On Breaking SAML: Be Whoever You Want to Be

Juraj Somorovsky[1], Andreas Mayer[2], Jörg Schwenk[1], Marco Kampmann[1], and Meiko Jensen[1]

[1]Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
[2]Adolf Würth GmbH & Co. KG, Künzelsau-Gaisbach, Germany
{*Juraj.Somorovsky, Joerg.Schwenk, Marco.Kampmann, Meiko.Jensen*}*@rub.de*,
Andreas.Mayer@wuerth.com

Abstract

The **S**ecurity **A**ssertion **M**arkup **L**anguage (*SAML*) is a widely adopted language for making security statements about subjects. It is a critical component for the development of federated identity deployments and Single Sign-On scenarios. In order to protect integrity and authenticity of the exchanged SAML assertions, the XML Signature standard is applied. However, the signature verification algorithm is much more complex than in traditional signature formats like PKCS#7. The integrity protection can thus be successfully circumvented by application of different XML Signature specific attacks, under a weak adversarial model.

In this paper we describe an in-depth analysis of 14 major SAML frameworks and show that 11 of them, including Salesforce, Shibboleth, and IBM XS40, have critical XML Signature wrapping (XSW) vulnerabilities. Based on our analysis, we developed an automated penetration testing tool for XSW in SAML frameworks. Its feasibility was proven by additional discovery of a new XSW variant. We propose the first framework to analyze such attacks, which is based on the information flow between two components of the Relying Party. Surprisingly, this analysis also yields efficient and practical countermeasures.

1 Introduction

The **S**ecurity **A**ssertion **M**arkup **L**anguage (*SAML*) is an XML based language designed for making security statements about subjects. SAML assertions are used as security tokens in WS-Security, and in REST based Single Sign-On (SSO) scenarios. SAML is supported by major software vendors and open source projects, and is widely deployed. Due to its flexibility and broad support, new application scenarios are defined constantly.

SAML ASSERTIONS. Since SAML assertions contain security critical claims about a subject, the validity of these claims must be certified. According to the standard, this shall be achieved by using XML Signatures, which should either cover the complete SAML assertion, or an XML document containing it (e.g. a SAML Authentication response).

However, roughly 80% of the SAML frameworks that we evaluated could be broken by circumventing integrity protection with novel XML Signature wrapping (XSW) attacks. This surprising result is mainly due to two facts:

- **Complex Signing Algorithm:** Previous digital signature data formats like PKCS#7 and OpenPGP compute a single hash of the whole document, and signatures are simply appended to the document. The XML Signature standard is much more complex. Especially, the position of the signature and the signed content is variable. Therefore, many permutations of the same XML document exist.
- **Unspecified internal interface:** Most SAML frameworks treat the Relying Party (i.e. the Web Service or website consuming SAML assertions) as a single block, assuming a joint common state for all tasks. However, logically this block must be subdivided into the signature verification module (later called RP_{sig}) which performs a cryptographic operation, and the SAML processing module (later called RP_{claims}) which processes the claims contained in the SAML assertion. Both modules have different views on the assertion, and they typically only exchange a Boolean value about the validity of the signature.

CONTRIBUTION. In this paper, we present an in-depth analysis of 14 SAML frameworks and systems. During this analysis, we found critical XSW vulnerabilities in 11 of these frameworks. This result is alarming given the importance of SAML in practice, especially since SSO frameworks may become a single point of attack. It clearly indicates that the security implications behind SAML and XML Signature are not understood yet.

Second, these vulnerabilities are exploitable by an attacker with far fewer resources than the classical network based attacker from cryptography: Our adversary may succeed even if he does not control the network. He does not need realtime eavesdropping capabilities, but can work with SAML assertions whose lifetime has expired. A single signed SAML assertion is sufficient to completely compromise a SAML issuer/Identity Provider. Using SSL/TLS to encrypt SAML assertions, and thus to prevent adversaries from learning assertions by intercepting network traffic, does not help either: The adversary may e.g. register as a regular customer at the SAML issuer, and may use his own assertion to impersonate other customers.

Third, we give the first model for SAML frameworks that takes into account the interface between RP_{sig} and RP_{claims}. This model gives a clear definition of successful attacks on SAML. Besides its theoretical interest, it also enables us to prove several *positive* results. These results are new and help to explain why some of the frameworks were not vulnerable to our attacks, and to give advice on how to improve the security of the other 11 frameworks.

Last, we show that XSW vulnerabilities constitute an important and broad class of attack vectors. There is no easy defense against XSW attacks: Contrary to common belief, even signing the whole document does not necessarily protect against them. To set up working defenses, a better understanding of this versatile attack class is required. A specialized XSW pentesting tool developed during our research will be released as open source to aid this understanding. Its practicability was proven by discovering a new attack vector on Salesforce SAML interface despite the fact that specific countermeasures have been applied.

Responsible Disclosure. All vulnerabilities found during our analysis were reported to the responsible security teams. Accordingly, in many cases, we closely collaborated with them in order to patch the found issues.

Outline. The rest of the paper is organized as follows. Section 2 gives a highlevel overview on SAML, and Section 3 adds details. The methodology of the investigation is explained in Section 4, and the detailed results are described in Section 5. In Section 6 we present the first fully automated XSW penetration test tool for SAML. Section 7 gives a formal analysis and derives two countermeasures. In Section 8 we discuss their practical feasibility. Section 9 presents an overview on related work. In the last section we conclude and propose future research directions.

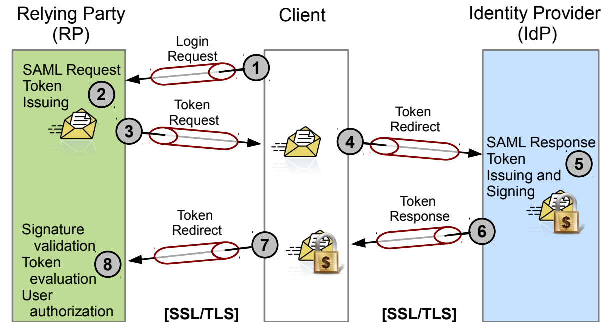

Figure 1: A typical Single Sign-On scenario: The user visits the RP, which generates a request token. He redirects this token to the IdP. The issued token is sent to the user and forwarded to the RP. Even though the channel is secured by SSL/TLS, the user still can see the token.

2 Motivation

In this section we introduce two typical SAML scenarios and some widely used SAML frameworks.

SAML-based Single Sign-On. Typical Internet users have to manage many identities for different web applications. To overcome this problem, Single Sign-On was developed. In this approach the users authenticate only once to a trustworthy Identity Provider (IdP). After a successful login of a user, the IdP issues security tokens on demand. These tokens are used to authenticate to Relying Parties (RP).

A simplified Single Sign-On scenario is depicted in Figure 1. In this setting, a user logged-in by the IdP first visits the desired RP (1). The RP issues a token request (2). This token is sent to the user (3) who forwards it to the IdP (4). The IdP issues a token response for the user including several claims (e.g. his access rights or expiration time). In order to protect the authenticity and integrity of the claims, the token is signed (5). Subsequently, the token is sent to the user (6), who forwards it to the RP (7). The RP validates the signature and afterwards grants access to the protected service or resource, if the user is authorized (8). This access control decision is based on the claims in the validated token.

Securing Web Services with SAML. Another typical application scenario is the use of SAML together with WS-Security [29] in SOAP [21] to provide authentication and authorization mechanisms to Web Services. SAML assertions are included as security tokens in the `Security` header.

SAML Providers and Frameworks. The evaluation presented in this paper was made throughout the last 18 months and includes prominent and well-used SAML frameworks, which are summarized in Table 1. Our analysis included the IBM hardware appliance

Framework/Provider	Type	Language	Reference	Application
Apache Axis 2	WS	Java	http://axis.apache.org	WSO2 Web Services
Guanxi	Web SSO	Java	http://guanxi.sourceforge.net	Sakai Project (www.sakaiproject.org)
Higgins 1.x	Web SSO	Java	www.eclipse.org/higgins	Identity project
IBM XS40	WS	XSLT	www.ibm.com	Enterprise XML Security Gateway
JOSSO	Web SSO	Java	www.josso.org	Motorola, NEC, Redhat
WIF	Web SSO	.NET	http://msdn.microsoft.com	Microsoft Sharepoint 2010
OIOSAML	Web SSO	Java, .NET	http://www.oiosaml.info	Danish eGovernment (e.g. www.virk.dk)
OpenAM	Web SSO	Java	http://forgerock.com/openam.html	Enterprise-Class Open Source SSO
OneLogin	Web SSO	Java, PHP, Ruby, Python	www.onelogin.com	Joomla, Wordpress, SugarCRM, Drupal
OpenAthens	Web SSO	Java, C++	www.openathens.net	UK Federation (www.eduserv.org.uk)
OpenSAML	Web SSO	Java, C++	http://opensaml.org	Shibboleth, SuisseID
Salesforce	Web SSO	—	www.salesforce.com	Cloud Computing and CRM
SimpleSAMLphp	Web SSO	PHP	http://simplesamlphp.org	Danish e-ID Federation (www.wayf.dk)
WSO2	Web SSO	Java	www.wso2.com	eBay, Deutsche Bank, HP

Table 1: Analyzed SAML frameworks and providers: The columns give information about type (Web Service or Browser-based SSO), programming language (if known), web site, and application in concrete products or frameworks.

