Master Thesis

Automatic Penetration Test Tool for Detection of XML Signature Wrapping Attacks in Web Services

Ruhr-Universität Bochum

Christian Mainka

22. May 2012

Lehrstuhl für Netz- und Datensicherheit
Ruhr-Universität Bochum
Universitätsstr. 150
D-44789 Bochum

Adviser:	M. Sc. Juraj Somorovsky Lehrstuhl für Netz- und Datensicherheit, Ruhr-Universität Bochum
Supervision:	Prof. Dr.-Ing. Jörg Schwenk Lehrstuhl für Netz- und Datensicherheit, Ruhr-Universität Bochum
	Prof. Dr.-Ing. Christof Paar Lehrstuhl Embedded Security, Ruhr-Universität Bochum

Abstract

SOAP based web services are a widespread technology for executing remote operations and transmitting structured data. They can be found in Service Oriented Architectures (SOAs) as Cloud interfaces, federated identity management, e-Government and military services. For ensuring integrity and authenticity, the XML Signature standards describes how to sign and verify document parts. However, the processing is very complex and it was shown that an attacker can modify the message without invalidating the signature. The so called XML Signature Wrapping (XSW) attack is one of the most discussed security issues in the web services community. Its practical relevance is evidenced by the attacks on the Amazon EC2 SOAP and the Eucalyptus Cloud web service interfaces.

This thesis presents a tool which unites all known XSW variants and is able to automatically create wrapping messages. It is integrated into the web services penetration testing framework WS-Attacker and can be used for easily attacking SOAP based web services.

Contents

1 Introduction

The use of Service Oriented Architectures (SOAs) and Cloud computing has been dramatically increased in the last few years. A fundamental technology for them are web services, which use XML based SOAP requests for executing remote operations. The area of application reaches from public Cloud interfaces up to management of federated identities, e-Government and military services. Due to the wide adoption of the SOAP technology, numerous extension specifications – mostly complex – have been released. For achieving data integrity and authenticity, the XML Signature standard can be applied. However, in 2005, McIntosh and Austel have found a fundamental weakness named XML Signature Wrapping (XSW) (1): In most applications, the method for detecting the signed elements by the signature verification logic is not identical to the method for detecting the payload used by the application logic. This difference allows an attacker to move the signed parts of a document to another location without invalidating the signature. The basic problem is that the signature itself does not protect the position of the signed element. The practical impact of the XSW attack was shown in (2), wherein the authors have successfully attacked the Amazon EC2 SOAP and the Eucalyptus Cloud web service interfaces.

The XSW attack is one of the most discussed security issues in the web services community. Since then, a lot of researches have been published to secure XML Signatures. Most of them are policy based or add non-standard compliant elements to enforce the position of the signed element (3, 4, 5). All these countermeasures have in common, that they try to secure signatures which detect the signed element by an ID attribute.

A different approach for selecting the signed element is using XPath expressions in combination with XPathFilter. The basic idea is to describe the location of the signed element by a string and thus protect its position within the XML document. The XPath grammar is very powerful and offers a lot of more or less complex functions for referencing the signed element. Some specific XPath expressions are vulnerable to XSW as well (6), so in general, the usage of an XPath expression does not prevent XSW attacks.

Gajek et al. have analyzed the whole XPath grammar and worked out a reduced one (7). The so called FastXPath grammar has a fast evaluation with respect to security. Nevertheless, even this referencing method could be successfully attacked by the namespace injection attack (8). Therefore, the authors abuses weaknesses in the interplay of XML Signature, XPath, XML Canonicalization and namespaces, so that a used namespace prefix will be resolved to two different URIs resulting in different element resolution for application- and verification logic.

As this shows, attacking XML Signatures by the XSW attack is very complex due to the large number of attack variants. In most cases, a web service developer does not have the time and the expertise to manually test his implementation for such weaknesses. In some cases, it is even not his own fault, because the relying framework has an implementation issue.

Motivated by the enormous number of techniques for attacking and detecting XSW and its huge complexity, this thesis will present a tool, which can automatically create and attack SOAP based web services which use XML Signature. It is implemented as an attack plugin for the web services penetration test framework WS-Attacker (9). The goal is to create a tool which is easy to use – for web service security experts and none-experts – and which covers all known attack scenarios. It is not meant to be a *one click wonder*, but rather a helper instrument for getting a quick overview and inspecting a web service implementation. The feasibility of the approach is proved by attacking three web services which use XML Signatures: Apache Axis2 which uses the standard security module Rampart, the XSpRES library and the IBM DataPower XI50. Therefore, the framework user just needs an eavesdropped signed message as input and the tool will ask him for the payload, update the included timestamps automatically, create and send the XSW messages to the server and inspect the answer if the attack was successful.

The thesis is structured and organized as follows. The next section gives an overview of the needed fundamentals for understanding SOAP based web services and XML Signature. Section 3 introduces the XSW attack and deals with its variants and countermeasures. Building on that, Section 4 presents generic algorithms for creating XSW messages out of arbitrary XML messages. The implementation details and the integration in the WS-Attacker framework are shown in Section 5 and evaluated in Section 6. The thesis concludes with future work in Section 7.

2 Foundations

In this section, the foundations for understanding the web service technology and the XML Signature standard are explained.

2.1 XML

The eXtended Markup Language (XML) defines a set of rules for encoding documents and data structures (10). A very basic example for this gives Listing 1.

```xml
<computer>
    <hardware>
        <keyboard layout="german"/>
        <monitor resolution="1920x1080"/>
    </hardware>
    <software>
        <os type="Linux">Ubuntu</os>
        <editor>vim</editor>
    </software>
</computer>
```

Listing 1: First XML example.

As one can see, there are different types of rules formatting XML documents. An element, e.g. `keyboard`, can have additional attributes, e.g. the `layout` attribute. The value `Ubuntu` is a special kind of attribute – the *text content*. It is always enclosed by the opening and closing element tag.

For a better readability, this thesis will use the tree notation for representing XML documents, as Figure 1 shows.

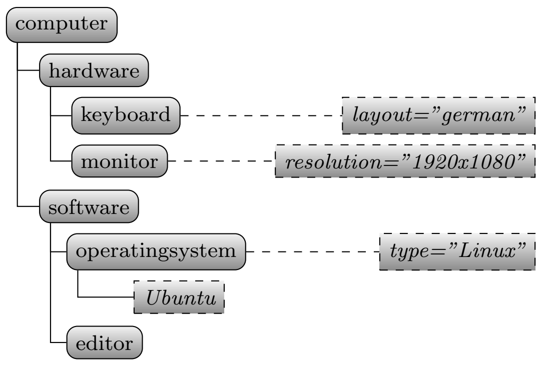


Figure 1: XML tree representation. Attributes are placed horizontally with a dashed rectangle border. Text contents have the same border but are placed below the corresponding element.

Elements have a solid, rounded border. Attributes are placed horizontally to the belonging element and have a dashed rectangle border. Text contents are placed like normal elements but with the same border as an attribute.

An additional feature of XML are namespaces (11, 12). They are used to clearly identify elements and attributes. An example document is shown in Figure 2.

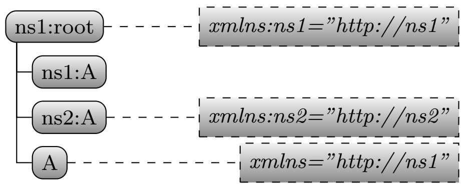


Figure 2: An example for XML namespaces.

The `root` element belongs to the namespace `http://ns1`, which is bound to the prefix `ns1`. It has three child elements with the same localname `A`. Nevertheless, only two of them are semantically the same. The first `A` element has the prefix `ns1`, but does not need to declare the namespace URI it belongs to, because it is inherited by its parent. The second `A` element uses its own namespace `ns2` which is bound to the URI `http://ns2`, so it is different to the first `A` element. The third one is special – it does not use any namespace prefix. However, it uses the same namespace URI as the first `A` element, so they are semantically the same.

The main benefit of using namespaces is that XML documents can embed other XML documents without conflicting with their element names. They can be distinguished by their namespace URI.

Note that the example descriptions in this thesis commonly left out namespaces for simplicity, e.g. instead of talking about `soap:Envelope`, the element is just named `Envelope` if the namespace is obvious.

2.1.1 XML Parser

For parsing XML documents, there are the following possibilities:

- ▷ The Document Object Model (DOM) reads the whole XML document into memory and creates an object for each node. A node can be the element itself, an attribute, the text content or a comment. They are all saved in a tree-like order, which allows easy access to each node and relationship information like child, parent and sibling nodes.
- ▷ A streaming based parser does not save any object into memory. It parses an XML stream and handles events if a new node starts or ends. This is extremely fast but more difficult to use, since the programmer can not go backwards, e.g. to get the parent node. While processing, he has to save all relevant data himself, e.g. ancestor or sibling nodes, and he can not directly access to other nodes. There are two different types of streaming based parsers:
  1. The Simple API for XML (SAX) parser reads a stream and sends out events if, e.g., a new element starts or ends. Thus, SAX is a *push* parser.
  2. The Streaming API for XML (StAX) parser works like a cursor: the programmer can ask for the next event, so StAX is a *pull* parser, which does not interact with the program actively.

Both models are widely used. At first glance, DOM seems to be the better parser type, as it is easier to use. Nevertheless, it is notable that parsing only a small document will result in much higher memory consumption. For a server, which has to process several XML documents per second, this can lead to a bottleneck in memory, being abused by denial of service attacks. However, the streaming based model also has its disadvantages. Although it is fast and consumes minimal memory, some XML specific operations, like XPath (see Section 2.5), are not fully supported.

2.2 XML Schema

XML Schema is a recommendation by the World Wide Web Consortium (W3C) for describing the structure of an XML document (13). Basically, it is a set of rules which can describe the structure for each contained element. This includes its allowed attributes, the type of its value (e.g. a string or integer), a description of its allowed child elements and how often they may occur.

```xml
<xs:schema xmlns:xs="http://..." xmlns:tns="http://...">

  <xs:element name="computer" type="tns:computer"/>
  <xs:complexType name="computer">
    <xs:sequence>
      <xs:element ref="tns:hardware" minOccurs="1"/>
      <xs:element ref="tns:software" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="hardware" type="tns:hardware"/>
  <xs:complexType name="hardware">
    <xs:sequence>
      <xs:any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="software" type="tns:software"/>
  <xs:complexType name="software">
    <xs:sequence>
      <xs:any namespace="##other" processContents="strict" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Listing 2: An XML Schema example.

Listing 2 gives an example of an XML Schema. It describes a `computer` element with two child elements. The `hardware` element must occur exactly once. This is defined by the attribute `minOccurs="1"`. The attribute `maxOccurs` is omitted, as its default value is "1". The computer element can also have a software element. It is optional because of the attribute minOccurs="0".

The elements hardware and software have special child elements. They are notated as xs:any, which means, that any kind of child elements are allowed. Their properties can be constrained by defining the following attributes:

- ▷ The namespace attribute defines its allowed namespace. Possible values are ##any (default value) and ##other to force a different one than its parent's namespace.
- ▷ The processContents attribute controls the behavior of the XML Schema handling for the specified child element. The default value is strict and means that the element must have an available XML Schema. If it is set to lax, the parser will lookup if an XML Schema is present and use it in this case. Otherwise, the element is not parsed, but this is not treated as an error.

The main idea of the xs:any element is to define an extension point for additional plugin elements. This way, the XML Schema can be expanded by just adding another Schema file to the parser. Whenever it finds an element at an xs:any position, the parser will lookup if it matches to the newly added Schema.

The downside of this approach is that xs:any elements can be used to place wrapper elements for moving signed parts, see Section 3.

2.3 SOAP based Web Services

A web service is the concrete implementation of a Service Oriented Architecture (SOA). Instead of defining how a services works, a web service just defines its interface. SOAP is a standard for describing message exchanges with a web service (14, 15). Although it is not bound to a specific protocol, commonly HTTP is used.

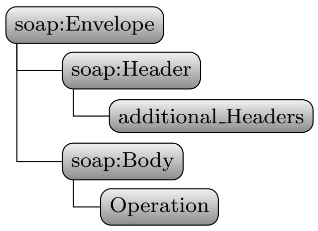


Figure 3: SOAP example. The root Envelope element has an optional Header element and a mandatory Body element.

Figure 3 shows a basic SOAP request. It has a root element named `soap:Envelope` which has an optional `soap:Header` and a mandatory `soap:Body` child element. The `Header` element is used to store meta information not meant to be used by the concrete application logic, e.g. a signature or some routing information. The `Body` element holds the payload. In most web service implementations, the first body child element is named as the operation to be performed. The available operations and the necessary structure of the request is stored in the Web Services Description Language (WSDL) (16, 17). Such a request can be seen as a kind of remote procedure call – the message holds the name of the operation to be executed and all necessary parameters. After the web service server has executed the operation, it sends the result as a SOAP response back to the invoker. If any kind of error occurs, the response contains a `soap:Error`.

2.4 XML Security

SOAP messages are commonly exchanged using HTTP. Thus, these messages can be easily protected using SSL/TLS (18). Unfortunately, this approach has some disadvantages.

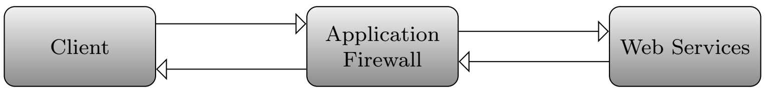


Figure 4: Security via SSL/TLS. A client wants to execute a web service operation which is protected by an application firewall.

Suppose the scenario of Figure 4. A client wants to invoke a web service operation which is protected by an application firewall. For doing this, the client has to establish an SSL/TLS connection with the firewall. Afterwards, the firewall has the following possibilities:

1. Sending the message to the web service using an unsecured connection.
2. Establishing a new SSL/TLS connection with the web service.

Obviously, both methods have its downside. Therefore, the OASIS group has maintained the WS-Security (19) standard, which describes how to:

- ▷ Sign and verify (parts of) an XML document (XML Signature, see Section 2.6).
- ▷ Encrypt and decrypt (parts of) an XML document (XML Encryption (20)).

▷ Add security tokens, e.g. timestamps or credentials, to an XML document.

For using such features, a WS-SecurityPolicy is needed (21). It is based on the WS-Policy standard (22) and describes which security features are required, to which document parts they shall be applied and which concrete algorithms must be used.

2.5 XPath

XPath is a query language for selecting nodes of an XML document (23, 24). An example for a valid XPath gives the Figure 5.

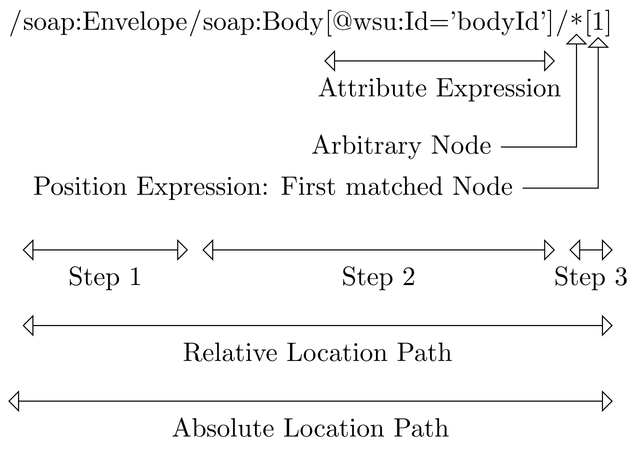


Figure 5: Example for an XPath expression.

The expression describes a node selection beginning with the document root (*absolute location path*), which is indicated by the first slash. Without this slash, the expression would be a *relative location path.* The next classification is a *step*, which can consist of a node name with or without an arbitrary number of expressions in square brackets. The most commonly used expressions select a node if it has an attribute with a specific value, e.g. the attribute `wsu:Id` with the value `bodyId` as shown in step 2, or if it has a specific position. The expression `[1]`, see step 3, is an abbreviation for `[index()=1]`, which means to select only the first matching node. The XPath standard offers a lot of functions for selecting a node by any kind of property – e.g. by its localname, its namespace or even if it has some specific ancestor or descendant node.

2.5.1 The XPath *descendant*-Axis

A special kind of selecting a node is using the `descendant-or-self` axis, abbreviated by a double-slash:

```
/root//*[@task='done']
```

Listing 3: Using an XPath expression with the `descendant-or-self` axis.

This expression would select any *descendant-or-self* node of the `root` element in the XML document which has an attribute named `task` with the value `done`. Note that the asterisk only selects any kind of direct child nodes, but not its descendants, e.g. /root/*[@task='done'] would only select the direct child nodes of the `root` element with the specified attribute. Furthermore, there is also a function for selecting any *descendant* node, so that the current (*self*) context node is not allowed. To summarize both methods, this thesis uses the notation *descendant-**.

2.5.2 FastXPath

A subset of the XPath grammar is FastXPath (7). It only uses simple forward axis – so no `descendant` or `ancestor` selection is possible. Additionally, FastXPath requires each step to have an attribute or position expression. Using this grammar, a very fast XPath evaluation is possible. As a side effect, FastXPath expressions are very resistant against some signature attacks. A detailed description of this grammar is given in Section 3.4.

2.5.3 PreXPath and PostXPath

For explaining the algorithms in Section 4, the definition of a PreXPath and a PostXPath must be introduced. Suppose the XPath /A/B/C/D/E/F. Then, step D as a part of the whole expression has the following properties:

- ▷ Its **PreXPath** is /A/B/C. It is defined as the concatenation of all steps before the specified step. If the step D is appended to this expression, it is called the **extended PreXPath** (/A/B/C/D).
- ▷ Its **PostXPath** is E/F. Note that this expression does not start with a slash – it is a relative location path.

2.6 XML Signature

2.6.1 Structure and Workflow

XML Signature is a recommendation by the W3C which defines a syntax for using digital signatures in an XML document (25, 26). It can be used for ensuring integrity and proofing authenticity of fragments or even the whole document. The basic structure of an XML Signature element can be seen in Figure 6.

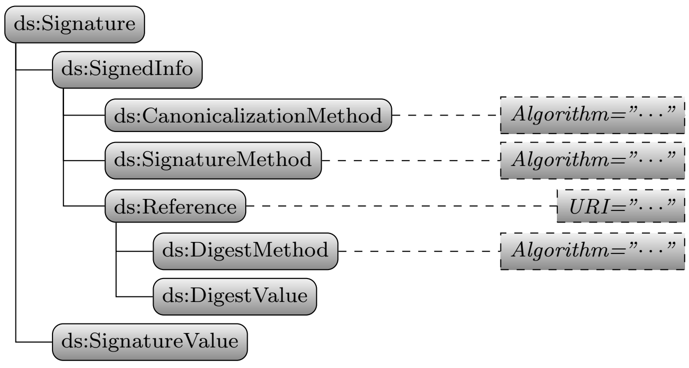


Figure 6: Structure of the XML Signature element.

The signing process undertakes the following flow: For each XML fragment to be signed, a `Reference` element is created and the `DigestValue` of the element specified by the `URI` attribute is computed using the algorithm specified in the `DigestMethod` element. Afterwards, the `SignedInfo` element is signed using the algorithm specified in the `SignatureMethod` element.

One problem by signing XML documents is that a signature would become invalid if the document structure changes without changing its semantics, e.g. by adding white-spaces between attribute definitions. For solving such problems, the element to be signed/hashed is normalized using an XML Canonicalization Algorithm as described in Section 2.6.2.

For embedding an XML Signature into a SOAP message, the `Signature` element is placed as a child of a WS-Security header. An example for such a so-called *detached signature* gives Figure 7.

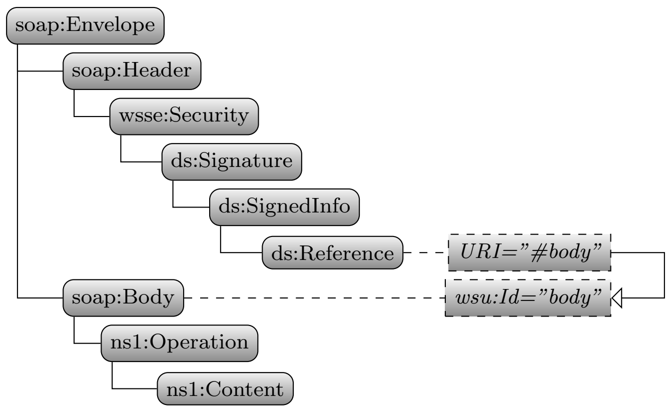


Figure 7: Signed SOAP message. Structure of the `Signature` element is simplified.

2.6.2 XML Canonicalization

The idea of an XML Signature is to ensure integrity and proof authenticity of a message. In general purposes, this means that the message must be exactly the same for sender and recipient. Nevertheless, in the context of XML, it just means that a message should be semantically the same.

