is now only used to produce the input for the hash. So an implementation can combine canonicalization and hash together. (There is a proposal to rename this `Canonicalization` element to `HashPrep`, to explain this intent, but canonicalization is a more familiar word).

This canonicalization is expressed as a combination of the following properties rather than an algorithm URI:

1. `inclusive` whether to do inclusive or exclusive dealing of namespaces. In exclusive mode the InclusiveNamespace parameter can be specified listing the prefixes that are to be treated in an inclusive mode

2. `ignoreComments` whether to ignore comments during canonicalization

3. `trimTextNodes` whether to trim (i.e. remove leading and trailing whitespaces) all text nodes when canonicalizing. Adjacent text nodes must be coalesced prior to trimming. (A better approach would be to remove only "non significant" whitespace, but that is not possible to determine without the schema.)

4. `serialization` whether to output in regular XML format, or some kind of compact XML format. A compact XML would result in fewer bytes going to the digestor which would speed it up. EXI is one such format. Another suggested format is to remove the tag name from the closing tag. i.e. instead of `<foo>bar</foo>` use `<foo>bar</>`

5. `preservePrefixes` whether the prefix name is significant. When there are QNames in content, prefixes are probably significant, otherwise they could be expanded out into URIs or converted into n1. n2, n3 etc

6. `sortAttributes`whether the attributes need to be sorted before canonicalization. In some environments the order of attributes changes in transit so sorting is important.

The combination of `inclusive="true"`, `ignoreComments="true"`, `trimTextNodes="false"`, `serialization="XML"`, `preservePrefixes="true"` and `sortAttributes="true"` is almost exactly equal to the current inclusive canonicalization with no comments algorithm. The only difference is with respect to entity expansion.

Canonicalization will not imply DTD validation and entity expansion. DTD processing makes time and resource requirements for core validation non-deterministic, introduces difficult-to-control resource resolution requirements and requires tight coupling between validators and signed content consumers to ensure they have the same view of DTDs.

The choice and order of DTD resolution and entity expansion relative to signature creation and validation would thus fall to application workflow outside of core XMLDSIG.This change will introduce additional complexity for applications relying on entities, but entity expansion as a mandatory part of signature validation is incompatible with core requirements of XMLDSIG.

Canonicalization may also be required for binary content. Rationale, explanation and design TBD.

4.5 Extensions in the new syntax

In the old syntax the extension points were two - defining new URI schemes and defining new Transforms. But in the new syntax, there are more possibilities. First thing is type/subtype - if a different type of data is being signed, then new type/subtypes should be defined. Whereas if it is a modification to selection process or the canonicalization process, then the new attributes/subelements should be defined under `Selection` or `Canonicalization`. For example

- The WS-Security SWA profile defines new transforms - AttachmentComplete and

AttachmentContentOnly. These transforms are to be used in conjunction with "cid:" URIs. The AttachmentContentOnly only fetches the attachment body only, and depending on the attachment mime type, it does xml canonicalization, text canonicalization or no canonicalization. The AttachmentComplete transform is slightly more complex - it fetches both the attachment body and attachment header and it canonicalizes the body depending on the mime type, and canonicalizes the attachment headers as well.

In the new syntax this can be defined much more cleanly. type should "...xml", "...binary" or "...text" depending on the mime content type of the attachment, and subtype should be "...soapAttachment". Canonicalization will not be part of this transform but rather part of the `Canonicalization` element, which should have have new mechanism for canonicalizing text data and for canonicalization mime headers.

- The WS-Security STR-Transform, on the other hand is a more regular transform, so it can be defined as a new attribute `replaceSTRwithST="true/false"` of the `Selection` element. Note, when defining new attribute the interaction should also be clearly defined. I.e. if there is both an envelopedSignature attribute and a replaceSTRWithST attribute then the interaction from them should be defined. In this case it doesn't matter which one is done first.

5 Acknowledgments

Thanks to John Boyer for his suggestions on this topic.

6 References

BradHill
Complexity as the Enemy of Security: Position Paper for W3C Workshop on Next Steps for XML Signature and XML Encryption, Brad Hill, 25-26 September 2007, http://www.w3.org/2007/xmlsec/ws/papers/04-hill-isecpartners/

ByteRangeTransform
email, Chris Solc, 6 October 2008, http://lists.w3.org/Archives/Public/public-xmlsec/2008Oct/0011.html

C14N
Canonical XML 1.1, John Boyer, Glenn Marcy. W3C Recommendation 2 May 2008, http://www.w3.org/TR/2008/REC-xml-c14n11-20080502/.

C14N-REQS
XML Canonicalization Requirements, James Tauber, Joel Nava. W3C Note, 5 June 1999, http://www.w3.org/TR/1999/NOTE-xml-canonical-req-19990605.

Thompson
Radical proposal for Vnext of XML Signature, Henry Thompson. Position paper, 26 September 2007, http://www.w3.org/2007/xmlsec/ws/papers/20-thompson/.

WSS_SWA
Web Services Security, SOAP Messages with Attachments (SwA) Profile 1.1 http://www.oasis-open.org/committees/download.php/16672/wss-v1.1-spec-os-SwAProfile.pdf

XMLDSIG
XML-Signature Syntax and Processing, D. Eastlake, J. R., D. Solo, M. Bartel, J. Boyer , B. Fox , E. Simon. W3C Recommendation, 12 February 2002, http://www.w3.org/TR/xmldsig-core/.

XMLDSIG-REQS
XML-Signature Requirements, Joseph Reagle. W3C Working Draft, 14 October 1999, http://www.w3.org/TR/xmldsig-requirements.

XMLDSIG2nd

XML Signature Syntax and Processing (Second Edition), W3C Recommendation 10 June 2008 http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/

XMLSecNextSteps
Workshop Report W3C Workshop on Next Steps for XML Signature and XML Encryption, W3C, 25-26 September 2007, http://www.w3.org/2007/xmlsec/ws/report.html

XPathFilter2Issues
email, Pratik Datta, 29 October 2008, http://lists.w3.org/Archives/Public/public-xmlsec/2008Oct/0047.html

XProc
XProc: An XML Pipeline Language, Walsh, N., Milowski A., Thompson, H., W3C Candidate Recommendation, 26 November 2008. http://www.w3.org/TR/2008/CR-xproc-20081126/. The status of this document is draft work in progress and it is subject to change.

XpathExcludeReinclude
email, John Boyer, 29 October 2008, http://lists.w3.org/Archives/Public/public-xmlsec/2008Oct/0048.html