XS40, which is applied as an XML Security Gateway. Other examples of closed source frameworks are the Windows Identity Foundation (WIF) used in Microsoft Sharepoint and the Salesforce cloud platform. Important open source frameworks include OpenSAML, OpenAM, OIOSAML, OneLogin, and Apache Axis 2. OpenSAML is for example used in Shibboleth and the SDK of the electronic identity card from Switzerland (SuisseID). OpenAM, formerly known as SUN OpenSSO, is an identity and access management middleware, used in major enterprises. The OIOSAML framework is e.g. used in Danish public sector federations (e.g. eGovernment business and citizen portals). The OneLogin Toolkits integrate SAML into various popular open source web applications like Wordpress, Joomla, Drupal, and SugarCRM. Moreover, these Toolkits are used by many OneLogin customers (e.g. Zendesk, Riskonnect, Zoho, KnowledgeTree, and Yammer) to enable SAML-based SSO. Apache Axis2 is the standard framework for generating and deploying Web Service applications.

3 Technical Foundations

In this section we briefly introduce the SAML standard and XML Signature wrapping attacks. Additionally, for readers unfamiliar with the relevant W3C standards, we present XML Signature [14] and XML Schema [36].

3.1 XML Signature

The XML Signature standard [14] defines the syntax and processing rules for creating, representing, and verifying XML-based digital signatures. It is possible to sign a whole XML tree or only specific elements. One XML Signature can cover several local or global resources. A signature placed within the signed content is called an enveloped signature. If the signature surrounds the signed parts, it is an enveloping signature. A detached signature is neither inside nor a parent of the signed data.

```
<Signature ID?>
   <SignedInfo>
      <CanonicalizationMethod/>
      <SignatureMethod/>
      (<Reference URI? >
         (<Transforms>)?
         <DigestMethod>
         <DigestValue>
       </Reference>)+
   </SignedInfo>
   <SignatureValue>
   (<KeyInfo>)?
   (<Object ID?>)*
</Signature>
```

Figure 2: XML Signature data structure ("?": zero or one occurrence; "+": one or more occurrences; "*": zero or more occurrences).

An XML Signature is represented by the `Signature` element. Figure 2 provides its basic structure. XML Signatures are two-pass signatures: the hash value of the resource (`DigestValue`) along with the used hash algorithm (`DigestMethod`) and the `URI` reference to the resource are stored in a `Reference` element. Additionally, the `Transforms` element specifies the processing steps which are applied prior to digesting of the resource. Each signed resource is represented by a `Reference` element in the `SignedInfo` element. Therefore, `SignedInfo` is a collection of hash values and `URI`s. The `SignedInfo` itself is protected by the signature. The `CanonicalizationMethod` and the `SignatureMethod` element specify the algorithms used for canonicalization and signature creation, and are also embedded in `SignedInfo`. The Base64-encoded value of the computed signature is deposited in the `SignatureValue` element. In addition, the `KeyInfo` element facilitates the transport of signature relevant key management information. The `Object` is an optional element that may contain any data.

```
<saml:Assertion Version ID IssueInstant>
  <saml:Issuer>
  <ds:Signature>?
  <saml:Subject>?
  <saml:Conditions>?
  <saml:Advice>?
  <saml:AuthnStatement>*
  <saml:AuthzDecisionStatement>*
  <saml:AttributeStatement>*
</saml:Assertion>
```

Figure 3: SAML assertion structure.

3.2 XML Schema

The W3C recommendation XML Schema [36] is a language to describe the layout, semantics, and content of an XML document. A document is deemed to be *valid*, when it conforms to a specific schema. A schema consists of a content model, a vocabulary, and the used data types. The content model describes the document structure and the relationship of the items. The standard provides 19 primitive data types to define the allowed content of the elements and attributes.

Regarding to our evaluation of SAML based XML Signature Wrapping attacks there is one important element definition in XML Schema. The `any` element allows the usage of any well-formed XML document in a declared content type. When an XML processor validates an element defined by an `any` element, the `processContents` attribute specifies the level of flexibility. The value `lax` instructs the schema validator to check against the given namespace. If no schema information is available, the content is considered valid. In the case of `processContents="skip"` the XML processor does not validate the element at all.

3.3 SAML

SAML is an XML standard for exchanging authentication and authorization statements about *Subjects* [11]. Several profiles are defined in [10]. The most important profile is the Browser SSO profile, which defines how to use SAML with a web browser.

A SAML assertion has the structure described in Figure 3. The issuing time of the assertion is specified in `saml:IssueInstant`. All attributes are required.

The `saml:Issuer` element specifies the SAML authority (the IdP) that is making the claim(s) in the assertion. The assertion's `saml:Subject` defines the principal about whom all statements within the assertion are made. The `saml:*Statement` elements are used to specify user-defined statements relevant for the context of the SAML assertion.

To protect the integrity of the security claims made by the Issuer, the whole `saml:Assertion` element must be protected with a digital signature following the XML

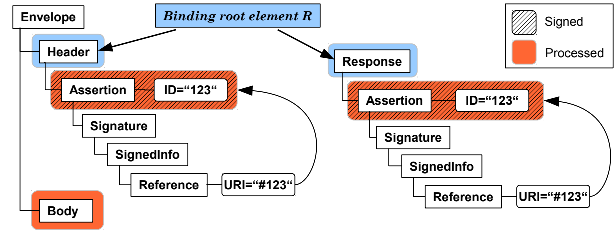

Figure 4: SAML message examples (SOAP and REST): The SAML assertion is put into a root element *R* and signed using an enveloped signature. When signing the SOAP body, an additional detached signature is used.

Signature standard. Therefore, the SAML specification [11] requires that either the `saml:Assertion` element or an ancestor element must be referenced by the `Signature` element, with an *enveloped* XML Signature ([11], Section 5.4.1). Furthermore, `Id`-based referencing must be used ([11], Section 5.4.2), which opens the way for XSW attacks.

In REST based frameworks, the SAML assertion is typically put into an enveloping `Response` element. Frameworks applying SOAP insert the SAML assertions into the SOAP header (or the `Security` element inside of the SOAP header). For clarification purposes, consider that the SAML assertions are signed using *enveloped* XML Signatures and are put into some binding root element *R* (see Figure 4).

3.4 XML Signature Wrapping Attacks

XML documents containing XML Signatures are typically processed in two independent steps: signature validation and function invocation (business logic). If both modules have different views on the data, a new class of vulnerabilities named *XML Signature Wrapping attacks (XSW)* [27, 23] exists. In these attacks the adversary modifies the message structure by injecting forged elements which do not invalidate the XML Signature. The goal of this alteration is to change the message in such a way that the application logic and the signature verification module use different parts of the message. Consequently, the receiver verifies the XML Signature successfully but the application logic processes the bogus element. The attacker thus circumvents the integrity protection and the origin authentication of the XML Signature and can inject arbitrary content. Figure 5 shows a simple XSW attack on a SOAP message.

XSW attacks resemble other classes of injection attacks like XSS or SQLi: in all cases, the attacker tries to force different views on the data in security modules (e.g. Web Application Firewalls) and data processing modules (HTML parser, SQL engine).

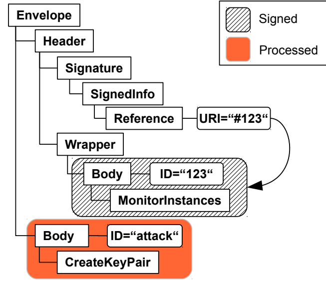

Figure 5: A simple XML Signature wrapping attack: The attacker moves the original signed content to a newly created `Wrapper` element. Afterwards, he creates an arbitrary content with a different `Id`, which is invoked by the business logic.

