XML Signature Best Practices

W3C Working Draft 26 February 2009

This version:
http://www.w3.org/TR/2009/WD-xmldsig-bestpractices-20090226/
Latest version:
http://www.w3.org/TR/xmldsig-bestpractices/
Previous version:
http://www.w3.org/TR/2008/WD-xmldsig-bestpractices-20081114/

Editors:
Frederick Hirsch, Nokia
Pratik Datta, Oracle

Abstract

This document collects best practices for implementors and users of the XML Signature specification [XMLDSIG]. Most of these best practices are related to improving security and mitigating attacks, yet others are for best practices in the practical use of XML Signature, such as signing XML that doesn't use namespaces, for example.

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 Working Draft of "XML Signature Best Practices".

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 practices in this document have been found generally useful and safe. However, they do not constitute a normative update to the XML Signature specification, and might not be applicable in certain situations.

This document was developed by the XML Security Working Group.

A diff-marked version of this specification which highlights changes against the previous version is available.

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

Publication as a Working Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. The group does not expect this document to become a W3C Recommendation. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

---

1 Overview

The XML Signature specification [XMLDSIG] offers powerful and flexible mechanisms to support a variety of use cases. This flexibility has the downside of increasing the number of possible attacks. One countermeasure to the increased number of threats is to follow best practices, including a simplification of use of XML Signature where possible. This document outlines best practices noted by the XML Security Specifications Maintenance Working Group, the XML Security Working Group, as well as items brought to the attention of the community in a Workshop on Next Steps for XML Security [XMLSecNextSteps]. While most of these best practices are related to improving security and mitigating attacks, yet others are for best practices in the practical use of XML Signature, such as signing XML that doesn't use namespaces.

2 Best Practices

2.1 For Implementors: Reduce the opportunities for denial of service attacks

XML Signature may be used in application server systems, where multiple incoming messages are being processed simultaneously. In this situation incoming messages should be assumed to be possibly hostile with the concern that a single poison message could bring down an entire set of web applications and services.

Implementation of the XML Signature specification should not always be literal. For example, reference validation before signature validation is extremely susceptible to denial of service attacks in some scenarios. As will be seen below, certain kinds of transforms may require an enormous amount of processing time and certain external URI references can lead to possible security violations. One recommendation for implementing the XML Signature Recommendation is to first "authenticate" the signature, before running any of these dangerous operations.

Best Practice 1: Mitigate denial of service attacks by executing potentially dangerous operations only after authenticating the signature.

Validate the ds:Reference elements for a signature only after establishing trust, for example by verifying the key and validating ds:SignedInfo first.

XML Signature operations should follow this order of operations:

1. *Step 1* fetch the verification key and establish trust in that key
2. *Step 2* validate SignedInfo with that key
3. *Step 3* validate the references

In step 1 and step 2 the message should be assumed to be untrusted, so no dangerous operations should be carried out. But by step 3, the entire Signed info has been authenticated, and so all the URIs and transforms in the SignedInfo can be attributed to a responsible party. However an implementation may still choose to disallow these operations even in step 3, if the party is not trusted to perform them.

In step 1, if the verification key is not known beforehand and needs to be fetched from KeyInfo, the care should be taken in its processing. The KeyInfo can contain a RetrievalMethod child element, and this could contain dangerous transforms, insecure external references and infinite loops (see Best Practice #5 and examples below for more information).

Another potential security issue in step 1 is the handling of untrusted public keys in KeyInfo. Just because an XML Signature validates mathematically with a public key in the KeyInfo does not mean that the signature should be trusted. The public key should be verified before validating the signature value.

For example, keys may be exchanged out of band, allowing the use of a KeyValue or X509Certificate element directly. Alternatively, certificate and path validation as described by RFC 5280 or some other specification can be applied to information in an X509Data element to validate the key bound to a certificate. This usually includes verifying information in the certificate such as the expiration date, the purpose of the certificate, checking that it is not revoked, etc.

Key Validation is typically more than a library implementation issue, and often involves the incorporation of application specific information. While there are no specific processing rules required by the XML Signature specification, it is critical that applications include key validation processing that is appropriate to their domain of use.

Best Practice 2: Establish trust in the verification/validation key.

Establish appropriate trust in a key, validating X.509 certificates, certificate chains and revocation status, for example.

