RUHR
UNIVERSITÄT
BOCHUM

RUB

Master Thesis

Automated Penetration Testing for SAML-based SSO Frameworks

Author:
Benjamin Sanno

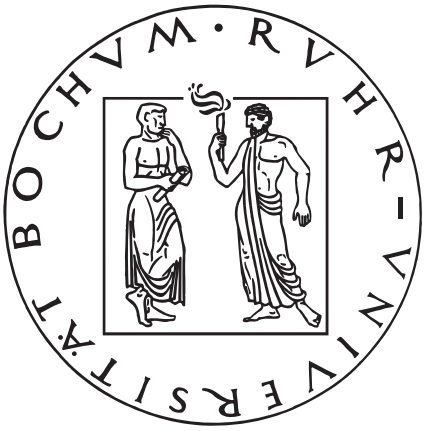
Supervisor:
Prof. Dr. Jörg Schwenk
Vladislav Mladenov
Christian Mainka

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the

Group of XML- and Webservice Security
Department of Network- and Data Security

July 2013



Declaration of Authorship

I, Benjamin Sanno, declare that this thesis titled, 'Automated Penetration Testing for SAML-based SSO Frameworks' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at the Ruhr-University Bochum.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.
- The source code which is included with this work has been written by me, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

RUHR-UNIVERSITY BOCHUM

Abstract

Faculty of Electronics and Information Technology
Department of Network- and Data Security

Master of Science

Automated Penetration Testing for SAML-based SSO Frameworks

by Benjamin Sanno

Single Sign-on (SSO) is a beneficial solution to establish authentication among several parties. These entities are an Identity Provider (IdP), a Service Provider (SP) and a user. As well as SOAP, the SAML recommendation is an authentication framework based on XML Signature and is often used in the context of SSO. Unfortunately, XML Signature Wrapping (XSW) attacks exist, which pose a threat to those authentication frameworks. The reason for this is basically, that different SPs have proprietary implementations and therefore, some may be vulnerable. The approach of this work is to do a detailed security analysis of proprietary SAML implementations. Instead of doing the analysis manually, security of an SP can be measured automatically by a program. An automatic tool has been developed in this thesis and its task is to emulate all parties of the authentication process but the SP. As a result, a new product exists for developers and penetration testers. It is universally applicable and automatic, hence easy and quick to use. Its sophisticated software design is optimized in terms of extensibility and modularity. In addition, the test tool uses machine learning to deal with unexpected behavior of the SP. Finally, four service providers were tested. Three of these could not be outwitted using the test tool. One provider seems to be vulnerable. Despite the XSW attack is known since 2005, it still represents a serious threat to services.

Contents

List of Figures

AaaS Authentication-as-a-Service . . . 1

ACS Assertion Consumer Service . . . 15

B2B Business to Business . . . 59

BCP Business Continuity Planning . . . 30

CRM Customer Relationship Management . . . 5

DMS Document Management System . . . 5

DoS Denial of Service . . . 30

ERP Enterprise Resource Planning . . . 5

HTML Hypertext Markup Language

HTTPS Hypertext Transfer Protocol Secure . . . 59

HTTP Hypertext Transfer Protocol . . . iv

IDE Integrated Development Environment . . . 49

IdP Identity Provider . . . ii

IDS Intrusion Detection System . . . 54

JDK Java Development Kit . . . 49

MitM Man-in-the-Middle . . . 29

OASIS Organization for the Advancement of Structured Information Standards . . . 8

PEM Privacy Enhanced Mail . . . 53

RFC Request For Comments . . . 13

SaaS Software as a Service . . . 59

SAML Security Assertion Markup Language . . . 1

SOAP SOAP specification . . . 2

SOA Service-Oriented Architecture . . . 1

SP Service Provider . . . ii

SSL Secure Sockets Layer . . . 12

SSO Single Sign-on . . . ii

TLS Transport Layer Security . . . 12

UML Unified Modeling Language . . . 38

URI Uniform Resource Identifier 6

URL Uniform Resource Locator 9

W3C World Wide Web Consortium 3

XHTML Extensible HyperText Markup Language 14

XML Extensible Markup Language 1

XSS Cross-Site Scripting 2

XSW XML Signature Wrapping ii

Chapter 1

Introduction

The Internet is a network that allows people and machines to communicate fast and easy to the world. We share and exchange much data that sometimes contain critical information. There is a strong trend to provide IT-services over the Internet. A usual scenario is to sell information-processing services to the customers instead of shipping stand-alone software, stored on a data medium. In order to offer web applications and services to their users, an enterprise should establish a Service-Oriented Architecture (SOA). These services should comply with established security standards, that address data privacy or non-repudiation. Furthermore, users should be accountable for their use of the services. Authentication-as-a-Service (AaaS) is one aspect of SOA and Single Sign-on (SSO) can be seen as a concept to establish AaaS.

Especially in the business environment, the SSO concept is used to simplify the authentication procedure, that employees are faced with. An employee is a user that requires access to several services, which are hosted by service providers. The idea is to establish an identity provider, which manages the authentication to several service providers for the user. As a result, the user needs to log in only once. To authenticate a user, a Security Assertion Markup Language (SAML) message is sent to a service provider. SAML is a specification that defines protocols and messages which are compliant to Extensible Markup Language (XML). The XML Signature standard is used to sign the messages and to make its statements verifiable. Despite that, a service provider which accepts signed SAML messages can be vulnerable to XSW, such as Somorovsky et al.

have shown. In addition, a large number of possibilities, i.e. maliciously altered messages, exist which could be a threat for XML-based procedures like SAML [8]. Because of the large amount of possibilities, an XML-based service cannot be checked manually without substantial effort.

There are several authentication frameworks, e.g. OpenID, WS-Trust, OAuth, SOAP specification (SOAP) or SAML. All of these procedures can have security weaknesses, but a comprehensive security analysis is far beyond this work. A detailed analysis of the SOAP protocol, for example, is elucidated in [25] and [10]. This thesis is about a profound SAML security inspection.

The topic of the master thesis is "Automatic Penetration Testing for SAML-based SSO Frameworks". It stems from previous work by some researchers around Juraj Somorovsky [25, 26]. Based on the XML Signature Wrapping (XSW) attack published by Michael McIntosh and Paula Austel from IBM Research [13], Somorovsky demonstrated the practicability of the idea on SAML-based SSO frameworks. A couple of frameworks were tested manually, and some of them were vulnerable. This analysis is very extensive and unfortunately, in comparison "to prominent attacks such as SQL-Injection or Cross-Site Scripting (XSS), there is currently no penetration test tool that is capable of analyzing the security of XML interfaces" [11]. This was before Christian Mainka developed a framework for web security testing, which is called "WS-Attacker" and already published as a Sourceforge Project [1][2]. However, it does not provide an extension to test browser-based SAML authentication yet.

To address this issue, the motivation for this work is to implement a penetration test tool in Java that can check service providers, which authenticate their users by SAML, for the XSW vulnerability. In addition, it should be automatic and universal so that as many service providers as possible can be tested with little effort.

[1] See ws-attacks.org project at URL: http://www.ws-attacks.org.

[2] See the Sourceforge project 'WS-Attacker Framework' at http://sourceforge.net/projects/ws-attacker/.

Chapter 2

Foundations

This chapter provides a sufficient explanation of the fundamental knowledge base that is necessary to understand the subsequent chapters. The next sections elucidate SSO as a concept and other important technical aspects to understand the test tool and its design. In preparation for this chapter, the reading of core SAML documents that are [14, 16] is recommended.

2.1 XML

[1] XML is an acronym for Extensible Markup Language. It is a recommendation by the World Wide Web Consortium (W3C) consortium and published in 1998 [2]. Each XML document has a strict tree structure, because there are rules how elements must be placed. An XML element is the fundamental building block and consists of a start-tag, an end-tag and content. For each start-tag exists an end-tag with the same name in the document, otherwise it would not be standard compliant. The biggest advantage of XML is that custom tag names can be defined and there exist only a few rules, which must be followed. Hence, XML documents are very flexible and extensible.

Each XML document consists of two abstract ranges: a header part and a body part. Typically, the header specifies general or meta information, e.g. the document's XML version type, and the body contains specific information. Listing 2.1 is a simple XML document and its header is the first line `<?xml version='1.0'?>`.

[1] This section 2.1 is based on my bachelor thesis.

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

Listing 2.1: Simple XML example [4]

2.2 Single Sign-On Scenario

Single Sign-on (SSO) is an authentication concept. Its main benefit is to support users to manage their digital identities and credentials. It is one possibility to solve the problem a typical user is faced with who is using many services over the Internet. Without the assistance of an SSO system, the user has to manage all authentication processes by himself. In contrast, an SSO infrastructure can help to reduce the effort for each user to authenticate himself to multiple services during one session. Figure 2.1 illustrates the participating parties: the user, one asserting entity as well as multiple relying entities.

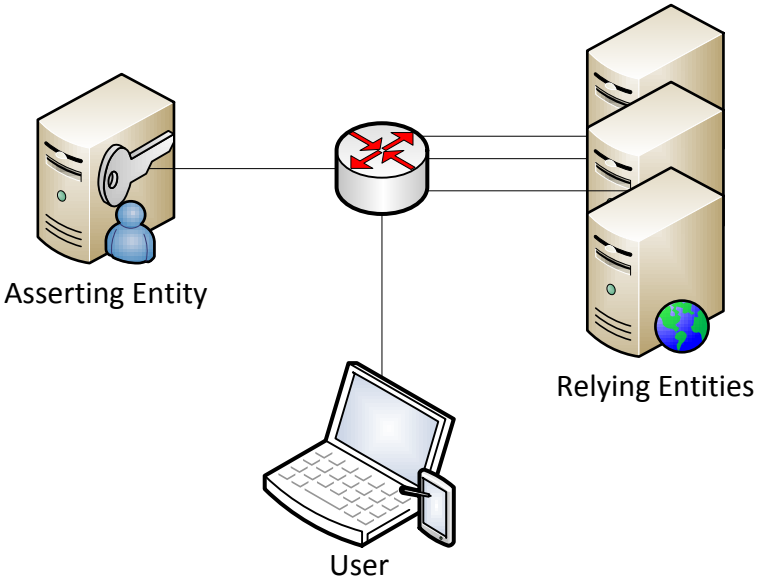


Figure 2.1: Basic SSO Scenario

2.2.1 The Relying Entity

The SAML glossary defines a relying entity as a "system entity that decides to take an action based on information from another system entity. For example, a SAML relying party depends on receiving assertions from an asserting party (a SAML authority)

about a subject." [18, page 8] In other words, a relying entity provides an information-processing service, which is typically an Enterprise Resource Planning (ERP) system or just a significant part of it. This can be basic Customer Relationship Management (CRM) functionality or a Document Management System (DMS). Therefore, each relying party provides a service that is claimed by users. Because of that, a relying party is called a **Service Provider (SP)** [2]. Obviously, a SSO architecture is only useful, if a subject has to authenticate against multiple service providers.