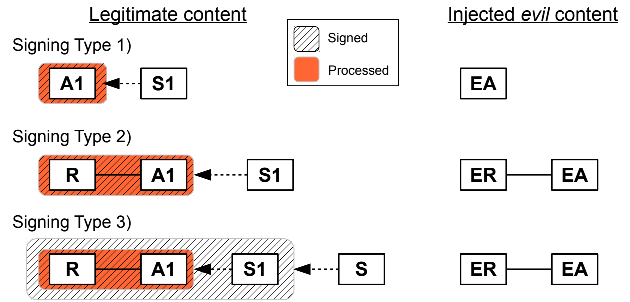

Figure 6: Types of signature applications on SAML assertions on the left. The new malicious content needed to execute the attacks depicted on the right, accordingly.

4 XSW Attacks on SAML

In this section we first characterize the assumed threat model. Second, we describe the basic attack principle that underlies our analysis of the 14 frameworks.[1]

4.1 Threat Model

As a prerequisite the attacker requires an arbitrary signed SAML message. This could be a single assertion A or a whole document D with an embedded assertion, and its lifetime can be expired. After obtaining such a message, the attacker modifies it by injecting evil content, e.g. an evil assertion EA. In our model we assume two different types of adversaries, which are both weaker than the classical network based attacker:

1. Adv_{acc}. To obtain an assertion, this attacker registers as a user of an Identity Provider IdP. Adv_{acc} then receives, through normal interaction with IdP, a valid signed SAML assertion A (probably as a part of a larger document D) making claims about Adv_{acc}. The attacker now adds additional claims EA about any other subject S, and submits the modified document D' (A') to RP.
2. Adv_{intc}. This adversary retrieves SAML assertions from the Internet, but he does not have the ability to read encrypted network traffic. This can be done either by accessing transmitted data directly from unprotected networks (sniffing), or in an ”offline” manner by analyzing proxy or browser caches. Since SAML assertions should be worthless once their lifetime expired, they may even be posted in technical discussion boards, where Adv_{intc} may access them.

[1] Please note that from now on we distinguish between the document D and the root element R. This is to make clear the distinction between the element referenced by the XML signature, and the document root: Even if the root element R of the original document D is signed, we may transform this into a new document D' with a new evil root ER, without invalidating the signature.

4.2 Basic Attack Principle

As described in the previous section, XML Signatures can be applied to SAML assertions in different ways and placed in different elements. The only prerequisite is that the `Assertion` element or the protocol binding element (ancestor of `Assertion`) is signed using an enveloped signature with `Id`-based referencing. In this section we analyze the usage of SAML assertions in different frameworks and the possibilities of inserting malicious content. Generally, SAML assertions and their signatures are implemented as depicted in Figure 6:

1. The first possible usage of signatures in SAML assertions is to insert the XML Signature *S1* as a child of the SAML assertion *A1* and sign only the `Assertion` element *A1*. This type can be used independently of the underlying protocol (SOAP or REST).
2. The second type of signature application in SAML signs the whole protocol binding element R. The XML Signature can be placed into the SAML assertion *A1* or directly into the protocol binding root element R. This kind of signature application is used in different SAML HTTP bindings, where the whole `Response` element is signed.
3. It is also possible to use more than one XML Signature. The third example shows this kind of signature application: the inner signature *S1* protects the SAML assertion and the outer signature S additionally secures the whole protocol message. This kind of signature application is e.g. used by the SimpleSAMLphp framework.

In order to apply XSW attacks to SAML assertions, the basic attack idea stays the same: The attacker has

to create new malicious elements and force the assertion logic to process them, whereas the signature verification logic verifies the integrity and authenticity of the original content. In applications of the first signature type, the attacker only has to create a new *evil assertion EA*. In the second and third signing types, he also has to create the whole *evil root ER* element including the *evil assertion*.

4.3 Attack Permutations

The attacker has many different possibilities where to insert the malicious and the original content. To this end, he has to deal with these questions:

- At which level in the XML message tree should the malicious content and the original signed data be included?
- Which `Assertion` element is processed by the assertion logic?
- Which element is used for signature verification?

By answering these questions we can define different attack patterns, where the original and the malicious elements can be permuted (Figure 7). We thus get a complete list of attack vectors, which served as a guideline for our investigations.

For the following explanations we only consider signing type 1) defined in Figure 6. In this signing type only the `Assertion` element is referenced.

The attack permutations are depicted in Figure 7. In addition, we analyze their SAML standard conformance and the signature validity:

1. Malicious assertion, original assertion, and signature are left on the same message level: This kind of XML message can have six permutations. None of them is SAML standard compliant, since the XML Signature does not sign its parent element. The digest value over the signed elements in all the messages can be correctly validated. We can use this type of attack messages if the server does not check the SAML conformance.
2. All the three elements are inserted at different message levels, as child elements of each other, which again results in six permutations: Messages 2-a and 2-b show examples of SAML standard conforming and cryptographically valid messages. In both cases the signature element references its parent – the original assertion *A1*. Message 2-c illustrates a message which is not SAML standard conform as the signature signs its child element. Nevertheless, the message is cryptographically valid. Lastly, message 2-d shows an example of an invalid message since the signature would be verified over both assertions. Generally, if the signature is inserted as the child of the root element, the message would also be either invalid or not SAML standard compliant.
3. For the insertion of these three elements we use two message levels: Message 3-a shows an example of a valid and SAML compliant document. By constructing message 3-b, the signature element was moved to the new malicious assertion. Since it references the original element, it is still valid, but does not conform to the SAML standard.

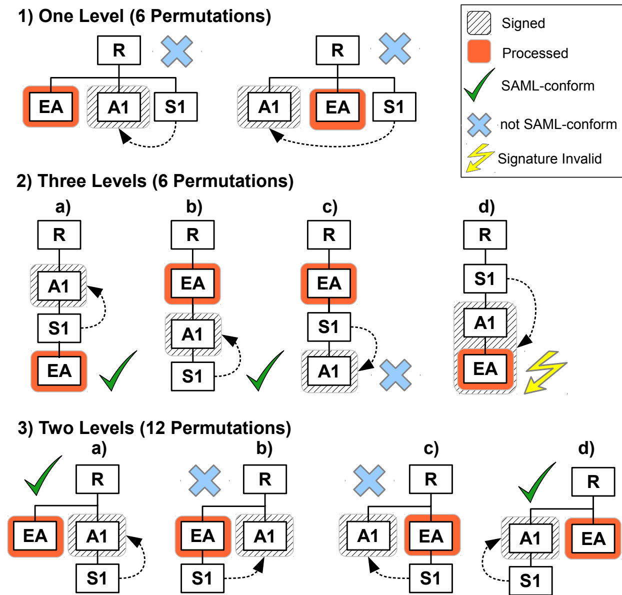

Figure 7: Possible variants for XSW attacks applied on messages with one signed SAML assertion divided according to the insertion depth of the evil assertion *EA*, the original assertion *A1* and the signature *S1*. The various permutations are labeled according to their validity and SAML-conformance.

The analysis shown above can similarly be applied to messages with different signing types (see Figure 6).

5 Practical Evaluation

We evaluated the above defined attacks on real-world systems and frameworks introduced in Section 2. In this section we present the results.

5.1 Signature Exclusion Attacks

We start the presentation of our results with the simplest attack type called *Signature exclusion attack*. This attack relies on poor implementation of a server's security logic, which checks the signature validity only if the signature is included. If the security logic does not find the `Signature` element, it simply skips the validation step.

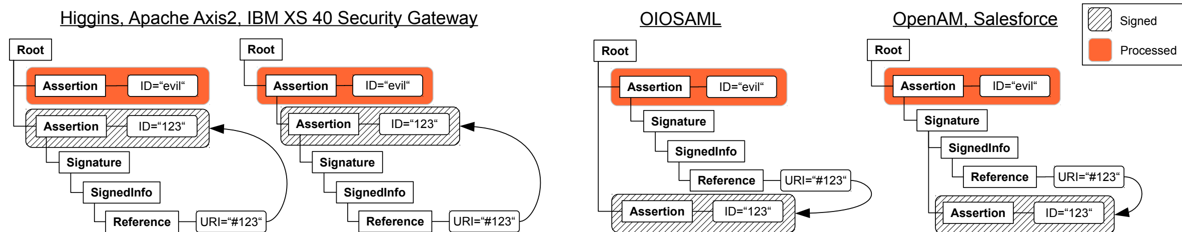

Figure 8: XML tree-based illustration of refined XSW attacks found in Type 1 signature applications.

The evaluation showed that three SAML-based frameworks were vulnerable to these attacks: Apache Axis2 Web Services Framework, JOSSO, and the Java-based implementation of SAML 2.0 in Eduserv (other versions of SAML and the C-implementation in Eduserv were not affected).