2.1.1 Example: XSLT transform that causes denial of service

The following XSLT transform contains 4 levels of nested loops, and for each loop it iterates over all the nodes of the document. So if the original document has 100 elements, this would take 100^4 = 100 million operations. A malicious message could include this transform and cause an application server to spend hours processing it. The scope of this denial of service attack is greatly reduced when following the best practices described above, since it is unlikely that an authenticated user would include this kind of transform. XSLT transforms should only be processed for References, and not for KeyInfo RetrievalMethods, and only after first authenticating the entire signature and establishing an appropriate degree of trust in the originator of the message.

Example: dos_xslt.xml

```
<Transform Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116">
  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
      <xsl:for-each select="//. | //@*">
        <xsl:for-each select="//. | //@*">
          <xsl:for-each select="//. | //@*">
            <foo/>
          <xsl:for-each>
      <xsl:for-each>
    <xsl:for-each>
  </xsl:stylesheet>
<Transform>
```

As discussed further, below, support for XSLT transforms may also expose the signature processor or consumer to further risks in regard to external references or modified approvals. An implementation of XML Signature may choose not to support XSLT, may provide interfaces to allow the application to optionally disable support for it, or may otherwise mitigate risks associated with XSLT.

Best Practice 3: Consider avoiding XSLT Transforms

Arbitrary XSLT processing might lead to denial of service or other risks, so either do not allow XSLT transforms, only enable them for trusted sources, or consider mitigation of the risks.

2.1.2 Example: XPath Filtering transform that causes denial of service

The following XPath Transform has an expression that simply counts all the nodes in the document, but it is embedded in special document that has a 100 namespaces ns0 to ns99 and a 100 <e2> elements. The XPath model expects namespace nodes for each in-scope namespace to be attached to each element, and since in this special document all the 100 namespaces are in scope for each of the 100 elements, the document ends up having 100x100 = 10,000 NamespaceNodes.Now in an XPath Filtering transform, the XPath expression is evaluated for every node in the document. So it takes 10,000 x 10,000 = 100 million operations to evaluate this document. Again the scope of this attack can be reduced by following the above best practices

Example: dos_xpath.xml

```
       <dsig:Transform Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
        <dsig:XPath>count(//. | //@* | //namespace::*)</dsig:XPath>
       </dsig:Transform>
```

An implementation of XML Signature may choose not to support the XPath Filter Transform, may provide interfaces to allow the application to optionally disable support for it, or otherwise mitigate risks associated with it. Another option is to support a limited set of XPath expressions - which only use the ancestor or self axes and do not compute string-value of elements. Yet another option is to use the XPath Filter 2.0 transform instead, because in this transform, the XPath expressions are only evaluated once, not for every node of the transform.

Best Practice 4: Try to avoid or limit XPath transforms

Complex XPath expressions (or those constructed together with content to produce expensive processing) might lead to a denial of service risk, so either do not allow XPath transforms or take steps to mitigate the risk of denial of service.

2.1.3 Example: Retrieval method that causes an infinite loop

The KeyInfo of a signature can contain a RetrievalMethod child element, which can be used to reference a key somewhere else in the document. RetrievalMethod has legitimate uses; for example when there are multiple signatures in the same document, these signatures can use a RetrievalMethod to avoid duplicate KeyInfo certificate entries. However, referencing a certificate (or most other KeyInfo child elements) requires at least one transform, because the reference URI can only refer to the KeyInfo element itself (only it carries an Id attribute). Also, there is nothing that prevents the RetrievalMethod from pointing back to itself directly or indirectly and forming a cyclic chain of references. An implementation that must handle potentially hostile messages should constrain the RetrievalMethod elements that it processes - e.g. permitting only a same-document URI reference, and limiting the transforms allowed.

Example: dos_retrieval_loop1.xml

```
<RetrievalMethod xml:id="r1" URI="#r1"/>
```

Example: dos_retrieval_loop2.xml

```
<RetrievalMethod Id="r1" URI="#r2"/>
<RetrievalMethod Id="r2" URI="#r1"/>
```

Best Practice 5: Try to avoid or limit RetrievalMethod support with KeyInfo

RetrievalMethod can cause security risks due to transforms, so consider limiting support for it.