```
<x attribute1="one" attribute2="two" />
<x attribute2='two' attribute1='one'          />
```

Listing 4: Two semantically same elements.

The example Listing 4 shows two elements named `x` which are semantically identical: Both elements have the same attributes with the same values, but the order is different, they use different quotation signs and the latter one has additional white-spaces at the end. Standard digest algorithms which compute their values by using the string representation as input lead to different hash values and thus to an invalid signature.

XML Canonicalization is a recommendation by the W3C which normalizes a document (27, 28). Semantically identical XML fragments result in the same string representation so that standard digest algorithms can be applied. Basically, the XML Canonicalization method does the following tasks:

- ▷ Sort attributes alphabetically.
- ▷ Normalize line breaks (Windows, UNIX,...), quotation signs, ...

▷ Add default attributes.

However, for handling namespaces, there exist two different methods:

▷ The Inclusive Canonicalization method adds any namespace declarations which can be seen by an element to its attributes. This also includes declarations of prefixes which are not used in any child elements. A problem of this approach is that XML Signatures can become invalid if a namespace declaration is added to any parent of a signed fragment, e.g. the root element. This causes the Inclusive Canonicalization to add the namespace declaration to each signed element. Thus, hashing them results in different values leading to an invalid signature.

▷ The Exclusive Canonicalization method only adds namespace declarations to an element if it is used for the first time. Namespace declarations in child elements, which use a prefix declared in an ancestor element, are removed. This behavior is called *visible utilized*.

For each of these two methods, there exist two sub-methods: One that omits comments, and one that does not, so that there are four canonicalization methods in total.

2.6.3 XPathFilter

Commonly, signed elements are referenced by an ID attribute. Therefore, the signature logic searches for an element anywhere in the XML document which has the specified attribute and value. Then, the element is added to the list of elements which are hashed. Another approach for referring elements is using XPath expressions, see Section 2.5. Currently, there exist two specifications for using XPaths in XML Signatures: XPathFilter1 and XPathFilter2 (29, 30).

The first version has a bad performance: It allows using an XPath expression as a boolean filter. The expression must be evaluated against every node in the referenced XML fragment. If the node fulfills the condition, the node is added to the list of objects which are used for calculating the digest value.

In contrast to this, XPathFilter2 has a different working flow: The XPath expression starts at the element defined by the `URI` in the `Reference` element and then evaluated the specified XPath. All matched elements will be used for calculating the digest value. Multiple XPaths can be combined using the set operations *intersect*, *union* and *subtract*.

Note that a common scenario for this is to use `URI=""`, which means to let the XPath begin at the document root.

An example for using XPathFilter2 gives Figure 8.

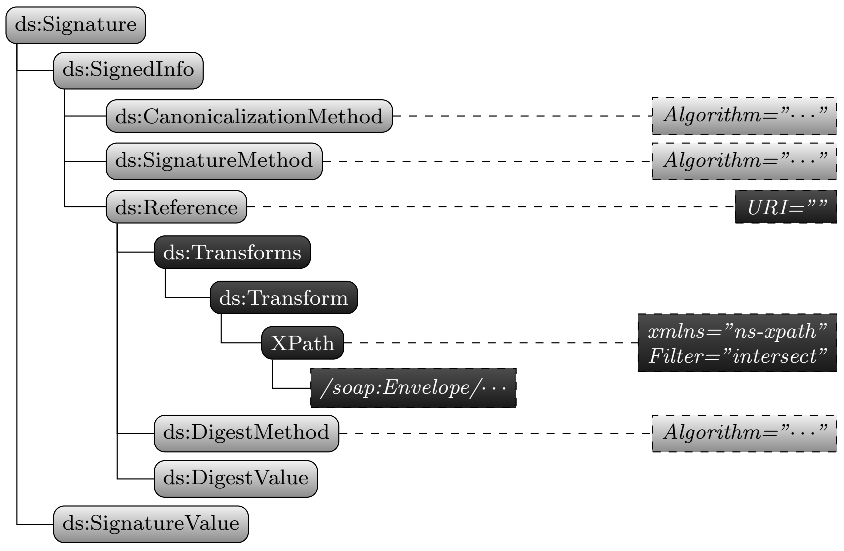


Figure 8: An XML Signature using XPathFilter2.

The structure is very similar to the use of ID attributes, but instead of selecting an element by an attribute value, the URI only selects the start point for the XPath expression. For this purpose, the `URI` attribute must be set to the empty string. Additionally, an `XPath` element is added as a transformation to the `Reference` element and its text-content holds the concrete XPath expression. The set operation – *intersect* in this case – is defined by its `filter` attribute.

2.6.4 Handling Multiple Elements to Sign

In most cases, there is the need to protect multiple elements in an XML document. A very common example for this is protecting the message's payload and a timestamp. XML Signature offers different possibilities for realizing this:

1. The XML document can have multiple `Signature` elements. Note that this allows an attacker to create a new message by mixing two old ones. It is also a bit slower than the next purpose, as the verification of each `Signature` element needs an expensive asymmetric operation.

2. The `Signature` element can have multiple `Reference` elements. The benefit of this is that the `SignatureValue` protects all referenced elements together. So, there is a connection between all signed elements.
3. Use XPathFilter2: Only one `Signature` element with exactly one `Reference` element is necessary. For each element to be signed, an XPath is added to the list of transformations. The first one uses `filter="intersect"`, all other ones use the `union` filter.

Note that in some scenarios, approaches two and three might be impossible, e.g. if there is an operation chain where different instances have to sign parts of the XML document. In this case, mixture of the first one with the other ones can be used.

2.6.5 XML Signature Verification in Software

The previous section clarifies that the XML Signature standard is very powerful and complex. Because of this, most common web services do not implement the standard itself, but rather use external libraries for the signature creation and verification. This approach allows to separate security- and application logic, but the downside of this is that the developer of the web service has to take care, that both, the security- and the application logic, use the same element for its processing. At first view, this seems to be absolutely logically, but remember the referencing method used by an ID based XML Signature. It detects the signed element by searching for any element which has a specified attribute value. For the application logic, this attribute needs not to be important. The developer could just process the first child of the `Body` element, as he expects the signed element to be there. Obviously, this is not enforced by the signature verification logic.

3 Attacking XML Signatures

McIntosh and Austel published the first basic scenario for attacking XML Signatures in 2005 (1). They named the attack XML Signature Wrapping (XSW), because the basic idea is to move the original signed element to a wrapper element on a different position and replace it with a new payload. Since then, more and more different techniques for attacking and protecting XML Signatures have been published and will be presented in this section.

3.1 Attacking ID based XML Signatures

The most frequently used scenario for XML Signature is to refer the signed elements by an ID attribute. This method is easy to understand for humans and simple to implement for developers. However, it has the big disadvantage that the signature itself only protects the content of the signed elements but not its location within the document. The signed element can be moved to another location – vertically and horizontally in the document structure – without invalidating the signature.

Figure 9 gives an example for constructing an XSW message which still bypasses the signature verification process but changes the payload used by the application logic.

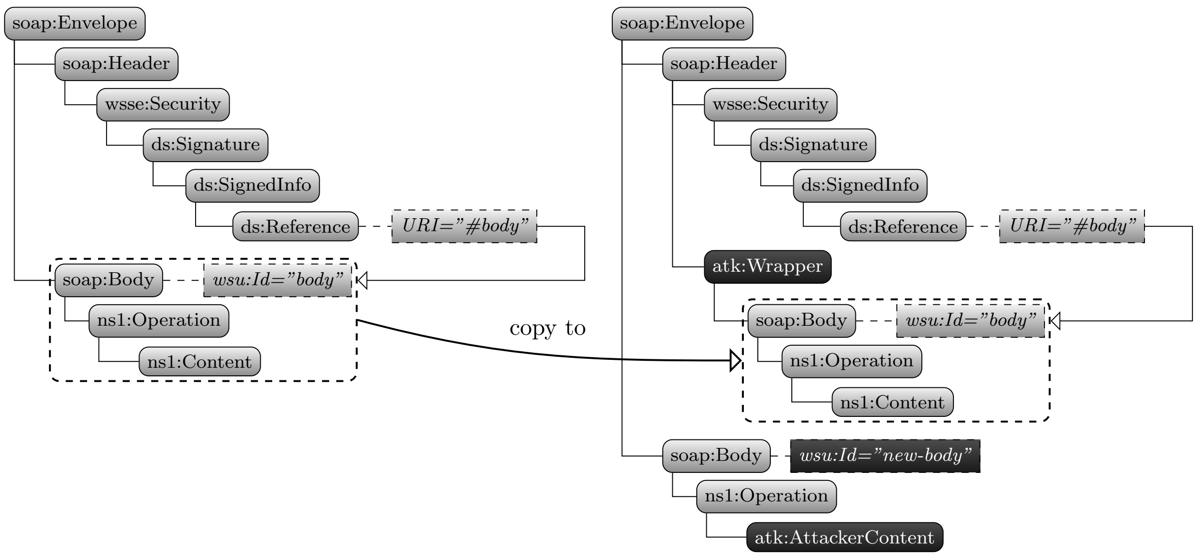


Figure 9: Creating an XSW message for an ID referencing based XML Signature. The original signed message is shown on the left side. The XSW message on the right side is constructed by copying the signed element to a `Wrapper` element and modifying the signed content to the attackers' needs.

The original message has a signed `Body` element which is referenced by the ID attribute `#body`. The attack message on the right has a new `Wrapper` element placed as a child of the `Header` element. Its child is a copy of the original signed `Body` element. Note that the ID attribute still has the value `#body`. The original `Content` element is replaced by a new `AttackerContent` element and the ID attribute of its ancestor `Body` element is changed to `new-body`, so that the signature verification logic will not use it. There might also be other attack scenarios in which the attribute value remains the same as in the `Wrapper` element or it removed completely. The success of this attack depends on the implemented application- and verification logic.

The main problem why this attack works is that the signature verification- and the application logic use different methods for detecting their elements. The signature verification logic looks for an element which has the attribute `wsu:Id="body"` and uses it to compute the digest value. The application logic, in contrast, does not care about the attribute `wsu:Id="body"` – it just uses the first child element of the `Body` element in the SOAP message. Obviously, these referencing methods are not equivalent, as the example attack message shows.

3.2 Countermeasures for ID based XML Signatures

Since the discovery of XSW, several countermeasures for preventing this attack have been proposed. The authors of the original paper themselves suggested a policy based approach to prevent it. This can be summarized summarized to the following rules (namespaces are omitted):

- ▷ The `Signature` element must be present in the `Security` header.
- ▷ The `Reference` element must be a child of the `Signature` element and refer to the `Body` element which itself must be a child of the `Envelope` root element.
- ▷ The position of the `Timestamp` element must be /Envelope/Header/Security/ ↷ Timestamp.
- ▷ The signature verification key must be provided by an X.509 certificate and the CA must be confidential.

Nevertheless, Gruschka and Lo Iocano showed that these proposed checks are not sufficient to prevent XSW (31).

Another WS-Policy based approach was created by Bhargavan et. al (4). The authors developed a policy adviser and proofed its theoretical correctness in a wide class of WS-Security protocols, but although the Amazon EC2[1] Cloud fulfilled these security requirements, it could be successfully attacked (2). This example shows that a theoretically secure policy is not sufficient for getting real-world security. There might be a gap between the formal WS-SecurityPolicy and its concrete implementation. Another problem with this approach is that very strong restrictions to the security policy are made. Many elements are claimed to be signed and the whole `Body` must be signed in every case. This reduces the flexibility of the SOAP security mechanism.

Rahaman et. al used a different approach for securing XML Signatures (5). Instead of forcing the elements to be signed by a WS-SecurityPolicy, the authors introduced a way to embed the elements' position directly in the XML document. The goal of this so-called *inline approach* is to fix the signed elements position so that each movement within the document results in an invalid signature verification. This is realized by including a `SoapAccount` element for each referenced element in the XML Signature, which holds the following information:

- ▷ The number of child elements of the SOAP root element `Envelope`.

[1]`https://aws.amazon.com/de/ec2/`

- ▷ The number of child elements of the `Header` element.
- ▷ The number of references in each XML Signature.
- ▷ The successors and predecessors of each signed object.

Although the idea of fixing the position seems to be plausible, this solution has some disadvantages: At first, the `SoapAccount` element is not standardized, thus, the resulting XML Signature is not standards-compliant. Secondly, the saved properties do not prevent XSW in general. An attacker could modify the message structure while fulfilling the secured account information of the inline approach as showed in (32).

3.3 Attacking XPath Based XML Signatures

XML Signatures can use XPath expressions for referencing signed elements. The advantage is that this method allows to describe the position of the signed element within the XML document. Liao et. al introduced two notations for this (6):

- ▷ The **hashed subtree** contains the nodes which are the input for the digest algorithm. For simplicity, this can be seen as the signed element[2].
- ▷ The **protected subtree** contains the hashed subtree and other nodes which are necessary to locate it, e.g. ancestor elements or attributes.

Suppose the XPath /soap:Envelope/soap:Header/ns:Signed. Figure 10 then shows the difference of the protected and the hashed subtree nodes for an example SOAP message.

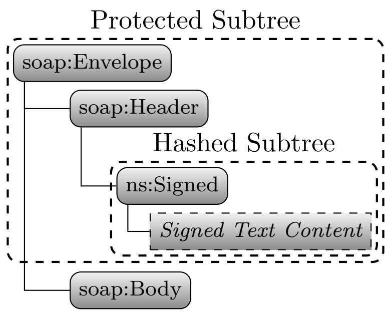


Figure 10: Visualization of the hashed- and the protected subtree.

[2]Note that one XPath can also match multiple elements, attributes or text contents.

In the example, the location of the `Signature` element is omitted. Remembering the structure of an XML Signature and the usage of XPathFilter2 as described in Section 2.6.3, the XPath expression is located in the `Reference` element. This implies, that the expression is protected by the signature itself. Thus, if an attacker tries to build an XSW message, the structure of the protected subtree must remain.

Although using XPaths seems to be able to protect the position of a signed object, so that XSW attacks might be detected, it must be mentioned that using them is not a countermeasure in general. There are some possibilities for attacking specific XPath expressions.

An example for this is the *descendant* axis specification. It selects a succeeding element which can be located anywhere below the current context node.

Consider the XPath expression //soap:Body`[1]`. It queries any `Body` element which is a descendant of the document root. In this case, for selecting nodes by the descendant axis, the abbreviated form // (*double-slash*) is used. This is equivalent to the XPath expression /descendant−or−self::node()/soap:Body`[1]`. Note, the XPath begins with the descendant axis selection, but it is also possible to use it in the middle of an expression, e.g. /soap:Envelope//soap:Body`[1]`

Assume the example message given by Figure 11 and the XPath expression //soap: ↷ Body`[1]` as mentioned before.

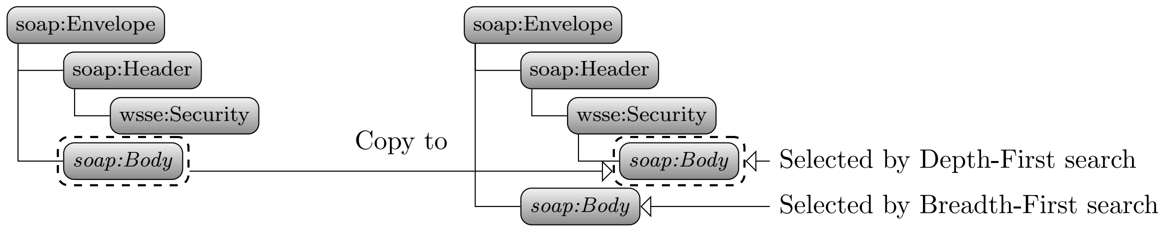


Figure 11: Comparing Depth-First and Breadth-First search: Which `Body` element is selected with the XPath //soap:Body`[1]`?

With regard to semantics, the XPath selects the first `Body` element in the XML document. However, an attacker can copy the original `Body` element and place it as a child of the `Header` element. It now depends on the XPath implementation which element is selected. If the implementation uses a Depth-First search, the element located at /soap:Envelope/soap:Header/soap:Body is selected. In case of a Breadth-First search, the other one is used. This illustrates the ambiguousness of the descendants axis. Commonly, the Depth-First search is used[3], which allows an attacker to create an XSW message as shown on the right side of Figure 11.

In some scenarios, it is also possible to attack XPaths which do not contain a *descendant-** axis specifier. Consider the example given by Figure 12.

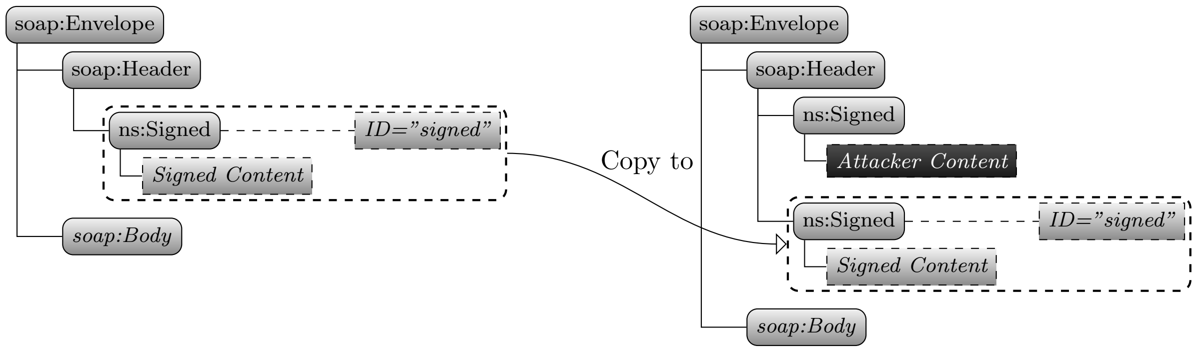


Figure 12: Creating an XSW message which is signed by using the XPath /soap:Envelope/soap:Header/ns:Signed`[@ID='signed']`.

The signed content is selected by the XPath /soap:Envelope/soap:Header/ns:Signed `[@ID='signed']`. Note that the signed element `Signed` has the attribute `ID='signed'`, which is used by the XML Signature verification logic. If the application logic ignores this attribute and just selects the first child element named `Signed` of the `Header` element, an attacker could construct the XSW message as shown on the right side. Therefore, he places a copy of the signed content after the original content. Afterwards, he removes the `ID` attribute from the original `Signed` element and places his payload in it.

Further examples use a mixture of both attack scenarios. A common example for this is an XPath with the following structure:

```
//*[@ID='signed']
```

This expression does semantically the same as selecting the signed element by only using its attribute – this is equivalent to an ID based referencing signature as shown in (7). An attacker therefore has to process both attacks:

[3]Tested with Java6 and the javax.xml.xpath.XPath package.

1. First, he has to find another location within the XML document which also matches the descendant axis specifier.
2. Second, he has to adjust the attributes in the signed and the wrapper elements.

A more detailed description is given in Section 4.

3.4 Countermeasures for XPath Based XML Signatures

As the previous section shows, the use of XPath as referencing method does not bring any security in general. The leakage of ID based referencing is that it does not protect the position of the signed element in the XML document. An XPath expression seems to have the possibilities to do this, but this does not affect the whole XPath grammar. Especially the case of transforming an ID based reference into an XPath expression clarifies this. Gajek et al. analyzed the XPath grammar in points of security and efficiency. Their results, the so-called FastXPath grammar, has the following structure:

```
FastXPath           ::= '/' RelativeFastXPath
RelativeFastXPath   ::= Step
                      | RelativeFastXPath '/'Step
Step                ::= QName PredicatePosition?
PredicatePosition   ::= Position Predicate?
                      | Predicate Position?
Position            ::= '[' [1-9][0-9]* ']'
Predicate           ::= '[' PredicateExpr ']'
PredicateExpr       ::= PredicateStep
                      | PredicateExpr 'and' PredicateStep
PredicateStep       ::= '@' QName '=' Literal
Literal             ::= '"' [^"]* '"'
                      | "'" [^']* "'"
```

Listing 5: The FastXPath grammar.

Listing 6 gives an example FastXPath expression, which selects the first `Body` child named `Operation`, as this is used in most implementations by the application logic.