2.2.2 The Asserting Entity

On the right side of Figure 2.1, the asserting entity is shown. The tasks of this entity are to "create, maintain, and manage identity information for principals [users] and provide principal authentication to other service providers" [12, page 22]. Because of that, this entity is called the **Identity Provider (IdP)** in the literature. The asserting party implements the most significant functionality in the SSO architecture. It represents an authority who, in particular, must be trustworthy and should only use secure algorithms and protocols. Its task is to manage multiple digital identities and credentials of users. At the beginning of a new session, a user has to authenticate himself once to the asserting entity. This initial authentication facilitates the entity to create security tokens. Then, the user can access protected resources on relying entities.

2.3 XML Signature

XML signature is a recommendation published by the W3C. The motivation is to standardize an extension for the XML recommendation, so that XML documents can provide "integrity, message authentication, and/or signer authentication services for data of any type" [27]. Consequently, XML signature is a specification about a specialized XML element called `<Signature>`. The recommendation defines the element structure, specifies a collection of methods and algorithms, and gives security advice.

The basic schema definition of this specialized element is shown in Listing 2.2. According to that XML Schema definition, a `<Signature>` node consists of at least two child nodes: `<SignedInfo>` and `<SignatureValue>`, but beyond that it is possible to extend this

[2] For a more precise definition of terms see [18] and [12].

```
<element name="Signature" type="ds:SignatureType"/>
<complexType name="SignatureType">
   <sequence>
      <element ref="ds:SignedInfo"/>
      <element ref="ds:SignatureValue"/>
      <element ref="ds:KeyInfo" minOccurs="0"/>
      <element ref="ds:Object" minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
   <attribute name="Id" type="ID" use="optional"/>
</complexType>
```

Listing 2.2: Complex Type Definition of SignatureType [27]

```
<element name="SignedInfo" type="ds:SignedInfoType"/>
   <complexType name="SignedInfoType">
   <sequence>
      <element ref="ds:CanonicalizationMethod"/>
      <element ref="ds:SignatureMethod"/>
      <element ref="ds:Reference" maxOccurs="unbounded"/>
   </sequence>
   <attribute name="Id" type="ID" use="optional"/>
</complexType>
```

Listing 2.3: Complex Type Definition of SignedInfoType [27]

fragment almost unlimited. A concrete example message that is signed in compliance to the XML signature specification is depicted on page 74.

The `<SignedInfo>` element specifies which transformation, hash, and sign algorithms and methods are used during the signature value compilation process. As a result, "XML-signatures are generated from a hash over the canonical form of a signature manifest" [23]. `<SignedInfo>` is an element of complex type. Listing 2.3 is the XML Schema definition of the `<SignedInfo>` node. It defines each algorithm that is used for canonicalization and signature creation.

`<SignedInfo>` includes a `<Reference>` element. The reference can be a unique attribute value, e.g. a character sequence of numbers and word characters. In this case, the `<Reference>` element has a Uniform Resource Identifier (URI) attribute, which is an ID. It is the label of the signed element. Another and more sophisticated referencing method can be an XPath expression. A `<Transforms>` node is appended as a child to the `<Reference>` node and the URI attribute value is usually set to URI="" (but other values are allowed too).

The `<DigestMethod>` "identifies the digest algorithm to be applied to the signed object" [27]. In other words, it specifies how the referenced data is hashed. The resulting

```
<element name="Reference" type="ds:ReferenceType"/>
<complexType name="ReferenceType">
   <sequence>
      <element ref="ds:Transforms" minOccurs="0"/>
      <element ref="ds:DigestMethod"/>
      <element ref="ds:DigestValue"/>
   </sequence>
   <attribute name="Id" type="ID" use="optional"/>
   <attribute name="URI" type="anyURI" use="optional"/>
   <attribute name="Type" type="anyURI" use="optional"/>
</complexType>
```

LISTING 2.4: Complex Type Definition of ReferenceType [27]

binary value is then base64 encoded. Finally, the character sequence is put into the `<DigestValue>` element.

The latter element encloses the signature value which is a base64 encoded character string. This value represents the result of a hash and sign procedure applied to a referenced element within the XML document.

The last two elements in the sequence are `<KeyInfo>` and `<Object>`. `<KeyInfo>` is optional and usually contains the public credentials, e.g. X509 certificate or an public key. Whereas, "Object" can be any XML fragment with any tag name. Even the number of occurrences is unbounded. This characteristic of the `<Signature>` definition is exploited by the XSW attack introduced in section 2.5.1.

Finally, a `<Signature>` node can have an identifier attribute of type `ID`. It is typically a random and unique sequence of numbers and word characters. Furthermore, the element can be referenced using an XPath expression. In this case, the `URI` attribute value is set to an empty string, and a `<Transform>` element is appended as a child to `<Transforms>`.

The creation af a signature is an important procedure the IdP has to be capable of, so that it can create security tokens. The signature creation process has two steps: creation of the digest value and calculation of the signature value. Both procedures are characterized as pseudo-code in Appendix A.1 (Algorithm 2 and Algorithm 3). In addition, the verification of a signed security token is done by the SP. The signature verification process also has two steps: see Algorithm 4 and Algorithm 5 in the Appendix for details.

In general, different types of signatures exist that are detached, enveloped, and enveloping signature. One variant of these is important for this work: that is the enveloped

signature. The other two types are described in the Appendix A, for the sake of completeness.

2.3.1 Enveloped Signature

"The signature is over the XML content that contains the signature as an element. The content provides the root XML document element. Obviously, enveloped signatures must take care not to include their own value in the calculation of the SignatureValue" [27, section 10]. Figure 2.2 is an illustration of an enveloped structure. The grey area is the character value that is signed, whereas the white rectangle encloses the `<Signature>` element which, of course, is not signed, although it is a child node of the referenced `<Document>` element. To be precise, the referenced XML fragment must be an ancestor of the `<Signature>` element (in this case it is the parent node), otherwise it would be a detached structure as illustrated in Figure A.2. Referencing can be done ID-based or with an XPath expression.

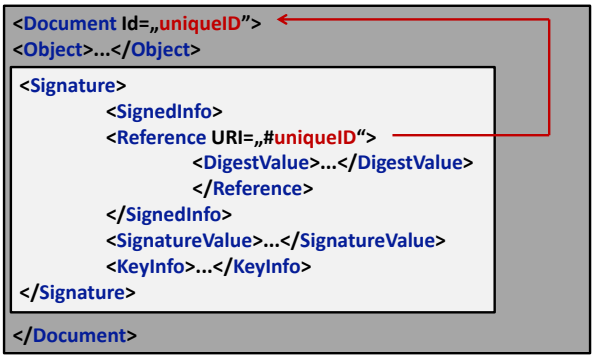


FIGURE 2.2: Enveloped Signature

2.4 SAML Standard

The SAML standard is published by the Organization for the Advancement of Structured Information Standards (OASIS) [3]. In essence, the standard can be described as an "XML-based framework for marshaling security and identity information" [12]. There are three SAML versions: V1.0, V1.1, and V2.0, but this work puts the focus on version 2.0. The full standard V2.0 consists of seven documents [14–17, 19–21].

[3]For more information see the website of OASIS Advancing Open Standards for the Information Society at `https://www.oasisopen.org/org`.

SAML is not a language. It is a specification that defines messages and its format, message encoding methods, message exchange protocols, and other recommendations. Important aspects of SAML are:

Message Format: A SAML message is an XML message with a particular format and special elements. The format and the naming conventions are strictly defined by ComplexType definitions in the SAML documents.

Message Exchange Protocols: How SAML request-response messages should be exchanged among the participating parties. For example, the IdP initiated or SP initiated message exchange protocols are introduced in this thesis.

Message Encodings: Message encoding recommendations describe how SAML messages should be encoded. For example, a SAML message can be encoded with Base64 and put into an HTTP POST message or an HTTP form control. A message can also be DEFLATE encoded. An `<AuthnRequest>` message can be base64 encoded, Uniform Resource Locator (URL)-encoded, and send via HTTP GET.

Bindings: Bindings recommend, how SAML messages should be mapped onto standard communication protocols like the HTTP methods POST and redirect, or also SOAP [16, page 5 section 1.1]. Therefore, a binding definition comprehends recommendations about: the message exchange, message encodings, message integrity and confidentiality, as well as metadata processing, caching, transport layer security levels, and error reporting.

2.4.1 SAML Message Format

There exist two types of SAML messages: a request type and a response type. Listing 2.5 is the abstract type definition for each SAML response message without an `<Assertion>` element. Based on the type definition, a SAML response can have an `<Issuer>`, `<Signature>`, or `<Status>` element. The order of these nodes is strict (as signalized by `<sequence>`), and all elements but the `<Status>` element are optional. In addition, some attributes can be set within the root element, which is the `<Response>`. Some of these attributes are optional, and some are required as shown in the Listing.

