XML Signature Best Practices

Editor's Draft 14 April 2008

$Revision: 1.1 $ $Date: 2008/08/22 18:17:26 $

This version:
: http://www.w3.org/TR/2008/WD-xmldsig-bestpractices-20080414/

Latest version:
: http://www.w3.org/TR/xmldsig-bestpractices/

Editors:
: Frederick Hirsch, Nokia
: Pratik Datta, Oracle

Copyright © 2008 W3C® (MIT, ERCIM, Keio), All Rights Reserved. W3C liability, trademark and document use rules apply.

Abstract

This Note serves to record Best Practices for XML Signature [XMLDSIG].

Status of this Document

This document is an editors' copy that has no official standing.

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/.

Table of Contents

1 Overview

This document outlines best practices for the use of XML Signature as noted by the W3C XML Security Specifications

Maintenance WG, the XML Security WG, and members of the W3C community that participated in the Workshop on Next Steps for XML Security [XMLSecNextSteps]. 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.

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.

Frederick Hirsch 8/22/08 2:37 PM
Deleted: ~~This Note outlines best practices noted by the XML Security Specifications Maintenance WG 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 mitigating attacks, some are for other issues - e.g. signing xml that doesn't use namespaces.~~

2 Best Practices

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

XML signature implementations are often 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, and it is not acceptable for a single poison message to bring down an entire set of web applications and services.

An implementation that literally follows the XML signature spec and performs the reference validation before the signature validation is extremely susceptible to denial of service attacks. As will be seen below, certain kinds of transforms require an enormous amount of processing time, certain retrieval method constructs lead to infinite loops and certain external URI references can lead to security violations. An implementation should first "authenticate" the signature, before running any of these dangerous operations. This will allow trust in the signing party to be assessed prior to *to performing potentially dangerous operations and possibly reducing the risks in processing the signature*.

Frederick Hirsch 8/22/08 2:24 PM
Deleted: at

Frederick Hirsch 8/22/08 2:24 PM
Deleted: ,

Frederick Hirsch 8/22/08 2:25 PM
Deleted: this

Frederick Hirsch 8/22/08 2:25 PM
Deleted: substantially reduce the attack surface, as it is lot less likely that an authenticated party will send a malicious message

Frederick Hirsch 8/22/08 2:39 PM
Deleted: [bhill: I don't feel this is an appropriate assumption for a security specification. I would rephrase: *"as trust in the responsible party can be assessed prior to performing potentially dangerous operations and a malicious message can be attributed to the responsible party."*]

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

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 still disallow these operations even in step 3*, if the party is not trusted to perform them*.

Frederick Hirsch 8/22/08 2:26 PM
Deleted: trusted

Frederick Hirsch 8/22/08 2:27 PM
Deleted: can

Best Practice 2: Take care when processing RetrievalMethod.

In step 1, if the verification key is not known beforehand and needs to be fetched from KeyInfo, the implementation should be very careful. The KeyInfo can have a RetrievalMethod, and this could have bad transforms, insecure external references and infinite loops (See examples below). RetrievalMethods do have some legitimate advantages though, for example when there are multiple signatures in the same document, these signatures can use a RetrievalMethod to avoid duplicate KeyInfo certificate entries. *An* implementation *that must handle potentially hostile messages*

Frederick Hirsch 8/22/08 2:27 PM
Deleted: So the

Frederick Hirsch 8/22/08 2:27 PM
Deleted: *A*

Frederick Hirsch 8/22/08 2:27 PM
Deleted: *which*

should choose to allow only very constrained RetrievalMethods - e.g. those that do not have any transforms, and only one level of indirection using a local URI.

Frederick Hirsch 8/22/08 2:27 PM
Deleted: may

Another potential security issue in step 1, is 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 implementation should at first validate the public key. If the KeyInfo is a X509Certificate element, the certificate needs to be validated. This involves verifying information in the certificate (for example, the expiration date, the purpose of the certificate, checking that it is not revoked, etc), and potentially building and validating a chain of certificates to a trusted certificate authority. See RFC 3280 for more information. If the KeyInfo is For an RSA or DSA KeyValue,-then there is no way to validate the key, so these should not be normally trusted; unless the keys has already been exchanged out of band, and the implementation only uses the KeyInfo to compare against the OOB exchanged key. Key Validation is typically more than an implementation issue, and often involves application specific information.

Best Practice 3: Establish trust in the verification/validation key.

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. But as mentioned before the scope of this denial of service attack is greatly reduced if the implementation follows the above best practices, because it is unlikely that an authenticated user would include this kind of transform. *As discussed previously,*

Frederick Hirsch 8/22/08 2:27 PM
Deleted:

XSLT transforms should only be processed for References, 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. To totally eliminate *these* kind*s* of attack an implementation can choose to not support XSLT at all or provide a mechanism to allow the application to optionally disable support for it.