```
/soap:Envelope[1]/soap:Body[1]/ns:Operation[1]
```

Listing 6: An example FastXPath expression.

The FastXPath uses only direct forward axis selection, which allows to use a SAX/StAX parser to parse the XML document. It is faster than a DOM parser and also requires less RAM. By using this subset of the XPath grammar, the security is also improved, as the FastXPath grammar explicitly denies the usage of the *descendant* axis.

Nevertheless, the selection of an element according to its attribute value is still allowed. In cases where the application logic ignores such attributes, the same attacks as shown in Section 3.3 are possible.

3.5 Namespace Injection Attack

XPath expressions can select a node by using its namespace prefix, e.g. /**soap**: ↷ Envelope/**soap**:Body/**ns**:Operation. However, this concept itself does not contain any information about the mapping of a prefix to its concrete URI. Jensen et al. developed a technique named *namespace injection attack*, which overrides namespace declarations in such a way, that the signature verification- and the application logic resolve one and the same prefix into different namespaces URIs (8).

Consider the SOAP message given in Figure 13.

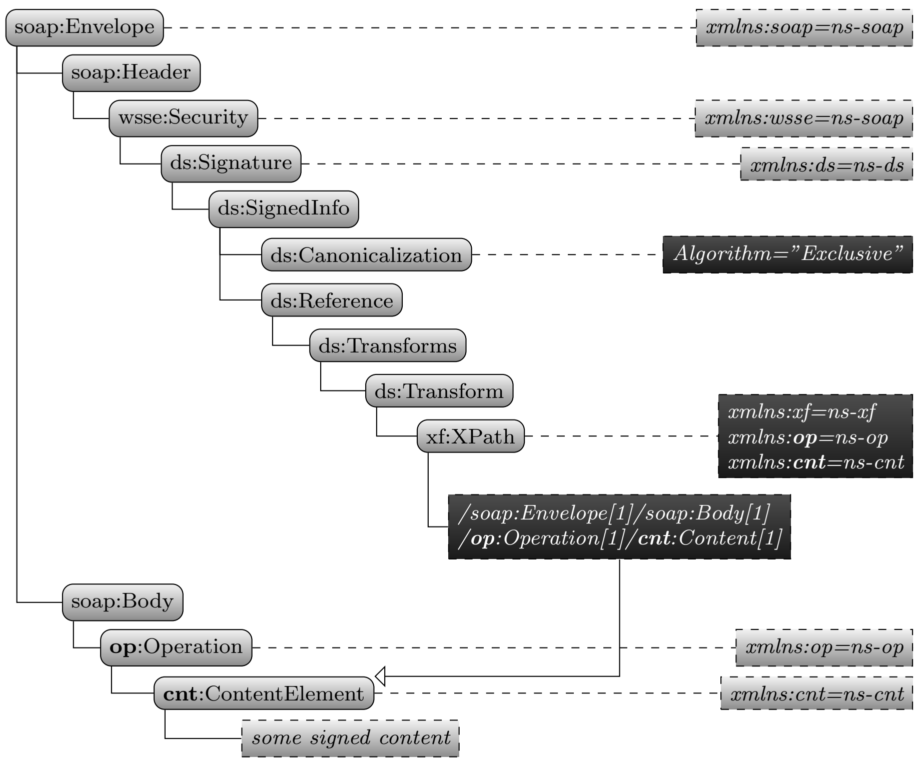


Figure 13: Prerequisites for namespace injection.

It shows a basic SOAP request which uses a FastXPath to selected the `cnt:ContentElement` located in the `soap:Body` part. The XPath itself uses the three prefixes *soap*, *op* and *cnt*. If the signature verification logic tries to detect the hashed subtree, it has to resolve those prefixes. Therefore, it looks for a fitting namespace declaration in the `xf:XPath` element and uses it if it is found. Otherwise, its ancestors are traveled upwards and it is searched for a matching declaration. Afterwards, the hashed subtree and the `ds:SignedInfo` element are canonicalized and the hash value on them is computed. Note that only the canonicalized form of the element is hashed, thus, only this form is protected by the XML Signature. In the case of Exclusive Canonicalization, all namespace declarations, which are not used in the hashed element, are removed during the canonicalization process (visible utilized, see Section 2.6.2). The prefixes used in the `xf:XPath` element are handled as text content, so that they are removed by the Exclusive Canonicalization if they are not needed. So, the canonicalized form of the example `xf:XPath` element only keeps the namespace declaration of the `xf` prefix, because the prefixes `op` and `cnt` are not used in the `ds:SignedInfo` element.

Using the namespace injection technique, an XSW message can be constructed as shown in Figure 14.

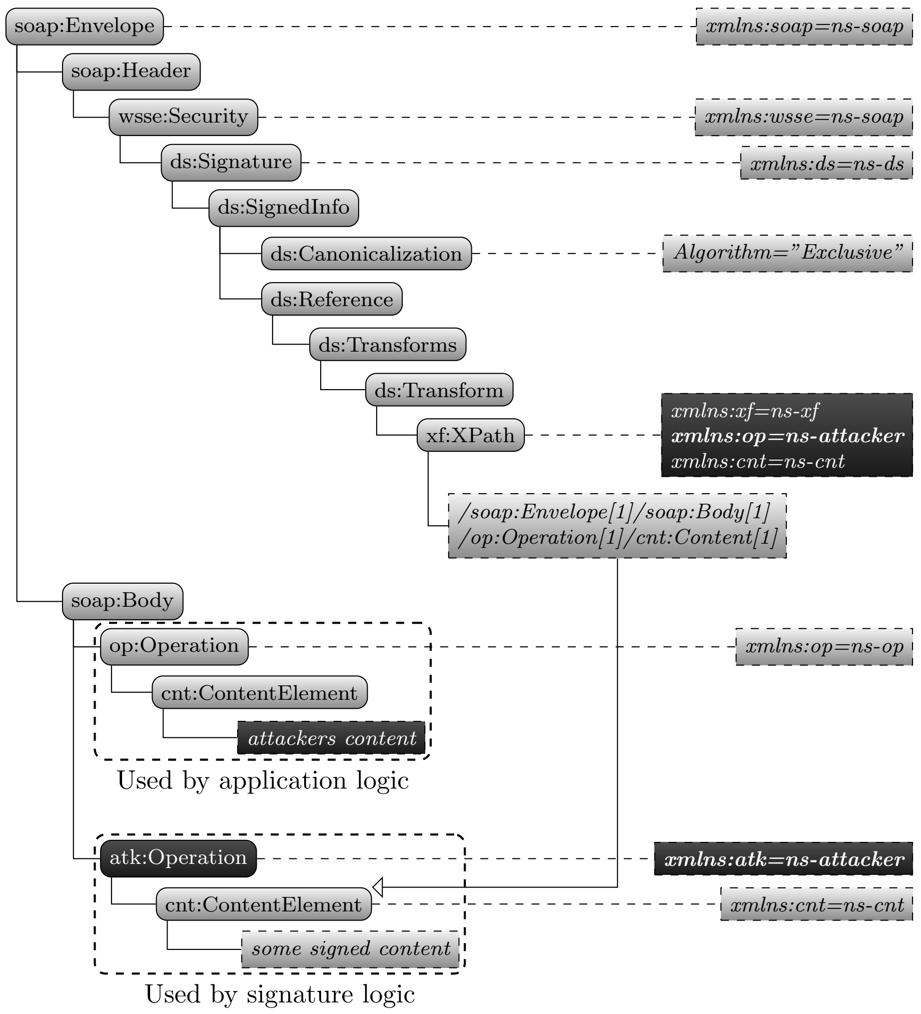


Figure 14: Example namespace injection attack message.

The message is constructed by the following steps:

- ▷ The `op:Operation` element is doubled.
  - → The first one is used by the application logic, so it holds the attacker content.
  - → The second one is used by the signature verification logic. Its prefix is changed to `atk` which is bound to an attacker namespace URI.
- ▷ The namespace declaration for the `op` prefix in the `xf:XPath` element is changed so that it uses the attacker URI.

The attack works, because the application logic simply looks in the `soap:Body` for an `Operation` element which is bound to the operation namespace[4], thus it uses the attacker content for the execution. The signature verification logic has to resolve the `op` prefix. It searches in the `xf:XPath` element and finds the declaration which binds the prefix `op` to the attacker URI, so it will use the `atk:Operation` element for computing the hash. As the Exclusive Canonicalization method is used, the namespace declaration for `op` in the `xf:XPath` element is omitted as described before and the hash value for the `ds:SignedInfo` element unchanged – the XML Signature is still valid.

As a countermeasure, Jensen et. al. proposed to embed the namespace URI in the XPath expression. For this purpose, the `namespace-uri()` and `local-name()` function can be used. Suppose the FastXPath of Listing 7.

```
/soap:Envelope[1]/soap:Body[1]
```

Listing 7: A sample FastXPath which uses prefixes.

The XPath can be transformed, so that it does not use the prefixes to select an element, but directly the concrete namespace URI. This is shown in Figure 8.

```
/*[local-name()="Envelope" and namespace-uri()="..."][1]    ↷
   /*[local-name()="Body" and namespace-uri()="..."][1]
```

Listing 8: Prefix-free transformed FastXPath.

Using a FastXPath expression as input, the authors showed that this transformation still allows a streaming based procession (SAX/StAX). An obvious disadvantage is the bad readability for humans. Although the example XPath only uses two prefixes, the length of the expression is much longer, even without showing concrete namespace URIs.

An attempt to combine the usability of FastXPath with the security of the transformed XPath expressions is shown in (33). The authors implemented an Axis2[5] module which uses a FastXPath based WS-SecurityPolicy and automatically transformed these expressions into its prefix-free equivalent.

[4] In a simpler scenario, the application logic just uses the first child of the `Body` element.
[5] `https://axis.apache.org/axis2/`

3.6 XML Schema Validation as Protection for XSW

Another more general approach for defending XSW is using XML Schema validation as shown in (34).

The general idea of XML Schema validation is to constrict the XML document in such a way, that an attacker can not find any possible position to place his wrapper element. This does not directly affect the XML Signature verification process – it is only started if the Schema validation was successful.

The main problem of this approach is its difficulty to apply it on SOAP based web services. The SOAP standard Schemas use the <xs:any/> property which allows an element to have any kind of child elements. Obviously, this is not compatible with the idea to constrict the whole XML document.

As a solution, the authors of the paper introduced the idea of *Hardened Schemas*, which remove all <xs:any/> declarations and embed SOAP extensions, like WS-Security and XML Signature.

Unfortunately, this procedure is not standard compliant, so that it must be explicitly integrated in the web service.

4 Algorithms for Automatic XML Signature Wrapping

The previous section showed different techniques for attacking XML Signatures and examples for the construction of XSW messages in different scenarios. In the following, algorithms for creating such messages with arbitrary XML documents as input are presented. The goal consists in combining these algorithms to create one generic algorithm which just uses any signed SOAP message as input and creates reasonable XSW messages out of it.

4.1 Terminology

For understanding the algorithms described in the next sections, three important terms must be introduced.

- ▷ The *signed element* is the root element of the hashed subtree. When using ID references, it is the element which has the matching attribute. In case of an XPath reference, it is commonly defined by its last step.
- ▷ The *payload element* replaces the signed element. This element will be used by the application logic. Note that it is also possible to use a different element name than the one of the signed element and try to let the application logic use it.
- ▷ The *wrapper element* is the element which holds the signed element, but is located at a different position within the XML document. In some cases, the name of the wrapper element is the same as the name of the signed element, meaning, no real `Wrapper` element is used.

If the attack is successful, the payload element replaces the signed element, which is then located in the wrapper element as shown in Figure 15.

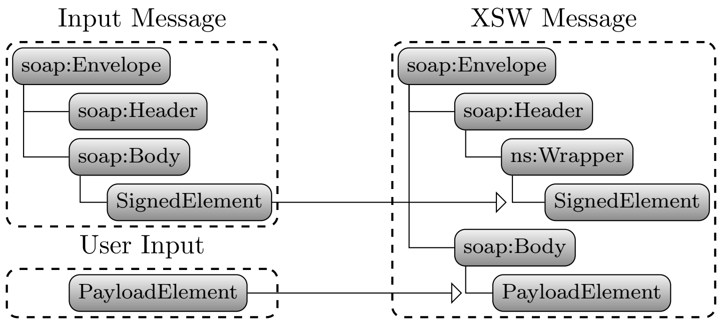


Figure 15: Terminology of *signed element*, *payload element* and *wrapper element*.

4.2 Complexity

First of all, the complexity of XSW must be analyzed. The example attacks in Section 3 showed only one specific XSW message. However, for creating this message, a lot of decisions have been made. Thus, there is not just one possible attack message, in fact, the number of possibilities is very high. Figure 16 gives an overview about the complexity of creating such XSW messages.

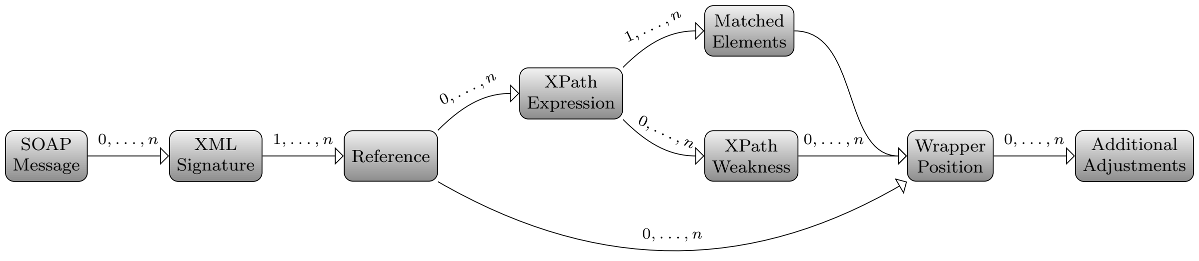


Figure 16: XSW complexity overview. There are a lot of one-to-many relationships, e.g. a `Reference` element can have multiple XPath expressions. This results in a huge number of possible XSW messages for one and the same input document.

In general, an XSW message places one wrapper element for each signed element, but in detail, it looks like the following:

- ▷ One SOAP message can have multiple XML Signatures.
- ▷ Each `Signature` element has at least one `Reference` element.

- ▷ A `Reference` element can have a `URI` to select the signed element by its `ID` attribute. However, it can also use multiple XPath expressions.
  - → If it uses XPath expressions, each one can match multiple elements, so that each one must be re-positioned (wrapped) in the XML document.
  - → One XPath can have multiple weaknesses and the attack abuses one of them, see Section 4.6.
- ▷ At least, if an element for placing the wrapper is found, there are several possibilities for doing this, e.g. the wrapper can be placed as the first child or any other child position.
- ▷ As a post processing step, there might exist some additional adjustments, e.g. change the value of an ID attribute.

This summary clarifies, that the number of possible XSW messages increases with each of these steps. Even small messages with only one signed element can have hundreds of alternative attack messages. Nevertheless, all these possibilities are deterministic. This allows to create an algorithm which can return the maximum number of possibilities and each of them can be accessed by its index.

4.3 Transforming ID References to XPath Expressions

The transformation of the `ID` referencing method to an XPath equivalent is an important fact for creating an XSW algorithm. As shown in Section 3.3, an XML Signature, which accesses the signed element by the attribute `wsu:Id='signedID'`, can be transformed to the XPath expression `//*[@wsu:Id='signedID']`. Both methods select the same element, but this transformation can simplify the building of wrapping attack messages. Instead of creating two algorithms – one for handling ID references and one additional for XPath references – only the latter is necessary. Thus, in the following, this thesis will concentrate on attacking XPath referencing XML Signatures.

4.4 Handling Timestamps

Timestamps are used to specify a time slot for a SOAP message. If it is expired, the message is rejected. The basic structure of a WS-Security `Timestamp` element can be seen in Figure 17.

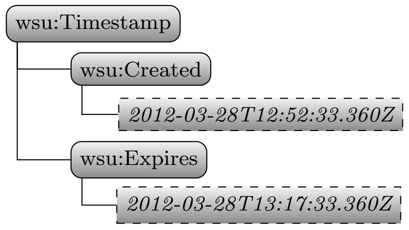


Figure 17: A sample `Timestamp` element.

If such an element is signed, the XSW attack should detect it and automatically use an updated, valid timestamp as the payload element. Therefore, the format must be kept, i.e.:

- ▷ Compute the lifetime of the timestamp as the difference of `Expires` and `Created`.
- ▷ The standard allows a timestamp to use milliseconds. The payload must keep this format if it is used.

Afterwards, the payload element can be easily created and use the same format, so that the server application can not distinguish it from a regular timestamp.

4.5 Analyzing XML Schema

The complexity of creating XSW messages mentioned in Section 4.2 showed that in general there are many possibilities to place a wrapper element. However, not each position makes sense, e.g. placing the wrapper as a child of the `SignatureValue` element invalidates an XML Signature. Those positions can be statically ignored, but another problem appears if the web service validates XML Schema. Wrapper elements at wrong positions will cause the SOAP request to raise a Schema validation error. Therefore, it is important to create Schema-valid SOAP messages, as they will be accepted no matter whether the web service uses a Schema validator or not.

Section 3.6 showed that the `xs:any` property of elements can be used to place wrapper elements. Suppose an attacker who wants to create an XSW message out of an arbitrary input document. Then, he has to answer the question, where to put the wrapper element, if the XSW message should remain Schema valid. The following algorithm will return a list of elements which allows to place `xs:any` child elements below a specified element in the XML document.

Algorithm 1 (*XML Schema Analyzer*)