A message example that shows a minimalistic SAML response is depicted in Listing 2.6. The example includes only required elements. This can help to find superfluous elements

```
<complexType name="StatusResponseType">
   <sequence>
      <element ref="saml:Issuer" minOccurs="0"/>
      <element ref="ds:Signature" minOccurs="0"/>
      ...removed for simplicity...
      <element ref="samlp:Status"/>
   </sequence>
   <attribute name="ID" type="ID" use="required"/>
   <attribute name="InResponseTo" type="NCName" use="optional"/>
   <attribute name="Version" type="string" use="required"/>
   <attribute name="IssueInstant" type="dateTime" use="required"/>
   <attribute name="Destination" type="anyURI" use="optional"/>
   <attribute name="Consent" type="anyURI" use="optional"/>
</complexType>
```

LISTING 2.5: Complex Type Definition of StatusResponseType [14, page 39]

```
<?xml version='1.0'?>
<samlp:Response xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
                xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
                ID="ResponseID"
                Version="2.0"
                IssueInstant="TimeStamp">
   <samlp:Status>
      <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success" />
   </samlp:Status>
   <saml:Assertion>
   ...removed for simplicity...
   </saml:Assertion>
</samlp:Response>
```

LISTING 2.6: Minimalistic SAML Response Example

to avoid checks on the side of the SP. In combination with another ComplexType definition which is not shown here, there can be one or more `<Assertion>` elements in a `<Response>` tree. To address confidentiality, the assertion can also be encrypted and in this case the XML tag is labeled `<EncryptedAssertion>`. The `<Assertion>` element is typically an essential child node of the `<Response>` element. As part of the SAML response, it is sent from the responder (IdP) to the requester (SP).

An assertion is a security token used for access control. In general, there are different types of tokens: a token can be a physical device, e.g. a key, or a information, e.g. a password or a signed statement. In case of the SAML authentication protocol, an assertion is a security token which represents a certified statement. The statement describes the right to get authorization to a protected resource.

Listing 2.7 is the ComplexType definition of the `<Assertion>` element. Based on the definition an `<Assertion>` can have the following childs: `<Issuer>`, `<Signature>`, `<Subject>`, and others which has been removed for the sake of simplicity. The `<Issuer>` is the only element which is required. So, in contrast to the type definition of the

```
<element name="Assertion" type="saml:AssertionType"/>
<complexType name="AssertionType">
   <sequence>
      <element ref="saml:Issuer"/>
      <element ref="ds:Signature" minOccurs="0"/>
      <element ref="saml:Subject" minOccurs="0"/>
      ...removed for simplicity...
   </sequence>
   <attribute name="Version" type="string" use="required"/>
   <attribute name="ID" type="ID" use="required"/>
   <attribute name="IssueInstant" type="dateTime" use="required"/>
</complexType>
```

LISTING 2.7: Complex Type Definition of AssertionType [14, page 17]

`<Response>`, the `<Issuer>` element is not optional. The three required attributes are `Version`, `ID`, and `IssueInstant`.

The `<Subject>` element is an important optional node, which is used generally for access control. For example, the "bearer" method, which can be specified in the `<Subject>` element, is used to assert that the `<Subject>` is the owner of a URL and thus its content. Hence, the requesting user should be authorized by the SP to access it. Of course, if there is no signature, the statements within the assertion remain a claim, that can be wrong or malicious. Therefore, the statements must be certified very well. The XML Signature recommendation was previously introduced in section 2.3, and is applied in this context to certify `SAML` messages. Three variants exist to sign a SAML message:

Response Only The XML document is signed completely. The signature is added as a child node of the root element, which is the `<Response>` itself. Because the `<Response>` element is a parent of the `<Signature>` node, this type of XML Signature is enveloped. The `<Signature>` element must be the second child after the sibling `<Issuer>`. A strict type definition shows Listing 2.5 which is the official complex type recommendation for a `<Response>` element.

Assertions Only In this case, the only element which is signed is the `<Assertion>` node. The `<Signature>` element should be a child node of the `<Assertion>` as defined by Listing 2.7. This structure is called an enveloped XML signature. As a result, a valid signature over the `<Assertion>` element assures that the statements within it are published by the claimed identity. However, only the assertion and its child nodes are signed and hence, the integrity of other nodes cannot be assured.

Response and Assertion Finally, the `<Response>` and the `<Assertion>` element can be signed. If we assume that the user and the IdP have own certificates, then the

`<Assertion>` can be signed by the private key which corresponds to the user and, additionally, the `<Response>` can be signed by the private key of the IdP. In this scenario, the response is certified by the IdP and the assertion itself is certified by the user certificate.

Putting all together, the result is a full security token. A realistic SAML response is depicted in the Appendix A in Section A.4.

2.4.2 SAML Protocol Bindings

A SAML protocol binding is a mapping of SAML messages to a representation that can be transmitted by an HTTP client over the network interface. A binding specification also contains recommendations for the message exchange flow. The message exchange is described later in 2.4.3 on page 15.

All bindings must use HTTP with Secure Sockets Layer (SSL) or Transport Layer Security (TLS). Because the standard defines several different bindings, any provider has the choice of which types are offered. For this work, two out of six default bindings are introduced and implemented in the test tool. In subsequent section, the HTTP POST and the HTTP Redirect bindings are explained.

HTTP POST Binding Maps SAML messages to HTTP POST messages. The transmission method should be HTTP POST.

HTTP Redirect Binding "The HTTP Redirect binding defines a mechanism by which SAML protocol messages can be transmitted within URL parameters" using a URL encoding technique. The transmission method should be HTTP GET [16, page 15].

2.4.2.1 HTTP Redirect Binding

The HTTP Redirect binding specifies a SAML conversation and it is defined on pages 15-21 in [16]. In this context, redirection means that SAML message parameters are transmitted by HTTP redirect messages. Figure 2.3 depicts the most important part of this binding specification. It represents graphically the various message encoding steps

and the final HTTP transmission method, which are defined by the HTTP Redirect binding.

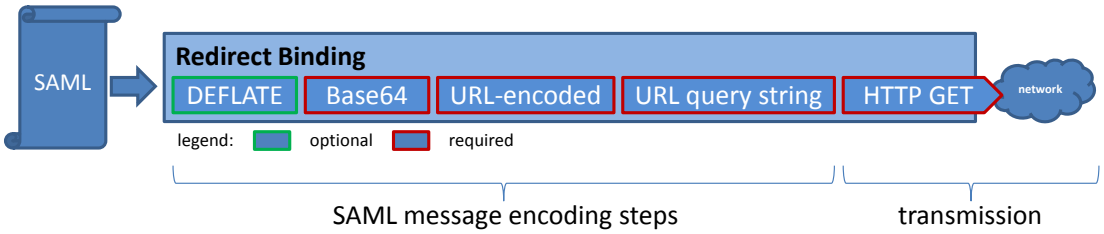


FIGURE 2.3: HTTP Redirect Binding

Base64 encoding (see Request For Comments (RFC) 2045), URL-encoding and the resulting character sequence MUST be placed "entirely within the URL query string" [16, page 17]. Therefore, these message encoding steps are required and marked as red. Whereas, the DEFLATE encoding (RFC 1951) in advance is optional and marked by color green. Finally, the transport protocol method is specified as HTTP GET and marked as red.

To be more precise, the base64 representation of the SAML message is URL encoded and the resulting character sequence is appended to the location URL. The location URL is part of the HTTP header, and parameters can only be appended as a query string. The subsequent table exemplifies how SAML messages should be encoded, so that they can be appended to URIs as a parameter value. Generally, the compliant location URL is a character sequence whose pattern is always conform to this syntax definition:

syntax	`<scheme name> : <hierarchical part> [?<query>] [#<fragment>]`

examples	`https://identityprovider.edu?SAMLRequest=...`
	`https://identityprovider.edu?SAMLResponse=...`
	`https://identityprovider.edu?SAMLRequest=...&RelayState=...`
	`https://idp.edu?SAMLResponse=...&SigAlg=...&Signature=...`

The table lists some examples for parameter names and their combination. The length of the query part is *theoretically* infinite with respect to the standard, but limited practically due to HTTP client implementations. Of course, no whitespaces and no word-wrap is allowed. Finally, the location URL must be integrated into an HTTP redirect message, which is typically an HTTP GET request. A specific example is depicted in Listing 2.8.

```
1 HTTP/1.1 302 Moved Temporarily
2 Content-Type: text/html; charset=UTF-8
3 Cache-control: no-cache, no-store
4 Location: https://www.malloryidp.org?SAMLRequest=...&RelayState=...
5 Date: Fri, 17 May 2013 20:01:26 GMT
```

LISTING 2.8: SAML Redirect Binding Example

Another binding is the HTTP POST binding, which is explained next.

2.4.2.2 HTTP POST Binding

Pages 21-26 of [16] define how SAML messages can be transmitted over the HTTP POST method. The binding specifies in which way messages are transmitted and redirected. Also, it depends on the binding, which message format the SP expects. The XML representation of the SAML message is base64 encoded and embedded in an HTML form control (see section 17.2 and 17.13 in [22]).

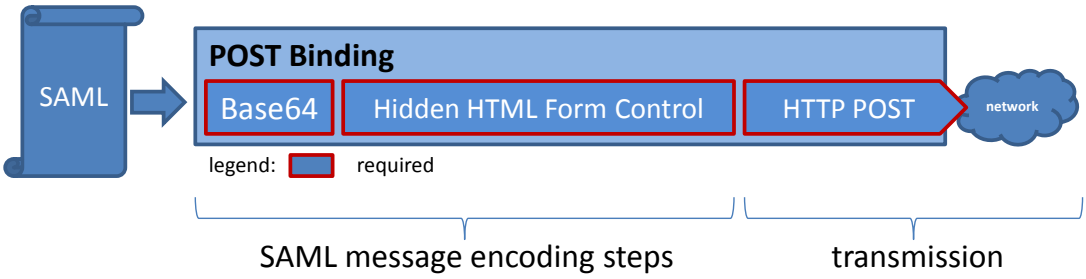


FIGURE 2.4: HTTP POST Binding

Figure 2.4 is a block diagram that illustrates the most important part of the HTTP POST binding: the SAML message is base64 encoded, embedded into a HTTP form control and finally posted using HTTP POST. An example HTTP form control message taken from the standard is shown in Listing 2.9.

The first three lines are the HTTP request header, and the second part is the request message body. It contains an Extensible HyperText Markup Language (XHTML) document: the root element is an `<html>` tag and it has one child: the `<body>` node. The `<body>` consists of a JavaScript node and a `<form>` element. The browser of the user processes the `onload` event handler and its containing JavaScript command. As a result, the browser submits the form on load.

```
HTTP/1.1 200 OK
Date: 21 Jan 2004 07:00:49 GMT
Content-Type: text/html; charset=iso-8859-1

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/
    xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
   <body onload="document.forms[0].submit()">
      <noscript>...</noscript>
      <form action="https://IdentityProvider.com/SAML/SLO/Response" method="post">
         <div>
            <input type="hidden" name="RelayState" value="..."/>
            <input type="hidden" name="SAMLResponse" value="..."/>
         </div>
         <noscript> <div> <input type="submit" value="Continue"/> </div> </noscript>
      </form>
   </body>
</html>
```

Listing 2.9: SAML response embedded into HTML form control [16, page 25]

The content of the form has two attributes set: an `action` and a `method`. The value of the `action` attribute specifies the Assertion Consumer Service (ACS) of the SP. Additionally, the SAML message and other parameters are embedded in the form.

