The Curse of Namespaces in the Domain of XML Signature

Meiko Jensen
Meiko.Jensen@rub.de

Lijun Liao
Lijun.Liao@rub.de

Jörg Schwenk
Joerg.Schwenk@rub.de

Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany

ABSTRACT

The XML signature wrapping attack is one of the most discussed security issues of the Web Services security community during the last years. Until now, the issue has not been solved, and all countermeasure approaches proposed so far were shown to be insufficient.

In this paper, we present yet another way to perform signature wrapping attacks by using the *XML namespace injection* technique. We show that the interplay of XML Signature, XPath, and the XML namespace concept has severe flaws that can be exploited for an attack, and that XML namespaces in general pose real troubles to digital signatures in the XML domain. Additionally, we present and discuss some new approaches in countering the proposed attack vector.

Categories and Subject Descriptors

H.3.5.f [**XML/XSL/RDF**]: XML Namespaces and XPath; M.3.0.b [**Web Services Communication Protocols**]: WS-Security; M.13.0.a [**Security Concerns of Service-Oriented Solutions**]: digital signatures in SOAP messages

General Terms

Security, Standardization

Keywords

XML namespaces, signature wrapping, XML namespace injection, prefix-free canonicalization, XML Signature

1. INTRODUCTION

XML Wrapping attacks (also named XML Rewriting attacks in the literature) were described in [1], and this severe security problem still persists in security related XML standards. While the core problem is still unsolved, we present a new, sophisticated variant of this attack, which easily circumvents all proposed countermeasures, and sheds a light on the complexity of XML based security standards.

SWS'09, November 13, 2009, Chicago, Illinois, USA.

Our attack makes use of the namespace concept of XML, which is used to render element and attribute definitions unique. We exploit the fact that in the most popular canonicalization algorithm, *Exclusive Canonicalization*, only the prefix abbreviations are fixed by the XML Signature, whereas the namespace definition remains unprotected. The basic idea is to define a new (non-signed) element that replaces the signed element from the original document, where new prefixes are introduced to abbreviate the correct namespace(s). The abbreviations in the signed element remain unchanged, but will be bound to another namespace, and thus become invalid for the application logic.

To prevent this kind of attacks, namespace definitions must be included in the signed element. Unfortunately, the canonicalization algorithm that includes namespace definitions, *Inclusive Canonicalization*, is only defined optional. We thus describe a solution based on an extension of *Exclusive Canonicalization* (which hopefully is implemented), and several other options to avoid this kind of attacks. However, one should keep in mind general defenses against wrapping attacks, as described in [1, 2, 3, 4].

The paper is structured as follows. The next section introduces the important specifications and prerequisites for the paper's contents, Section 3 then provides previous and related work. After that, Section 4 describes the *namespace injection* technique and attack pattern, and Section 5 enumerates a list of countermeasure approaches and improvements. The paper concludes with future work in Section 6.

2. FOUNDATIONS

2.1 XML Namespaces

The main idea behind the introduction of the XML Namespaces specification [5] was to make XML element names globally unique so that merging issues with identical element names originating from different contexts could be prevented. The use of namespace uris for this purpose provides an unambiguous way to create unique identifiers by using appropriate domain names. Further, it enables a second useful functionality of lodging descriptive contents at the address specified by the url (but this is not required).

Additionally, the XML Namespace specification introduces an abbreviation mechanism for urls: the namespace *prefixes*. These short strings can be added to an element's tag name (separated by a colon) in order to have that element being bound to the namespace url the prefix refers to. Namespace prefixes are *not* required to be globally unique. Their only purpose is to bind XML elements to namespace uris.

Therefore, prior to using a namespace prefix in an XML document, it must be bound to a certain namespace uri. This is done by adding *namespace declarations*. A namespace declaration looks like `xmlns:`*prefix*`="`*uri*`"`, and binds the prefix to the given namespace uri for the whole subtree of this element. Alternatively, an element can be bound by using the `xmlns="`*uri*`"` notation.

Generally, the choice of where and how to place namespace bindings within an XML document is due to the XML creator. However, all of these approaches result in the very same XML semantics when being parsed, as the only information required is the namespace uri a certain XML element is bound to. Thus, the prefix concept provides XML with high flexibility in terms of namespace bindings, and XML parsers usually are required to be able to cope with all of them.

2.2 XML Signature

XML Signature [6] defines an XML syntax for digital signatures. It provides similar functionality as PKCS#7 [7], however is more extensible and tailored towards signing of XML documents. The basic structure is illustrated in Figure 1.

When generating an XML Signature, at first, each resource to be signed is transformed—if necessary—and canonicalized. Then, the digest value over the resulting data object is calculated and stored in the `DigestValue` of a `Reference`, along with the `Transforms` and `DigestMethod`. Further, the element `SignedInfo` with all of its child elements (cf. Figure 1) is created. Finally, the `SignatureValue` is calculated over the canonicalized `SignedInfo` using the algorithms specified in `SignatureMethod`.