```
Input:  ▷ Element $x$ which is highest possible wrapper position.
        ▷ XML Schema definition files. Files for SOAP, XML Security, ..., can
          be set as default.
Output: List $\mathcal{L}$ of allowed extension points.
  1. Initiate the list of allowed extension points $\mathcal{L}$ with an empty list.
  2. If $x$ has the xs:any property, then append $x$ to $\mathcal{L}$.
  3. For any child element $y$ of $x$ in the working document or in the XML Schema
     definition:
        ▷ Go to Step 2 with $x \leftarrow y$.
```

The idea of the algorithm is to search recursively for elements which allow any child elements. It is notable, that the algorithm does not only look for elements in the working XML document. If the Schema allows an element to have one specific child element, which is currently not present in the document, it will also be used to look for the `xs:any` property.

As an example, consider the following XML Schema, which is notated as a grammar for simplicity.

- ▷ $\mathtt{A} \rightarrow \mathtt{B} + \mathtt{C}$
- ▷ $\mathtt{B} \rightarrow \textit{any} + \mathtt{D} + \mathtt{E}$
- ▷ $\mathtt{E} \rightarrow \textit{any}$
- ▷ $\mathtt{C} \rightarrow \mathtt{F} + \textit{any}$

The Schema analyzer algorithm is now used to find extension points for the XML document on the left side of Figure 18.

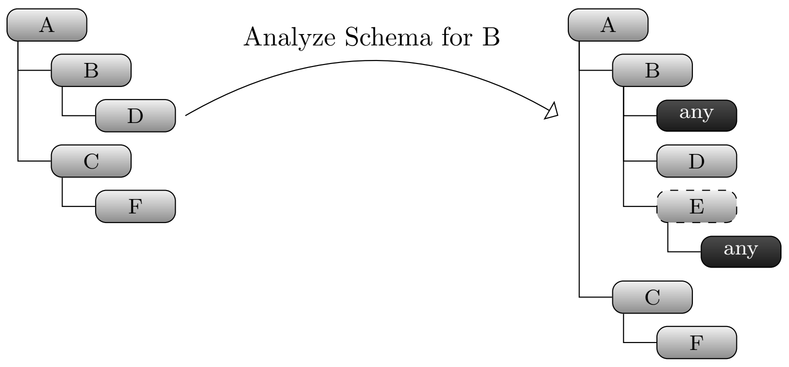


Figure 18: Example for the Schema analyzer algorithm.

Suppose that the attacker wants to place the wrapper element somewhere below the element `B`. Note that the attacker could also ask for extension points below the root element. This is what he would usually do for ID based references, but some XPath expression, e.g. /A/B//∗`[@ID='signed']` need extension points at different positions. The algorithm then processes the following steps:

1. $\mathcal{L} = \{\}$, $x \leftarrow$ `B` (Initialization)
   ▷ $x =$ `B` has `xs:any` property with rule `B` $\rightarrow$ *any* $+$ `D` $+$ `E`
   ⇒ $\mathcal{L} = \{$`B`$\}$
2. $y \leftarrow$ `D` (from input document).
   ▷ No more rules for `D`.
3. $y \leftarrow$ `E` (from rule `B` $\rightarrow$ *any* $+$ `D` $+$ `E`).
   ▷ $x =$ `E` has `xs:any` property with rule `E` $\rightarrow$ *any*
   ⇒ $\mathcal{L} = \{$`B,E`$\}$

The algorithm found two possibilities for placing a wrapper element. The first possibility is placing it as a direct child of element `B`. The second possibility needs to create the element `E` as a child of element `B`. Note that this element is not part of the input document. Its validity is only defined in the XML Schema.

A use-case for this scenario is the `Object` element in an XML Signature. The XML Signature Schema definition allows this element to occur as a child of the `Signature` element. Commonly, this can be used to store the signed content of a classical XML document, which is called an enveloping signature. However, in the case of SOAP messages, detached signatures are commonly used, which means, that the signed object is located outside of the `Signature` element. If the XML Schema definition is searched only for elements which are allowed *and* available in the current working document, the `Object` element can not be found.

4.6 XPath Weakness Algorithms

The XPath grammar is a very powerful language. In Section 3, multiple techniques for creating XSW messages are presented. As shown before, an ID based XML Signature can be treated as if it references the signed object by an XPath expression, see Section 4.3. Thus, for creating such XSW messages, only attacks on XPaths are relevant. The following sections will present algorithms which abuse a specific aspect of an XPath expression to create an XSW message.

4.6.1 Attribute Weakness

An XPath expression can select a node by its attribute name and value. However, this does only mean that the signature verification logic uses this attribute to detect the signed object. It does not mean that the application logic does the same – maybe it just checks, if an attribute with the same name is present, but does not check, if it has the same value. It is also possible that the application logic does not verify the attribute's presence at all.

Suppose an attacker who wants to create an XSW message for the input message shown on the left side of Figure 19, which is signed with the XPath given by Listing 9. Note that the `Signature` element is omitted in the figure for simplicity.

```
/soap:Envelope/soap:Header/ns:Order[@num='1']/ns:Shipping
```

Listing 9: Example XPath which uses an attribute to select a node.

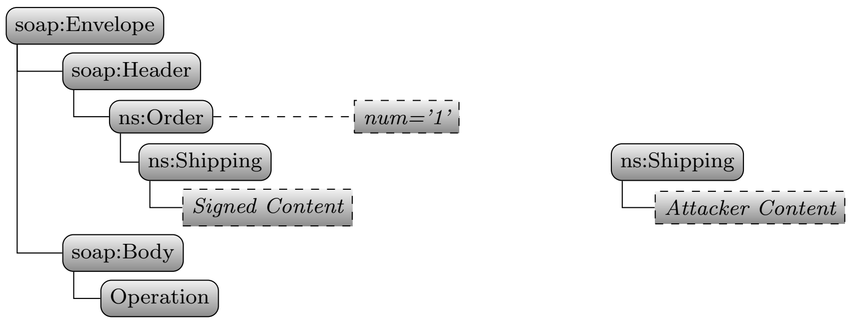


Figure 19: Example message to use for the XPath of Listing 9. The payload element is located on the right side.

If the attacker wants to create a classical XSW message (see Section 3.3, Figure 12), he first has to find the element which is selected by the XPath step that uses the attribute expression. This element will be called the *signed post part* element $\mathcal{SPP}$.

In the example, the $\mathcal{SPP}$ element is the `Order` element, because it is matched by the step ns:Order`[`**@num=**`'1']`. Note that the attribute expression does not select the signed element (the `Shipping` element) directly – it only specifies one of its ancestor elements, the `Order` element.

Generally, the signed post part element $\mathcal{SPP}$ can be detected by Algorithm 2.

Algorithm 2 (*Signed Post Part by PreXPath*)

```
Input:
  ▷ The signed element $\mathcal{S}$.
  ▷ An XPath step $x$.
Output: The signed post part element $\mathcal{SPP}$.
  1. Evaluate the extended PreXPath of step $x$ to get a list of matches $\mathcal{M}$.
  2. For each match $m$ in $\mathcal{M}$:
       ▷ If $m$ is an ancestor of $\mathcal{S}$, return $\mathcal{SPP} \leftarrow m$.
```

The algorithm evaluates the extended PreXPath to find elements which are matched by the input XPath step x. The for loop is then used to identify which one of those matched elements is the ancestor of the signed element – this is the signed post part element $\mathcal{SPP}$. It would also be possible to find this element by the PostXPath. Therefore, one has to start at the signed element and evaluate the PostXPath. If it matches, the signed post part element is found. Otherwise one has to retry with its parent element, until a match is found. However, as this method requires multiple XPath evaluation, the former method is used.

To apply Algorithm 2 to the example input message, one has to set $\mathcal{S} \leftarrow$ `Shipping` and $x =$ ns:Order`[`**@num=**`'1']`. Note that in this case, $\mathcal{M}$ does only contain one element: `Order`, which is an ancestor of $\mathcal{S}$.

The next step is to create the *payload post part* element $\mathcal{PPP}$. This is shown in Figure 20.

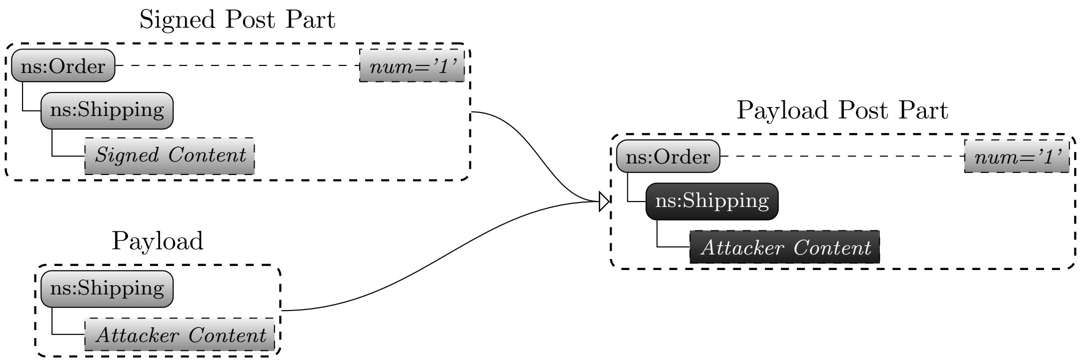


Figure 20: Creating the payload post part.

It can easily be seen, that for the example attack message, the payload post part is created by using the payload element `Shipping` and adding the `Order` element as its parent node. For generic messages, this is realized by Algorithm 3.

Algorithm 3 (*Payload Post Part*)

```
Input:
  ▷ The signed post part element $\mathcal{SPP}$.
  ▷ The signed element $\mathcal{S}$
  ▷ The payload element $\mathcal{P}$.
Output: The payload post part element $\mathcal{PPP}$.
  1. Create a deep copy of $\mathcal{SPP} =: tmp$.
  2. Replace the corresponding $\mathcal{S}$ in tmp, which is a descendant of $\mathcal{SPP}$ or it is
     $\mathcal{SPP}$ itself, with $\mathcal{P}$.
  3. Return $\mathcal{PPP} \leftarrow tmp$.
```

Putting all together, the XPath attribute weakness algorithm has the following flow:

Algorithm 4 (*XPath Attribute Weakness*)

```
Input:
  ▷ The signed element $\mathcal{S}$, and the whole XML document where it is located.
  ▷ The payload element $\mathcal{P}$. It is not part of the input document.
  ▷ The XPath attribute expression $\mathcal{N} = \mathcal{V}$ with attribute name $\mathcal{N}$ and its
    value $\mathcal{V}$. It is a part of the step $x$, which belongs to the XPath used to
    select $\mathcal{S}$.
Output: An XSW message.
  1. Detect the position of the signed post part element $\mathcal{SPP}$ by using Algo-
     rithm 2 with input $\mathcal{S}$ and $x$.
  2. Create the payload post part element $\mathcal{PPP}$ by using Algorithm 3 with input
     $\mathcal{SPP}$, $\mathcal{S}$ and $\mathcal{P}$.
  3. Place the $\mathcal{PPP}$ element as a sibling of the $\mathcal{SPP}$ element.
  4. Change the attribute $\mathcal{N}$ of the $\mathcal{PPP}$ element. There are three possibilities:
     4.1. Remove the whole attribute $\mathcal{N}$ from $\mathcal{PPP}$.
     4.2. Change the value $\mathcal{V}$ to an arbitrary value.
     4.3. Leave $\mathcal{N} = \mathcal{V}$ as it is.
```

The created XSW message can be seen in Figure 21. The three possibilities are analogous to Algorithm 4.

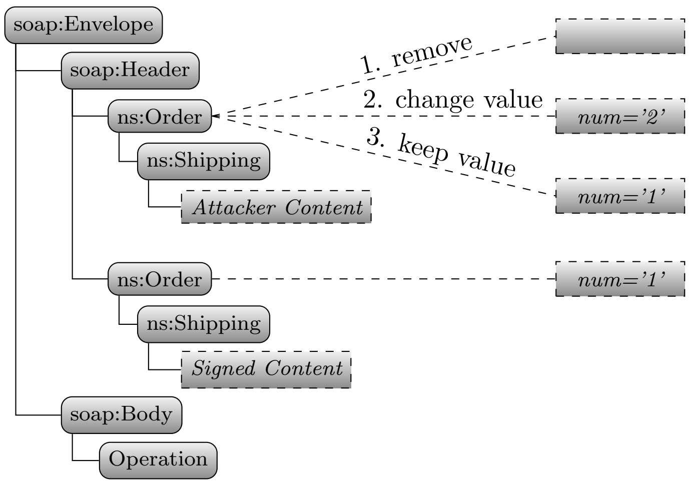


Figure 21: XSW message for Figure 19 after Algorithm 4.

However, the algorithm itself offers even more possibilities. The $\mathcal{PPP}$ element could also be placed after the $\mathcal{SPP}$ element. The more siblings the $\mathcal{SPP}$ element has, the more possibilities exist, as shown in Section 4.2.

4.6.2 Descendant Weakness

XPath expressions can select nodes which are located anywhere below a specified element. This feature is realized by using the *descendant-** axis selection, as shown in Section 2.5.1. However, when using such a function in the context of XML Signature, an attacker can try to use the following technique to create an XSW message.

As an example, suppose the XPath in Listing 10 and the SOAP request of Figure 22.

```
/soap:Envelope//ns:Operation/ns:Signed
```

Listing 10: Example XPath which uses the *descendant-or-self* axis to select a node.

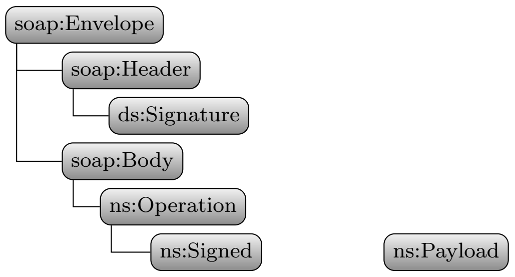


Figure 22: Example message to use for the XPath /soap:Envelope//ns:Operation/ ↷ ns:Signed. The attacker is going to include the payload element on the right side.

The main problem is that the protected subtree does not contain every element on the path from the document root down to the signed element as a step in the given XPath expression. The element `Body` is not a part of it. Thus, the attacker can split the expression at the descendant step named x into the PreXPath /soap:Envelope and the PostXPath ns:Operation/ns:Signed. Obviously, both are part of the protected subtree, so the attacker has to keep their structures. Figure 23 visualizes which part of the document is selected by those XPath parts.

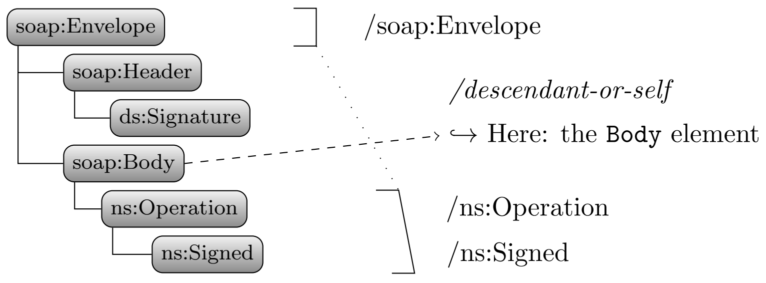


Figure 23: Visualization of the protected parts.

Commonly, an attacker knows the position of the signed element $\mathcal{S}$, which is the `Signed` element in this case. For finding this element, he only has to evaluate the whole XPath expression. However, he needs to find the signed post part element $\mathcal{SPP}_{\text{post}}$, in this case the `Operation` element, which is the highest element selected by the PostXPath. For detecting it, there are again the same two possibilities as mentioned in Section 4.6.1. Thus, Algorithm 2 with the signed element $\mathcal{S}$ and the first step *after* the *descendant-** step x can be used as input parameters.

Applying this algorithm to the example, the ns:Operation step is used and internally expanded to the extended PreXPath /soap:Envelope//ns:Operation, so that the correct `Operation` element is found.

Additionally, the signed post part element $\mathcal{SPP}_{\text{pre}}$, matched by the PreXPath must be detected. In this case, it is obviously the `Envelope` element. Furthermore, it can be identified by using Algorithm 2 with the signed element $\mathcal{S}$ and the first step *before* the *descendant-** step x as input parameters. The attacker then knows that he has to place the wrapper element, which must contain the $\mathcal{SPP}_{\text{post}}$ element `Operation`, somewhere below the $\mathcal{SPP}_{\text{pre}}$ element `Envelope`.

The next step is to decide where to place exactly the $\mathcal{SPP}_{\text{post}}$ element `Operation`. To reduce the number of possibilities, the attacker can try to create an XML Schema valid XSW message. Therefore, he can use Algorithm 1 with the $\mathcal{SPP}_{\text{pre}}$ element `Envelope` as input to get a list of extension points, i.e. $\mathcal{L} = \{\texttt{Header}, \texttt{Body}, \texttt{Object}, \dots\}$. Note that for this result, the SOAP request is assumed to use version 1.2. For version 1.1, the `Envelope` element itself would be contained, too. Note that the `Object` element is defined in the XML Signature Schema, but the element itself is not yet contained in the input document. The attacker then has to choose $\mathcal{E}$ as one of those possibilities, e.g. $\mathcal{E} \leftarrow$ `Object`. As this element does not exist yet, it has no child elements, thus, the attacker determines to put the wrapper element as the first child of the `Object` element.

Afterwards, the attacker has to decide if a special wrapper element should be used. This is shown in Figure 24.

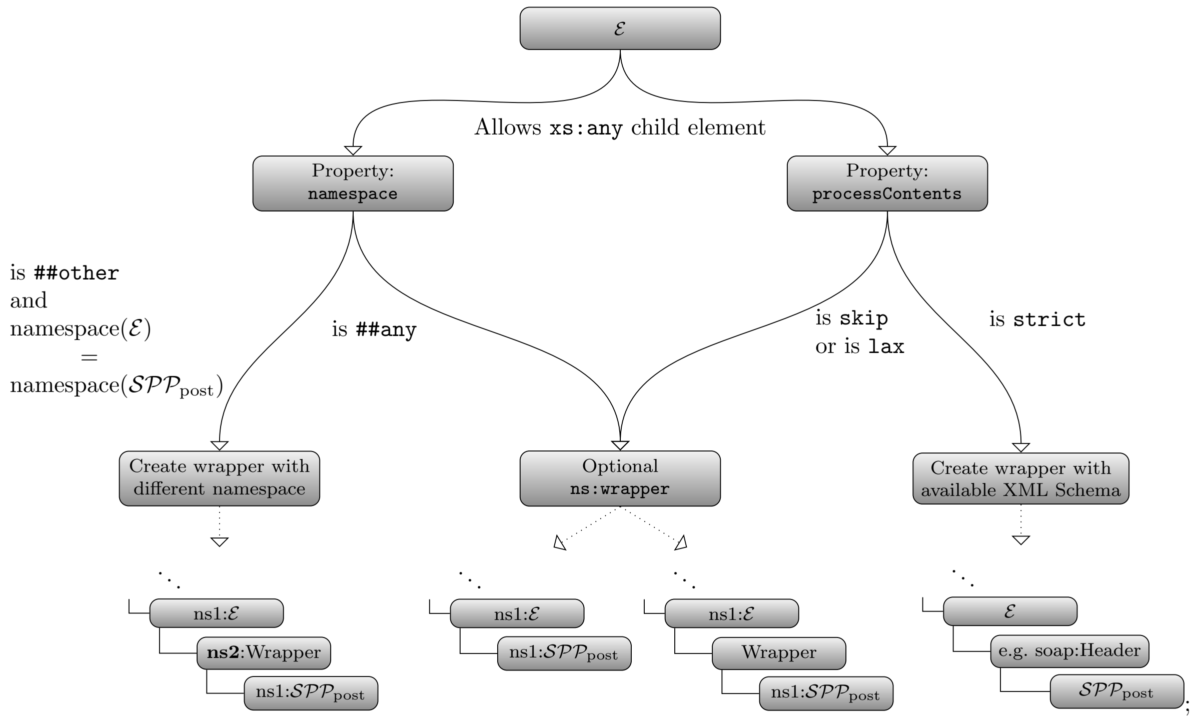

$\mathcal{E}$
Allows xs:any child element
Property: namespace
Property: processContents
is ##other and namespace($\mathcal{E}$) = namespace($\mathcal{SPP}_{\text{post}}$)
is ##any
is skip or is lax
is strict
Create wrapper with different namespace
Optional ns:wrapper
Create wrapper with available XML Schema
ns1:$\mathcal{E}$
ns2:Wrapper
ns1:$\mathcal{SPP}_{\text{post}}$
Wrapper
e.g. soap:Header
$\mathcal{SPP}_{\text{post}}$

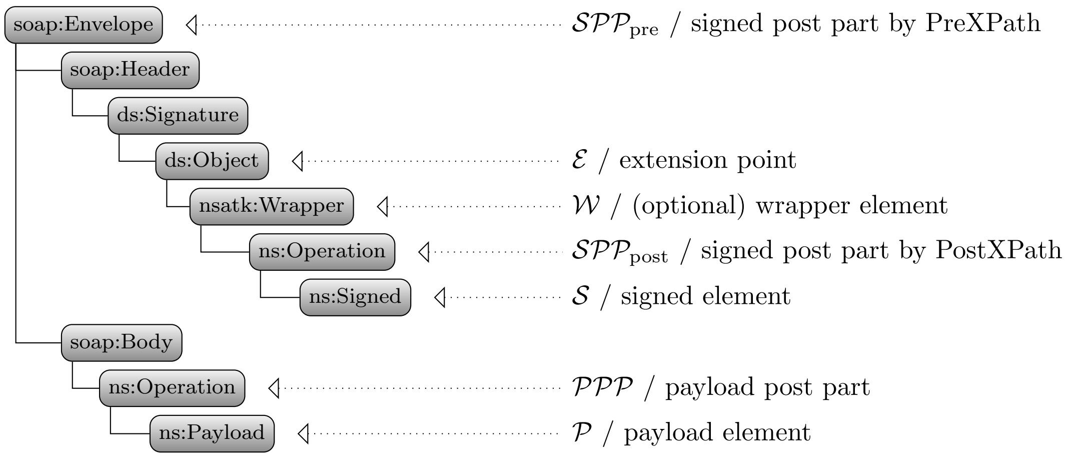
Figure 24: Overview of XML Schema properties and their influence for creating wrapper elements.

An attacker needs to create a specific wrapper in the following two scenarios:

1. The XML Schema has the property `namespace="##other"` for the element $\mathcal{E}$ and the $\mathcal{SPP}_{\text{post}}$ element uses the same namespace as the element $\mathcal{E}$. In this case, there must be inserted a wrapper element between $\mathcal{E}$ and $\mathcal{SPP}_{\text{post}}$ which has a different namespace.
2. If the Schema definition for the element $\mathcal{E}$ has the property `processContents="strict"`, the XML Schema definition for the child element must be available. As a trick, one can insert a `soap:Header` element, whose Schema definition is always present.

In the example, the element $\mathcal{E}$ = `Object` has the properties `processContents="lax"` and `namespace="##any"`, so that there are no restrictions and the attacker does not need to create a special wrapper element. Note that in some scenarios, both restrictions may occur and the wrapper element must fulfill them all. Nevertheless, for didactic reasons, the example attacker decided to use an `ns:Wrapper` element $\mathcal{W}$. Thereafter, the attacker places the wrapper element $\mathcal{W}$ (`ns:Wrapper`) as a child of the element $\mathcal{E}$ (`ds:Object`). The signed element is now relocated.

The last step is to create the payload post part element $\mathcal{PPP}$. Therefore, Algorithm 3 can be used with input $\mathcal{SPP}_{\text{post}}$, $\mathcal{S}$ and the payload element $\mathcal{P}$. Afterwards, the $\mathcal{PPP}$ element is moved to the exact position, where the $\mathcal{SPP}_{\text{post}}$ element was moved from. The final XSW message can be seen in Figure 25.



soap:Envelope
$\mathcal{SPP}_{\text{pre}}$ / signed post part by PreXPath
soap:Header
ds:Signature
ds:Object
$\mathcal{E}$ / extension point
nsatk:Wrapper
$\mathcal{W}$ / (optional) wrapper element
ns:Operation
$\mathcal{SPP}_{\text{post}}$ / signed post part by PostXPath
ns:Signed
$\mathcal{S}$ / signed element
soap:Body
ns:Operation
$\mathcal{PPP}$ / payload post part
ns:Payload
$\mathcal{P}$ / payload element

Figure 25: Final XSW message which abuses the XPath *descendant-** axis.

All in all, Algorithm 5 summarizes the previous steps for creating an XSW message by abusing the XPath *descendant-** axis.

Algorithm 5 (*XPath Descendant Weakness*)