By applying this attack on JOSSO and Eduserv the attacker had to remove the `Signature` element from the message, since if it was found, the framework tried to validate it. On the other hand, the Apache Axis2 framework did not validate the `Signature` element over the SAML assertion at all, even if it was included in the message. Apache Axis2 validated only the signature over the SOAP body and the `Timestamp` element. The signature protecting the SAML assertion, which is included separately in the `Assertion` element, was completely ignored.

5.2 Refined Signature Wrapping

Ten out of 14 systems were prone to refined XSW attacks.

Classified on the three different signature application types given in Figure 6, five SAML-based systems failed in validating Type 1 messages, where only the assertion is protected by an XML Signature. Figure 8 depicts the XML tree-based illustration of the found XSW variants. Starting from left to right, Higgins, Apache Axis2, and the IBM XS 40 Security Gateway were outfoxed by the two depicted permutations. In the first variant it was sufficient to inject an evil assertion with a different `Id` attribute in front of the original assertion. As the SAML standard allows to have multiple assertions in one protocol element, the XML Schema validation still succeeded. The second attack type embedded the original assertion as a child element into the evil assertion *EA*. In both cases the XML Signature was still standard conform, as enveloped signatures were applied. This was broken in the case of OIOSAML by using detached signatures. In this variant the original `Signature` element was moved into the *EA*, which was inserted before the legitimate assertion. The last shown permutation was applicable to the cloud services of Salesforce

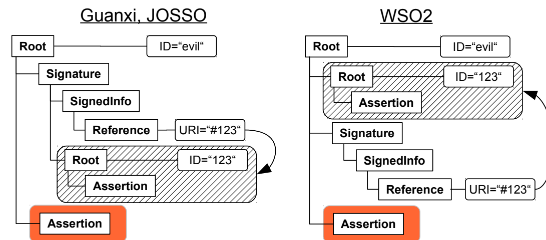

Figure 9: XML tree-based illustration of refined XSW attacks found in Type 2 signature applications.

and the OpenAM framework. At this, the genuine assertion was placed into the original `Signature` element. As both implementations apply XML Schema for validating the schema conformance of a SAML message, this was done by injecting them into the `Object` element, which allows arbitrary content. Again, this is not compliant to the SAML standard because this mutation transforms the enveloped to an enveloping signature. Finally, the OneLogin Toolkits were prone to all shown attack variants as they did not apply XML Schema, validated the XML Signature independent of it's semantic occurrence and used a fixed reference to the processed SAML claims (`/samlp:Response/saml:Assertion[1]`).

We found three susceptible implementations, which applied Type 2 messages, where the whole message is protected by an XML Signature. We depict the attacks on these implementations in Figure 9. In the Guanxi and JOSSO implementations the legitimate root element was inserted into the `Object` element in the original `Signature`. The `Signature` node was moved into the *ER* element which also included the new evil assertion. In the case of WSO2, it was sufficient to place the original root element into the *ER* object. Naturally, someone would expect that enforcing full document signing would eliminate XSW completely. The both given examples demonstrate that this does not hold in practice. Again, this highlights the vigilance required when implementing complex standards such as SAML.

Finally, we did not find vulnerable frameworks that applied Type 3 messages, where both the root and the as-

sertion are protected by different signatures. Indeed, one legitimate reason is, that most SAML implementations do not use Type 3 messages. In our practical evaluation, only SimpleSAMLphp applied them by default. Nevertheless, this does not mean that XSW is not applicable to this message type in practice.

5.3 OpenSAML Vulnerability

The attack vectors described above did not work against the prevalently deployed OpenSAML library. The reason was that OpenSAML compared the `Id` used by the signature validation with the `Id` of the processed assertion. If these identifiers were different (based on a string comparison), the signature validation failed. Additionally, XML messages including more than one element with the same `Id` were also rejected. Both mechanisms are handled in OpenSAML by using the Apache Xerces library and its XML Schema validation method [34]. Nevertheless, it was possible to overcome these countermeasures with a more sophisticated XSW attack.

As mentioned before, in OpenSAML the Apache Xerces library performs a schema validation of every incoming XML message. Therefore, the `Id` of each element can be defined by using the appropriate XML Schema file. This allows the Xerces library to identify all included `Ids` and to reject messages with `Id` values which are not unique (e.g. duplicated). However, a bug in this library caused that XML elements defined with `xsd:any` content were not processed correctly. More concretely, the content of the elements defined as `<xsd:any processContents="lax">` were not checked using the defined XML Schema. Therefore, it was possible to insert elements with arbitrary – also duplicated – `Ids` inside an XML message. This created a good position for our wrapped content.

It is still the question which of the extensible elements could be used for the execution of our attacks. This depends on two processing properties:

1. Which element is used for assertion processing?
2. Which element is validated by the security module, if there are two elements with the same `Id`?

Interestingly, the two existing implementations of Apache Xerces (Java and C++) handled element dereferencing *differently*.

For C++, the attacker had to ensure that the original signed assertion was copied before the evil assertion. In the Java case, the legitimate assertion had to be placed within or after the evil assertion. In summary, if two elements with the same `Id` values occurred in an XML message, the XML security library detected only the first (for C++) or the last (for Java) element in the message. This property gave the attacker an opportunity to use