Frederick Hirsch 8/22/08 2:49 PM
Deleted: this

Best Practice 4: Consider avoiding XSLT Transforms

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, *b*ut 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

Frederick Hirsch 8/22/08 3:03 PM
Deleted: . B

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>
```

To totally eliminate this kind of attack an implementation can choose to not support XPath Filter transform at all or provide a mechanism to allow the application to optionally disable support for 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 5: Try to avoid or limit XPath transforms

2.1.3 Example: Retrieval method that causes an infinite loop

The KeyInfo of a signature can have a RetrievalMethod, which can be used to reference a key somewhere else in the document. However there is nothing that prevents the RetrievalMethod from pointing back to itself directly or indirectly, forming a cyclic chain of references. RetrievalMethods have other problems too - they can include the above XPath or XSLT transform, and they can also have insecure external references, so RetrievalMethod should be avoided or constrained.

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

2.1.4 Example: Problematic external references

An XML Signature can use URIs to reference keys or to reference data to be signed. Same document references are fine, but external references to the file system *or* to other web sites can cause exceptions or cross site attacks. For example a message could have 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

Frederick Hirsch 8/22/08 2:40 PM
Deleted: The

Frederick Hirsch 8/22/08 2:40 PM
Deleted: message

Frederick Hirsch 8/22/08 2:40 PM
Deleted: s

Frederick Hirsch 8/22/08 2:41 PM
Deleted: http based

Frederick Hirsch 8/22/08 2:41 PM
Deleted: , because

Frederick Hirsch 8/22/08 2:41 PM
Deleted: which

Frederick Hirsch 8/22/08 2:41 PM
Deleted: some

Frederick Hirsch 8/22/08 2:41 PM
Deleted: n

Frederick Hirsch 8/22/08 2:41 PM
Deleted: hr

Frederick Hirsch 8/22/08 2:41 PM
Deleted: which

Frederick Hirsch 8/22/08 2:42 PM
Deleted: But

Frederick Hirsch 8/22/08 2:42 PM
Deleted: this

Frederick Hirsch 8/22/08 2:42 PM
Deleted: which

Frederick Hirsch 8/22/08 2:42 PM
Deleted: ,

Frederick Hirsch 8/22/08 2:43 PM
Formatted: Default Paragraph Font

Frederick Hirsch 8/22/08 2:43 PM
Formatted: Default Paragraph Font

Frederick Hirsch 8/22/08 2:43 PM
Deleted: xmlsec

Frederick Hirsch 8/22/08 2:43 PM
Deleted: will

Frederick Hirsch 8/22/08 2:43 PM
Deleted: try

Frederick Hirsch 8/22/08 2:43 PM
Deleted: and

effect of adding an extra holiday.

Implementations should 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.

Implementations should follow the recommendations in section 2.3 to provide cached references to the content as verified, 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, as with a "web bug".

Implementations that support XSLT transforms may further wish 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 Service uses a "cid:" URL for referencing data inside attachments, 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

Frederick Hirsch 8/22/08 2:44 PM
Deleted: [bhill: I am not sure if Cross-Site Request Forgery should strictly be in-scope for discussion of XML Signature best practices. External references can cause problems such as denial of service conditions if allowed to retrieve arbitrarily large amounts of data or from a slow-responding source. They can cause information disclosures in concert with other transforms - many XSLT implementations would allow valid XML data imported via the file:// method as discussed above to be leaked to an external URL. Finally, they can lead to time-of-check, time-of-use flaws in applications that do not use cached reference retrieval (as in 2.3) or rely on long-lived signatures (issues addressed by XAdES)]

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: dos_toomanytranforms.xml
```
     <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 Transforms Allowed.

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. For an application, it is completely meaningless to invoke a xmlsec library call to verify a signature, without knowing what the signature is really signing. 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.

There rules need to be modified slightly for WSSecurity, which adds some extra transforms - an STRTransform could be used in place of an C14N transform, and for SWA Attachment, Attachment Content/Complete transform could be used in place of base64

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:

Frederick Hirsch 8/22/08 2:29 PM
Deleted: xp

- *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: Check the reference URIs and transforms when verifying the signature

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.