```
Input:
    ▷ The signed element $\mathcal{S}$, and the whole XML document where it is located.
    ▷ The payload element $\mathcal{P}$. It is not part of the input document.
    ▷ The XPath descendant step $x$, which belongs to the XPath used to select $\mathcal{S}$.
Output: An XSW message.
    1. Detect the signed post part element $\mathcal{SPP}_{\text{post}}$ by Algorithm 2 with input $\mathcal{S}$ and next_step($x$).
    2. Detect the signed post part element $\mathcal{SPP}_{\text{pre}}$ by using Algorithm 2 with input $\mathcal{S}$ and previous_step($x$).
    3. Place the wrapper by the following steps:
        ▷ Get a List $\mathcal{L}$ of Schema weaknesses by using the Schema Analyzer Algorithm 1 with input $x = \mathcal{SPP}_{\text{pre}}$.
        ▷ Choose one element $\mathcal{E}$ of $\mathcal{L}$. If it is not contained in the current document, it must be created at the defined position.
        ▷ Choose $i \in \{0, \ldots, \text{number\_of\_child\_elements}(\mathcal{E})\}$.
        ▷ Detect if a special wrapper element is necessary. If true, set $\mathcal{W}$ to this element, otherwise set $\mathcal{W} \leftarrow \mathcal{SPP}_{\text{post}}$.
        ▷ Place the wrapper $\mathcal{W}$ as the $i$-th child of element $\mathcal{E}$.
    4. Create the payload post part element $\mathcal{PPP}$ by using Algorithm 3 with input $\mathcal{SPP}$, $\mathcal{S}$ and $\mathcal{P}$.
    5. Place $\mathcal{PPP}$ at the old position of $\mathcal{SPP}_{\text{post}}$
```

This algorithm itself does only abuse the *descendant-** weakness. When attacking real web services, there might be XPath expressions which contain additional attribute expressions. A common example might be a transformed ID reference, e.g. `//*[@ID ↷ ='signed'`. For creating an XSW message in such scenarios, the attribute weakness shown in Section 4.6.1 must be applied afterwards. Note that in such cases, the algorithm must not duplicate the signed post part element, as this is processed by the descendant weakness algorithm. It only needs to adjust the attribute values of the $\mathcal{SPP}$ and the $\mathcal{PPP}$ elements.

4.6.3 Namespace Injection Attack

The last included XPath weakness algorithm uses the namespace injection technique. This one has – compared to the attribute- and the descendant weakness – some more prerequisites:

1. The XPath step x uses a prefix $\rho_{\text{org}} = uri_{\text{org}}$.
2. The XML Signature uses Exclusive Canonicalization.
3. The prefix ρ_{org} is not protected by the XML Signature.
4. The step x is not the first step.

The first prerequisite is obvious – the namespace injection attack needs the presence of a prefix. The second one is a bit special. While the other presented attacks do not depend on any special XML Signature characteristic, the namespace injection attack requires the usage of Exclusive Canonicalization. This is fundamental, as the attack directly manipulates the hashed subtree. Nevertheless, those changes do not affect its canonicalized form – the signature will remain valid. Additionally, the prefix ρ_{org} must not be protected by the XML Signature, because the idea of this attack is to insert a namespace declaration in an element, e.g. the `xf:XPath` element. Every prefix, that is not used, will be removed be the Exclusive Canonicalization. Thus, the only declaration kept in this element will be the `xf` prefix, so this one can not be overwritten. However, this will be no restriction in most cases. The last requirement is that the step x must not be the first step, as the namespace injection attack will double the element matched by this step. This is simply not possible for the root element of a SOAP request.

The formal procedure for the namespace injection attack is shown in Algorithm 6.

```
Algorithm 6 (Namespace Injection Weakness)
  Input:
    ▷ The signed element $\mathcal{S}$, and the whole XML document where it is located.
    ▷ The payload element $\mathcal{P}$. It is not part of the input document.
    ▷ The XPath step $x$, which belongs to the XPath used to select $\mathcal{S}$. It uses a
      prefix $\rho_{\text{org}}$ which belongs to the namespace URI $uri_{\text{org}}$.
  Output: An XSW message.
    1. Detect the position of the signed post part element $\mathcal{SPP}$ by using Algo-
       rithm 2 with input $\mathcal{S}$ and $x$.
    2. Create the payload post part element $\mathcal{PPP}$ by using Algorithm 3 with input
       $\mathcal{SPP}$, $\mathcal{S}$ and $\mathcal{P}$.
    3. Insert $\mathcal{PPP}$ as a sibling of $\mathcal{SPP}$.

    4. Choose an injection prefix $\rho_{\text{atk}}$ and its corresponding URI $uri_{\text{atk}}$.
    5. For $\mathcal{SPP}$ and each child element of $\mathcal{SPP}$, do:
         ▷ If the element uses prefix $\rho_{\text{org}}$ and $uri_{\text{org}}$, then change it to $\rho_{\text{atk}}$ and
           $uri_{\text{atk}}$.
    6. Locate the xf:XPath element of the used XPath within the input document.
       Insert the namespace declaration $\rho_{\text{org}} = uri_{\text{atk}}$.
       It is also possible to insert this declaration in one of its ancestors but before
       the original declaration $\rho_{\text{org}} = uri_{\text{org}}$ is defined.
```

The first steps are similar to the other algorithms. The attacker has to detect the signed post part element $\mathcal{SPP}$ and then creates the payload post part element $\mathcal{PPP}$. Afterwards, the $\mathcal{PPP}$ element is inserted as a sibling of the $\mathcal{SPP}$ element. Note that there are again two possibilities for doing this, namely before and after the $\mathcal{SPP}$ element.

In the next phase, the attacker must inject the namespace declaration. Therefore, he chooses his attacker namespace ρ_{atk} and the corresponding uri_{atk}. Then he searches for every occurrence of the original prefix ρ_{org} in the $\mathcal{SPP}$ element and all of its descendants. If a match is found, the prefix is replaced with the attacker prefix ρ_{atk} and the namespace URI is also changed to the attacker URI uri_{atk}.

The last step is to insert the namespace declaration $\rho_{\text{org}} = uri_{\text{atk}}$. Note that this will map the original prefix ρ_{org} to the URI uri_{atk}, which is defined by the attacker. Thus, this declaration must be placed in the `xf:XPath` element, because the signature verification logic will start at this element to resolve the prefix ρ_{org}. There are still more possible positions for placing this declaration. As the namespace resolution process will start searching for the prefix declaration in the `xf:XPath` element, going upwards if nothing is found, it could also be declared in one of its ancestors. The only restriction to this is, that it must not overwrite the original declaration. If, e.g. $\rho_{\text{org}} = wsse$ and uri_{org} is the URI of WS-Security, the ρ_{atk} URI must be declared as a descendant of the `wsse:Security` element. Otherwise, the signature verification logic will not find the `ds:Signature` element, because the namespace for the `wsse:Security` element has changed.

4.7 High Level Algorithm

The previous sections described algorithms for creating XSW messages for specific scenarios. Algorithm 7 merges those techniques to one general procedure.

Algorithm 7 (*XSW Algorithm*)