When validating an XML Signature, each digest specified in `Reference` is verified by retrieving the corresponding resource information and applying the described transformations and the specified digest algorithm. The resulting value is compared to the content of the `DigestValue`; validation fails if any of them does not match. Then, the content of the `SignedInfo` is serialized using the specified canonicalization method, and the signature is verified using the specified algorithm.

2.3 Canonicalization

Digital signatures require the contents covered by the signature to be truly identical on signature application and verification, for else a digital signature gets invalid. However, in the case of digital signatures on XML documents, slight changes to the signed XML fragments are tolerable, as long as the document's contents remain identical to an XML parser. For example, the use of XML comments or the extra addition of whitespaces between an element's attributes are not of significance to an XML parser, thus any of such modifications are not required to immediately invalidate the signature.

In order to realize this flexibility, the XML Signature specification [6] introduced the concept of *canonicalization*. This is a set of operations to be performed on the signed XML contents prior to signature application or verification in order to hide irrelevant character-level modifications of the underlying XML document.

As it turned out, especially the canonicalization of namespace declarations is a real problem of the XML Signature specification. This can be determined by the fact that the

```
<Signature xmlns=".../xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod
        Algorithm=".../xml-c14n11"/>
    <SignatureMethod
        Algorithm=".../xmldsig#dsa-sha1"/>
    <Reference uri="#signedContent">
      <Transforms>
        <Transform
            Algorithm=".../xml-c14n11"/>
      </Transforms>
      <DigestMethod
          Algorithm=".../xmldsig#sha1"/>
      <DigestValue>g6EsdUK...</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>DHG2Q...</SignatureValue>
</Signature>
```

Figure 1: XML Signature Data Structure

XML Signature specification provides two different canonicalization algorithms to choose, each of them having some special parameters, benefits, and flaws. These two, called *Inclusive Canonicalization* [8] and *Exclusive Canonicalization* [9], are to be described next.

2.3.1 Inclusive Canonicalization (Inc-C14N)

The first, most simple approach in canonicalizing namespace declarations in the context of XML Signature consists in leaving all namespace declarations within the signed subtree in place, and only including all namespace declarations that were specified outside the signed subtree but also cover the signed subtree be included in the signed subtree's root element. Thus, if e.g. a namespace prefix x with namespace uri ns_x is defined within any ancestor of the signed subtree's root element, the *Inclusive Canonicalization* algorithm requires the declaration of x to be repeated within the root element of the signed subtree—regardless of whether it is used within the subtree or not.

However, this approach has lead to severe interoperability issues in the past, since every addition of namespace declarations e.g. at a SOAP `Envelope` element lead to invalidation of every XML Signature within that message, as the new namespace declaration is considered on signature verification, but was not present on signature calculation. Thus, the hash values differ, and the verification will result in a fault.

2.3.2 Exclusive Canonicalization (Exc-C14N)

In order to solve the interoperability issues of Inc-C14N, a new algorithm was included in the XML Signature specification, namely the *Exclusive Canonicalization*. It was intended to only contain those namespace declarations that really were required for parsing the signed contents, and to have only their namespace declarations become embedded in the signed subtree. Thus, if another namespace is introduced after signature generation, this won't affect the signature's validity any more.

However, when trying to identify the namespaces required within a certain subtree, Exc-C14N makes use of a certain, rather complex definition, called the *"visibly utilized"* namespaces of a subtree. This definition refers to the set of all namespaces that have at least one element or at-

```
<Transform Algorithm=".../REC-xpath-19991116">
   <XPath xmlns:soap="ns-soap">
      ancestor-or-self::soap:Body
   </XPath>
</Transform>
```

Figure 2: XPath transform example

```
<Transform Algorithm=".../xmldsig-filter2">
   <XPath Filter="intersect"
          xmlns="http://.../xmldsig-filter2"
          xmlns:soap="ns-soap">
      //soap:Body
   </XPath>
</Transform>
```

Figure 3: XPath Filter 2 transform example

tribute from that namespace occurring within the signed contents. Thus, if a namespace is visibly utilized, its namespace declaration—regardless of its position within the signed subtree—is kept in place. If that namespace happened to be declared outside of the signed subtree, its declaration is moved to the first element that visibly utilizes it.

2.4 ID, XPath, and XPath Filter 2

An important capability of the XML Signature specification is the possibility to have multiple digital signatures being applied to arbitrary parts of the very same XML document. However, this capability requires a reliable way to describe references to arbitrary subtrees or nodesets of the document. The XML Signature specification actually suggests three different approaches for describing such references, which may also be used in conjunction with each other. They are *ID-based referencing*, *XPath tranforms*, and *XPath Filter 2 transforms*.

The ID-based referencing is the easiest and most common referencing scheme, which relies on the use of the *fragment* part of a reference uri. For instance, a reference to the uri `"#n34"` within the `uri` attribute of a `Reference` element implies that the element with an `ID` attribute of value `n34` is targeted by that particular reference. Thus, the digital signature will cover the whole subtree that is rooted at the element of ID `n34`. Another valid approach is to set the `uri` attribute to `""` to reference a document's root element.