2.1.4 Example: Problematic external references

An XML Signature message can use URIs to references keys or to reference data to be signed. Same document references are fine, but external references to the file system or other web sites can cause exceptions or cross site attacks. For example, a message could have a URI reference to "file://etc/passwd" in its KeyInfo. Obviously there is no key present in file://etc/passwd, but if the xmlsec implementation blindly tries to resolve this URI, it will end up reading the /etc/passwd file. If this implementation is running in a sandbox, where access to sensitive files is prohibited, it may be terminated by the container for trying to access this file.

URI references based on HTTP can cause a different kind of damage since these URIs can have query parameters that can cause some data to be submitted/modified in another web site. Suppose there is a company internal HR website that is not accessible from outside the company. If there is a web service exposed to the outside world that accepts signed requests it may be possible to inappropriately access the HR site. A malicious message from the outside world can send a signature, with a reference URI like this http://hrwebsite.example.com/addHoliday?date=May30. If the XML Security implementation blindly tries to dereference this URI when verifying the signature, it may unintentionally have the side effect of adding an extra holiday.

When implementing XML Signature, it is recommended to take caution in retrieving references with arbitrary URI schemes which may trigger unintended side-effects and/or when retrieving references over the network. Care should be taken to limit the size and timeout values for content retrieved over the network in order to avoid denial of service conditions.

When implementing XML Signature, it is recommended to follow the recommendations in section 2.3 to provide cached references to the verified content, as remote references may change between the time they are retrieved for verification and subsequent retrieval for use by the application. Retrieval of remote references may also leak information about the verifiers of a message, such as a "web bug" that causes access to the server, resulting in notification being provided to the server regarding the web page access. An example is an image that cannot be seen but results in a server access [WebBug-Wikipedia] .

When implementing XML Signature with support for XSLT transforms, it can be useful to constrain outbound network connectivity from the XSLT processor in order to avoid information disclosure risks as XSLT instructions may be able to dynamically retrieve content from local files and network resources and disclose this to other networks.

Some kinds of external references are perfectly acceptable, e.g. Web Services Security uses a "cid:" URL for referencing data inside attachments, and this can be considered to be a same document reference. Another legitimate example would be to allow references to content in the same ZIP or other virtual file system package as a signature, but not to content outside of the package.

The scope of this attack is much reduced by following the above best practices, because with that only URIs inside a validated SignedInfo section will be accessed. But to totally eliminate this kind of attack, an implementation can choose not to support external references at all.

Best Practice 6: Control External References

To reduce risks associated with ds:Reference URIs that access non local content, it is recommended to be mitigate risks associated with query parameters, unknown URI schemes, or attempts to access inappropriate content.

2.1.5 Example: Denial of service caused by too many transforms

XML Signature spec does not limit the number of transforms, and a malicious message could come in with 10,000 C14N transforms. C14N transforms involve lot of processing, and 10,000 transforms could starve all other messages.

Again the scope of this attack is also reduced by following the above best practices, as now an unauthenticated user would need to at first obtain a valid signing key and sign this SignedInfo section with 10,000 C14N transform.

This signature has a 1000 C14N and a 1000 XPath transforms, which makes it slow. This document has a 100 namespaces ns0 to ns99 and a 100 <e2> elements, like in the XPath denial of service example. Since XPath expands all the namespaces for each element, it means that there are 100x100 = 10,000 NamespaceNodes All of these are processed for every C14N and XPath transform, so total operations is 2000 x 10,000 = 20,000,000 operations. Note some C14N implementations do not expand all the Namespace nodes but do shortcuts for performance, to thwart that this example has an XPath before every C14N.

Example:

```
<Transform Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
  <XPath>1</XPath>
</Transform>
```

```
        <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315">

        <Transform Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
          <XPath>1</XPath>
        </Transform>
        <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315">

           ... repeated 1000 times
```

To totally eliminate this kind of attack, an implementation can choose to have an upper limit of the number of transforms in each Reference

Best Practice 7: Limit number of ds:Reference transforms allowed.

Too many transforms in a processing chain for a ds:Reference can produce a denial of service effect, consider limiting the number of transforms allowed in a transformation chain.

2.2 For Applications: Check what is signed

XML Signature offers many complex features, which can make it very difficult to keep track of what was really signed. When implementing XML Signature it is important to understand what is provided by a signature verification library, and whether additional steps are required to allow a user to see what is being verified. The examples below illustrate how an errant XSLT or XPath transform can change what was supposed to have been signed. So the application should inspect the signature and check all the references and the transforms, before accepting it. This is done much easier if the application sets up strict rules on what kinds of URI references and transforms are acceptable. Here are some sample rules.