Based on the `method` command within the form control, the browser sends an HTTP POST message to the ACS URL stated by the `action` attribute value. The parameters which are defined in the hidden input tags represent the encoded SAML message and the `RelayState`. Both are transmitted within the body of the HTTP POST message.

It is recommended to use HTTP POST binding instead of HTTP Redirect binding. Word-wrapping is allowed, and processing of the POST parameter value is considered much more robust compared to parsing of the location header. The content length is set as a header value and therefore, the data size is known before it is transmitted. Data that is transmitted as a POST parameter can even be chunked. This is called *chunked transfer encoding* and has a few advantages over using the location header for transmission. In practice, the HTTP POST Binding should be used to avoid unpredictable behavior due to long URL parameters.

2.4.3 Message Exchange Protocol

A protocol is a collection of rules, which defines how messages should be exchanged between communicating parties. Derived from the SSO concept in section 2.2 on page 4, SAML defines three parties: the SAML requester, the SAML responder, and the user.

Each of them might initiate a conversation. However, the user is usually the initiating entity, because it chooses the resource to be accessed. To do so, the user utilizes a browser to initiate the SSO authentication process. Because a browser is an essential part of the procedure, this scenario can be called "browser-based SSO".

Once the user has signalized his request for access, the subsequent procedure can be of two types: initiated by the Service Provider (SP) or initiated by the Identity Provider (IdP) depending on which party is involved after the user has utilized his browser. Both procedures are introduced in the following two sections.

2.4.3.1 Service Provider Initiated SSO

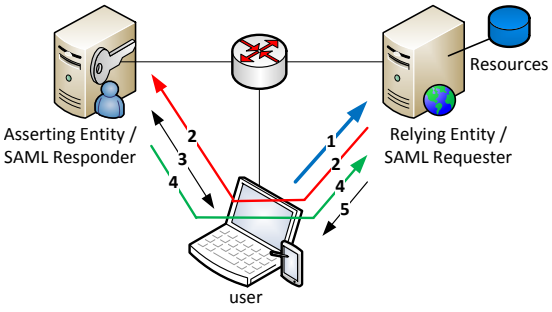


FIGURE 2.5: Scenario of SP initiated SSO message exchange

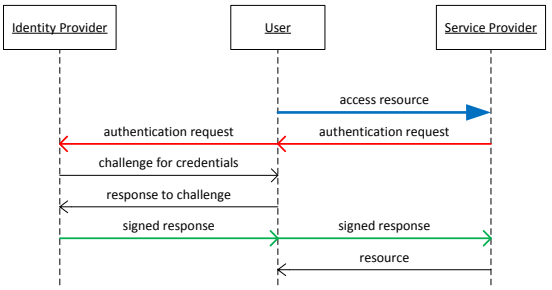


FIGURE 2.6: Sequence Diagram of SP initiated SSO

On the left side, Figure 2.5 is depicted, and it shows the well-known SSO scenario (see Figure 2.1), which is now covered with an illustration of the SP initiated message exchange flow. On the right side, Figure 2.6 is a sequence diagram, which is a more formal representation of the same exchange. Both illustrations are based on Figure 1 of [26, page 2] and also [12, page 42]. Red arrows represent the message transmission that initiates the authentication process. Green arrows represent the transmission of the security token, i.e. the SAML response which is sent, in response to the authentication request. The blue arrow indicates the user request for a particular resource.

1. **Access Resource** First of all, the user signalizes his intention to access a protected resource stored on the SP system.

2. **Request Authentication** Because the user has not been authenticated to the SP yet, the provider does the following: the system creates an initial authentication request and sends it to the user who redirects the request message immediately to

the IdP. The request contains a relocation URL to specify that it should be redirected to the IdP. Then, the forwarding of the request is performed by the user. At this point, it becomes clear why this protocol type is called that way: the SP is initiating the authentication process. More precise: the user receives an HTTP message from the service provider, which contains an encoded `<AuthnRequest>` message. This message may contain some meta information like the expected binding choice. Then, the browser of the user redirects this message to the consumer service of the IdP. This process is usually completely hidden from the user and requires no user interaction.

3. **Authenticate** In case the user is not authenticated against the IdP beforehand, a challenge response procedure is executed at this step in the protocol. The user is usually challenged to type in a password for the requested account. If the password is accepted, the protocol is continued by the IdP. This step is omitted, if the user is previously authenticated to the IdP.

4. **Send Response** Derived from the authentication request, a security token, i.e. a SAML response, is created and send to the user. Then, the response is forwarded to the SP by the user.

5. **Transmit Resource** The service provider verifies the validity of the security token and if this check succeeds the resource is transmitted back to the user.

2.4.3.2 Identity Provider Initiated SSO

In contrast to the SP initiated SSO scenario, the Service Provider (SP) is passive and just reacts to incoming messages. Both Figures 2.7 and 2.8 are quite similar to the

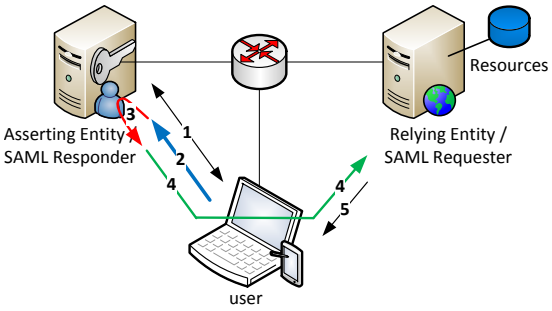


FIGURE 2.7: Scenario of IdP initiated SSO message exchange

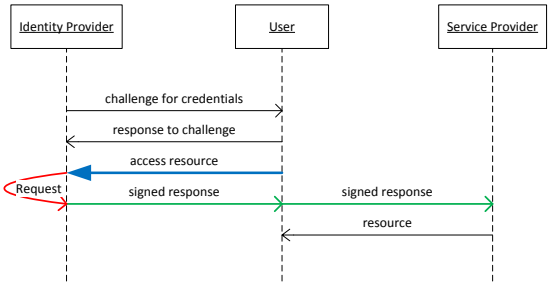


FIGURE 2.8: Sequence Diagram of IdP initiated SSO

previous two. On the left side, there is a more realistic view, and on the other side a formal sequence diagram. The illustrations are based on [12, page 45]. The markings with colors (blue, red, and green) mean the same as previously defined.

1. **Authenticate** First of all, the user is authenticated against the IdP. A challenge response procedure is executed. The user is usually challenged to type in a password for the requested account. If the password is accepted, the user ets an overview to all resources that now can be accessed.

2. **Access Resource** The user signalizes his intention to access a protected resource. In comparison of both protocols, this step is significantly different from the equivalent protocol message in the SP initiated procedure, which is step 1: On the one hand, the user sends a request message to the SP directly, to get access to the resource. On the other hand, he initializes the protocol by signalizing it to the IdP (e.g. a mouse click on a button).

3. **Request Authentication** The request for authentication is equivalent to step 2 of the SP initiated protocol sequence. The difference is, that such a message doesn't really exist in the IdP initiated protocol. This is because it is implicit with respect to the access message of step 2. Therefore, this request and the corresponding check is handled internally by the IdP.

4. **Send Response** A security token, i.e. a SAML response, is created and send to the user. Then, the response is forwarded to the SP by the user.

5. **Transmit Resource** The service provider is always listening to incoming responses. It has a dedicated component for that, namely the Assertion Consumer Service. It verifies the validity of the security token and if this check succeeds the requested resource is transmitted back to the user.

2.4.4 Combinations of Message Exchange and Message Encoding

This part is about possible combinations of message exchange protocols (e.g. SP init or IdP init) with bindings (see Table 2.1). In addition, the SAML binding specification gives a recommendation which combinations should be used in practice. The SAML binding document [16] is a list of best practices which define a realistic and complete

scenario for each binding type. For example, the HTTP POST binding section in the specification describes a best practice scenario in which HTTP POST binding and HTTP Redirect binding are used in the context of the SP initiated message exchange flow.

Message Encoding / Message Exchange	HTTP POST Binding	HTTP Redirect Binding	HTTP POST and Redirect Bindings
SP init	Yes	Yes	Yes
IdP init	Yes	Yes	No

TABLE 2.1: Protocol flow and encoding combinations ([12, page 42, 45])

For example, the HTTP Redirect, the HTTP POST binding, and the HTTP Artifact binding[4] can be composed with the SP initiated message exchange. So there can be three different types of bindings coupled with one message exchange protocol. In case of IdP initiated message exchange, there can only be one binding combined with the protocol. The SAML standard typically recommends the combination of bindings with the SP initiated protocol [16, page 18, 23, 26]. In addition, an editor of the SAML specification document introduces more combinations (see Eve Maler about SAML Bindings in [12, page 42, 45]).

2.4.4.1 Combinations with the SP Initiated Protocol

Figure 2.9 shows which message encoding method is used for each transmission. Two messages, one in step two and another in step four, are labeled with "BINDING" in the illustration. The transmission method of these SAML messages can be any suitable binding. The immediately following messages are transmitted using message encoding methods, which depend on those binding choices. For example, if HTTP Redirect binding is chosen, the subsequent message is sent via the HTTP GET method and encoded as specified in the standard. The other messages of the protocol are not specified by the binding's choice in this scenario. For example, the SAML standard does not define how step 3 of the protocol sequence is done. Because of that, this arrows of are labeled "unspecified".

[4]The HTTP Artifact binding is not introduced in this work.