The XPath and XPath Filter 2 referencing schemes rely on the XPath language [10] for referring to arbitrary nodesets of an XML document. This *de-facto* standard in terms of XML referencing is used in the context of XML Signature by two specific *transforms* that can be applied to the ID-referenced subtree prior to signature value calculation.

The first transform, the original XPath transform, enables the signing party to specify a Boolean expression that must be evaluated against every DOM node of the referenced subtree. If that evaluation turns out to `true`, the corresponding DOM node is kept in the nodeset to be protected by the digital signature. Otherwise, the DOM node is excluded and thus will not be covered on signature value calculation. As an example, the XPath transform shown in Figure 2 will result in that the whole subtree rooted at the `Body` element of a SOAP message will be covered by the signature. For each descendant of `Body` (and the `Body` element itself), the evaluation of that XPath expression will result in `true`, whereas for any node of the SOAP header, it will result in a `false`, excluding them from the signed subtree—even if the `uri` reference attribute refers to the document root.

A major disadvantage of the original XPath transform consists in its complexity and high probability of unintended misconfigurations. Additionally, it turned out that its formal semantics drastically differed from most user's intuitive interpretations, as they expected a different processing approach—namely specifying an XPath that is evaluated against the root of the signed subtree only, then pointing to the subtree roots or nodesets to be signed. This—more intuitive—referencing scheme was later-on adapted in the specification, resulting in the XPath Filter 2 transform [11]. This transform, which is based on foundations of the set operations $\cup$, $\cap$, and $\setminus$, enabled a more intuitive referencing mechanism. For the example in Figure 3, the signature protects both the `Routing` header—including its descendants—and the whole SOAP body, without any necessity to specify two separate references or to create the very complex XPath expression required for achieving the same results with the original XPath transform. Thus, if ID-based referencing is not sufficient (see e.g. next section), the use of the XPath Filter 2 transform is the preferable choice.

2.5 XML Signature Wrapping Attacks

In 2005, McIntosh and Austel [1] first identified the threat of *Signature Wrapping Attacks*, also known as *XML rewriting attacks*. This XML-specific attack pattern misuses the referencing flexibility of XML Signature to trick the processing application so that arbitrary XML data of the attacker's choice is treated as if being signed by a legitimate user.

The key vulnerability exploited by the signature wrapping attack consists in that the XML processing in presence of XML Signatures usually is done twice: once for the validation of the digital signature, and once for the real application that uses the XML data (i.e. the Web Service application). The issue is that each of these two steps accesses the XML contents using a different approach. The XML Signature processing locates the `Signature` element in the SOAP header, then uses the reference IDs given to find the signed contents. The application parser instead uses tree-based navigation to locate the data it is interested in. Usually, both referencing schemes should end up at the very same XML contents, but in the signature wrapping case, the attacker moves the signed contents with its ID to another location, while placing its own contents at the original structure position.

An example for such a signature wrapping attack is given in Figures 4 and 5. Figure 4 shows the original SOAP message that was created and signed by a legitimate Web Service client, but was eavesdropped by the attacker. Then, the attacker modifies the SOAP message as shown in Figure 5. As can be seen, the signed contents now no longer reside on their intended structural position. Nevertheless, the XML Signature for the subtree of ID `signedContent` still remains valid. Thus, when the receiving side processes the new SOAP message altered by the attacker, it will firstly identify a valid digital signature on an element with ID `signedContent`, then process the contents at `/soap:Envelope/soap:Body` for the service application. This way, the attacker's operation is performed, using the legitimation of a valid user's valid signature.

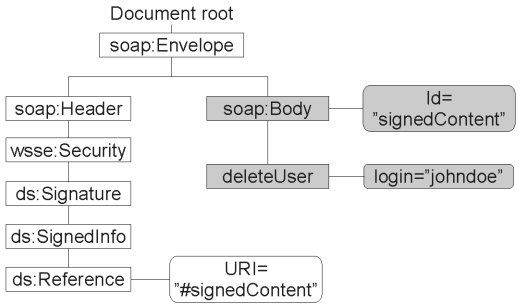

Figure 4: Signature Wrapping: Original SOAP message

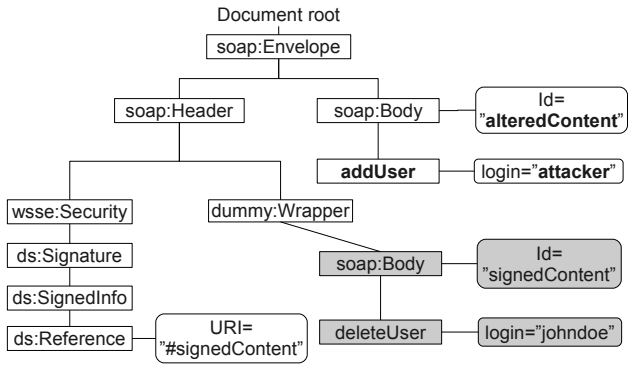

Figure 5: Signature Wrapping: Modified SOAP message

3. RELATED WORK

McIntosh and Austel [1] showed how to protect against certain wrapping attacks by improving the security policy to be followed by sender and receiver. On the other hand, they also show how to counterfeit each new security policy by a new, more sophisticated wrapping attack.