- *For simple disjoint signatures:* Reference URI must use local ID reference, and only one transform - C14N
- *For simple enveloped signatures:* References URI must use local ID reference, and two transforms - Enveloped Signature and C14N, in that order
- *For signatures on base64 encoded binary content:* Reference URI must local ID references, and only one transform - Base64 decode.

These sample rules may need to be adjusted for the anticipated use. When used with web services WS-Security, for example, consider the STR Transform in place of a C14N transform, and with SWA Attachment, Attachment Content/Complete transform could be used in place of a base64 transform.

Sometimes ID references may not be acceptable, because the element to be signed may have a very closed schema, and adding an ID attributes would make it invalid. In that case the element should be identified with an XPath filter transform. Other choices are to use an XPath Filter 2 transform, or XPath in XPointer URI, but support for these are optional. However XPath expressions can be very complicated, so using an XPath makes it very hard for the application to know exactly what was signed, but again the application could put in a strict rule about the kind of XPath expressions that are allowed, for example:

- *For XPath expressions* The expression must be of the farm : ancestor-or-self:elementName. This expressions includes all elements whose name is elementName. Choosing a specific element by name and position requires a very complex XPath, and that would be too hard for the application to verify

Best Practice 8: Enable verifier to automate "see what is signed" functionality.

Enable the application to verify that what is signed is what was expected to be signed, by providing access to id and transform information.

2.2.1 Base Approval example

Consider an application which is processing approvals, and expects a message of the following format where the where the Approval is supposed to be signed

Example: Expected message for approval verification

```
<Doc>
  <Approval xml:id="ap" >...</Approval>
  <Signature>
     ...
       <Reference URI="ap"/>
    ...
  </Signature>
</Doc>
```

It is not sufficient for the application to check if there is a URI in the reference and that reference points to the Approval. Because there may be some transforms in that reference which modify what is really signed

2.2.2 Modified Approval Example: XPath transform that causes nothing to be selected for signing

In this case there is XPath transform, which evaluates to zero or false for every node, so it ends up selecting nothing. So even though the signature seems to sign the Approval, it actually doesn't. The application should reject this document.

Example: Insecure Approval verification message

```
<Doc>
  <Approval xml:id="ap">...</Approval>
  <Signature>
     ...
       <Reference URI="ap">
           <Transforms>
                           <Transform Algorithm="...XPath...">
                                  <XPath>0</XPath>
                                </Transform>
           </Transforms>     ...
       </Reference>
  </Signature>
```

```
</Doc>
```

2.2.3 Modified Approval Example: XSLT transform that causes nothing to be selected for signing

Similar to the previous example, this one uses an XSLT transform which takes the incoming document, ignores it, and emits a "<foo/>" . So the actual Approval isn't signed. Obviously this message needs to be rejected.

Example: Insecure Approval verification message

```
<Doc>
  <Approval xml:id="ap">...</Approval>
  <Signature>
     ...
       <Reference URI="ap">
           <Transforms>
              <Transform Algorithm="...xslt...">
              <xsl:stylesheet>
                    <xsl:template match="/">
                      <foo/>
                    </xsl:template>
                 </xsl:stylesheet>
              </Transform>
           </Transforms>     ...
       </Reference>
  </Signature>
</Doc>
```

2.2.4 Modified Approval Example: Wrapping attack

This one is a different kind of problem - a wrapping attack.There are no transforms here, but notice that Reference URI is not "ap" but "ap2". And "ap2" points to another <Approval> element that is squirreled away in an Object element. An Object element allows any content. The application will be fooled into thinking that the approval element is properly signed, it just checks the name of what the element that the Reference points to. It should check both the name and the position of the element.

Best Practice 9: When checking a reference URI, don't just check the name of the element

To mitigate attacks where the content that is present in the document is not what was actually signed due to various transformations, verifiers should check both the name and position of an element as part of signature verification.