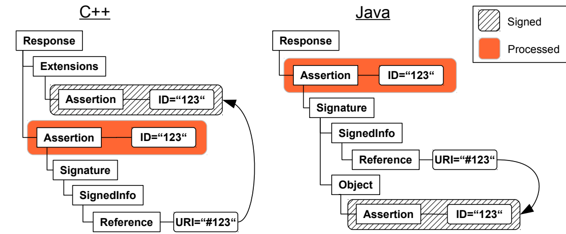

Figure 10: XSW attack on OpenSAML library.

```
<element name="Extensions" type="samlp:ExtensionsType"/>
<complexType name="ExtensionsType">
  <sequence>
    <any namespace="##other" processContents="lax"
  maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

Figure 11: XML Schema definition of the `Extensions` element.

e.g. the `Extensions` element for the C++ library, whose XML Schema is defined in Figure 11. However, the `Extensions` element is not the only possible position for our wrapped content. The schemas of SAML and XML Signature allow more locations (e.g. the `Object` element of the `Signature`, or the `SubjectConfirmationData` and `Advice` elements of the `Assertion`).

The previously described behavior of the XML schema validation forced OpenSAML to use the wrapped original assertion for signature validation. In contrast, the application logic processed the claims of the evil assertion. In Figure 10, we present the concrete attack messages of this novel XSW variant.

The successful attack on OpenSAML shows that countering the XSW attack can become more complicated than expected. Even when applying several countermeasures, the developer should still consider vulnerabilities in the underlying libraries. Namely, one vulnerability in the XML Schema validating library can lead to the execution of a successful XSW attack.

5.4 Various Implementation Flaws

While reviewing the OneLogin Toolkit, we discovered another interesting flaw: the implementation did not care about what data was actually signed. Therefore, any content signed by the IdP was sufficient to launch a XSW attack. In our case we used the metadata of the IdP [2] and created our own self-made response message to successfully attack OneLogin.

[2]The SAML Metadata [12] describes properties of SAML entities in XML to allow the easy establishment of federations. Typically, the metadata is signed by the issuer and publicly available.

Besides the fact that a SAML system has to check what data is signed, it is also essential to verify by whom the signature was created. In an early version of SimpleSAMLphp, which applied Type 3 messages, we observed that an attacker could forge the outer signature of the response message with any arbitrary key. In short, the SimpleSAMLphp RP did not verify if the included certificate in the `KeyInfo` element is trustworthy at all. The key evaluation for the signed assertion was correctly handled.

5.5 Secure Frameworks

In our evaluation of real-world SAML implementations we observed that Microsoft Sharepoint 2010 and SimpleSAMLphp were resistant to all applied test cases. Based on these findings the following questions arise: How do these systems implement signature validation? In which way do signature validation and assertion processing work together? Due to the fact that the source code of Sharepoint 2010 is not publicly available, we were only able to analyze SimpleSAMLphp.

According to this investigation the main signature validation and claims processing algorithm of SimpleSAMLphp performs the following five steps to counteract XSW attacks:

1. **XML Schema validation:** First, the whole response message is validated against the applied SAML schemas.
2. **Extract assertions:** All included assertions are extracted. Each assertion is saved as a DOM tree in a separate variable. The following steps are only applied on these segregated assertions.
3. **Verify what is signed:** SimpleSAMLphp checks, if each assertion is protected by an enveloped signature. In short, the XML node addressed by the `URI` attribute of the `Reference` element is compared to the root element of the same assertion. The XML Signature in the assertion is an enveloped signature if and only if both objects are identical.
4. **Validate signature:** The verification of every enveloped signature is exclusively done on the DOM tree of each corresponding assertion.
5. **Assertion processing:** The subsequent assertion processing is solely done with the extracted and successfully validated assertions.

When not considering the signature exclusion bug found in the OpenAthens implementation and its Java-based assertions' processing, this framework was also resistant to all the described attacks. The analysis of its implementation showed that it processes SAML assertions similarly to the above described SimpleSAMLphp framework.

Frameworks / Providers	**Signing type**	**Signature exclusion**	**Refined XSW**	**Sophisticated XSW**	**Not vulnerable**
Apache Axis 2	1)	X	X		
Guanxi	2)		X		
Higgins 1.x	1)		X		
IBM XS40	1)		X		
JOSSO	2)	X	X		
WIF	1)				X
OIOSAML	1)		X		
OpenAM	1)		X		
OneLogin	1)		X		
OpenAthens	1)	X			
OpenSAML	1)			X	
Salesforce	1)		X		
SimpleSAMLphp	3)				X
WSO2	2)		X		

Table 2: Results of our practical evaluation show that a majority of the analyzed frameworks were vulnerable to the refined wrapping techniques.

5.6 Summary

We evaluated 14 different SAML-based systems. We found 11 of them susceptible to XSW attacks, while the majority were prone to refined XSW. One prevalently used framework (OpenSAML) was receptive to a new, more subtle, variant of this attack vector. In addition, three out of the tested frameworks were vulnerable to Signature Exclusion attacks. We found two implementations, which were resistant against all test cases. The results obtained from our analysis are summarized in Table 2.

6 XSW Penetration Test Tool for SAML

Motivated on our crucial findings from the extensive frameworks' analysis and the vast amount of possible attack permutations, we implemented the first fully automated penetration test tool for XSW attacks in SAML-based frameworks. In this section we briefly describe the basic design decisions for our tool. Afterwards, we motivate its usage by revisiting the Salesforce SAML interface. This interface yielded a new possibility for an interesting XSW attack even after a deep investigation with different handcrafted messages.

Our tool will be integrated into the WS-Attacker framework[3] and offered as open source to support the huge Web Services and SSO developers' community.

6.1 Penetration Test Tool

According to the theoretical and practical analysis of different SAML frameworks (see Section 4, 5), we gained the following general knowledge about XSW attacks:

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

- **XML Schema validation:** Some of the SAML frameworks check message conformance to the underlying XML schema. Therefore, it is necessary to use XML schema extension points for placing the wrapped content. If the extension elements are not provided in the message, they have to be explicitly included.
- **Order and position:** The order and position of signed and executed elements in the message tree can force the different processing modules to have inconsistent data views.
- **Processing of the Ids:** Several SAML frameworks explicitly check, if the `Id` in the handled assertion is also used in the `Reference` of the XML Signature. Application of this countermeasure alone does not work, as there is still the option to use more elements with equal `Ids`.
- **Placement of the `Signature` element:** The `Signature` element can be placed in the newly created evil assertion or stay in the original assertion (cf. the attacks on Higgins, Apache Axis2 and IBM XS40 in Figure 8). Both cases must be considered.
- **Signature exclusion:** In three out of 14 frameworks implementation bugs caused that the signature validation step was omitted.
- **Untrusted signatures:** It is essential to check that the signature was created with a trustworthy key. Otherwise, the attacker can forge a signature with any arbitrary key and embed the corresponding certificate in the `KeyInfo` element.

Based on this knowledge, we developed a library, which allows the systematic creation of a vast amount of different SAML attack vectors. Its processing can be summarized in the following steps. First, the library takes a signed XML document containing a SAML assertion and analyzes the usage of XML Signature. The element referenced by the signature is stored as a string. Subsequently, it creates a new malicious message including an evil assertion with modified content (e.g. the `NameID` and/or `Timestamp` element). Then, it searches dynamically for extension points in the XML Schema documents (e.g. XML Schemas for SAML, HTTP binding, XML Signature, or SOAP). It places the extension elements into the malicious message (e.g. a new `Object` element is created and placed into the given `Signature` element). Afterwards, the library embeds the stored original referenced element into each of the possible malicious message elements. For each position, a combination of different attack vectors – considering changes in the `Ids` of the newly created elements and the positions of the `Signature` elements – are created. For completeness, test cases for signature exclusion and untrusted signatures are provided. With these attack vectors, developers can systematically test the security of their (newly) developed SAML libraries.

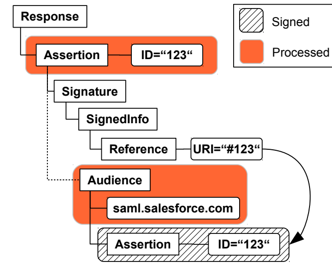

Figure 12: A successful XSW attack performed against the patched Salesforce SAML interface.

6.2 Salesforce SAML Interface Revisited

After reporting the XSW vulnerability to Salesforce, the security response team developed a simple and promising countermeasure: the SAML interface solely accepted messages containing *one* `Assertion` element[4]. On request of the Salesforce security team, we investigated the fixed SAML interface with handcrafted messages containing wrapped contents in different elements. Our manual analysis did not reveal any new attack vectors. Every message containing more than one `Assertion` element was automatically rejected. Therefore, we first considered this interface to be secure.

A few months later, after finishing the development of our penetration test tool, we decided to retest the Salesforce SAML interface and prove the feasibility of our approach. Surprisingly, the automated penetration test tool revealed a new successful attack variant by inserting the wrapped content into the `Audience` element – a descendant of the `Conditions` element. This element typically contains a `URI` constraining the parties that can consume the issued assertion. The wrapped message is depicted in Figure 12. As can be seen in the figure, both `Assertion` elements needed to contain the same `Id` attribute.