In addition, the complex security policies employed are not presented in XML syntax, thus they have to be hard-coded into the application. By doing so, one would lose all advantages of service-oriented architectures, because services can no longer be loosely coupled.

Rahaman, Schaad and Rits [12, 4, 13] proposed the inline approach for early detection of wrapping attacks. Our previous work [2] demonstrated that this approach is still vulnerable to wrappping attacks.

Sinha, Benameur, Kadir, and Fenet [14, 15] extended the inline approach by adding an element that contains the depth information of the signed object, keeps other information besides the name of the signed object's parent, or identifies the parent of the signed object by adding an `"Id"` attribute and keeping its value and the name of its parent. However, also these extensions do not eliminate signature wrapping attacks completely, as discussed in [3].

4. NAMESPACE INJECTION ATTACKS

While the XML Signature specification [6] suggests both canonicalization methods—Inc-C14N and Exc-C14N (cf. Section 2.3)—WS-Security [16] prefers the latter one. Even more, the renowned WS-I Basic Security Profile [17] explicitly disallows the use of Inc-C14N. Exc-C14N is more flexible; however, it introduces a severe security risk, as we will show next.

The basic vulnerability exploited for the XML namespace injection attack lies in the default behavior of canonicalization of namespace mappings. As a digital signature for XML covers all element nodes, text nodes, and attribute nodes of the XML document, a modification on any of these automatically would cause an invalidation of the signature. The only critical part of the XML structure that is not signed by default is the mapping of namespace prefixes to namespace urls[1]. This is due to the issue that a namespace prefix can be defined at an arbitrary ancestor node of the root element of the subtree to be protected by the signature.

[1]There are other XML components that are not signed by default, but these are usually not of relevance for the interpretation of an XML document's semantics

Thus, the W3C working group on XML Signature decided to cope with such namespace prefix mappings by using *canonicalization* (cf. Section 2.3). The main purpose of this prior-to-hash transformation is to embed all namespace prefix mappings into the hashed—and thus protected—part of the XML document. Unfortunately, there are may possible scenarios—e.g. due to misconfiguration or unawareness of the issue—where a namespace prefix mapping is *not* embedded correctly into the protected subtrees of the XML document. Thus, a modification to these unprotected namespace prefix mappings does not change the hashed element's syntactical appearance, and thus does not invalidate the digital signature.

ID-based referencing is most widely used to select the signed XML element. This is, however, vulnerable to signature wrapping attacks, because the position of the signed element is not protected by the signature. A more secure method is to apply XPath to fix the position, using XPath transforms or XPath Filter 2 (cf. Section 2.4). In both cases, the signed element's position is fixed by one or more XPath expressions (cf. Figures 2, 3). For clarity, most XPath expressions use prefixes instead of the represented namespaces to specify elements and attributes. These prefixes are defined in the ancestor elements of the text node that represents the XPath expression (e.g. in the `XPath` element).

As all XPath transforms or XPath Filter 2 transforms are declared within the `Reference` elements, which in turn are contained in the `SignedInfo` element, their canonical form is protected by the signature (cf. Section 2.2).

When Exc-C14N is applied, it embeds a prefix/namespace pair only if the prefix is used by the element tags or attributes within the subtree that is to be canonicalized (cf. Section 2.3.2). The prefixes contained in XPath expressions are not considered, since XPath expressions are considered only as normal text nodes. That means, an adversary can modify the namespace uri bound to a prefix without invalidating the signature—as long as the prefix used in the XPath expression is not used in any elements or attributes elsewhere within the `SignedInfo` subtree.

Consider the example in Figure 6. Exc-C14N is used to canonicalize the `SignedInfo` element. The signature intends to protect the element `ReplyTo` which is colored in gray. Apparently the signature covers the element located by the XPath `/soap:Envelope/soap:Header/wsse:Security/wsa:ReplyTo` whose prefixes are all defined within the el-

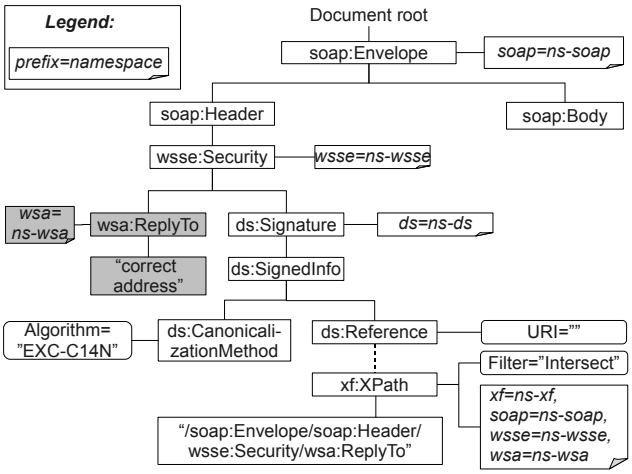

Figure 6: *Exclusive Canonicalization* without `InclusiveNamespaces`

ement `XPath`. However, the XPath expression is only a text node, hence the prefixes within it will not be automatically added to the canonicalization method.