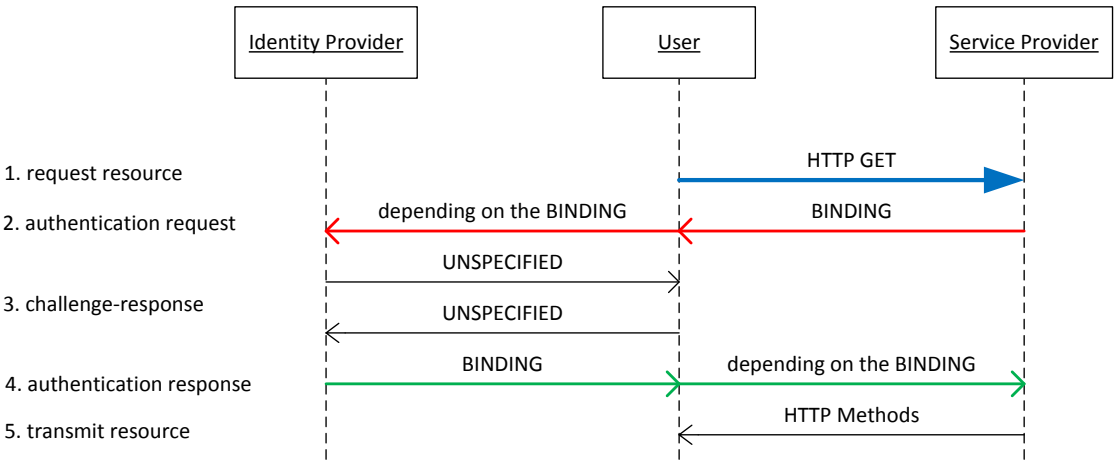


FIGURE 2.9: SP initiated SAML message exchange protocol

The SP initiated message exchange can be combined with two bindings, because there are two steps in the protocol flow to do so: step 2 and step 4. This is why the SAML binding document specifies possible compositions, which are illustrated in the Table 2.2.

Compositions	IdP init	SP init
HTTP POST	Yes	possible
HTTP Redirect	Yes	possible
HTTP Artifact	No	possible
HTTP Redirect and POST	No	Yes
HTTP Redirect, POST, Artifact	No	Yes

TABLE 2.2: Binding compositions for the SP initiated message exchange [16].

To be precise, Figure 2.9 can be combined with the HTTP POST binding in step four of the protocol sequence. If the HTTP POST binding is requested by the SP (in step 2), the user receives an HTTP POST message containing an HTTP form control. As a result the SP receives an HTTP POST message in response (step 4). This HTTP POST message has a body part and it contains the SAML response as base64 encoded POST parameter.

2.4.4.2 Combinations with the IdP Initiated Protocol

Of course, the IdP initiated message exchange can also be combined with different binding methods. Table 2.2 lists possible combinations. In this scenario, the IdP has

full control and the SP is always passive. Because the SP is in a reactive position, it accepts all messages that fit to any binding it can process.

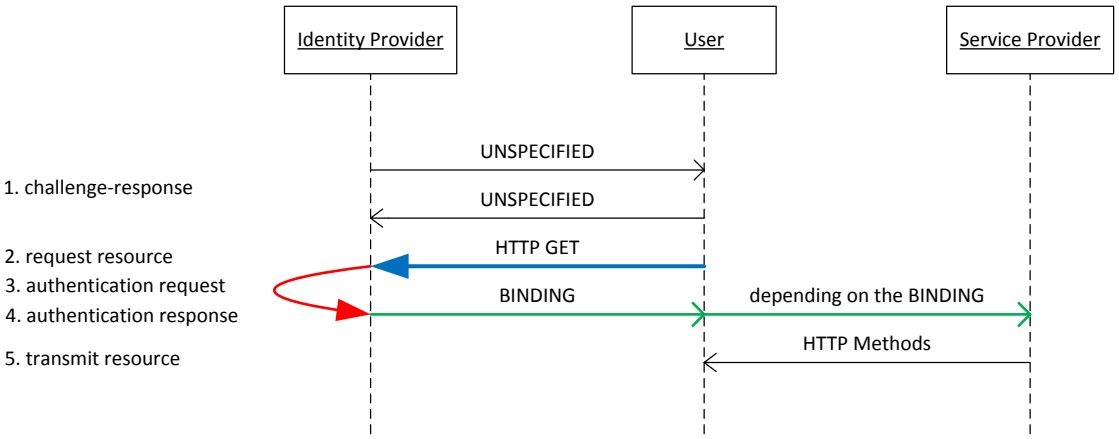


FIGURE 2.10: IdP initiated SAML Message Exchange Protocol

The HTTP POST binding as well as the Redirect binding can be used in combination with the IdP initiated protocol. The user requests a resource from the IdP, which sends an HTTP POST binding message to the user's browser (containing an HTTP form control). It creates an HTTP POST message with respect to the commands embedded in the binding message and then, the SAML message is sent within the HTTP POST body to the SP.

In case of Redirect binding, the SAML message is encoded into the URL query string as previously explained in section 2.4.2.1.

The remaining part of this chapter deals exclusively with attack techniques on the SAML framework and the XML Signature specification. At the end, countermeasures against these attacks are discussed.

2.5 XSW on SAML Messages

The XML Signature Wrapping (XSW) attack was primarily published by Michael McIntosh and Paula Austel in year 2005 [13]. Assuming that there is a separation of security- and application logic, this can lead to serious security problems. Under that circumstance, the security logic verifies the signature and the application logic processes the statement. This can cause a discrepancy between signed statements and actually executed commands. Hence, the core security issue is that, in some cases, the signature

verification and command execution units access different nodes within the XML document. The position of the signed element is declared in the `<SignedInfo>` element and referenced by an ID attribute value (URI="uniqueID"). The `<SignedInfo>` element is signed too and hence integrated in the signature value.

The problem is that some manipulations to the SAML response cannot be detected. For example, if only the `<Assertion>` element is signed and the position within the tree structure is not verified, it is possible to copy this node to another location within the `<Response>`. Unfortunately, the signature remains valid because enveloped XML signature is used, which is not beware of the position of the verified element.

Some researchers could demonstrate that this attack can be applied on real world systems and could be exploited successfully [10, 25]. All protocols that use XML messages and XML Signature to sign data are theoretically threatened. This attack can be applied to SOAP and SAML based authentication protocols, because both use XML signatures and the signed payload is usually referenced by an ID or an XPath expression.

From the perspective of an attacker, the motivation is to mimic all values of a genuine response of the victim, except those which are necessary for signature validation and those which he wants to manipulate. To be precise, the XSW attack does **not** exploit a weakness in the XML Signature verification process A.1.1 on page 70. The intention is to create a SAML response that is almost equal to a genuine message of the victim but extended by a valid signature of the attacker. There are three known types to mimic a victim's response and using the attacker's signature. As a precondition, each variant should be conform to the SAML recommendation. In the following, all three types are exemplified.

2.5.1 XSW on ID-based Signatures in SAML Messages

Figure 2.11 illustrates all different types by an abstract node representation. On the right side, three possibilities for a manipulated SAML message are shown. The left part of Figure 2.11 is a schematic node representation of a SAML response as shown in Listing A.1, but without some less important nodes. XML element nodes are represented by rectangles with a continuous line and attributes are connected to those blocks but have a

dashed line. The blue arrow represents the virtual link which is set by the URI attribute. The `<Assertion>` node is the payload of the message and framed by a green line.

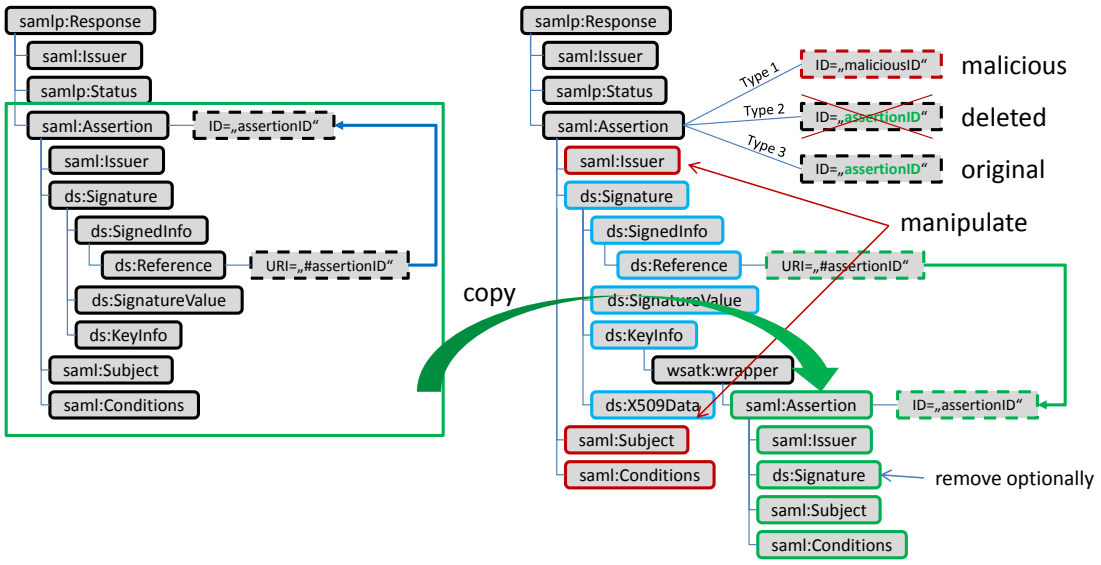


FIGURE 2.11: Schematic XML Signature Wrapping - Three Types

The wrapping procedure is as follows: First, the genuine `<Assertion>` element is copied to another location within the same document. With respect to the ComplexType definition, a standard conform position is within the `<KeyInfo>` node. A `<wsatk:wrapper>` element is created, and the `<Assertion>` is set as a child. Consequently, two `<Assertion>` elements exist in the tree. In general, there are many different positions to place the wrapper element.

Because the `<Signature>` element is omitted during the XML Signature validation process, it can be optionally deleted. However, if the signature logic checks the presence of this node, it should not be deleted to get a stronger attack vector. The purpose to copy the original `<Assertion>` is that the signature verification logic needs this data as input for the validation.

At this point, all other nodes are the same as in the original SAML message. If this response is sent, it would be accepted. Nevertheless, we want to access the victim not the attacker and therefore, some nodes have to be manipulated. These are marked with color red. In case of type one, the `<Assertion>` gets a new attribute value (malicious ID). This node becomes the payload of the attack.