Frederick Hirsch 8/22/08 2:50 PM
Deleted: which

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

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. An implementation should provide a mechanism to inspect a signature and return was signed. *This is especially important when implementations allow references to content retrieved over the network so that an application does not have to dereference such references again. A second dereference raises the risk that what 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

Frederick Hirsch 8/22/08 2:29 PM
Deleted: *if*

Frederick Hirsch 8/22/08 2:30 PM
Deleted: *which*

Frederick Hirsch 8/22/08 2:30 PM
Deleted: *would*

Frederick Hirsch 8/22/08 2:30 PM
Deleted: *otherwise*

Frederick Hirsch 8/22/08 2:31 PM
Deleted: *again, with no*

Frederick Hirsch 8/22/08 2:31 PM
Deleted: *as to*

Frederick Hirsch 8/22/08 2:50 PM
Deleted: s

Frederick Hirsch 8/22/08 2:32 PM
Deleted: it

Frederick Hirsch 8/22/08 2:32 PM
Deleted: s

should provide a way to cache this byte array and return it to the 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.*

Frederick Hirsch 8/22/08 2:32 PM
Deleted: *which*

2.3.2 Return pre c14n data

While the above mechanism lets 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 the expected byte array.

Frederick Hirsch 8/22/08 2:32 PM
Deleted: which

Frederick Hirsch 8/22/08 2:32 PM
Deleted: with

Frederick Hirsch 8/22/08 2:32 PM
Deleted: *expected*

A better but more complicated approach is to return the pre-c14n data as a nodeset. For this the implementation should run through all the transforms except the last c14n transform - the output of this should be nodeset. If there are multiple references in the signature, the implementation should compute a union of these nodesets and return them. 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 exptected nodeset.

Frederick Hirsch 8/22/08 2:51 PM
Deleted: Because

Frederick Hirsch 8/22/08 2:51 PM
Deleted: that

Frederick Hirsch 8/22/08 2:51 PM
Deleted: , and they cannot

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 10: Unless impractical, sign all parts of the document.

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

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).

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

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 since application may change.

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.

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.

2.4.3 Use Timestamps tokens issued by Timestamp authorities for long lived signatures

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 feldged 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. As such, time-stamps generated by well-operating TSAs are trusted time indications which prove that what was time-stamped actually existed at the time indicated, whereas any time indication inserted by the signatory is not more than a claim made by the generator of the signature.

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

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.

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.

WSS

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

XMLDSIG

XML Signature Syntax and Processing, Second Edition. 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

Frederick Hirsch 8/22/08 2:33 PM
Deleted: -

Frederick Hirsch 8/22/08 2:34 PM
Deleted: D. Eastlake, J. R., D. Solo, M. Bartel, J. Boyer , B. Fox , E. Simon

Frederick Hirsch 8/22/08 2:33 PM
Deleted: 12 February 2002

5 XML Signature Best Practices Change Log

Date	Author	Description
20080414	FJH	Created first draft based on material from Pratik Datta http://lists.w3.c maintwg/2008Apr/0007.html
20080415	FJH	Added material from Hal Lockhart http://lists.w3.org/Archives/Public/ maintwg/2008Apr/0018.html and change log. Correction regarding po Formatting fix ups.
20080519	FJH	Added material from Hal Lockhart http://lists.w3.org/Archives/Public/ maintwg/2008Apr/0018.html on timestamp nonce as agreed at last W http://www.w3.org/2008/05/06-xmlsec-minutes.html#item15 (**2.4.2 time and Nonces to protect against Replay Attacks**).
20080519	FJH	Added material from Sean Mullan on legacy XML, see http://lists.w3.o maintwg/2008Apr/0029.html, as accepted at WG meeting, see http:// minutes.html#item14 (**2.5 Signing XML without namespace inf**

		editorial modifications, also spell check on document. Added link targe
200805 19	FJH	Added updated material from Pratik Datta regarding transforms and d http://lists.w3.org/Archives/Public/public-xmlsec-maintwg/2008May/a meeting, see http://www.w3.org/2008/05/06-xmlsec-minutes.html#it **what is signed** and **2.1 For Implementors: Reduce the oppo attacks**). Included link to OASIS DSS discussion.
200806 07	PD	Reorganized the document wih information from the email exchanges see http://lists.w3.org/Archives/Public/public-xmlsec-maintwg/2008M http://lists.w3.org/Archives/Public/public-xmlsec-maintwg/2008May/ http://lists.w3.org/Archives/Public/public-xmlsec-maintwg/2008May/ http://lists.w3.org/Archives/Public/public-xmlsec-maintwg/2008May/ http://lists.w3.org/Archives/Public/public-xmlsec-maintwg/2008Jun/0 http://lists.w3.org/Archives/Public/public-xmlsec-maintwg/2008Jun/0 http://lists.w3.org/Archives/Public/public-xmlsec-maintwg/2008Jun/0
200806 09	PD	Cleared the confusion between DSS timestmap and WSS timestamp. Cl or "dateTime". Added a new section on DSS XAdes Timestamps.
200806 23	PD	Made the edits suggested by Sean Mullan http://lists.w3.org/Archives/ maintwg/2008Jun/0014.html and and Juan Carlos http://lists.w3.org/A maintwg/2008Jun/0019.html