Example: Insecure Approval verification message

```
<Doc>
  <Approval xml:id="ap">...</Approval>
  <Signature>
     ...
```

```
        <Reference URI="ap2"/>
     ...
     <Object>
       <Approval xml:id="ap2">...</Approval>
     </Object>
   </Signature>
 </Doc>
```

2.3 For Implementors: provide a mechanism to determine what was signed

As shown above, it is very hard for the application to know what was signed, especially if the signature uses complex XPath expressions to identify elements. When implementing XML Signature some environments may require a means to provide a means to be able to return what was signed when inspecting a signature. This is especially important when implementations allow references to content retrieved over the network, so that an application does not have to retrieve such references again. A second dereference raises the risk that that is obtained is not the same -- avoiding this guarantees receiving the same information originally used to validate the signature. This section discusses two approaches for this

2.3.1 Return pre digested data

While doing reference validation, the implementation needs to run through the transforms for each reference, the output of which is a byte array, and then digest this byte array. The implementation should provide a way to cache this byte array and return it tot he application. This would let the application know exactly what was considered for signing This is the only recommended approach for processors and applications that allow remote DTDs, as entity expansion during C14N may introduce another opportunity for a malicious party to supply different content between signature validation and an application's subsequent re-processing of the message.

2.3.2 Return pre C14N data

While the above mechanism let the application know exactly what was signed, it cannot be used by application to programmatically compare with what was expected to be signed. For programmatic comparison the application needs another byte array, and it is hard for the application to generate a byte array that will match byte for byte with the expected byte array.

Best Practice 10: Offer interfaces for application to learn what was signed.

Returning pre-digested data and pre-C14N data may help an application determine what was signed correctly.

A better but more complicated approach is to return the pre-C14N data as a nodeset. This should include all the transforms except the last C14N transform - the output of this should be nodeset. If there are multiple references in the signature,the result should be a union of these nodesets. The application can compare this nodeset with the expected nodeset. The expected nodeset should be a subset of the signed nodeset

DOM implementations usually provide a function to compare if two nodes are the same - in some DOM implementations just comparing pointers or references is sufficient to know if they are the same, DOM3 specifies a "isSameNode()" function for node comparison.

This approach only works for XML data, not for binary data. Also the transform list should follow these rules.

- The C14N transform should be last transform in the list. Note if there no C14N transform, an inclusive C14N is implicitly added
- There should be no transform which causes data to be converted to binary and then back to a nodeset. The reason is that this would cause the nodeset to be from a completely different document, which cannot be compared with the expected nodeset.

2.4 For Applications: prevent replay attacks

2.4.1 Sign what matters

By electing to only sign portions of a document this opens the potential for substitution attacks.

Best Practice 11: Unless impractical, sign all parts of the document.

Signing all parts of a document helps prevent substitution and wrapping attacks.

To give an example, consider the case where someone signed the action part of the request, but didn't include the user name part. In this case an attacker can easily take the signed request as is, and just change the user name and resubmit it. These Replay attacks are much easier when you are signing a small part of the document. To prevent replay attacks, it is recommended to include user names, keys, timestamps, etc into the signature.

A second example is a "wrapping attack" [McIntoshAustel] where additional XML content is added to change what is signed. An example is where only the amounts in a PurchaseOrder are signed rather than the entire purchase order.

2.4.2 Make Effective use of signing time and Nonces to protect against Replay Attacks

Best Practice 12: Use a nonce in combination with signing time

A nonce enables detection of duplicate signed items.

In many cases replay detection is provided as a part of application logic, often and a by product of normal processing. For example, if purchase orders are required to have a unique serial number, duplicates may be automatically discarded. In these cases, it is not strictly necessary for the security mechanisms to provide replay detection. However, since application logic may be unknown or change over time, providing replay detection is the safest policy.

Best Practice 13: Do not rely on application logic to prevent replay attacks

Supporting replay detection at the security processing layer removes a requirement for application designers to be concerned about this security issue and may prevent a risk if support for replay detection is removed from the application processing for various other reasons.

Nonces and passwords must fall under at least one signature to be effective. In addition, the signature should include at least a critical portion of the message payload, otherwise an attacker might be able to discard the dateTime and its signature without arousing suspicion.

Best Practice 14: Nonce and signing time must be signature protected.

A signature must include the nonce and signing time in the signature calculation for them to be effective, since otherwise an attacker could change them without detection.