Assuming that the service provider does not check the plausibility of the content, the attacker can manipulate the `<Issuer>`, `<Subject>`, and `<Conditions>` element in any way. Only the `<Signature>` element colored in blue (and the wrapper element) should not be manipulated, because it influences the validation process. The most obvious change is to replace the user name in the `<Subject>` node. If this manipulation is not detected, the service provider grants access to the victim's resources.

2.5.1.1 Type One

All three types including type one are shown in Figure 2.11. Each type is a direct result of the wrapping procedure but with different manipulations. In case of type one, the `<Assertion>`'s **ID attribute is set to a new value**. Additionally, some child nodes which are labeled red are altered with malicious values.

2.5.1.2 Type Two

For type two, the **attribute ID is removed** from the payload element, so that there are two `<Assertion>` elements but only the valid and wrapped instance has an ID attribute set. The malicious `<Assertion>` node has no ID attribute (see Figure 2.11 type 2).

2.5.1.3 Type Three

This type is a direct result of the wrapping procedure and no other manipulations are made except to the child elements of the malicious `<Assertion>` (red nodes). As a result, two `<Assertion>` elements exist in the tree with the **same ID attribute**. The schematic node tree that demonstrates this type of permutation is exemplified in 2.11 type 3.

The third attack type might work, if the application logic as well as the signature logic care about the ID attribute value: only if both IDs are equal, the authentication process continues. Nevertheless, another threat remains: the signature logic does not check the position of the `<Signature>` element within the XML document. The signature logic and the application logic process two elements with the same ID attribute value but at different positions! Therefore, the elements seem to be the same, but they are not

necessarily the same. For example, Amazon's Cloud service was vulnerable to this attack type until the year 2011 [25].

2.5.2 Countermeasures

As an introduction to the topic, Gajek at al. provide a detailed overview of countermeasures against wrapping attacks in [6]. Several countermeasures against XSW on SOAP as well as SAML have been worked out [7][3, pp. 8][26, pp. 11]. The following part is an overview of some countermeasures against XSW on SAML based on those papers.

2.5.2.1 Fixation of the Structure

"Most signature wrapping attacks base on modifying the structure of the original message from the legitimate sender" [7]. The fixation of the structure can be a solution against the XSW attack. However, this contrasts with the SAML recommendation, which has the advantage to provide high flexibility.

2.5.2.2 Check SAML Message Format

The idea is to implement a list of sophisticated checks, that are applied to the incoming SAML messages. These checks can be permanently implemented on receiver side. As a result, the verification logic acts as a filter component. The following checks can be applied:

Element Presence All nodes and attributes that are set as required in the SAML recommendation are present in the received message. And, a signature element must be present in the authentication message. In addition, an `<Assertion>` element exists and has an `ID` attribute set.

Element Position The `<Assertion>` element referenced by the `<Signature>` must be located at the position that is specified in the SAML standard. First, the `<Reference>` element must be a child of the `<Signature>` element and second, refer to an `<Assertion>` element, which itself must be a child node of the `<Response>` root element.

Quantity of Elements If there is more than one `<Assertion>` element with the same `ID` attribute in the SAML response, the application logic should reject the message.

Check Timestamps The application logic should check all timestamps and strictly reject old or expired responses.

Check IDs The application logic should check all IDs and reject the message if it contains an ID which has already been used.

2.5.2.3 Only Process what is Hashed

Another countermeasure can be the paradigm that only those parts of a SAML response are processed by the application logic which are hashed and signed (and thus its integrity is verified). Two concepts can be implemented:

Return Location Hint The signature verification logic returns a boolean value which represents the validation result and additionally, a location hint (e.g. a strict XPath expression) where the signed data is located in the XML document. The application logic evaluates the location hint and only executes commands within the signed part or rejects the message completely.

Filter Elements A similar idea is to filter out (i.e. omit) all elements except the XML element that has been verified by the signature logic. Afterwards, the logic passes the verified node as well as the signature verification result to the application logic. This countermeasure is great, because there is no possibility that the application logic processes an unsigned node of the XML document. This is exactly what the XSW attack exploits and therefore, this solution seems to be a very good countermeasure against it. However, some problems remain: for example, elements that are filtered out may contain relevant information. And if there are multiple signatures within the SAML response message, how to handle it? Third, what about data that is referenced by the signature and located outside the XML document? This data may also be stored in a non-XML compliant format.

2.5.2.4 Separation of Accounts

The service provider can use separation as a concept: Each account has a unique ACS endpoint, and the signature logic restricts the scope of resources, that can be accessed. Depending on the signature, which is included in the SAML message, the system grants access only to those resources that are semantically connected to the signature. As a result, even if an adversary is able to create a SAML message that passes all tests, access to a protected resource of another account is technically prevented. This security concept is implemented by Google to protect its SAML services, for example.

Chapter 3

Approach

This chapter is about my approach to test SAML-based SSO frameworks. The detailed elaboration of the approach is an important aspect of every security audit. The findings from this analysis form the basis for the design of the test tool and the experimental setup for the penetration tests.

First, the security threat is described in the problem statement section. Second, an attacker profile is defined for this work to be clear which preconditions are required. Third, it is given a list of potential targets and vulnerabilities which can be derived from SAML-based SSO scenarios. The section about attack vectors deals with those aspects. Finally, the attack scenario is derived from the preceding analysis and explained in detail. All aspects taken together, form the framework for the practical penetration tests.

3.1 Problem Statement

The authentication process of a web application can be vulnerable. Therefore, it is desirable to actually measure if the service is secure to known attacks. This is called penetration testing. It is extensive, because there are many possible attack vectors. The SAML message exchange can be a suitable and secure authentication protocol, if the implementation of the relying party is secure. For this work, a test tool is implemented to detect XSW vulnerability of a particular SP. XSW attacks are diverse, which means that there are different SAML message formats and several ways to execute an XSW attack.

Moreover, there are several applications for SAML, each with different procedures. The SAML bindings describe how to map SAML to those procedures. For all these aspects, it is time consuming and complex to test the authentication process.

3.2 Attacker Profile

The attacker profile is a list of characteristics, skills and possibilities which describe the realistic capabilities of an adversary. Based on SSO frameworks as a target, the following assumptions are defined about the attacker:

Network Access The attacker has continuous access to the SP for penetration testing, but he does not require any control over the network itself [26, page 2]. Realtime eavesdropping capabilities are not needed and a Man-in-the-Middle (MitM) is not necessary. The attacker has the same access options like a valid user.

Accounts It is easy for an adversary to create a new group, new users and other accounts on the same SP. The victim has an active and valid account at the target SP.

SAML This authentication method must be set up for the victim's account. The SAML endpoints of the SP are accessible via a network which the attacker has access to. The adversary knows a valid but expired SAML response of the victim or the expected message format. Therefore, he knows the register information without private credentials, in some cases. This can be a user name, eMail address, issuer value, ID format, and the ACS URL.

Black Box The attacker does not know the structure or any implementation details of the authentication and validation process of the service provider. In particular, the source code is unknown and no information about the security checks is public. However, if the SP responds with detailed error messages, it may be possible to deduce what parts are verified and how.

3.3 Attack Vectors

Whenever data is transmitted via a network, a MitM attack might be possible. An attacker could eavesdrop or tamper data, which poses a threat to confidentiality and integrity of that information. However, this is not what this work is about. No entity that has access to the message exchange is necessary.

3.3.1 Analysis of the Protocol

Additionally, there exist targeted attacks on SAML-based SSO frameworks: The analysis of the SAML specification can be successful to find a weakness (e.g. a replay attack or information leakage through message malleability), but the protocol can be usually considered safe. A formal analysis of SAML has been published by Armando et al. [1].

3.3.2 Hijack the User

Furthermore, a malicious or hijacked user can cause serious damage during the SSO authentication process. As described in the previous sections, the user is in control over all messages, and thus his behavior is critical for the protocol. If the user's behavior differs from the protocol, this violation may be exploited by an adversary. Although, the user is the least trustworthy of all entities participating in the protocol, he has most control. An adversary could also try to hijack one of the user's software processes and manipulate data on that system. If the user system is hijacked, which tends to be the easiest target, we can go phishing of assertions.

3.3.3 Attacks on the Identity Provider

An IdP is a single point of failure. This is especially attractive to attackers. Identity theft and Denial of Service (DoS) are typical attacks aimed at a centralized system architecture. Moreover, the potential is high to obtain many authentication credentials at once. If the IdP is not secure, reliable and available most of the time for both, i.e. the user and the SP, this represents a significant threat to the overall SSO architecture and must be addressed by the Business Continuity Planning (BCP) of involved companies.

Even if there is no adversary that is threatening the IdP, the failure of it still has an influence on the operational process of the customers.

3.3.4 Attacks on the Service Provider

The adversary can obtain the source code of the relying party in some cases (e.g. if Shibboleth is used [1]) to analyze the code. The review of the code, which is responsible for executing signature verification, authentication and authorization to resources, can be useful to find vulnerabilities.

Finally, another promising attack vector exists: Each SP has to decide whether the incoming SAML responses are valid and genuine or not. The decision process is based on XML Signature verification and additional format and plausibility checks. For this work, we exploit that the decision-making process is not always correct. Derived from this attack vector, the following section is about the attack scenario, which has been designed to test this attack idea in practice.

3.4 Attack Scenario

The idea is to emulate all involved parties by the test tool, except the service provider itself. In contrast to the standard-compliant SSO scenario on page 4, only two parties are involved in this attack scenario, which is shown in Figure 3.1. The third party which is the IdP is emulated by the test tool. This is possible, because an adversary can create his own asserting party with a self-signed certificate. There is no Chain-Of-Trust that certifies the asserting entity (the IdP). The test tool can be installed and executed on any computer system that has network access to a service provider.