As a result, the canonicalized `SignedInfo` subtree looks like shown in Figure 7. The namespaces for the prefixes *soap*, *wsse*, and *wsa* within the XPath expression are *not* protected by the signature.

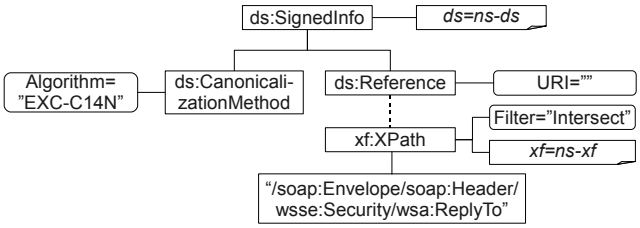

Figure 7: canonicalized `SignedInfo` from Figure 6

For the following explanation, we use `{ns}ElementName` to specify an element with local name `ElementName` and namespace uri `ns`.

To perform a rewriting attack, an adversary modifies the namespace associated with the unprotected prefix, adds a wrapper element with modified namespace, and moves the signed element to the wrapper element.

Figure 8 shows such an attack for the example given in Figure 6. The adversary changes the namespace uri associated with the prefix "wsse" from "ns-wsse" to "ns-attack" in the `xf:XPath` element. He then creates a new `{ns-attack}Security` element, and adds it as a sibling of the original `{ns-wsse}Security` element. Then, the `ReplyTo` element is modified, and the original version is moved from the `{ns-wsse}Security` element to the `{ns-attack}Security` element.

After receiving the SOAP message in Figure 8, the signature verification process initially searches for an XML Signature located at `/{ns-soap}Envelope/{ns-soap}Body/{ns-wsse}Security/{ns-ds}Signature`. Though the message contains two `Security` headers, note that only the one of namespace `ns-wsse` (and prefix `wsse`) matches the search

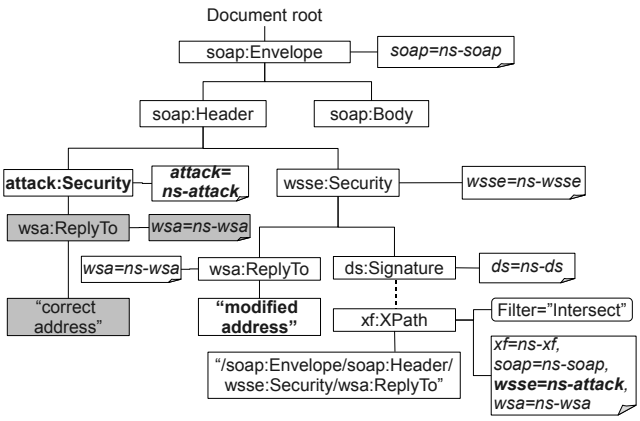

Figure 8: A wrapping attack on the SOAP message from Figure 6

criteria. Then, the signature validation extracts the contained XPath reference, and starts resolving all prefixes used within. This prefix resolution starts at the text node child of the `xf:XPath` element, thus the namespace bindings declared within that element are used. The resulting reference of the XPath expression `/soap:Envelope/soap:Body/wsse:Security/wsa:ReplyTo` then points to `/{ns-soap}Envelope/{ns-soap}Body/{ns-attack}Security/{ns-wsa}ReplyTo`, as the resolution of the `wsse` prefix within the XPath leads to a different binding than it had been on signature creation. Thus, the signature verification process follows that reference, ending up at the original `wsa:ReplyTo` subtree, even though it is no longer located at its original position. Its hash value, however, remains identical, and thus the signature verification succeeds.

Then, the application logic—here the WS-Addressing processing logic—accesses the SOAP message using the structural position of `/{ns-soap}Envelope/{ns-soap}Body/{ns-wsse}Security/{ns-wsa}ReplyTo`. Thus, it ends up at the malicious `ReplyTo` element inserted by the adversary, and falsely treats its contents as covered by the signature.

As can be seen, due to insufficient canonicalization, the modification of the `wsse` namespace binding is not detected on signature validation. Thus, it can be misused to change an XPath's target, which affects any kind of XPath-based referencing schemes, such as XPath transforms and XPath Filter 2 transforms.

5. ATTACK COUNTERMEASURES

The wrapping attacks described above can be avoided if all prefixes contained in the XPath expression are contained in the canonical form of the `SignedInfo` element. In general, there are three ways to achieve this goal. At first, prefixes in XPath expressions can be considered as in Inc-C14N. Secondly, prefix/namespace pairs can be specified explicitly in the wrapper element that contains the XPath expression, or thirdly, prefix-free XPath expressions can be used.