Web Services Security [WSS] defines a <Timestamp> element which can contain a Created dateTime value and/or a Expires dateTime value. The Created value obviously represents an observation made. The expires value is more problematic, as it represents a policy choice which should belong to the receiver not the sender. Setting an expiration date on a Token may reflect how long the data is expected to be correct or how long the secret may remain uncompromised. However, the semantics of a signature "expiring" is not clear.

WSS provides for the use of a nonce in conjunction with hashed passwords, but not for general use with asymmetric or symmetric signatures.

WSS sets a limit of one <Timestamp> element per Security header, but their can be several signatures. In the typical case where all signatures are generated at about the same time, this is not a problem, but SOAP messages may pass through multiple intermediaries and be queued for a time, so this limitation could possibly create problems. In general Senders should ensure and receivers should assume that the <Timestamp> represents the first (oldest) signature. It is not clear how if at all a <Timestamp> relates to encrypted data.

2.5 Enable Long-Lived Signatures

2.5.1 Timestamp Authorities

Best Practice 15: Use Timestamp tokens issued by Timestamp authorities for long lived signatures

Such time-stamps prove that what was time-stamped actually existed at the time indicated, whereas any other time indication is only a claim by the signer and is less useful in dispute resolution.

The X.509 Public Key Infrastructure Time-Stamp Protocol, RFC 3161 [RFC3161], describes the use of a time stamp authority to establish evidence that a signature existed before a given time, useful in applications where dispute resolution may be necessary.

ETSI has produced TS 101 903: "XML Advanced Electronic Signatures [XAdES], which among other ones, deals with the issue of long-term electronic signatures. It has defined a standard way for incorporating time-stamps to XML signatures. In addition to the signature time-stamp, which should be generated soon after the generation of the signature, other time-stamps may be added to the signature structure protecting the validation material used by the verifier. Recurrent time-stamping (with stronger algorithms and keys) on all these items, i.e., the signature, the validation material and previous time-stamps counters the revocation of validation data and weaknesses of cryptographic algorithms and keys. RFC 3161 and OASIS DSS time-stamps may be incorporated in XAdES signatures.

OASIS DSS core specifies a XML format for time-stamps based in XML Sig. In addition DSS core and profiles allow the generation and verification of signatures, time-stamps, and time-stamped signatures by a centralized server.

The XAdES and DSS Timestamps should not be confused with WSS Timestamps. Although they are both called Timestamps, the WSS <Timestamp> is just a xsd:dateTime value added by the signer representing the claimed time of signing. XAdES and DSS Timestamps are full fledged signatures generated by a Time-stamp Authority (TSA) binding together a the digest of what is being time-stamped and a dateTime value. TSAs are trusted third parties which operate under certain rules on procedures, software and hardware including time accuracy ensurance mechanisms.

2.5.2 Include time of signing in Long-Lived Signatures

Best Practice 16: Long lived signatures should include a xsd:dateTime field to indicate the time of signing just as a handwritten signature does.

The time of signing is an important consideration for use of long-lived signatures and should be included.

Note that in the absence of a trusted time source, such a signing time should be viewed as indicating a minimum, but not a maximum age. This is because we assume that a time in the future would be noticed during processing. So if the time does not indicate when the signature was computed it at least indicates earliest time it might have been made available for processing.

It is considered desirable for ephemeral signature to be relatively recently signed and not to be replayed. The signing time is useful for either or both of these. The use for freshness is obvious. Signing time is not ideal for preventing replay, since depending on the granularity, duplicates are possible.

A better scheme is to use a nonce and a signing time The nonce is checked to see if it duplicates a previously presented value. The signing time allows receivers to limit how long nonces are retained (or how many are retained).

2.6 Signing XML without namespace information ("legacy XML")

Best Practice 17: When creating an enveloping signature over XML without namespace information, take steps to avoid having that content inherit the XML Signature namespace.

Avoid enveloped content from inheriting the XML Signature namespace by either inserting an empty default namespace declaration or by defining a namespace prefix for the Signature Namespace usage.

When creating an enveloping signature over XML without namespace information, it may inherit the XML Signature namespace of the Object element, which is not the intended behavior. There are two potential workarounds:

1. Insert an xmlns="" namespace definition in the legacy XML. However, this is not always practical.
2. Insulate it from the XML Signature namespace by defining a namespace prefix on the XML Signature (ex: "ds").