```
Input: Signed SOAP request.
Output: One possible wrapped message.
  1. Analyze the message:
       ▷ Detect signed parts.
       ▷ Offer a payload for each part (user interaction required).
           → Take care of Timestamp elements and update them automatically
             as shown in Section 4.4.
  2. For each pair of signed- and payload element,
     detect where to move the signed part:
       ▷ If referenced by ID, convert it into an XPath as shown in Section 4.3.
       ▷ Analyze the XPath:
           → If it uses an attribute expression,
             use the attribute weakness Algorithm 4.
           → If it uses descendant-* step,
             use the descendant weakness Algorithm 5.
           → If it uses prefixes and the other prerequisites mentioned in Sec-
             tion 4.6.3 are fulfilled, use namespace injection attack Algorithm 6.
```

The high level algorithm needs to identify all signed parts within the SOAP requests. For each one, a payload element must be offered – the attacker has to set this element to the content he wants to execute. Afterwards, the algorithm needs to move each signed document part to its wrapper position. Therefore, each signed part is inspected. If it is accessed by an ID reference, it will be transformed to an XPath reference. Note that this transformation is not really applied to the concrete SOAP request – it is rather handled internally. Then, the used XPath is analyzed. If a specific part is used, e.g. the *descendant-** axis, the corresponding algorithm is applied.

The algorithm's final output is one XSW message, which has relocated all signed parts to another valid location within the input document. Additionally, all payload elements have taken the original position of the signed parts. Then, this message can be sent to the concrete web service for testing if it is accepted, which means that the signature is valid, contained timestamps are not expired and the custom payload is used by the application logic.

5 Implementation

The following section will discuss the design decisions made for implementing the algorithms of Section 4. The goal is to create an automatic penetration test tool for detection of XSW attacks in web services. The tool will analyze the message, identify the signed parts and relocate them within the SOAP request. Additionally, the user is asked to set the payload for each signed element and the tool will put it on the original position of the signed element, so that the application logic will hopefully use it.

5.1 Components

In this section, the basic components for building the XSW tool will be presented. The tool is autarkic, so that it could be easily integrated into any framework or even be used as a standalone application.

5.1.1 Signature Manager

The signature manager is the binding component between the XML Signature and the input XML document. The XSW components are independent of the SOAP standard (although the examples shown before always used SOAP messages for understandability), but as the goal is to create SOAP XSW messages, there must be a component which connects the SOAP characteristics to the XSW tool:

A signed SOAP request can have an arbitrary number of signed parts, e.g. by ID references or XPath expressions. For creating an XSW message, those have to be identified and for each part, a payload element must be created. This is the task of the signature manager as shown in Figure 26.

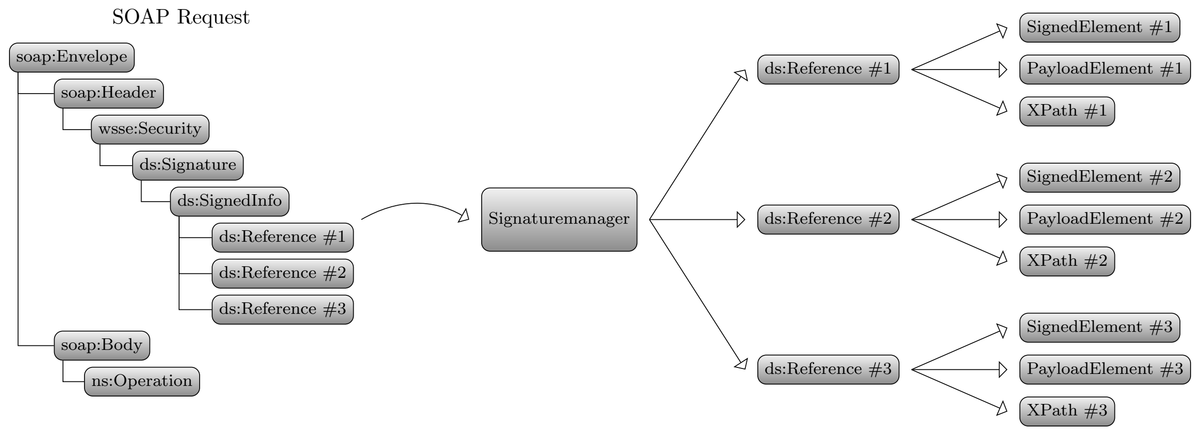


Figure 26: Task overview of the signature manager (simplified). A signed SOAP message can have multiple signed elements. Each one is accessed by a specified referencing mechanism, e.g. an XPath, and needs a payload for the attack message.

The signature manager takes a SOAP message as input. It searches for the `Signature` element and identifies its `Reference` elements. Additionally, it saves a payload element which corresponds to the signed element. It is initialized by a copy of the signed element, but can later be configured by the user.

Note that Figure 26 gives a slightly simplified structure of the signature manager. A `Reference` element of an XML Signature can select more than one signed element by using XPath expressions, especially when combining them with set-operations as shown in Section 2.6.3. In such cases, the signature manager will identify all of them and offer one payload for each. If the signed element is referenced by an ID attribute, the signature manager can transform it into an XPath expression as shown in Section 4.3.

5.1.2 XPath Parser

An essential part for analyzing XPath expressions is to be able to access all specific parts of it. Therefore, a parser has been built which gets an XPath string as an input and processes it, so that an object hierarchy as show in Figure 27 is available.

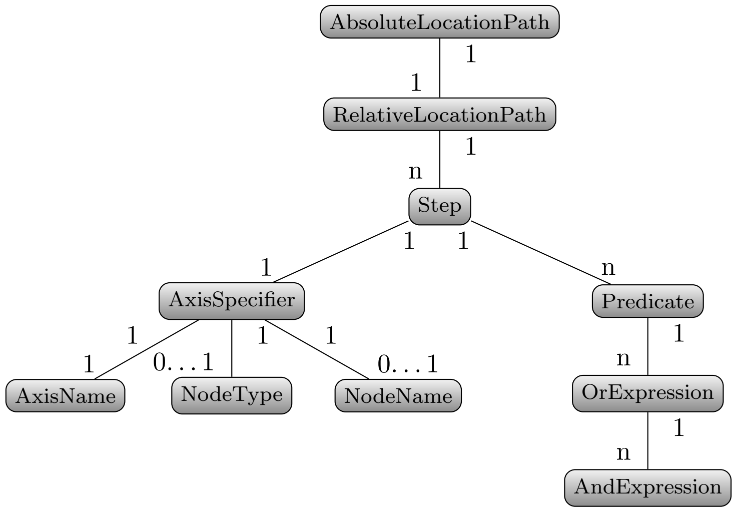


Figure 27: Splitting an XPath into its parts.

An XPath expression consists of various parts. They are associated in different relations. Some are one to one, others are one or zero to many. In such cases, they are accessible as a list.

An additional important aspect for parsing XPaths is, that some function can be abbreviated. The descendant−or−self axis, e.g. can be shortened by //, an attribute is accessible by attribute :: name or **@name**. This is the task of the XPath parser.

5.1.3 XPath Analyzer

The XPath analyzer is a bundle of concrete implementations for creating XSW messages. The algorithms presented in Section 4 are used for attacking XPath attribute expressions (Section 4.6.1), the *descendant-** axis selection (Section 4.6.2) and creating namespace injection messages (Section 3.5).

As shown in Section 4.2, the complexity of creating XSW messages is very high, even for small messages. Thus, it is very important to be able to create one specific XSW message out of a huge number. This is realized by a two phases approach:

1. During the **analysis phase**, the input message is processed and for every decision, a counter is introduced.
2. When it comes to the **apply phase**, index parameters are used to determine which decision should be used.

The analysis phase does the main work. The message is completely processed, but instead of creating a real XSW message, this phase just has to identify which possibilities are available. Consider that the wrapper element should be placed somewhere below an element $\mathcal{E}$. Let element $\mathcal{E}$ have $n = 3$ child elements. Then there are $n + 1 = 4$ possible positions for placing the wrapper element. There are also some static possibilities, e.g. for the attribute weakness, the attribute in the payload post part element can be untouched, changed or completely removed – so there are always three possibilities.

All these counters can be combined to compute the total number of possibilities. Every weakness implementation (attribute weakness, descendant weakness, namespace injection weakness) and the XPath analyzer implementation itself has a method named getNumberOfPossibilites() which returns this total number. In interaction with this procedure, the apply method abuseWeakness(**int** i) can then be used to create one specified XSW message.

The final penetration test tool will have to use all of these possibilities and send them to the server. By using the two phases approach, the creation of XSW messages is very fast, as the action consists only in walking down a decision tree and applying them.

5.1.4 Wrapping Oracle

The XPath analyzer has the task to wrap *one* signed part to another location and put the corresponding payload within the message. The wrapping oracle is the implementation of the high level Algorithm 7. It can be seen as an oracle, which gets a signed SOAP message as input and outputs one possible XSW message, so it relocates *each* signed element and places one payload for it.

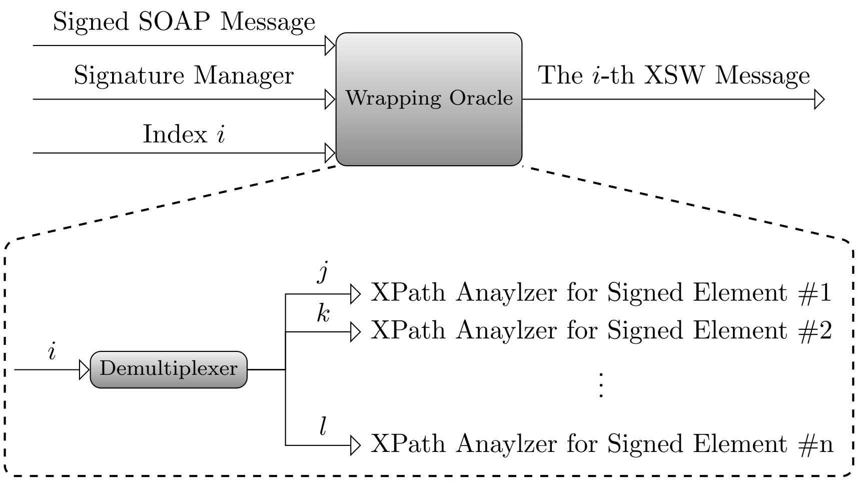

Signed SOAP Message
Signature Manager
Index i
Wrapping Oracle
The i-th XSW Message
i
Demultiplexer
j
k
l
XPath Anaylzer for Signed Element #1
XPath Anaylzer for Signed Element #2
XPath Anaylzer for Signed Element #n

Figure 28: The wrapping oracle.

As an additional parameter, the wrapping oracle needs a reference to the signature manager. This is essential, as the signature manager holds the payload elements. The last parameter specifies the index, which XSW message should be created. Internally, the index parameter is split to many indexes by a demultiplexer and forwarded to the single XPath analyzer instances as shown in Figure 28. Consequently, the wrapping oracle also has a method getNumberOfPossibilites() which returns the total number of possible XSW messages.

5.2 Integration in the WS-Attacker Penetration Test Framework

WS-Attacker is a modular framework for web services penetration testing (9). It is open source and the code is available on *sourceforge*[6], a free web-based code repository.

5.2.1 WS-Attacker Overview

The basic workflow of the WS-Attacker framework can be seen in Figure 29.

[6]`http://sourceforge.net/projects/ws-attacker/`

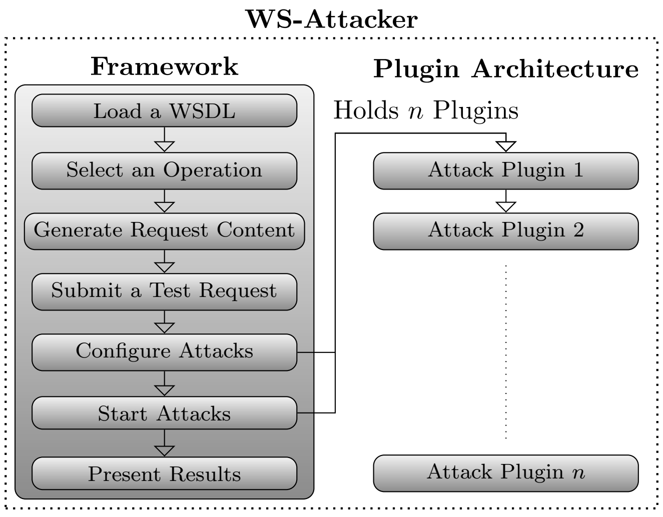


Figure 29: WS-Attacker general overview.

Its task is to load a WSDL. Afterwards, the user selects the operation to attack and the framework generates the basic request content, which has to be adjusted by the user, e.g. by setting the correct request parameters. Then, the user sends a test request to the web service and the response is saved to determine the *normal state* of the server. Hereafter, the user selects and configures the attack plugins. Each attack plugin represents one attack on a web service. Currently, there are two plugins maintained with the framework – one for SOAPAction Spoofing[7] and one for WS-Addressing Spoofing[8]. After selecting the attacks, the framework can run them one after another. Each attack will generate different outputs, e.g. log entries, and display whether the attack was successful. This is indicated by two facts:

1. A boolean value is used to show if the attack was successful in general.
2. An integer rating indicates the impact of the attack. Consider, e.g., a denial of service attack. The attacked server could be unavailable for just some minutes, or completely with the need to reboot.

[7]`http://clawslab.nds.rub.de/wiki/index.php/SOAPAction_Spoofing`
[8]`http://clawslab.nds.rub.de/wiki/index.php/WS-Addressing_spoofing`

5.2.2 Concrete Integration: Building the XSW Plugin

The main work for building an XSW plugin is done by the wrapping oracle, see Section 5.1.4. The oracle is able to use a signed SOAP request for creating the attack messages. It also has a method which returns the maximum number of possible attack messages. Those must be sent consecutively to the web service and the response has to be inspected for a successful attack. The basic idea for this can be seen in Figure 30.

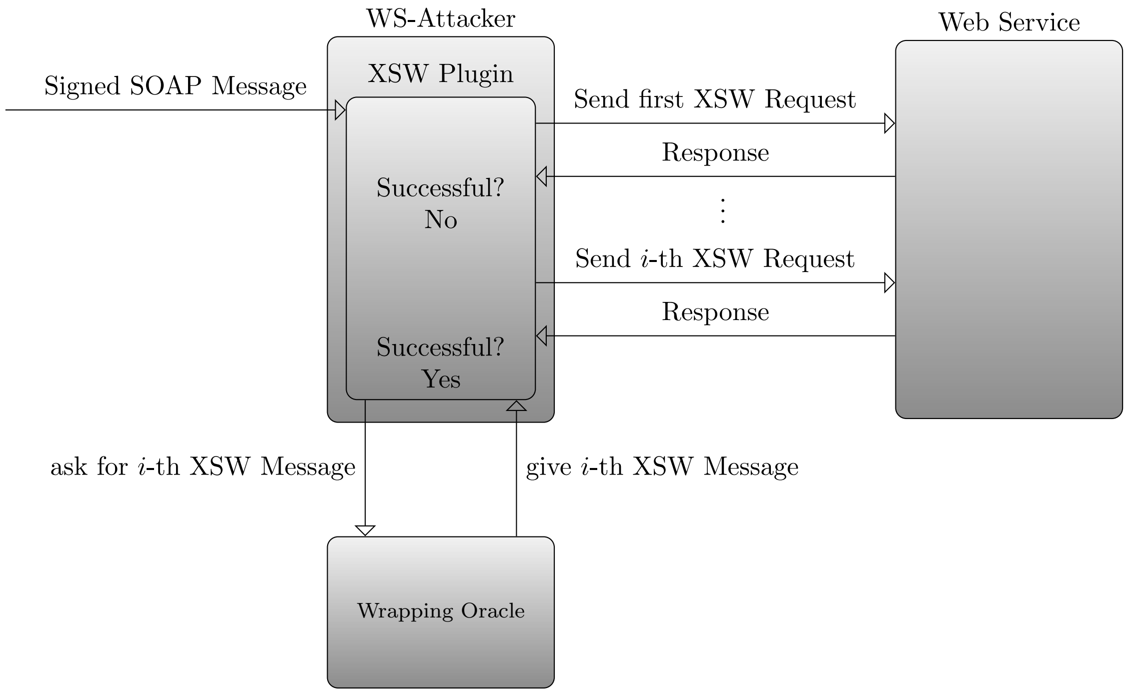


Figure 30: Integration of the XSW plugin.

An unanswered question is: How does the plugin come to know if the request is accepted by the application logic? At first glance, this seems to be very easy. The response of the web service must only not contain a SOAP error. However, when dealing with XSW messages, there is an additional problem. Every request sent to the server contains two payloads. The first one is the originally signed element, the other one is the payload element defined by the attacker/framework user. The plugin must be able to distinguish, which of them is used by the application logic. If the plugin would consider every non SOAP error response to be a successful attack, the web service could execute the original signed element and ignore the attacker's payload. Obviously, this is not a successful attack. To solve this problem, the user can simply define a characteristic string, which must be contained in the SOAP response. Only in such cases, the attack is counted as successful.

Furthermore, the WS-Attacker requires an attack plugin to rate its impact. This is implemented as follows:

0% : The web service uses transformed prefix-free FastXPath.

10% : The web service uses FastXPaths.

20% : There are multiple reasons for this:
- ▷ The web service uses XPaths, but could not be successfully attacked.
- ▷ The web service uses both, ID References and XPaths.
- ▷ The web service uses ID References but could not be successfully attacked.

100% : The web service could be successfully attacked.

Although the attack is only assessed to be successful if 100% are reached, the other ratings are used to give a general overview on the usage and integration of XML Signature into the web service.

The best rating from the view of a web service can only be reached if the signature uses prefix-free XPath expressions as described in Section 3.5. This is the only referencing method for which no attacks are currently published. 10% is given if the server uses FastXPath expressions. The only known attack for this is the namespace injection attack, which is also implemented into the XSW plugin. Note that a FastXPath is not vulnerable in general, see the prerequisites in Section 4.6.3. The rating of 20% is used to indicate interesting facts to the user if the attack was not successful. One example for this are complicated XPath expressions. The XSW plugin does not cover all functions of the XPath grammar – only the most important ones are handled. If the contained expression uses such functions, the automatic plugin might not be able to attack the web service although it could be vulnerable by a clever manual penetration tester. The plugin also notifies the user if the signature uses ID references. In general, they are attackable, unless there are no other security enforcements, e.g. a special WS-SecurityPolicy or an XML Schema parser. So if IDs are used, but the plugin could not successfully attack the web service, the user might be interested in this fact.

6 Evaluation

The evaluation for the automatic creation of XSW messages is split into two parts. At first, the general implementation of the wrapping oracle is used in Section 6.1 to build some XSW messages and validate them with standard Java methods. Secondly, the WS-Attacker XSW plugin is used for attacking three different web service framework implementations in Section 6.2.

6.1 Practical Evaluation of the Implementation Correctness

For the practical evaluation of the implementation correctness, JUnit[9] tests are used. The general setup for a test is the following:

1. Generate a test SOAP request.
2. Sign the test request
   - ▷ ... using ID references.
   - ▷ ... using one or more XPath expressions.
   - ▷ ... using a combination of both.
3. Use the signature manager to set the payloads.
4. Create a wrapping oracle.
5. For each possible XSW message.
   - ▷ Check if the signature is still valid.
     Abort with success if this is true.
6. Abort with failure.

As an example, a simple SOAP request is signed. It is signed by the following XPath expressions, which are applied by an XPathFilter2:

- ▷ //wsu:Timestamp`[1]`
- ▷ //ns1:payloadBody`[@wsu:Id='bodyToSign']`/ns1:signedElement`[1]`

The wrapping oracle will automatically detect the signed `Timestamp` element and update it. One output XSW message is shown in Listing 11.

[9]`http://www.junit.org/`

```
<soap:Envelope xmlns:soap="...">
  <soap:Header>
    <!-- Timestamp used by Signature Verification Logic -->
    <wsu:Timestamp xmlns:wsu="...">
      <wsu:Created>2011-11-28T21:01:12.100Z</wsu:Created>
      <wsu:Expires>2011-11-28T21:06:12.100Z</wsu:Expires>
    </wsu:Timestamp>
    <wsse:Security xmlns:wsse="...">
        <!-- Attackers Payload for Timestamp Element:
             It is just updated -->
      <wsu:Timestamp>
        <wsu:Created>2012-04-30T13:29:42.826Z</wsu:Created>
        <wsu:Expires>2012-04-30T13:34:42.826Z</wsu:Expires>
      </wsu:Timestamp>
      <ds:Signature xmlns:ds="...">
        <ds:SignedInfo>...</ds:SignedInfo>
        <ds:SignatureValue>...</ds:SignatureValue>
        <ds:KeyInfo>...</ds:KeyInfo>
        <!-- The Object Element is created by
             analyzing the XML Schema -->
        <ds:Object>
          <!-- Signed Body Element Wrapper -->
          <wsatk:wrapper xmlns:wsatk="...">
            <ns1:payloadBody xmlns:ns1="...">
              <ns1:signedElement>Original Content</
                 ns1:signedElement>
            </ns1:payloadBody>
          </wsatk:wrapper>
        </ds:Object>
      </ds:Signature>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <ns1:payloadBody xmlns:ns1="...">
      <!-- Attackers Payload is placed here -->
      <ns1:signedElement>ATTACKER CONTENT</ns1:signedElement>
    </ns1:payloadBody>
  </soap:Body>
</soap:Envelope>
```

Listing 11: One output of the wrapping oracle.

In this message, the `Timestamp` wrapper element is placed as the first child of the `Header` element. The signed body element is relocated in the `Object` element within the

Signature element. Note that the Object element was not part of the input message – it is created by analyzing the XML Signature Schema.

The implementation contains a lot of more JUnit tests, which ensure the functionality of all implemented algorithms, e.g. for the attribute weakness and the namespace injection attack. They can be regarded in the source code.

6.2 Real World Scenarios

In this section, the XSW WS-Attacker plugin will be used for attacking Apache Axis2 in Section 6.2.1, the XSpRES library in Section 6.2.2 and the IBM DataPower XI50 in Section 6.2.3.

6.2.1 Attacking Apache Axis2

As a first real-world scenario, the WS-Attacker penetration test framework uses the XSW plugin to attack an Apache Axis2 web service. For ensuring integrity and authenticity, the web service uses the Rampart security module (35, 36). The setup is based on the *sample02*[10] distributed with Rampart and uses the simplified WS-SecurityPolicy given by Listing 12.

```
<wsp:Policy wsu:Id="SigOnly" xmlns:wsu="..." xmlns:wsp="...">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:AsymmetricBinding xmlns:sp="...">
        <wsp:Policy>
          <sp:InitiatorToken> ... </sp:InitiatorToken>
          <sp:RecipientToken> ... </sp:RecipientToken>
          <sp:AlgorithmSuite> ... </sp:AlgorithmSuite>
          <sp:IncludeTimestamp/>
          <sp:OnlySignEntireHeadersAndBody/>
        </wsp:Policy>
      </sp:AsymmetricBinding>
      <sp:Wss10 xmlns:sp="..."> ... </sp:Wss10>
      <sp:SignedParts xmlns:sp="...">
        <sp:Body/>
      </sp:SignedParts>
```

[10] https://axis.apache.org/axis2/java/rampart/samples.html

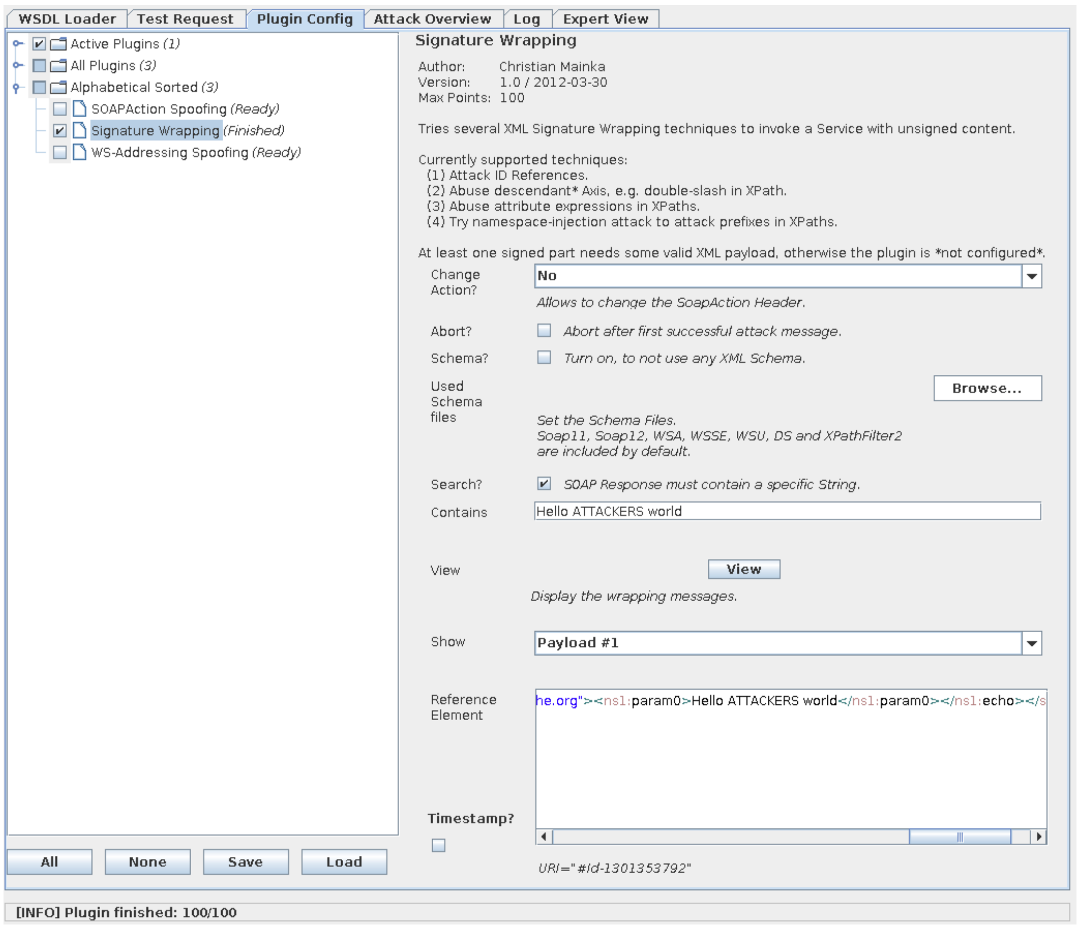
```
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

Listing 12: WS-SecurityPolicy used for the evaluation of the XSW WS-Attacker plugin for attacking Apache Axis2. The structure of the policy file is shortened to the most important parts.

The policy requires the SOAP request to have the following security assertions:

1. It must have a valid `Timestamp` element.
2. The whole `Body` and the `Timestamp` element must be signed.

If such a message successfully bypasses the Rampart security module, the application logic will use the element located in the `Body` and execute the corresponding operation, which is a simple echo operation in this case.



- ☑ Active Plugins *(1)*
- ☐ All Plugins *(3)*
- ☐ Alphabetical Sorted *(3)*
  - ☐ SOAPAction Spoofing *(Ready)*
  - ☑ Signature Wrapping *(Finished)*
  - ☐ WS-Addressing Spoofing *(Ready)*

Signature Wrapping

```
Author:     Christian Mainka
Version:    1.0 / 2012-03-30
Max Points: 100
```

Tries several XML Signature Wrapping techniques to invoke a Service with unsigned content.

Currently supported techniques:
 (1) Attack ID References.
 (2) Abuse descendant* Axis, e.g. double-slash in XPath.
 (3) Abuse attribute expressions in XPaths.
 (4) Try namespace-injection attack to attack prefixes in XPaths.