This scientifically interesting attack vector stayed unanalyzed as the Salesforce security team did not expose any concrete information about their SAML interface. However, this finding shows how complex the development of secure signature wrapping countermeasures is. This motivates for further development of automatic penetration test tools for XSW.

Salesforce security team afterwards implemented a countermeasure, which could successfully mitigate all our attack types. Its details were not revealed.

[4]This countermeasure is not standard-conform as one message can generally contain several assertions. Therefore, we do not consider this remedy in our countermeasure analysis in Section 7.

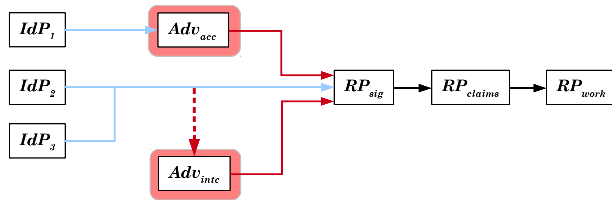

Figure 13: Overview of the components in our formal model.

7 Analysis and Countermeasures

In order to define what a successful attack on a SAML implementation is, we have to define the possibilities of the adversary, and the event that characterizes a successful attack. We do this in form of a game played between the adversary on one side, and IdP and RP on the other side. Additionally, we derive two different countermeasures. Their practical application is described in Section 8.

7.1 Data Model

A SAML assertion A can be sent to a Relying Party RP either as a stand-alone XML document, or as part of a larger document D. (D may be a complete SOAP message, or a SAML Authentication response.) To process the SAML assertion(s), the Relying Party (more specifically, RP_{claims}) searches for the `Assertion` element and parses it. We assume that A is signed, either stand-alone, or as part of D.

7.2 Identity Provider Model

We define an Identity Provider IdP to be an entity that issues signed SAML assertions, and that has control over a single private key for signing. Thus, companies like Salesforce may operate several IdPs, one for each domain of customers.

An Identity Provider IdP operates a customer database db_{IdP} and is able to perform a secure authentication protocol with any customer contained in this database. Furthermore, he has control over a private signing key, where the corresponding public key is trusted by a set of Relying Parties $\mathcal{RP} := \{RP_1, \ldots, RP_n\}$, either directly, or through means of a Public Key Infrastructure. After receiving a request from one of the customers registered in db_{IdP}, and after successful authentication, he may issue a signed XML document D, where the signed part contains the requested SAML assertion A.

7.3 Relying Party Model

We assume that processing of documents containing SAML assertions is split into two parts: (1) XML Signature verification RP_{sig}, and (2) SAML security claims processing RP_{claims} (see Figure 13). This assumption is justified since both parts differ in their algorithmic base, and because this separation was found in all frameworks. If RP_{claims} accepts, then the application logic RP_{work} of the Relying Party will deliver the requested resource to the requestor.

The XML Signature verification module RP_{sig} is configured to trust several Identity Provider public keys $\{pk_1, \ldots, pk_r\}$. Each public key defines a *trusted domain* within RP. After receiving a signed XML document D, RP_{sig} searches for a `Signature` element. It applies the referencing method described in `Reference` to retrieve the signed parts of the document, applies the transforms described in `Transforms` to these parts, and compares the computed hash values with the values stored in `DigestValue`. If all these values match, signature verification is performed over the whole `SignedInfo` element, with one of the trusted keys from $\{pk_1, \ldots, pk_r\}$. RP_{sig} then communicates the result of the signature verification (eventually alongside D) to RP_{claims}.

The SAML security claims processing module RP_{claims} may operate a customer database db_{RP}, and may validate SAML assertions against this database. In this case if the claimed identity is contained in db_{RP}, the associated rights are granted to the requestor. As an alternative, RP_{claims} may rely on authorization data contained in db_{IdP}. In this case, the associated rights will be contained in the SAML assertion, and RP_{claims} will grant these.

Please note that the definition of the winning event given below does not depend on the output of the signature verification part RP_{sig}, but on the SAML assertion processing RP_{claims}. This is necessary since in all cases described in this paper, signature verification was done correctly (as is *always* the case with XML Signature wrapping). Therefore, to be able to formulate meaningful statements about the security of a SAML framework, we must make some assumptions on the behavior of RP_{claims}.

There are many possible strategies for RP_{claims} to process SAML assertions: E.g. use the claims from the first assertion which is opened during parsing, from the first that is closed during parsing (analogously for the last assertion opened or closed), or issue an error message if more than one `Assertion` element is read.

7.4 Adversarial Model

Please recall the two different types of adversaries we have mentioned in our threat model in Section 4. Adv_{intc} is the stronger of the two: He has the ability to

partially intercept network traffic, e.g. by sniffing HTTP traffic on an unprotected WLAN, by reading past messages from an unprotected log file, or by a chosen ciphertext attack on TLS 1.0 along the lines of [5]. Please note that already this adversary is strictly weaker than the classical network based attacker known from cryptography. Adv_{acc}, our weaker adversary, only has access to the IdP and RP, i.e. he may register as a customer with IdP and receive SAML assertions issued about himself, and he may send requests to RP.

We define preconditions and success conditions of an attacker in the form of a game G. If Adv mounts a successful attack under these conditions, we say that Adv *wins* the game. This facilitates some definitions.

During the game G, the adversary has access to a validly signed document D containing a SAML assertion A issued by IdP. He then generates his own (evil) assertion EA, and combines it arbitrarily with D into an XML document D'. This document is then sent to RP.

Definition 1. *We say that the adversary (either Adv_{intc} or Adv_{acc})* wins *game G if RP, after receiving document D', with non-negligible probability $Pr(Win_{Adv})$ bases its authentication and authorization decisions on the security claims contained in EA.*

Remark: For all researched frameworks, the winning probability was either negligible or equal to 1. Within the term "negligible" we include the possibility that Adv issues a forged cryptographic signature, which we assume to be impossible in practice. If an adversary wins the game against a specific Relying Party RP, he takes over the trust domain for a specific public key pk within RP. Adv_{acc} may do this for all pk where he is allowed to register as a customer with the corresponding IdP who controls (sk, pk). Adv_{intc} can achieve this for all pk where he is able to find single signed SAML assertion A where the signature can (could in the past) be verified with pk.

7.5 Countermeasure 1: Only-process-what-is-hashed

We can derive the first countermeasure if we assume that RP_{sig} acts as a filter and only forwards the hashed parts of an XML document to RP_{claims}. The hashed parts of an XML document are those parts that are serialized as an input to a hash function, and where the hash value is stored in a `Reference` element. This excludes all parts of the document that are removed before hash calculation by applying a transformation, especially the enveloped signature transform.

Claim 1. *If RP_{sig} only forwards the hashed parts of D to RP_{claims}, then $Pr(Win_{Adv})$ is negligible.*

It is straightforward to see that EA is only forwarded to RP_{claims} if a valid signature for EA is available.

Please note that although this approach is simple and effective, it is rarely used in practice due to a number of subtle implementation problems. A variant of this approach is implemented by SimpleSAMLphp, where the RP imposes special requirements on the SAML authentication response, thus limiting interoperability. We discuss these problems in Section 8.

7.6 Countermeasure 2: Mark signed elements

In practice, RP_{sig} only returns a Boolean value, and the whole document D is forwarded to RP_{claims}. Since IdPhas to serve many different Relying Parties, we assume knowledge about the strategy of RP_{claims} only for RP_{sig}. One possibility to mark signed elements is to hand over the complete document D from RP_{sig} to RP_{claims}, plus a description where the validly signed assertions can be found.

A second possibility that is more appropriate for SAML is that RP_{sig} chooses a random value r, marks the validly signed elements with an attribute containing r, and forwards r together with the marked document. RP_{claims} can then check if the assertion processed contains r.

Let us therefore consider the second approach in more detail. For sake of simplicity we assume that only one complete element (i.e. a complete subtree of the XML document tree) is signed.

Claim 2. *Let D_{sig} be the signed subtree of D, and let $r \in \{0,1\}^l$ be the random value chosen by RP_{sig} and attached to D_{sig}. Then $Pr(Win_{Adv})$ is bounded by $max\{break_{sig}, 2^{-l}\}$.*

RP_{claims} (regardless of its strategy to choose an assertion) will only process EA if r is attached to this element. An adversary can achieve this by either generating a valid signature for EA (then r will be attached by RP_{sig}), or by guessing r and attaching it to EA.

8 Practical Countermeasures

In Section 5.5 we analyzed message processing of SimpleSAMLphp. This framework was resistant against all XSW attacks. One could therefore ask a legitimate question: Why do we need further countermeasures and why is it not appropriate to apply the security algorithm of SimpleSAMLphp in every system?

We want to make clear that SimpleSAMLphp offers both critical functionalities in one framework: signature validation (RP_{sig}) and SAML assertion evaluation

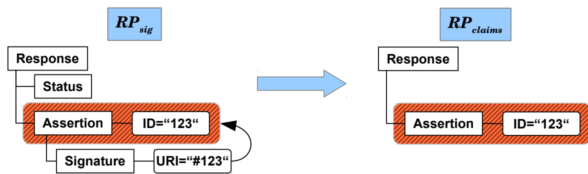