5.1 Explicitly Embedding Namespaces in Canonicalized XML

The prefixes used in the XPath expressions can be included in the `InclusiveNamespaces` element of `CanonicalizationMethod` within `SignedInfo`. This way, the namespace

declarations associated with such prefixes are processed as if using Inc-C14N. That means, all these prefixes are contained in the canonical form. For the example in Figure 6, the `CanonicalizationMethod` then looks as follows:

```
<ds:CanonicalizationMethod
    Algorithm=".../xml-exc-c14n#">
  <ec:InclusiveNamespaces
      PrefixList="soap wsse wsa"/>
  <ec:InclusiveNamespaces>
</ds:CanonicalizationMethod>
```

However, this requires that the canonicalization method and references must work together. A program should add the prefixes used in the XPath expression to the `InclusiveNamespaces` implicitly. However, as far as we know, this is not implemented in any open/available libraries for XML signatures, e.g. Apache XML Security [18] and Microsoft's .Net Framework [19]. Even worse, in some libraries (e.g. both libraries mentioned above), although it is possible to specify the `InclusiveNamespaces`, the specified prefixes will be ignored while canonicalizing the `SignedInfo`. In this case, if both signer and verifier ignore the `InclusiveNamespaces` element, the threat of namespace injection remains—if not becoming worse, as syntactically the attack vector seems to be disabled. Otherwise, if only one party correctly processes the `InclusiveNamespaces` element, the canonicalized XML representations differ, and thus the signature becomes invalid.

5.2 Defining Namespaces Explicitly

Another solution is to specify the prefix/namespace pairs explicitly in the wrapper that contains the XPath expressions. By doing so, the position fixed by the XPath expression is unique, independent from the environment. There are several ways to specify such pairs, either using attributes or as child element structures. As an example, consider the following:

```
<XPath xmlns="dummy">
  <PrefixNs>
    soap=ns-soap wsse=ns-wsse wsa=ns-wsa
  </PrefixNs>
  <XPathExpr>
    /soap:Envelope/soap:Header/wsse:Security/
    wsa:ReplyTo
  </XPathExpr>
</XPath>
```

Unfortunately, this solution cannot be applied to the existing XPath transform and XPath Filter 2 transform, because they are not extensible.

5.3 Using Prefix-free XPath Expressions

Actually, the most reliable countermeasure to the threats described above consists in placing the namespace uri directly into the XPath expression itself. This way, it is no longer necessary to resolve namespace prefixes contained in an XPath expression into their corresponding namespace uris. This also provides the benefit of that an XPath expression is no longer dependent from its surrounding XML document context (which usually provides the prefix-uri-mapping), but remains identical for arbitrary usage scenarios.

The XPath specification therefore provides some operators for accessing namespace uris in XPath, most importantly the `namespace-uri()` function, which actually returns the namespace uri of the evaluation context node (if not explicit nodeset parameter is provided). Thus, it is possible to use this function to create XPath expressions that are independent from the local namespace prefix resolution functionality. For example, the typical XPath expression `/soap:Envelope/soap:Body` with `soap` mapping to `ns-soap` can also be expressed like this:

```
/*[local-name()="Envelope" and namespace-uri()=
"ns-soap"]/*[local-name()="Body" and
namespace-uri()="ns-soap"]
```

As can be seen, that XPath expression does no longer contain any references to namespace prefixes, but ensures the referenced elements to belong to the correct namespaces. Please note that the use of the `local-name()` function is required here, as a test like

```
/Envelope[namespace-uri()="ns-soap"]
```

will not result in the expected matches. With that expression, the XPath parser would treat `/Envelope` like a reference to any child element of the document root that has local name `Envelope` and an empty namespace. As the SOAP message's `Envelope` element definitely has a namespace, the node test would fail already (irrespective of the `namespace-uri()` predicate, and the result of the XPath evaluation would be empty.

Using XPath expressions like this, the threat of namespace injection is countered, as the namespace resolution is no longer required to be identical for both XPath expression and referenced element. It solely depends on the referenced element to belong to the correct namespace, and the namespace declarations valid at the XPath expression can be neglected.

However, this solution obviously induces a serious rise in complexity of the XPath expression, and also raises the threat of misconfigurations. For instance, if the XPath developer accidentally uses `namespace-uri="..."` in one of the predicates, the XPath evaluation would treat this token no longer as a function call, but merely as a string statement. Thus, it would compare the string `"namespace-uri"` with the actual namespace uri string (which always returns false), and the XPath expression would select nothing. The threat here is that this does *not* induce a visible error neither in XPath processing nor in XML Signature processing. If, for instance, an XPath Filter 2 transform contains an `intersect` filter with such a faulty XPath expression, it will reduce the selected nodeset after transformation to become empty. Thus, the XML Signature implementation will calculate a hash value over the empty nodeset, and embed it as the `DigestValue` of the `Reference`. The XML Signature validator will do the same, resulting in identical hash values (over the empty nodeset), and thus it will approve the signature's validity, even though in fact none of the SOAP message's elements actually were protected against modification.

5.4 Excluding Prefixes

Another approach that is worth a thought consists in abandoning namespace prefixes completely from being covered by any digital signatures. Actually, the original purpose of namespace prefixes was to make it easier to define the namespace a certain XML element belongs to. The prefix merely was intended to act as a shortcut for circumventing the requirement to always repeat the long namespace urls. Thus, a prefix was not considered to be globally unique for a certain namespace nor was it thought to be of semantic

purpose for XML processing, apart from helping to bind elements to namespace uris.