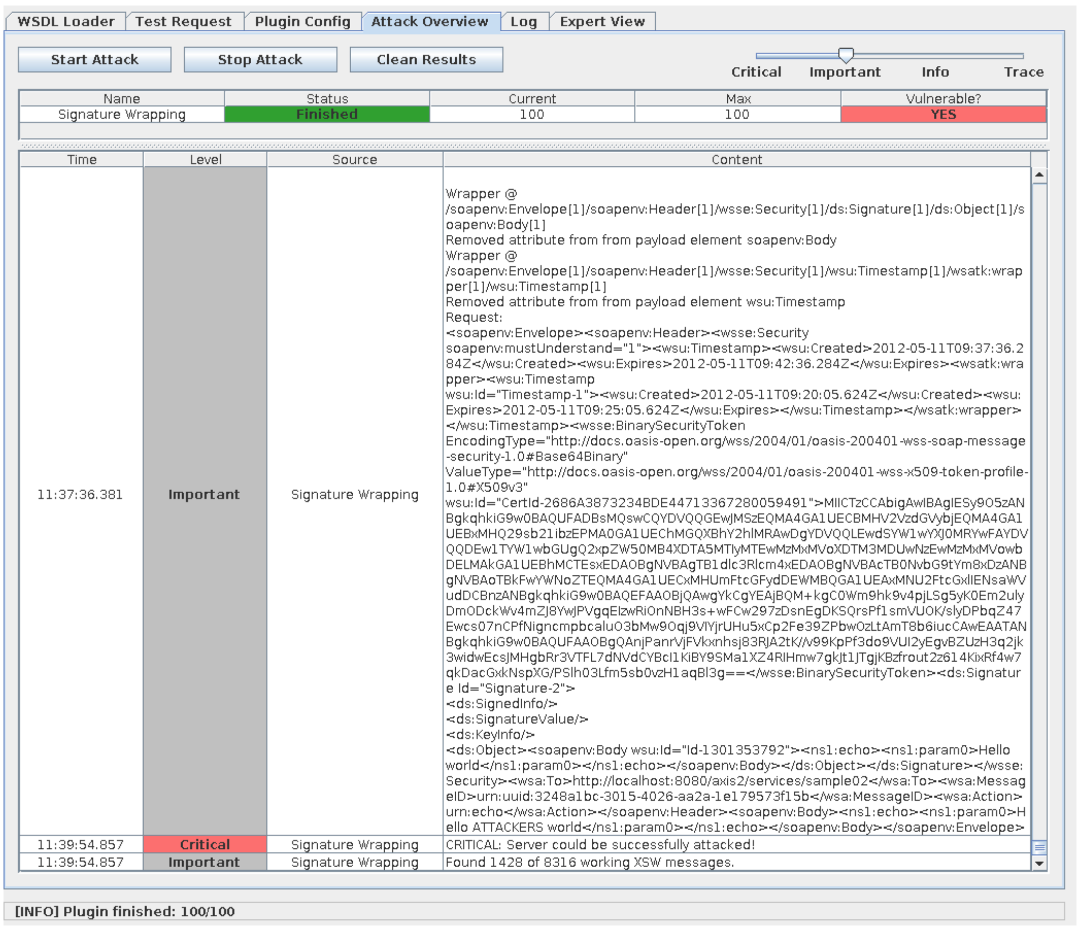
At least one signed part needs some valid XML payload, otherwise the plugin is *not configured*.

Change Action?	No *Allows to change the SoapAction Header.*
Abort?	☐ *Abort after first successful attack message.*
Schema?	☐ *Turn on, to not use any XML Schema.*
Used Schema files	*Set the Schema Files. Soap11, Soap12, WSA, WSSE, WSU, DS and XPathFilter2 are included by default.*
Search?	☑ *SOAP Response must contain a specific String.*
Contains	Hello ATTACKERS world
View	*Display the wrapping messages.*
Show	Payload #1
Reference Element	`he.org"><ns1:param0>Hello ATTACKERS world</ns1:param0></ns1:echo></s`
Timestamp?	☐ *URI="#Id-1301353792"*

[INFO] Plugin finished: 100/100

Figure 31: Configuration of the XSW plugin within the WS-Attacker framework.

Figure 31 shows how to configure the plugin. There is an option for changing the SOAPAction parameter. This is useful if the attacker's payload shall execute an operation which is different to the basic request. The *abort* option allows the plugin to stop after the first XSW message is accepted by the server. In this setup, the option is turned of to get the total number of working attack messages. Furthermore, it is possible to select XML Schema file for the Schema analyzer module. If the user does not upload any file, the plugin uses the Schemas for SOAP, WS-Addressing, XML Signature, WS-Security as default. It is also possible to not use any XML Schemas. This will lead to much more possible XSW messages, as each wrapper element can be placed at arbitrary positions, even at those where they are not allowed. The *search* option enables to specify a string which must be contained in the response as described before. In this case, it is set to the value specified in the payload element. There is also a *Payload #2* element, which can be selected by the drop-down box. It is automatically detected to be a `Timestamp` element and will be updated during the attack (not visible on the screenshot).



Name	Status	Current	Max	Vulnerable?
Signature Wrapping	Finished	100	100	YES

Time	Level	Source	Content
11:37:36.381	Important	Signature Wrapping	
11:39:54.857	Critical	Signature Wrapping	CRITICAL: Server could be successfully attacked!
11:39:54.857	Important	Signature Wrapping	Found 1428 of 8316 working XSW messages.

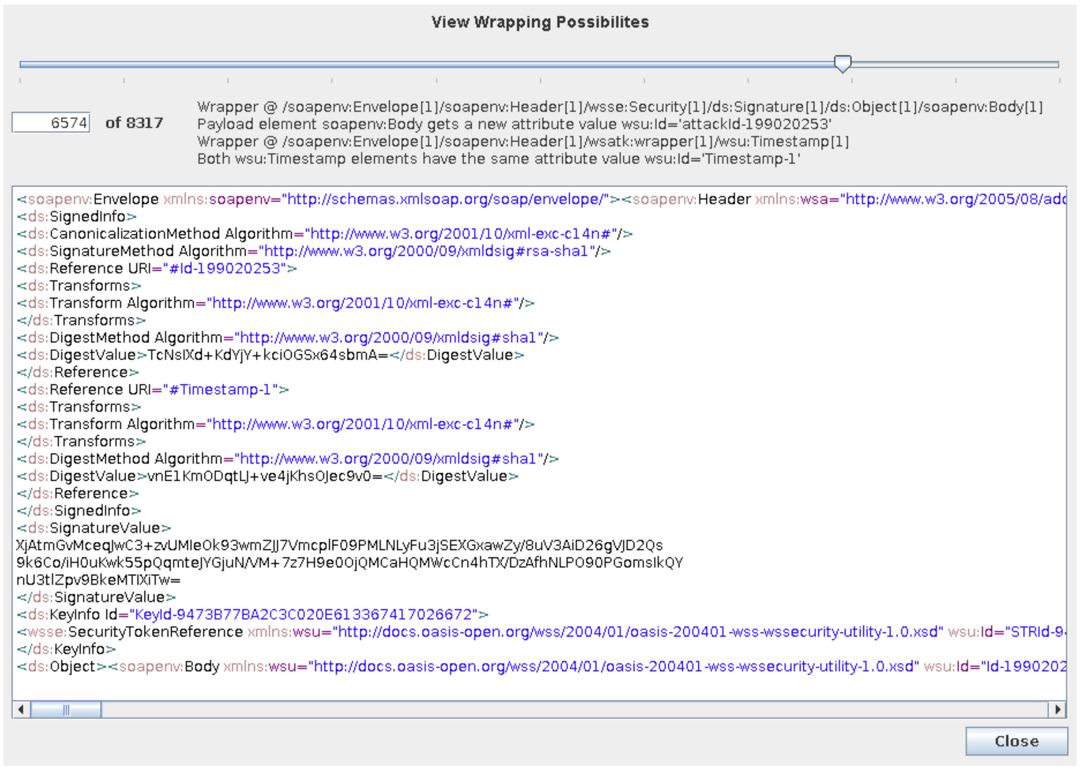
```
Wrapper @
/soapenv:Envelope[1]/soapenv:Header[1]/wsse:Security[1]/ds:Signature[1]/ds:Object[1]/s
oapenv:Body[1]
Removed attribute from from payload element soapenv:Body
Wrapper @
/soapenv:Envelope[1]/soapenv:Header[1]/wsse:Security[1]/wsu:Timestamp[1]/wsatk:wrap
per[1]/wsu:Timestamp[1]
Removed attribute from from payload element wsu:Timestamp
Request:
<soapenv:Envelope><soapenv:Header><wsse:Security
soapenv:mustUnderstand="1"><wsu:Timestamp><wsu:Created>2012-05-11T09:37:36.2
84Z</wsu:Created><wsu:Expires>2012-05-11T09:42:36.284Z</wsu:Expires><wsatk:wra
pper><wsu:Timestamp
wsu:Id="Timestamp-1"><wsu:Created>2012-05-11T09:20:05.624Z</wsu:Created><wsu:
Expires>2012-05-11T09:25:05.624Z</wsu:Expires></wsu:Timestamp></wsatk:wrapper>
</wsu:Timestamp><wsse:BinarySecurityToken
EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message
-security-1.0#Base64Binary"
ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-
1.0#X509v3"
wsu:Id="CertId-2686A3873234BDE44713367280059491">MIICTzCCAbigAwIBAgIESy9O5zAN
BgkqhkiG9w0BAQUFADBsMQswCQYDVQQGEwJMSzEQMA4GA1UECBMHV2VzdGVybjEQMA4GA1
UEBxMHQ29sb21ibzEPMA0GA1UEChMGQXBhY2hlMRAwDgYDVQQLEwdSYW1wYXJ0MRYwFAYDV
QQDEw1TYW1wbGUgQ2xpZW50MB4XDTA5MTIyMTEwMzMxMVoXDTM3MDUwNzEwMzMxMVowb
DELMAkGA1UEBhMCTEsxEDAOBgNVBAgTB1dlc3Rlcm4xEDAOBgNVBAcTB0NvbG9tYm8xDzANB
gNVBAoTBkFwYWNoZTEQMA4GA1UECxMHUmFtcGFydDEWMBQGA1UEAxMNU2FtcGxlIENsaWV
udDCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAjBQM+kgC0Wm9hk9v4pjLSg5yK0Em2uly
DmODckWv4mZJ8YwJPVgqElzwRiOnNBH3s+wFCw297zDsnEgDKSQrsPf1smVUOK/slyDPbqZ47
Ewcs07nCPfNigncmpbcaluO3bMw9Oqj9VlYjrUHu5xCp2Fe39ZPbwOzLtAmT8b6iucCAwEAATAN
BgkqhkiG9w0BAQUFAAOBgQAnjPanrVjFVkxnhsj83RJA2tK/v99KpPf3do9VUI2yEgvBZUzH3q2jk
3widwEcsJMHgbRr3VTFL7dNVdCYBcI1KiBY9SMa1XZ4RlHmw7gkJt1JTgjKBzfrout2z614KixRf4w7
qkDacGxkNspXG/PSlh03Lfm5sb0vzH1aqBl3g==</wsse:BinarySecurityToken><ds:Signatur
e Id="Signature-2">
<ds:SignedInfo/>
<ds:SignatureValue/>
<ds:KeyInfo/>
<ds:Object><soapenv:Body wsu:Id="Id-1301353792"><ns1:echo><ns1:param0>Hello
world</ns1:param0></ns1:echo></soapenv:Body></ds:Object></ds:Signature></wsse:
Security><wsa:To>http://localhost:8080/axis2/services/sample02</wsa:To><wsa:Messag
eID>urn:uuid:3248a1bc-3015-4026-aa2a-1e179573f15b</wsa:MessageID><wsa:Action>
urn:echo</wsa:Action></soapenv:Header><soapenv:Body><ns1:echo><ns1:param0>H
ello ATTACKERS world</ns1:param0></ns1:echo></soapenv:Body></soapenv:Envelope>
```

```
[INFO] Plugin finished: 100/100
```

Figure 32: Results of the XSW plugin.

The results of the final attack can be seen in Figure 32. The plugin detected that both elements are referenced by ID attributes (this is not visible on the screenshot). The server can be attacked, e.g. if the attacker moves the signed `Body` element to the `Object` element within the `Signature` element and furthermore, he has to move the `Timestamp` element to a custom wrapper element located in the updated, new `Timestamp` element. The server response is not visible on this screenshot, but the user can change the logging level to tracing by using the slider located on the top right to display it. In total, the plugin detected that 1428 of 8317 attack messages were successful and each of them can be inspected in detail in WS-Attacker's attack overview window.

As an additional feature, the plugin offers the *view* button, which can be used to inspect all possible XSW messages without sending them to the server.



```
Wrapper @ /soapenv:Envelope[1]/soapenv:Header[1]/wsse:Security[1]/ds:Signature[1]/ds:Object[1]/soapenv:Body[1]
Payload element soapenv:Body gets a new attribute value wsu:Id='attackId-199020253'
Wrapper @ /soapenv:Envelope[1]/soapenv:Header[1]/wsatk:wrapper[1]/wsu:Timestamp[1]
Both wsu:Timestamp elements have the same attribute value wsu:Id='Timestamp-1'
```

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/add
<ds:SignedInfo>
<ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
<ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
<ds:Reference URI="#Id-199020253">
<ds:Transforms>
<ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
</ds:Transforms>
<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
<ds:DigestValue>TcNsIXd+KdYjY+kciOGSx64sbmA=</ds:DigestValue>
</ds:Reference>
<ds:Reference URI="#Timestamp-1">
<ds:Transforms>
<ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
</ds:Transforms>
<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
<ds:DigestValue>vnE1KmODqtLJ+ve4jKhsOJec9v0=</ds:DigestValue>
</ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>
XjAtmGvMceqJwC3+zvUMIeOk93wmZJJ7VmcplF09PMLNLyFu3jSEXGxawZy/8uV3AiD26gVJD2Qs
9k6Co/iH0uKwk55pQqmteJYGjuN/VM+7z7H9e0OjQMCaHQMWcCn4hTX/DzAfhNLPO90PGomsIkQY
nU3tlZpv9BkeMTlXiTw=
</ds:SignatureValue>
<ds:KeyInfo Id="KeyId-9473B77BA2C3C020E61336741 7026672">
<wsse:SecurityTokenReference xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" wsu:Id="STRId-9
</ds:KeyInfo>
<ds:Object><soapenv:Body xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" wsu:Id="Id-1990202
```

Figure 33: Viewing all possible XSW messages.

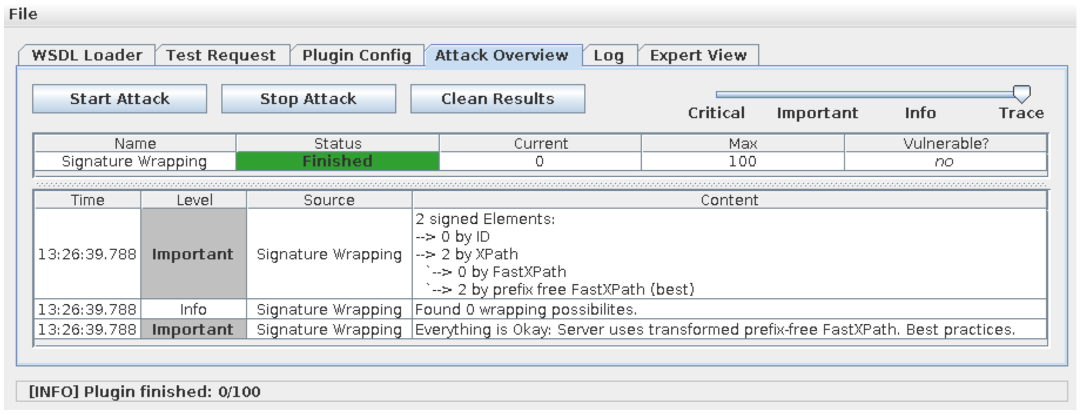
Figure 33 shows this feature. It can be seen, that there are 8317 possible XSW messages. This number has drastically been decreased by using the Schema analyzer. If the plugin does not use it, there are 32401 possibilities. Although this number is much higher, the 8317 messages created with respect to XML Schema is not a subset of the 32401 possibilities created without the Schema analyzer. The message shown in Figure 33 places a wrapper in the `Object` element, which is not contained in the input document, thus, it can only be created when using the XML Schema analyzer.

The idea of this slider is to give the penetration tester the ability to create custom XSW messages for manually attacking a web service. Note that if an attack was successful, the possibility number can be seen on the info logging level. As this number is deterministic, the slider can be used to create this message for a deeper, manual analysis.

6.2.2 Attacking XSpRES

XSpRES is a library for XML spoofing resistant electronic signatures (33). The authors' goal was to create a module for Apache Axis2 which is able to create and verify XML Signatures securely. It is built to resist all known attacks, including the attribute weakness (see Section 4.6.1), the descendant weakness (see Section 4.6.2) and the namespace injection attack (see Section 4.6.3).

The results of the attack can be seen in Figure 34.



Name	Status	Current	Max	Vulnerable?
Signature Wrapping	Finished	0	100	*no*

Time	Level	Source	Content
13:26:39.788	**Important**	Signature Wrapping	2 signed Elements: --> 0 by ID --> 2 by XPath \`--> 0 by FastXPath \`--> 2 by prefix free FastXPath (best)
13:26:39.788	Info	Signature Wrapping	Found 0 wrapping possibilites.
13:26:39.788	**Important**	Signature Wrapping	Everything is Okay: Server uses transformed prefix-free FastXPath. Best practices.

[INFO] Plugin finished: 0/100

Figure 34: Results of the XSpRES attack.

The attack was not successful – the XSW plugin did not even find a single wrapping possibility. Since the authors use transformed prefix-free FastXPath expressions for selecting the signed elements, the SOAP message is not attackable by the namespace injection attack – the XPath does not use any prefix. It also denies the use of the *descendant-** axis, as the basis for the transformed XPath is a FastXPath.

This XML Signature implementation clarifies, that the web service is not attackable by the currently known attack techniques, if the XPath expressions are used carefully and the signature verification- and the application logic are enforced to use the same elements.

6.2.3 Attacking IBM DataPower XI50

In the last scenario, the XSW plugin will be used for attacking the IBM DataPower XI50[11]. The XI50 is an XML Security gateway which is able to sign and verify XML Signatures using XPath expressions. However, it creates the signatures in a unconventional way. Instead of saving the XPath expressions in the corresponding element as a part of an XPathFilter, the DataPower uses ID references in the SOAP messages. The concrete XPaths for creating and verifying XML Signatures are only used in the backend, so that the SOAP message itself does not contain them. It looks like a standard signed SOAP message which uses ID references. Furthermore, the XI50 could not be successfully attacked if the `Timestamp` element is signed. The validation logic always detected, that there are two `Timestamp` elements within the message and throws a SOAP fault.

The setup uses the XPath shown in Listing 13 for signing the message given by Figure 35.