This was also discussed in the OASIS Digital Signature Services technical committee, see http://lists.oasis-open.org/archives/dss/200504/msg00048.html.

2.7 Avoiding default XML Schema values

Best Practice 18: Avoid use of default schema values or ensure that their values are always present in the instance document.

When an instance document is governed by a schema that makes use of default values there is a risk that signatures made over that instance will not verify. The reason is that the instance generator (and the signature process) will not include the default values but the recipient processing application, if parsing under the control of the schema, may fill in the defaults. The net result being that what is verified will not be what was signed.

2.8 Be aware of XML Schema Normalization

Best Practice 19: Avoid destructive schema validation operations before verifying signatures.

Applications relying on validation should either consider verifying signatures before schema validation, or select implementations that can avoid destructive DOM changes while validating.

Part of the validation process defined by XML Schema includes the "normalization" of lexical values in a document into a "schema normalized value" that allows schema type validation to occur against a predictable form.

Some implementations of validating parsers, particular early ones, often modified DOM information "in place" when performing this process. Unless the signer also performed a similar validation process on the input document, verification is likely to fail. Newer validating parsers generally include an option to disable type normalization, or take steps to avoid modifying the DOM, usually by storing normalized values internally alongside the original data.

Verifiers should be aware of the effects of their chosen parser and adjust the order of operations or parser options accordingly. Signers might also choose to operate on the normalized form of an XML instance when possible.

2.9 For Implementors: be aware of certificate encoding issues

Best Practice 20: Do not re-encode certificates, use DER when possible with the X509Certificate element.

Changing the encoding of a certificate can break the signature on the certificate if the encoding is not the same in each case. Using DER offers increased opportunity for interoperability.

Although X.509 certificates are meant to be encoded using DER before being signed, many implementations (particularly older ones) got various aspects of DER wrong, so that their certificates are encoded using BER, which is a less rigorous form of DER. Thus, following the X.509 specification to re-encode in DER before applying the signature check will invalidate the signature on the certificate.

In practice, X.509 implementations check the signature on certificates exactly as encoded, which means that they're verifying exactly the same data as the signer signed, and the signature will remain valid regardless of whether the signer and verifier agree on what constitutes a DER encoding. As a result, the safest course is to treat the certificate opaquely where possible and avoid any re-encoding steps that might invalidate the signature.

The `X509Certificate` element is generically defined to contain a base64-encoded certificate without regard to the underlying ASN.1 encoding used. However, experience has shown that interoperability issues are possible if encodings other than BER or DER are used, and use of other certificate encodings should be approached with caution. While some applications may not have flexibility in the certificates they must deal with, others might, and such applications may wish to consider further constraints on the encodings they allow.

3 Acknowledgments

This document records best practices related to XML Signature from a variety of sources, including the W3C Workshop on Next Steps for XML Signature and XML Encryption [XMLSecNextSteps]

4 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/

Gajek
: *Towards a Semantic of XML Signature: Position Paper for W3C Workshop on Next Steps for XML Signature and XML Encryption*, Sebastian Gajek, Lijun Liao, and Jörg Schwenk, 25-26 September 2007, http://www.w3.org/2007/xmlsec/ws/papers/07-gajek-rub/

McIntoshAustel
: *XML signature element wrapping attacks and countermeasures.* M. McIntosh and P. Austel. In Workshop on Secure Web Services, 2005.

RFC3161
: *Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)*, IETF RFC-3161, C. Adams, P. Cain, D. Pinkas, R. Zuccherato. August 2001. http://www.ietf.org/rfc/rfc3161.txt

WSS
: *Web Services Security v1.1*, OASIS Standard, February 2006. http://www.oasis-open.org/specs/index.php#wssv1.1

WebBug-Wikipedia
: *Web Bug - Wikipedia*, http://en.wikipedia.org/wiki/Web_bug

XAdES
: *ETS TS 101 903: "XML Advanced Electronic Signatures (XAdES)" v1.3.2 March 2006.*, http://webapp.etsi.org/workprogram/Report_WorkItem.asp?WKI_ID=21353

XMLDSIG
: *XML-Signature Syntax and Processing*, Second Edition, D. Eastlake, J. R., D. Solo, M. Bartel, J. Boyer , B. Fox , E. Simon. W3C Recommendation, 10 June 2008, http://www.w3.org/TR/xmldsig-core/.

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