[1] Shibboleth is a free and open-source software that provides Single Sign-On capabilities. An IdP as well as an SP implementation is available. http://shibboleth.net/

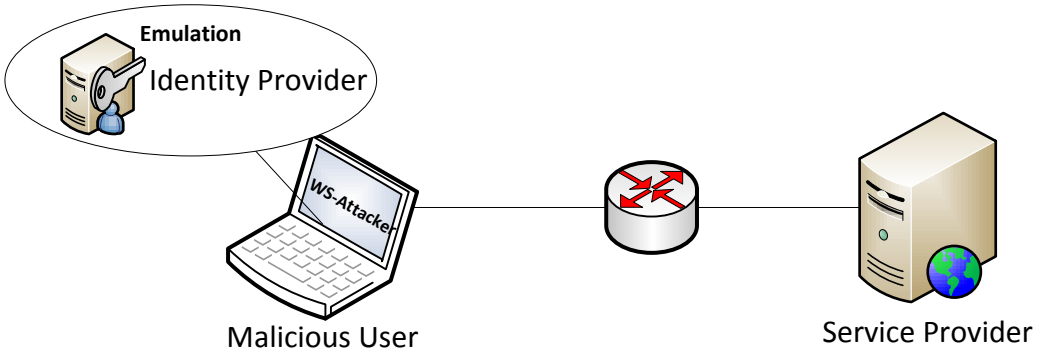


FIGURE 3.1: SSO Attack Scenario

Two types of vulnerabilities can be distinguished and tested with the test tool: the XSW vulnerability and a signature verification vulnerability of the service provider.

3.4.1 XSW Vulnerability

Figure 3.2 illustrates the XSW vulnerability. The service provider has a database which stores users and certificates. In this scenario, two groups with two users exist. In this case, only the first group is attacked.

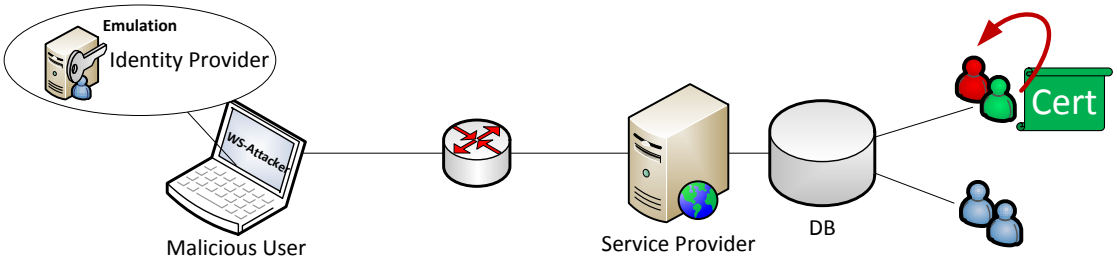


FIGURE 3.2: Attack Scenario: Type One

In the following, some definitions are made:

Account An account is the basic instance, which can be created on SP side. It has at least one user (the account owner itself) and has control over the certificate as well as security configurations.

User A user is one of many identities that have access to the account. Each user can have particular rules and possibilities within the account.

Group A group is a collection of users that share the *same certificate for authentication.* A group is associated with only one account. An account can have multiple groups.

For example, a company has several users in a work group and all have access to their account for collaboration. The administrator has set *one* certificate for the authentication of these employees of the work group.

The attacking user is represented by the green person and the hijacked user is colored in red. The attacker is an internal adversary, which means he is a member of the company. In this scenario, one of the colleagues is the malicious user. He wants to access the same account, but with another user name. Thus, the attack is directed against the own group, which is protected by the same certificate.

To execute the attack, the malicious user creates a valid SAML response, that is signed by the genuine certificate. Then, this message is wrapped, so that the victim's user role is requested. As a result, the malicious user is authenticated as the victim's user. Now he can access and manipulate the account data in the victim's name.

3.4.2 Signature Verification Vulnerability

The second kind of vulnerability can be characterized as a serious implementation and configuration error on SP side. A service provider that is threatened this vulnerability has no secure separation between *accounts with different certificates*. Figure 3.3 illustrates the scenario.

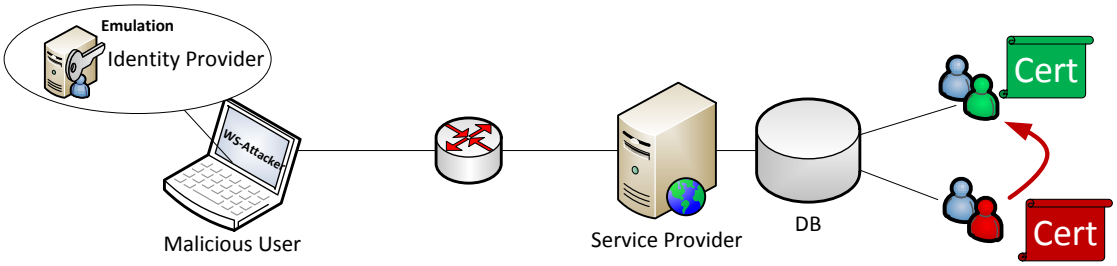


FIGURE 3.3: Attack Scenario: Type Two

Usually, account data and certificates are stored in one single database. The application and signature logic of the SP has access to all of these entries. The attacker is an external adversary, which means he is not a member of the victim's company. He has no access to the certificate and no valid user *within the target group*.

However, the adversary can create a valid account at the service provider with its own certificate. In this case, the red group is controlled by the adversary on the same service

provider as the victim. This is easily possible, because the attacker can create a new account with the public provider for himself. A particular user of the green group is the victim. The green group is an arbitrary work group of another company to be spied. Then, the SAML message which is signed by the certificate of the attacker is wrapped, so that the victim's user name is requested. The SAML request is manipulated, so that almost all values are same as in a genuine victim response. However, the signature is from the attacker not from the victim. As a result, the malicious user is authenticated as the green user, which is member of the other group (certificate verification vulnerability) and not the own group (XSW vulnerability).

The trick is as follows: the signature logic verifies the validity of the signature and the application logic receives the success signal. The application logic is confident that the SAML response to be processed is correct (valid signature) and executes the malicious payload (inserted based on XSW). The payload contains the command to provide access to the victim's account, which is protected by the victims certificate and not that one of the attacker. If the attack is successful, the application logic does respond with data of the victim, although the certificates *don't* match (because the signature logic has verified successfully the attacker's signature)

Chapter 4

Software Design Concepts

This chapter is about important design concepts which have been utilized or developed for this work. To meet the requirements with respect to scalability and flexibility, the software architecture of the test tool should be clever. In the following, the design concepts are described in detail.

On an abstract level, the test tool is composed of a couple of modules that interact with each other. The main goal of this concept is to manage the complexity of the software. Procedures that are related are composed in its own module.

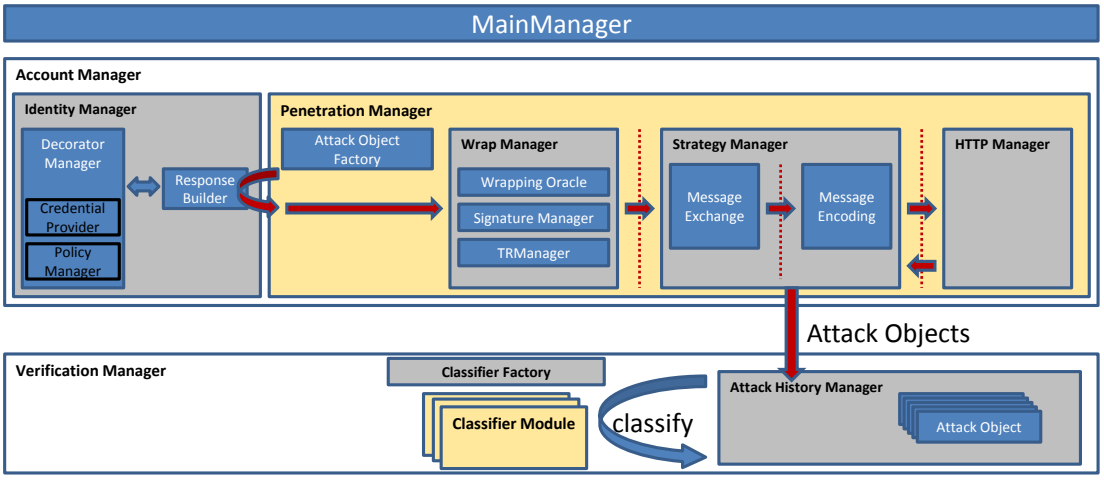


FIGURE 4.1: Module Hierarchy

Figure 4.1 is a high level illustration of the most important modules. All modules have hierarchical relationships and implement useful design concepts. The following sections

describe the functionality of each module and its internal design principles. All sections refer to the module hierarchy which is shown above.

4.1 Main Manager

This is the highest level class in the module hierarchy. Its task is to automize and create the other modules. Each module has its own automatisms, but in some cases a more abstract command and control unit is necessary. No special design concepts are implemented.

4.2 Account Manager

The account management is handled by the account manager, which is directly subordinate to the main manager and at the same hierarchy level as the verification manager. It stores the current status information of the account: e.g. the account name and the error states of components.

Its task includes the creation of the identity manager (4.3), the penetration manager (4.4) and the service manager. The service manager is not shown in the Figure 4.1, but its hierarchy level is the same as of the identity and the penetration manager. The service management provides information to the other managers about the SP like the service URI, the ACS URI, and the relay state.

Furthermore, the account manager is responsible to mediate the communication between the penetration manager and the identity manager. For example, if the penetration manager has an update for the identity manager, the account manager can allow or deny this changes.

4.3 Identity Manager

As shown in Figure 4.1, the identity manager is a sub-manager of the account manager. All processes of the IdP are implemented in this module, so that the purpose of this module is to manage the creation of SAML responses.

The most important classes are the decorator manager, the credential provider, and the response builder (several other classes of the identity manager are not depicted in the Figure). The following list describes the functionality of important subordinate managers in short.

Response Builder The response builder implements the management of two functions: First, it manages the build, marshal, and sign procedure.[1] And second, the component can parse a text file containing a SAML response into an OpenSAML Response object. The OpenSAML library [2] provides a message parser for that. The resulting OpenSAML Response object can be passed to the decorator manager on the left side in the Figure. The other way around is also possible: the response builder can create an OpenSAML Response object based on the configuration of the decorator manager. The concept of decoration is explained in detail in Section 4.3.1.