Figure 14: The see-what-is-signed approach applied in HTTP POST binding: After successful signature validation the security module RP_{sig} excludes all the unsigned elements and forwards the message to the module processing security claims RP_{claims} and the business logic.

(RP_{claims}). These two methods are implemented using the same libraries and processing modules. After parsing a document, the elements are stored within a document tree and can be accessed directly. This allows the security developers to conveniently access the same elements used in signature validation and assertion evaluation steps. However, especially in SOA environments there exist scenarios, which force the developers to separate these two steps into different modules or even different systems, e.g.:

- **Using a signature validation library:** Before evaluating the incoming assertion elements, the developer uses a DOM-based signature library, which returns `true` or `false` according to the message validity. However, the developer does not exactly know which elements were validated. If the assertion evaluation uses a different parsing approach (e.g. streaming-based SAX or StAX approach) or another DOM-library, the message processing could become error-prone.
- **XML Security gateways:** XML Security gateways can validate XML Signatures and are configured to forward only validated XML documents. If the developer evaluates a validated document in his application, he again has no explicit information about the position of the signed element. Synchronization of signature and assertion processing components in this scenario becomes even more complicated, if the developer has no information about the implementation of the security gateway (e.g. IBM XS40).

These two examples show that a convenient access to the same XML elements is not always given. Subsequently, we present two practical feasible countermeasures, which can be applied in complex and distributed real-world implementations. Both countermeasures result from our formal analysis in Section 7.

8.1 See-what-is-signed

The core idea of this countermeasure is to forward only those elements to the business logic module (RP_{claims}) that were validated by the signature verification module (RP_{sig}). This is not trivial as extracting the unsigned elements from the message context could make the further message processing in some scenarios impossible. Therefore, we propose a solution that excludes only the unsigned elements which do not contain any signed descendants. We give an example of such a message processing in Figure 14. This way, the claims and message processing logic would get the whole message context: in case of SOAP it would see the whole `Envelope` element, by application of HTTP POST binding it would be able to process the entire `Response` element. The main advantage of this approach is that the message processing logic does not have to search for validated elements because all forwarded elements are validated.

We want to stress the fact that by application of this approach *all* unsigned character nodes have to be extracted. Otherwise, the attacker could create an evil assertion *EA* and insert the signed original assertion into each element of *EA*. If RP_{sig} would not extract the character contents from *EA*, RP_{claims} could process its claims. However, by extracting the unsigned character nodes, the attacker has no possibility to insert his *evil* content, since it was excluded in RP_{sig}. Nevertheless, the subsequent XML modules can still access the whole XML tree.

This idea has already been discussed by Gajek et al. [17]. However, until now no XML Signature framework implements this countermeasure. It could be applied especially in the context of SAML HTTP POST bindings because the unsigned elements within the SAML response do not contain any data needed in RP_{claims}. We consider this countermeasure in these scenarios as appropriate because the SAML standard only allows the usage of `Id`-based referencing, exclusive canonicalization, and enveloped transformation. The authors explicitly state that this countermeasure would not work if XML Signature uses specific XSLT or XPath transformations.

8.2 Unique Identification (Tainting) of Signed Data

The second countermeasure represents another form of the *see-what-is-signed* approach. The basic idea is to uniquely identify the signed data in the RP_{sig} module and forward this information to the following modules. As described in our formal analysis, this could be done by generating a random value r, sending it to the next processing module (or as an attribute in the document root element), and attaching it to all the signed elements. We give an example of this countermeasure applied to a SOAP message in Figure 15.

The main drawback of this countermeasure is that the SAML XML Schema does not allow the inclusion of

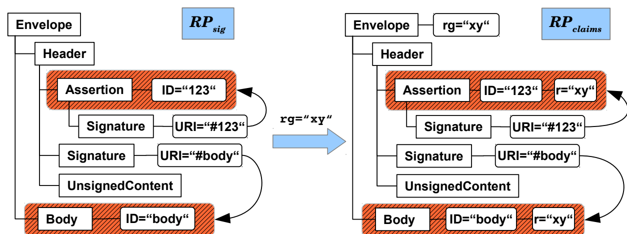

Figure 15: Unique identification of signed data applied on a SOAP message including two signed elements: The RP_{sig} module uniquely identifies the signed elements with a random value r and forwards this information along with the whole XML message.

new attributes: neither directly into the `Assertion` element nor the `Response` binding element. Therefore, the XML Schema validation of the assertion processing module would fail. For general application of this idea the SAML XML Schema needs to be extended.

Another possibility to implement this countermeasure is to use XML node types, which do not violate XML Schema, but are visible to the XML processors. For example, processing instructions, which are intended to carry instructions to the application belong to this group. They can be placed anywhere in the document without invalidating the XML Schema. Additionally, they can be conveniently found by processing XML trees with streaming and DOM-based parsers. Therefore, the presence of these XML nodes would help to find the validated data and thus allows to mitigate XSW attacks. We propose this technique for further discussion in the W3C XML Security Working Group and the OASIS consortium.

9 Related Work

XML Signature Wrapping (XSW). XSW attacks have first been described in [28] and [7]. Several countermeasures have been proposed over time.

McIntosh and Austel [28] have presented several XSW attacks and discussed (informal) receiver-sided security policies in order to prevent such exploits. They have however not given a definitive solution for this problem.

Bhargavan, Fournet and Gordon [7] have analyzed a formal approach in order to verify Web Services specifications. They have proposed a policy advisor [8], a tool that generates appropriate security policies for Web Services protocols. This approach is however not directly applicable to SAML.

Rahaman, Schaad and Rits [32, 30, 31] have refrained from policy-driven approaches and have introduced an inline solution. The authors have proposed to embed an `Account` element into the SOAP header. This element contains partial information about the structure of the SOAP message and the neighborhood of the signed element(s). The information preserves the structure of the data to be signed. However, Gajek et al. have shown that this approach does not prevent XSW attacks [16]. Benameur, Kadir, and Fenet [6] have extended the inline approach, but suffer from the same vulnerabilities.

Jensen et al. [24] have analyzed the effectiveness of XML Schema validation in terms of fending XSW attacks in Web Services. Thereby, they have used manually hardened XML Schemas. The authors have concluded that XML Schema validation is capable of fending XSW attacks, at the expense of two important disadvantages: for each application a specific hardened XML Schema without extension points must be created carefully. Moreover, validating of a hardened XML Schema entails severe performance penalties.

XPath and XPath Filter 2 are specified as referencing mechanisms in the XML Signature standard. However, the WS-Security standard proposes not to use these mechanisms, and the SAML standard mandates to use `Id`-based referencing instead. This is due to the fact that both standards are very complex. Gajek et al. [15] have evaluated the effectiveness of these mechanisms to mitigate XSW attacks in the SOAP context, and have proposed a lightweight variant `FastXPath`, which has lead to the same performance in a PoC implementation as by adapting the `Id`-based referencing.

Jensen et al. [23] have however shown that this approach does not completely eliminate XSW attacks: by clever manipulations of XML namespace declarations within a signed document, which take into account the processing rules for canonicalization algorithms in XML Signature, XSW attacks could successfully be mounted even against XPath referenced resources.

The impacts of practical XSW attacks have also been analyzed in [20, 33]. In these works new types of XSW attack have been applied on SOAP Web Service interfaces of Amazon and Eucalyptus clouds. The attacks have exploited different XML processing in distinct modules.

In summary, previous work has mostly concentrated on SOAP, and the results do not directly apply to all SAML use cases.

SAML and Single Sign-On Since SAML offers very flexible mechanisms to make claims about identities, there is a large body of research on how SAML can be used to improve identity management (e.g. [22, 39]) and other identity-related processes like payment or SIP on the Internet [25, 35]. In all these applications, the security of all SAML standards is assumed.

In an overview paper on SAML, Maler and Reed [26] have proposed *mutually authenticated* TLS as the basic

security mechanism. Please note that even if mutually authenticated TLS would be employed, it would not prevent our attacks because we only need a single signed SAML assertion from an IdP, which we can get through different means. Moreover, there exist specific sidechannels, which could be exploited by an adversary. Let us e.g. mention chosen-plaintext attacks against SSL/TLS predicted by [37] and refined by [5], or the Million Question attack by Bleichenbacher [9]. Other complications arise with the everlasting problems with SSL PKIs.

In 2003, T. Groß has initiated the security analysis of SAML [18] from a Dolev-Yao point of view, which has been formalized in [4]. He has found, together with B. Pfitzmann [19], deficiencies in the information flow between the SAML entities. Their work has influenced a revision of the standard.