5.4.1 Prefix Coverage in XML Parsers

Actually, neither the DOM nor the SAX XML processing libraries provide any means to easily access or modify the namespace prefixes or their declarations within an XML document. In fact, the SAX event `startElement`, for example, provides an application with an element's local name and namespace uri. Beyond these parameters, the SAX specification only list an extra parameter named `qname`, which *may* contain the full element's name (including the prefix and the `:` character), but are also allowed to be empty, depending on the actual SAX parser in duty. Thus, it is not always possible for any SAX-based application to determine namespace prefixes at all. The same applies to DOM parsers; as these are not required to provide namespace prefixes, and even do not provide any details on where in the document a prefix was defined.

Additionally, when it comes to re-serialization of a parsed and processed XML document, each serializer component is free to use any namespace prefixes and prefix definition styles it considers useful. It is allowed to declare all prefixes within the root element of the XML document, or to declare prefixes only where they are used for elements.

For the sole purpose of processing raw XML data, this liberal prefix usage does not cause any major difficulties, but for the specific context of digital signatures applied to XML documents, the impact of this usage flexibility is tremendous. For instance, consider a SOAP message that contains an XML Signature. Assume it to be parsed, processed, and re-serialized by any SOAP Intermediate, e.g. a company's gateway application. If the Intermediate decides to change any little detail on the namespace definitions within the SOAP message, it may invalidate the XML signature, even if the resulting XML document would be semantically identical to any other XML processing application. This is due to the issue that namespace prefix bindings are covered by the signature though they are not of semantic importance to any XML parser (as long as the elements are somehow linked to their correct namespaces).

5.4.2 Prefix-free Canonicalization (PFC14N)

In order to solve the issue outlined above, we suggest the use of a different approach in XML canonicalization, which is completely independent from the actual prefixes and namespace mapping approach used in an XML document. This canonicalization approach, which we called *Prefix-free Canonicalization*, relies on the plain use of prefix-free namespace declarations (`xmlns="uri"`) throughout the whole document.

The prefix-free canonicalization works as follows. Every element $\mathcal{X}$ with namespace $ns_{\mathcal{X}}$ and parent element $\mathcal{P}$ of namespace $ns_{\mathcal{P}}$ in the XML subtree to be canonicalized is transformed according to Algorithm 1.

Figures 9 and 10 show an example of an XML subtree before and after canonicalization of the root element `Envelope`. As can be seen, the resulting XML subtree still contains the very same element-namespace-bindings as the input did, but apart from the (few and predefined) prefixes required for proper namespace bindings of attributes, the resulting XML does not contain any prefixes. Thus, the outcome is completely independent from the namespace bind-

Algorithm 1 PFC14N($\mathcal{P}$, $\mathcal{X}$)

Require: Parent element $\mathcal{P}$, current element $\mathcal{X}$
Ensure: PFC14N-canonical form of $\mathcal{X}$

1: remove all prefix definitions and all namespace declarations from $\mathcal{X}$
2:
3: **if** $ns_{\mathcal{X}} \neq ns_{\mathcal{P}}$ **or** $\mathcal{P}$ is outside of the subtree to be canonicalized **then**
4: add the namespace declaration `xmlns="`$ns_{\mathcal{X}}$`"` to $\mathcal{X}$
5: **end if**
6:
7: **if** there are namespaces except $ns_{\mathcal{X}}$ associated with at least one attribute of $\mathcal{X}$ **then**
8: sort these namespaces in lexicographic order
9: $i \leftarrow 0$
10: **for each** namespace $ns_{\mathcal{A}}$ **do**
11: add the prefix definition `xmlns:ns`i`="`$ns_{\mathcal{A}}$`"` to $\mathcal{X}$
12: $i = i + 1$
13: **end for**
14: **end if**
15:
16: add the attributes in lexicographic order
17:
18: **return** $\mathcal{X}$

```
<soap:Envelope xmlns:soap="http://soap.ns"
               xmlns:legacy="http://legacy.ns"
               xmlns:ids="http://identity.ns">
  <soap:Body ids:ID="myBody">
    <my:operation xmlns:my="http://my.app.ns">
      <my:customer xmlns="http://my.app.ns">
        <name>John Doe</name>
        <cc:creditcard
             xmlns:cc="http://creditCard.ns">
          <cc:number>1234 5678</cc:number>
          <cc:validity>10/11</cc:validity>
        </cc:creditcard>
      </my:customer>
    </my:operation>
  </soap:Body>
</soap:Envelope>
```

Figure 9: Example input for PFC14N

```
<Envelope xmlns="http://soap.ns">
  <Body xmlns:ns0="http://identity.ns"
        ns0:ID="myBody">
    <operation xmlns="http://my.app.ns">
      <customer>
        <name>John Doe</name>
        <creditcard
             xmlns="http://creditCard.ns">
          <number>1234 5678</number>
          <validity>10/11</validity>
        </creditcard>
      </customer>
    </operation>
  </Body>
</Envelope>
```

Figure 10: Canonicalization output for PFC14N

ing approach utilized in the original XML document, even if it was processed and rewritten by any other Intermediate instance. Nevertheless, if this canonicalization approach is used prior to hash value calculation, an XML Signature applied to the document before Intermediate interference still remains valid after reserialization, even if the Intermediate applies changes to the namespace definition approaches.