```
//*[namespace-uri()='http://microsoft.com/webservices/' and ↷
   local-name()='GetPrimeNumbers']/*[namespace-uri()='http:// ↷
   microsoft.com/webservices/' and local-name()='max'][1]
```

Listing 13: Used XPath for attacking IBM DataPower XI50.

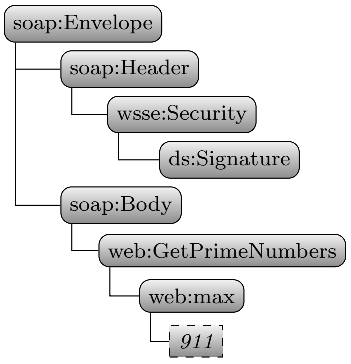


Figure 35: Example message used as the input for the WS-Attacker framework. The signed element `max` is selected by an XPath.

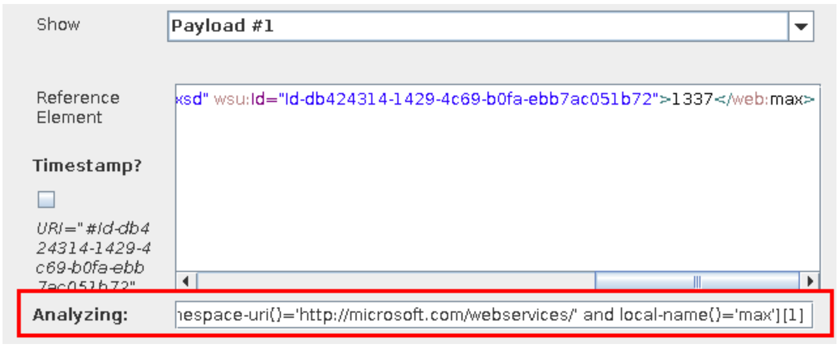
Semantically, the XPath selects a `GetPrimeNumbers` element that has has a `max` child element. This one will be signed. The `GetPrimeNumbers` element can be any descendant

[11] `https://www-01.ibm.com/software/integration/datapower/xi50/`

(or self) of the document root.

As the message itself does not contain any XPath expression, the tool is extended for this attack scenario as follows:

Commonly, the XSW tool would convert the ID reference into an XPath expression of the form //∗`[wsu:Id='x']`. As this is not the XPath used internally by the XI50, the framework user has to set it manually as shown in Figure 36.


Show: Payload #1

Reference Element: `xsd" wsu:Id="Id-db424314-1429-4c69-b0fa-ebb7ac051b72">1337</web:max>`

Timestamp? ☐

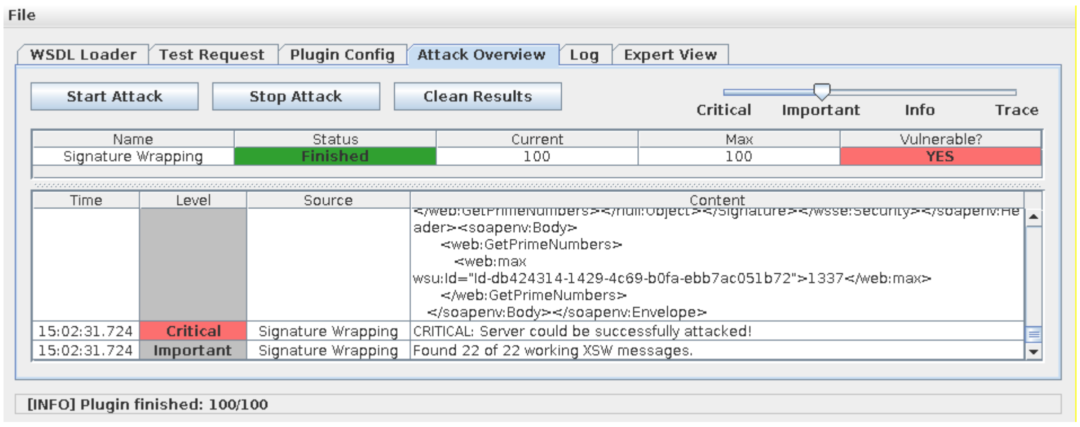
URI="#Id-db424314-1429-4c69-b0fa-ebb7ac051b72"

Analyzing: `nespace-uri()='http://microsoft.com/webservices/' and local-name()='max'][1]`

Figure 36: XSW plugin configuration for attacking IBM DataPower XI50.

The tool will then analyze the custom XPath, which is the same one as the server-side uses for the verification logic. Without this setting, the XSW tool would only try to relocate the `max` element, but for a successful attack, it must have a parent element named `GetPrimeNumber`. This can only be achieved by setting the XPath manually.

The results for the attack can be seen in Figure 37. The attack was successful and each of the 22 wrapping possibilities was accepted.



Name	Status	Current	Max	Vulnerable?
Signature Wrapping	Finished	100	100	YES

Time	Level	Source	Content
			</web:GetPrimeNumbers></null:Object></Signature></wsse:Security></soapenv:Header><soapenv:Body> <web:GetPrimeNumbers> <web:max wsu:Id="Id-db424314-1429-4c69-b0fa-ebb7ac051b72">1337</web:max> </web:GetPrimeNumbers> </soapenv:Body></soapenv:Envelope>
15:02:31.724	Critical	Signature Wrapping	CRITICAL: Server could be successfully attacked!
15:02:31.724	Important	Signature Wrapping	Found 22 of 22 working XSW messages.

```
[INFO] Plugin finished: 100/100
```

Figure 37: XSW plugin results after attacking IBM DataPower XI50.

7 Conclusion

The scenario of XSW is no longer of theoretical nature. Recent attacks on the Amazon EC2 SOAP and the Eucalyptus Cloud web service interfaces have underlined the importance of the weakness.

This thesis presented a tool for the automatic creation of XSW messages, which is integrated into the web services penetration testing framework WS-Attacker. The goal was to build a tool which is able to test a web service implementation against all known XSW variants. To reach this, the attack techniques were analyzed and generic algorithms to apply them were derived. This includes the XPath attribute weakness, the XPath descendant-* axis weakness and the namespace injection attack technique. The program is able to operate with ID based XML Signatures by converting the referencing method to an XPath equivalent and can also handle signatures which use XPathFilter2. By creating Schema valid messages by means of an XML Schema analyzer, the number of possible attack messages can be kept as minimal as possible and additionally, hidden wrapper positions, which are not directly contained in the input message, can be detected.

The tool has been used to attack different web service implementations. Firstly, the Apache Axis2 web service framework which uses the wide-spread Rampart security module was analyzed. Rampart was configured with a default WS-SecurityPolicy which uses ID based XML Signatures and could be successfully attacked. Furthermore, the XSpRES library, which uses prefix-free transformed FastXPaths, was attacked. As the design goal of this library was to prevent all known XSW techniques, it was not very surprising that no XSW weakness could be detected. So, its security could be successfully validated. Lastly, the IBM DataPower XI50, a web service security gateway, was analyzed. The attacks on it turned out to be successful only in specific scenarios, namely if no timestamps are used. The XI50 allowed to use an arbitrary XPath for selecting the signed element, meaning that when the used XPath had a weakness, e.g. it used the descendant axis, it could be successfully attacked.

In a future work, the tool could be extended for attacking XML Signatures which use XPathFilter1 and XPointer. It would also be possible to expand the attack vectors by changing the SOAP version from 1.2 to 1.1, as in this version the `Envelope` element is allowed to have any child element. The current implemented version also only supports one type of ID references: the `wsu:Id` which is specified by the WS-Security standard.

Other XML Signature applications, e.g. a custom developed signature logic which uses different spelled ID attribute, are not supported but can be easily integrated.

The modular structure of the implementation also offers the possibility to create XSW messages apart from the SOAP context, because the creation of such messages basically uses an arbitrary XML document. Thus, the operative range of the tool could be extended to any applications which use XML documents and XML Signatures, e.g. SAML, Microsoft Word documents or XML based databases which use XML Signatures for integrity protection.

Appendix

References

[1] Michael McIntosh and Paula Austel. XML signature element wrapping attacks and countermeasures. In *SWS '05: Proceedings of the 2005 Workshop on Secure Web Services*, pages 20–27, New York, NY, USA, 2005. ACM Press.

[2] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces. In *The ACM Cloud Computing Security Workshop (CCSW)*, October 2011.

[3] Mohammad Ashiqur Rahaman, Andreas Schaad, and Maarten Rits. Towards secure soap message exchange in a soa. In *Proceedings of the 3rd ACM workshop on Secure web services*, SWS '06, pages 77–84, New York, NY, USA, 2006. ACM. ISBN 1-59593-546-0. doi: 10.1145/1180367.1180382. URL `http://doi.acm.org/10.1145/1180367.1180382`.

[4] Karthikeyan Bhargavan, Cedric Fournet, Andrew D. Gordon, and Greg O'Shea. An advisor for Web Services Security policies. In *SWS '05: Proceedings of the 2005 Workshop on Secure Web Services*, pages 1–9, New York, NY, USA, 2005. ACM Press.

[5] Mohammad Ashiqur Rahaman, Rits Marten, and Andreas Schaad. An inline approach for secure soap requests and early validation. OWASP AppSec Europe, 2006.

[6] Lijun Liao, Meiko Jensen, Florian Kohlar, and Nils Gruschka. On interoperability failures in ws-security: The xml signature wrapping attack. *Electronic Business Interoperability: Concepts, Opportunities and Challenges, Information Science Reference*, 2011.

[7] Sebastian Gajek, Meiko Jensen, Lijun Liao, and Jörg Schwenk. Analysis of signature wrapping attacks and countermeasures. In *ICWS*, pages 575–582. IEEE, 2009.

[8] Meiko Jensen, Lijun Liao, and Jörg Schwenk. The curse of namespaces in the domain of xml signature. In Ernesto Damiani, Seth Proctor, and Anoop Singhal, editors, *SWS*, pages 29–36. ACM, 2009. ISBN 978-1-60558-789-9.

[9] Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. Penetration Testing Tool for Web Services Security. In *IEEE 2012 Services Workshop on Security and Privacy Engineering (SPE2012)*, June 2012.

[10] Tim Bray, Jean Paoli, Eve Maler, François Yergeau, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. URL `http://www.w3.org/TR/2008/REC-xml-20081126/`.

[11] Tim Bray, Richard Tobin, Henry S. Thompson, Dave Hollander, and Andrew Layman. Namespaces in XML 1.0 (third edition). W3C recommendation, W3C, December 2009. URL `http://www.w3.org/TR/2009/REC-xml-names-20091208/`.

[12] Richard Tobin, Andrew Layman, Tim Bray, and Dave Hollander. Namespaces in XML 1.1 (second edition). W3C recommendation, W3C, August 2006. URL `http://www.w3.org/TR/2006/REC-xml-names11-20060816`.

[13] C. M. Sperberg-McQueen, Henry S. Thompson, Murray Maloney, Henry S. Thompson, David Beech, Noah Mendelsohn, and Shudi (Sandy) Gao. W3C xml schema definition language (XSD) 1.1 part 1: Structures. Last call WD, W3C, December 2009. URL `http://www.w3.org/TR/2009/WD-xmlschema11-1-20091203/`.

[14] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1, 2000. URL `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`.

[15] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik F. Nielsen, Anish Karmarkar, and Yves Lafon. Soap version 1.2 part 1: Messaging framework (second edition). Technical report, April 2007. URL `http://www.w3.org/TR/soap12-part1/`.

[16] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. W3c note, World Wide Web Consortium, March 2001. URL `http://www.w3.org/TR/wsdl`.

[17] Jean J. Moreau, Roberto Chinnici, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. Candidate recommendation, W3C, March 2006.

[18] T. Dierks and C. Allen. RFC 2246: The TLS protocol version 1. Technical report, January 1999. URL `ftp://ftp.internic.net/rfc/rfc2246.txt`. Status: PROPOSED STANDARD.

[19] Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). *OASIS Standard*, 2006.

[20] Donald Eastlake, Joseph Reagle, Takeshi Imamura, Blair Dillaway, and Ed Simon. XML Encryption Syntax and Processing. *W3C Recommendation*, 2002.

[21] Chris Kaler and Anthony Nadalin. Web Services Security Policy Language (WS-SecurityPolicy) 1.1. 2005.

[22] Asir S. Vedamuthu, David Orchard, Frederick Hirsch, Maryann Hondo, Prasad Yendluri, Toufic Boubez, and Ümit Yalçinalp. Web services policy 1.5 - framework. Technical report, September 2007. URL `http://www.w3.org/TR/ws-policy/`.

[23] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Xml path language (xpath) version 1.0. Technical report, World Wide Web Consortium, 1999. URL `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

[24] Michael Kay, Don Chamberlin, Jonathan Robie, Mary F. Fernández, Jérôme Siméon, Scott Boag, and Anders Berglund. XML path language (XPath) 2.0. W3C recommendation, W3C, January 2007. URL `http://www.w3.org/TR/2007/REC-xpath20-20070123/`.

[25] Donald Eastlake, David Solo, and Joseph Reagle. XML-signature syntax and processing. first edition of a recommendation, W3C, February 2002. URL `http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/`.

[26] Frederick Hirsch, David Solo, Joseph Reagle, Donald Eastlake, and Thomas Roessler. XML signature syntax and processing (second edition). W3C recommendation, W3C, June 2008. URL `http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/`.

[27] John Boyer. Canonical XML version 1.0. W3C recommendation, W3C, March 2001. URL `http://www.w3.org/TR/2001/REC-xml-c14n-20010315`.

[28] Joseph Reagle, Donald E. Eastlake 3rd, and John Boyer. Exclusive XML canonicalization version 1.0. W3C recommendation, W3C, July 2002. URL `http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/`.

[29] James Clark and Steven DeRose. XML path language (XPath) version 1.0. W3C recommendation, W3C, November 1999. URL `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[30] Joseph Reagle, John Boyer, and Merlin Hughes. XML-signature XPath filter 2.0. W3C recommendation, W3C, November 2002. URL `http://www.w3.org/TR/2002/REC-xmldsig-filter2-20021108/`.

[31] Nils Gruschka and Luigi Lo Iacono. Vulnerable Cloud: SOAP Message Security Validation Revisited. In *ICWS '09: Proceedings of the IEEE International Conference on Web Services*, Los Angeles, USA, 2009. IEEE.

[32] Sebastian Gajek, Lijun Liao, and Jörg Schwenk. Breaking and fixing the inline approach. In *Proceedings of the 2007 ACM Workshop on Secure Web Services (SWS'07)*, pages 37–42, Fairfax, Virginia, USA, November 2007. Association for Computing Machinery.

[33] Christian Mainka, Meiko Jensen, Luigi Lo Iacono, and Jörg Schwenk. XSpRES: Robust and Effective XML Signatures for Web Services. In *Closer 2012: 2nd International Conference on Cloud Computing and Services Science*, April 2012.

[34] M. Jensen, C. Meyer, J. Somorovsky, and J. Schwenk. On the effectiveness of xml schema validation for countering xml signature wrapping attacks. In *Securing Services on the Cloud (IWSSC), 2011 1st International Workshop on*, pages 7 –13, September 2011. doi: 10.1109/IWSSCloud.2011.6049019.

[35] The Apache Software Foundation. Apache Axis2 - Next Generation Web Services, . URL `http://ws.apache.org/axis2/`.

[36] The Apache Software Foundation. Apache Rampart - Axis2 Security Module, . URL `https://axis.apache.org/axis2/java/rampart/`.

[37] Christian Mainka, Meiko Jensen, Juraj Somorovsky, and Jörg Schwenk. Ws-attacker. URL `http://sourceforge.net/projects/ws-attacker/`.

List of Figures

Listings

Glossary

A

Apache Axis2 Apache eXtensible Interaction System, commonly used web service framework created by the Apache Software Foundation (35).. 6, 61, 62, 65, 70, 78

C

Cloud Cloud Computing, using shared ressources to compute as a service rather than a product.. 2, 5, 22, 70

E

E-Government electronic government, digital interactions between citizens and the government.. 2, 5

I

IBM DataPower XI50 web service security gateway by IBM.. 6, 61, 66–68, 70, 77, 78

N

Namespace used to clearly identify XML elements and attributes (11, 12).. 5, 8, 9, 11, 13, 17, 21, 27–30, 44, 47, 48, 76

P

Parser A parser syntatically analyzes a text and seperates it into tokens, e.g. words. In the context of XML, the tokens are *nodes* like *elements*, *attributes* and *text contents*.. 9

- **DOM** The DOM parser reads the whole XML document into memory and builds an object for each node.. 9, 26
- **SAX** The *Simple API for XML* parses an XML stream and sends an event if a new element starts or ends.. 9, 26, 30
- **StAX** The StAX parser is a *pull* parser and thus works like a cursor: the programmer can ask for the next event, e.g. next element start.. 9, 26, 30

Penetration test method for evaluating security on computer systems.. 2, 6, 51, 54, 55, 61, 65, 70

R

Rampart Apache Axis2 security module. Can be used for securing SOAP messages according to the WS-Security specifications (36).. 6, 61, 62, 70

S

SAML Security Assertion Markup Language. 71

SOA Service Oriented Architecture, abstract model of software architecture.. 2, 5, 11

SOAP SOAP is a standard which describes message exchange with a web service (14, 15).. 2, 5, 6, 11, 12, 15, 21–23, 27, 28, 31–35, 37, 42, 43, 47, 49, 51, 52, 54, 56, 57, 59, 62, 66, 70, 71, 76, 77

W

W3C World Wide Web Consortium, organization for standards in the World Wide Web.. 9, 14, 16

Web service concrete implementation of SOA.. 2, 5–7, 11, 12, 19, 31, 35, 46, 50, 51, 55–59, 61, 65, 66, 70, 76

WS-Attacker automatic penetration test framework (37).. 2, 6, 55, 58, 59, 61, 62, 64, 67, 70, 77, 78

WSDL Web Services Description Language, specification for creating a client's web service message (16, 17).. 11, 56

WS-Policy specifies how to add policies to a WSDL (22).. 13, 22

WS-Security extension for SOAP to specify security in web services (19).. 12, 15, 22, 31, 34, 48, 62, 70

WS-SecurityPolicy specifies security policy assertions for WS-Security (21).. 13, 22, 30, 58, 61, 62, 70, 78

X

XML eXtended Markup Language, textual data format to encode documents, commonly used for message exchange (10).. 5–9, 15–17, 65, 71, 76, 79

XML document general term for a document which uses XML as description language, e.g. a SOAP message.. 2, 5, 7–9, 12–20, 22–26, 31–33, 35–37, 40, 42, 43, 45, 47, 49–51, 65, 67, 71, 76

XML Canonicalization Process to convert an XML document into a normalized representation.. 5, 15, 16

Exclusive Canonicalization Canonicalization method which includes as little as possible namespace declarations.. 17, 28, 29, 47

Inclusive Canonicalization Canonicalization method which includes all visible namespace declarations in every element.. 17

XML Encryption standard for encryption of XML documents (20).. 12

XML Schema well-founded commendation from W3C to define the structure of an XML document (13).. 9–11, 30, 31, 35–37, 43, 44, 58, 60, 62, 65, 70, 77

XML Security generic term for security features in XML.. 12, 35, 66

XML Signature also XML Digital Signature, standard for creating signatures in XML documents (25, 26).. 2, 5–7, 12, 14–20, 22, 23, 25, 28, 29, 31–35, 37, 38, 42, 43, 47, 51, 52, 58, 60, 62, 65, 66, 70, 71, 76

XPath XML standard for selecting parts of XML Documents (23, 24).. 5, 9, 13, 14, 17–19, 23–28, 30, 32–34, 37–40, 42, 43, 45–47, 49, 51–55, 58, 59, 66–68, 70, 76–78

FastXPath subset of the XPath grammar which only allowes forward referencing (7).. 5, 14, 26, 28, 30, 58, 66, 70, 77

PostXPath part of an XPath after a specified step, e.g. d/e in /a/b/c/d/e for step c. 14, 39, 42, 43

PreXPath part of an XPath before a specified step, e.g. /a/b in /a/b/c/d/e for step c. 14, 39, 42, 43

XPathFilter specification to describe how to select elements for cryptographic primitives within an XML document (29, 30).. 5, 17–19, 23, 59, 66, 70, 76

XSpRES XML Spoofing Resistant Electronic Signatures, proof-of-concept security libraray for Apache Axis2 which is designd to resist all known XSW techniques (33). 6, 61, 65, 66, 70, 77

XSW XML Signature Wrapping, technique for attacking signed web services (1).. 2, 5, 6, 20–25, 28, 30, 32–35, 38–43, 45–47, 49–51, 53–59, 61–66, 68, 70, 71, 76–78

Attribute weakness XSW technique which abuses the attribute selection of an XPath.. 38, 40, 46, 49, 54, 61, 65, 70

Descendant weakness XSW technique which abuses the *descendant-** axis selection of an XPath.. 42, 45, 46, 49, 54, 65, 70

Namespace injection XSW technique which overrides namespace declarations in XPath namespace resolution.. 5, 27, 29, 46, 47, 49, 53, 54, 58, 61, 65, 66, 70, 76

Eigenständigkeitserklärung

Hiermit versichere ich, Christian Mainka (Matrikelnummer: 108007212667), dass ich die Arbeit selbständig angefertigt, außer den im Quellen- und Literaturverzeichnis sowie in den Anmerkungen genannten Hilfsmitteln keine weiteren benutzt und alle Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, unter Angabe der Quellen als Entlehnung kenntlich gemacht habe.

Ort, Datum	Christian Mainka