In 2008, Armando et al. [3] have built a formal model of the SAML 2.0 Web Browser SSO protocol and have analyzed it with the model checker SATMC. By introducing a malicious RP they have found a practical attack on the SAML implementation of Google Apps. Another attack on the SAML-based SSO of Google Apps has been found in 2011 [2]. Again, a malicious RP has been used to force a user's web browser to access a resource without approval. Thereby, the malicious RP has injected malicious content in the initial unintended request to the attacked RP. After successful authentication on the IdP this content has been executed in the context of the user's authenticated session.

The fact that SAML protocols consist of multiple layers has been pointed out in [13]. In this paper, the Weakest Link Attack has enabled adversaries to succeed at all levels of authentication by breaking only at the weakest one.

Very recently, another work pointing out the importance of SSO protocols has been published by Wang et al. [38]. This work has analyzed the security quality of commercially deployed SSO solutions. It has shown eight serious logic flaws in high-profile IdPs and RPs (such as OpenID, Facebook, or JanRain), which have allowed an attacker to sign in as the victim user. The SAML-based SSO has not been analyzed.

10 Conclusion

In this paper we systematically analyzed the application of XSW attacks on SAML frameworks and systems. We showed that the large majority of systems exhibit critical security insufficiencies in their interfaces. Additionally, we revealed new classes of XSW attacks, which worked even if specific countermeasures were applied. We showed that the application of XML Security heavily depends on the underlying XML processing system (i.e. different XML libraries and parsing types). The processing modules involved can have inconsistent views on the same secured XML document, which may result in successful XSW attacks. Generally, these heterogeneous views can exist in all data formats beyond XML.

We proposed a formal model by analyzing the information flow inside the Relying Party and presented two countermeasures. The effectiveness of these countermeasures depends on the *real* information flow and the data processing inside RP_{claims}. Our research is a first step towards understanding the implications of the information flow between cryptographic and non-cryptographic components in complex software environments. Research in this direction could enhance the results, and provide easy-to-apply solutions for practical frameworks.

As another future research direction, we propose development of an enhanced penetration testing tool for XSW in arbitrary XML documents and all types of XML Signatures. This kind of tool presents a huge challenge as it should e.g. consider more difficult transformations like XPath or XSLT.

Acknowledgements

The authors would like to thank all the security teams and their developers for their cooperation, and would like to note that throughout the collaboration all the teams effectuated a productive and highly professional communication.

Moreover, we would like to thank Scott Cantor, David Jorm, Florian Kohlar, Christian Mainka, Christopher Meyer, Thomas Roessler, and the anonymous reviewers (of the USENIX Security Symposium and the IEEE Symposium on Security and Privacy) for their valuable remarks on the developed attacks and the paper content. Finally, we thank Alexander Bieber for the Sharepoint 2010 test bed.

This work was partially funded by the Sec2 project of the German Federal Ministry of Education and Research (BMBF, FKZ: 01BY1030).

References

[1] *IEEE International Conference on Web Services, ICWS 2009, Los Angeles, CA, USA, 6-10 July 2009* (2009), IEEE.

[2] ARMANDO, A., CARBONE, R., COMPAGNA, L., CUÉLLAR, J., PELLEGRINO, G., AND SORNIOTTI, A. From Multiple Credentials to Browser-Based Single Sign-On: Are We More Secure? In *Future Challenges in Security and Privacy for Academia and Industry*, J. Camenisch, S. Fischer-Hbner, Y. Murayama, A. Portmann, and C. Rieder, Eds., vol. 354 of *IFIP Advances in Information and Communication Technology*. Springer Boston, 2011.

[3] ARMANDO, A., CARBONE, R., COMPAGNA, L., CUÉLLAR, J., AND TOBARRA, M. L. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*, V. Shmatikov, Ed. ACM, Alexandria and VA and USA, 2008.

[4] BACKES, M., AND GROSS, T. Tailoring the dolev-yao abstraction to web services realities. In *SWS* (2005), E. Damiani and H. Maruyama, Eds., ACM, pp. 65–74.

[5] BARD, G. V. A Challenging but Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL. In *SECRYPT* (2006), M. Malek, E. Fernández-Medina, and J. Hernando, Eds., INSTICC Press, pp. 99–109.

[6] BENAMEUR, A., KADIR, F. A., AND FENET, S. XML Rewriting Attacks: Existing Solutions and their Limitations. In *IADIS Applied Computing 2008* (Apr. 2008), IADIS Press.

[7] BHARGAVAN, K., FOURNET, C., AND GORDON, A. D. Verifying policy-based security for web services. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security* (2004), pp. 268–277.

[8] BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND O'SHEA, G. An advisor for web services security policies. In *SWS '05: Proceedings of the 2005 workshop on Secure web services* (New York, NY, USA, 2005), ACM, pp. 1–9.

[9] BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1. In *CRYPTO* (1998), pp. 1–12.

[10] CANTOR, S., KEMP, J., MALER, E., AND PHILPOTT, R. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, 15.03.2005, 2005. `http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf`.

[11] CANTOR, S., KEMP, J., PHILPOTT, R., AND MALER, E. Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, 15.03.2005, 2005. `http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf`.

[12] CANTOR, S., MOREH, J., PHILPOTT, R., AND MALER, E. Metadata for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, 15.03.2005, 2005. `http://docs.oasis-open.org/security/saml/v2.0/saml-metadata-2.0-os.pdf`.

[13] CHAN, Y.-Y. Weakest link attack on single sign-on and its case in saml v2.0 web sso. In *Computational Science and Its Applications - ICCSA 2006*, M. Gavrilova, O. Gervasi, V. Kumar, C. Tan, D. Taniar, A. Lagan, Y. Mun, and H. Choo, Eds., vol. 3982 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 507–516. 10.1007/11751595_54.

[14] EASTLAKE, D., REAGLE, J., SOLO, D., HIRSCH, F., AND ROESSLER, T. XML Signature Syntax and Processing (Second Edition), 2008. `http://www.w3.org/TR/xmldsig-core/`.

[15] GAJEK, S., JENSEN, M., LIAO, L., AND SCHWENK, J. Analysis of signature wrapping attacks and countermeasures. In *ICWS* [1], pp. 575–582.

[16] GAJEK, S., LIAO, L., AND SCHWENK, J. Breaking and fixing the inline approach. In *SWS '07: Proceedings of the 2007 ACM workshop on Secure web services* (New York, NY, USA, 2007), ACM, pp. 37–43.

[17] GAJEK, S., LIAO, L., AND SCHWENK, J. Towards a formal semantic of xml signature. W3C Workshop Next Steps for XML Signature and XML Encryption, 2007.

[18] GROSS, T. Security Analysis of the SAML SSO Browser/Artifact Profile. In *ACSAC* (2003), IEEE Computer Society, pp. 298–307.

[19] GROSS, T., AND PFITZMANN, B. SAML artifact information flow revisited. In *In IEEE Workshop on Web Services Security (WSSS)* (Berkeley, May 2006), IEEE, pp. 84–100.

[20] GRUSCHKA, N., AND IACONO, L. L. Vulnerable cloud: Soap message security validation revisited. In *ICWS* [1], pp. 625–631.

[21] GUDGIN, M., HADLEY, M., MENDELSOHN, N., MOREAU, J.-J., AND NIELSEN, H. F. SOAP Version 1.2 Part 1: Messaging Framework. *W3C Recommendation* (2003).

[22] HARDING, P., JOHANSSON, L., AND KLINGENSTEIN, N. Dynamic security assertion markup language: Simplifying single sign-on. *Security Privacy, IEEE 6*, 2 (march-april 2008), 83 – 85.

[23] JENSEN, M., LIAO, L., AND SCHWENK, J. The curse of namespaces in the domain of xml signature. In *SWS* (2009), E. Damiani, S. Proctor, and A. Singhal, Eds., ACM, pp. 29–36.

[24] JENSEN, M., MEYER, C., SOMOROVSKY, J., AND SCHWENK, J. 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* (sept. 2011), pp. 7 –13.

[25] LUTZ, D., AND STILLER, B. Combining identity federation with payment: The saml-based payment protocol. In *Network Operations and Management Symposium (NOMS), 2010 IEEE* (april 2010), pp. 495 –502.

[26] MALER, E., AND REED, D. The venn of identity: Options and issues in federated identity management. *Security Privacy, IEEE 6*, 2 (march-april 2008), 16 –23.

[27] MCINTOSH, M., AND AUSTEL, P. XML Signature Element Wrapping Attacks and Countermeasures. In *SWS '05: Proceedings of the 2005 workshop on Secure web services* (New York, NY, USA, 2005), ACM Press, pp. 20–27.

[28] MCINTOSH, M., AND AUSTEL, P. XML signature element wrapping attacks and countermeasures. In *Workshop on Secure Web Services* (2005).

[29] NADALIN, A., KALER, C., MONZILLO, R., AND HALLAM-BAKER, P. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). *OASIS Standard* (2006).

[30] RAHAMAN, M. A., MARTEN, R., AND SCHAAD, A. An inline approach for secure soap requests and early validation. OWASP AppSec Europe, 2006.

[31] RAHAMAN, M. A., AND SCHAAD, A. Soap-based secure conversation and collaboration. In *ICWS* (2007), pp. 471–480.

[32] RAHAMAN, M. A., SCHAAD, A., AND RITS, M. Towards secure soap message exchange in a soa. In *Workshop on Secure Web Services* (2006).

[33] SOMOROVSKY, J., HEIDERICH, M., JENSEN, M., SCHWENK, J., GRUSCHKA, N., AND IACONO, L. L. All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces. In *The ACM Cloud Computing Security Workshop (CCSW)* (Oct. 2011).

[34] THE APACHE SOFTWARE FOUNDATION. Apache Xerces.

[35] TSCHOFENIG, H., FALK, R., PETERSON, J., HODGES, J., SICKER, D., AND POLK, J. Using saml to protect the session initiation protocol (sip). *Network, IEEE 20*, 5 (sept.-oct. 2006), 14 –17.

[36] VAN DER VLIST, E. *XML Schema*. O'Reilly, 2002.

[37] WAGNER, D., AND SCHNEIER, B. Analysis of the SSL 3.0 protocol. In *In Proceedings of the Second USENIX Workshop on Electronic Commerce* (1996), USENIX Association, pp. 29–40.

[38] WANG, R., CHEN, S., AND WANG, X. Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *IEEE Symposium on Security and Privacy (Oakland), IEEE Computer Society* (May 2012).

[39] YONG-SHENG, Z., AND JING, Y. Research of dynamic authentication mechanism crossing domains for web services based on saml. In *Future Computer and Communication (ICFCC), 2010 2nd International Conference on* (may 2010), vol. 2, pp. V2–395 –V2–398.