5.4.3 Merits and Flaws of PFC14N

The obvious merit of PFC14N is that it allows modifications to an XML document on an application-unaware level of processing. Thus, the huge threat of interoperability failures caused by namespace processing issues can be resolved completely. Additionally, due to the low configuration requirements for this canonicalization approach—in comparison to Inc-C14N or Exc-C14N—the threat of misconfigurations causing vulnerabilities as described above can be mitigated by far.

On the other hand, at first glance, the prefix-free canonicalization seems to cause a lot of processing overhead, as every XML element has to be modified. This obviously causes a lot of overhead, which may affect processing performance. As we are not yet able to provide a full evaluation on the real performance implications, it is arguable if the merits outvalue the flaws here. Nevertheless, an argument on the real performance implications consists in that the namespace uri for each XML element of the document is attached to the element's parser representation besides any canonicalization already. Thus, the canonicalization requires few more than using this data on hash value calculations. Thus, that task can be optimized by integrating it into the hash calculation step.

Other benefits of this canonicalization approach cover its streamability (PFC14N can be done in a one-time pass on the XML document on event-based parsing) and reduction on the canonicalized message's size (the approach removes all prefixes and prefix declarations), but to our consideration these are not the major competing factors for a canonicalization algorithm. Please also note that the PFC14N algorithm already contains a slight optimization in that it only includes an element's namespace declaration if its parent's namespace differs. Thus, if all descendants of a certain element belong to the same namespace, this namespace is declared only once.

6. CONCLUSION AND FUTURE WORK

In this paper, we have shown that the up-to-date processing of XML namespaces in the domain of XML Signatures has severe flaws, causing a vulnerability to XML signature wrapping attacks, and a general threat of accidental signature invalidation. We have described the problem of *namespace injection*, and exemplified an attack scenario based on this technique. Further, we discussed several countermeasure approaches to this threat. To resume, it must be stated that XML Signature is not secure today, and that the flexibility given in defining XML namespaces poses yet another severe threat to its proper application and reliability. Future work consists in developing appropriate countermeasures and supervising the standardization and adaptation of these countermeasures in real-world scenarios.

7. REFERENCES

[1] M. McIntosh and P. Austel, "Xml signature element wrapping attacks and countermeasures," in *SWS*, 2005, pp. 20–27.

[2] S. Gajek, L. Liao, and J. Schwenk, "Breaking and fixing the inline approach," in *SWS*, 2007, pp. 37–43.

[3] S. Gajek, M. Jensen, L. Liao, and J. Schwenk, "Analysis of signature wrapping attacks and countermeasures," in *IEEE International Conference on Web Services (ICWS 2009)*. IEEE CS, July 2009.

[4] M. A. Rahaman, R. Marten, and A. Schaad, "An inline approach for secure SOAP requests and early validation," OWASP AppSec Europe, 2006.

[5] T. Bray, D. Hollander, A. Layman, and R. Tobin, *Namespaces in XML 1.0 (Second Edition)*, W3C Std., 2006.

[6] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon, *XML-Signature Syntax and Processing (Second Edition)*, W3C Recommendation, June 2008.

[7] B. Kaliski, *PKCS#7: Cryptographic Message Syntax Standard, Version 1.5*, IETF RFC 2315, Mar. 1998.

[8] J. Boyer, *Canonical XML Version 1.0*, W3C Recommendation, Mar. 2001.

[9] J. Boyer, D. Eastlake, and J. Reagle, *Exclusive XML Canonicalization, Version 1.0*, W3C Recommendation, July 2002.

[10] Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simon, *XML Path Language (XPath), Version 2.0*, W3C Recommendation, 2007.

[11] J. Boyer, M. Hughes, and J. Reagle, *XML-Signature XPath Filter 2.0*, W3C Recommendation, Nov. 2002.

[12] M. A. Rahaman, A. Schaad, and M. Rits, "Towards secure SOAP message exchange in a soa," in *Workshop on Secure Web Services*, 2006.

[13] M. A. Rahaman and A. Schaad, "SOAP-based secure conversation and collaboration," in *IEEE International Conference on Web Services (ICWS 2007)*. IEEE CS, 2007, pp. 471–480.

[14] A. Benameur, F. A. Kadir, and S. Fenet, "XML Rewriting Attacks: Existing Solutions and their Limitations," in *IADIS Applied Computing 2008*, 2008.

[15] S. K. Sinha and A. Benameur, "A formal solution to rewriting attacks on SOAP messages," in *SWS*, 2008, pp. 53–60.

[16] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker, *Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)*, OASIS Std., 2006.

[17] M. McIntosh, M. Gudgin, K. S. Morrison, and A. Barbir, "Basic security profile version 1.0," *Web Services Interoperability Organization Deliverables*, 2007.

[18] (2007, Mar.) The apache xml security java version. [Online]. Available: http://santuario.apache.org/Java/index.html

[19] (2009, May) Microsoft .net framework. [Online]. Available: http://www.microsoft.com/NET/