Decorator Manager A SAML message has a tree structure and is based on the XML specification. Both standards are flexible and hence the messages can be manifold. The IdP module should be capable to create any SAML message. To address this, we chose a suitable design pattern for its inner logical structure: the decorator pattern (Section 4.3.1). The decorator manager is a sophisticated implementation of this pattern. The builder can create decorator objects and extensions. Both represent a particular structure, i.e. a template or scheme, for SAML messages. These messages can then be generated multiple times based on this template. They have the same structure, but different values (timestamps, digest value, etc.). The decorator concept is flexible and hence the scheme can be altered. Consequently, a user of the test tool can create almost all kinds of SAML messages.

The credential provider provides cryptographic functionality to the decorator manager. It is at the lowest level of the hierarchy. At some point of the creation process, the message contains all necessary elements, but without cryptographic security. The signature is missing: there is no digest value, no signature value and no key info set in the message. The responsibility of the credential provider is to create

[1] Marshalling is the process of transforming an OpenSAML object to a data format (DOM, Doc, XML).

[2] https://wiki.shibboleth.net/confluence/display/OpenSAML/Home

(or load) RSA key pairs and X.509 certificates. It is bundled with the decorator manager and hence, even lower in the module hierarchy.

The policy manager stores many restrictions and SAML recommendations that influence the creation process of the decorator manager. It is at the lowest level of the hierarchy similar to the credential provider. Its responsibility is to provide official tag names and attribute values to the Decorator Builder. To create an XML message, which is conform to a SAML response message, the software must know how to name the elements; which attributes are allowed for each element; and what values can be set, if it is restricted.

4.3.1 Decorator Pattern

In the context of software architecture, there exist several patterns to simplify development [5, pp. 79]. The decorator pattern is one of these design patterns. It is similar to subclassing. Figure 4.2 depicts an abstract illustration of the decorator design pattern in Unified Modeling Language (UML) notation.

This pattern is used to create dynamically a composition of basic objects and multiple extensions to these. The fundamental idea is to alter the functionality of an object at run-time by addition and recombination of extensions. Each extension is independent and flexible to use. They are independent, because a modification of one extension does not affect the others. And their usage is flexible, because a general interface is implemented by all instances and therefore, the order in which extensions are added or processed is variable. Moreover, a nested tree structure is possible by implementing a decorator pattern within an extension.

These characteristics of the decorator pattern make this software design a suitable representation of SAML responses.

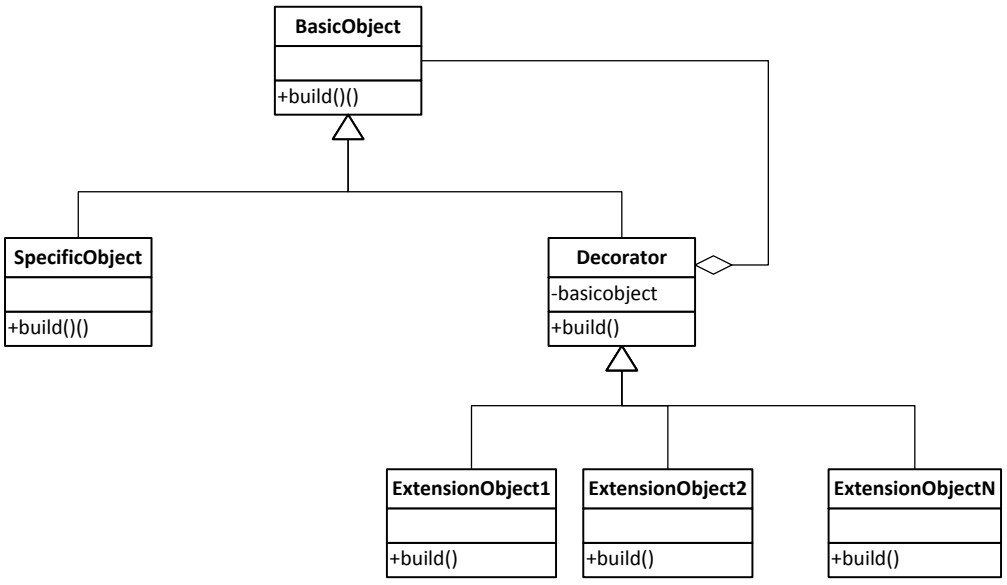


FIGURE 4.2: UML Illustration of the Decorator Pattern

The decorator pattern is some kind of linked list. It is apparent from the illustration in Figure 4.3. The Figure shows the same decorator pattern but in a different view, which emphasises the pointers of each object.

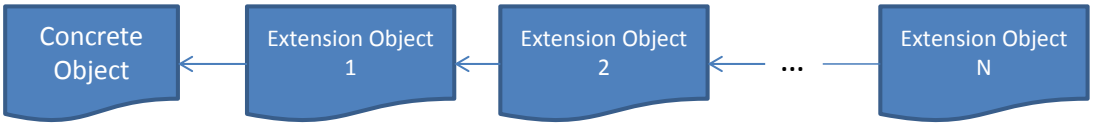


FIGURE 4.3: Decorator Pattern Linked List Illustration

Depending on how the extensions are implemented, the order of execution can change. However, the inherent execution sequence of the decorator pattern is as follows: First, the basic object is created, and then extension objects 1 to N are created. Second, the build method of extension object N is called. Object N itself points to extension object N-1, and that build() method is called. In the following, the execution process continues through all extensions up until extension object 1. In other words, the object creation is carried out in forward direction, whereas method call is propagated backwards through the list.

4.4 Penetration Manager

This manager is responsible for the communication procedure that is conducted with the service provider during a SAML-based authentication process. In addition, attacking this process is a complex task and therefore, an attack management system is also implemented by the penetration manager.

SAML response messages must pass through a series of processing steps before they can be transmitted to a service provider. Each step can be varied and thus might be different in comparison of service providers. Therefore, a pipeline structure is implemented to prepare the messages for transmission as illustrated in Figure 4.4. A pipeline is a collection of software components that communicate with each other only in sequence. The components can be arranged side by side in a single row to illustrate which entities communicate with each other predominantly. This software design is chosen because of its advantages: It allows to represent a process sequence that has multiple stages. Its characteristics are modularity, flexibility and expandability.

To address a more robust execution and to facilitate dynamic interaction with the test tool, the penetration manager runs in its own thread [3].

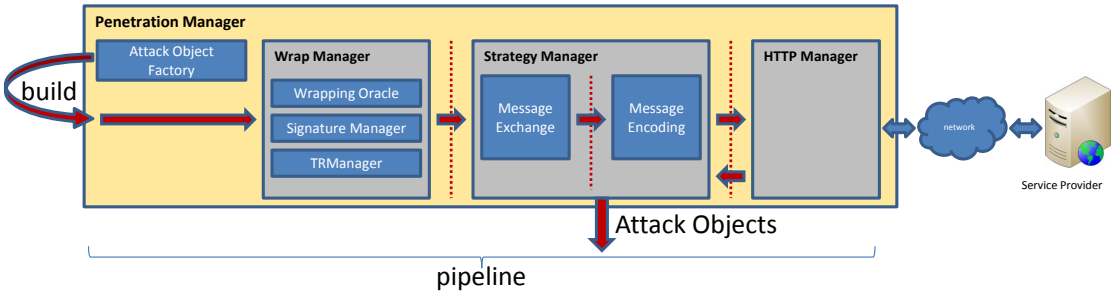


FIGURE 4.4: Penetration manager pipeline

The following list describes each of these pipeline components in short:

4.4.1 Attack Object Factory

This is the origin of each Attack Object instance. This factory creates an empty object each time the penetration manager executes a new run. Then, the identity manager is

[3] If the background of a rectangle is yellow in any Figure of this thesis, this indicates that this part of the software is executed in its own thread.

called to build a fresh SAML Response object. The reference to this instance is set in the new Attack Object.

4.4.2 Wrap Manager

Second column in the row is the wrapping module. The new Attack Object is passed to the wrap manager for process. Its task is to generate a wrapped SAML response message based on the XSW attack. Figure 4.5 represents the internal structure of the module and it illustrates, how the XSW wrapping library of Christian Mainka is integrated into the test tool (for details about the library see [10, pp. 51]).

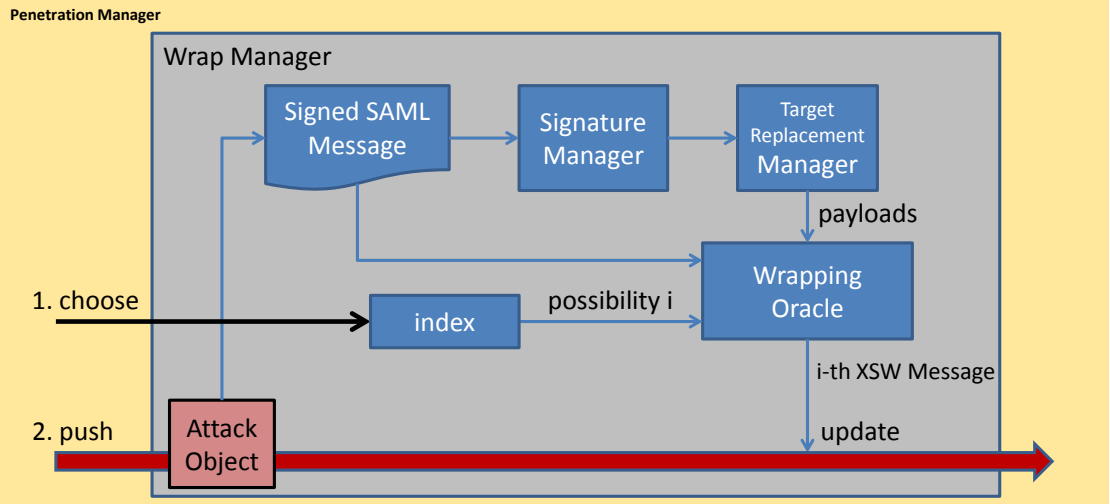


FIGURE 4.5: Wrapping Oracle Integration (based on Figure 28 in [10, page 55])

The penetration manager is in control over the wrap manager and for each run of the pipeline, a possibility is chosen by the penetration management. Once the XSW possibility is set, the attack object is pushed through the wrap manager. This starts the internal molding process:

First, the signature manager is used to identify signed parts of the SAML response. Signed parts are declared as payload elements, so that they can be manipulated. The TR manager (target-replacement manager) is responsible to do malicious manipulations to those payloads. Furthermore, the tool provides an automatism to set up the right values for the malicious manipulations.

Second, the original SAML message as well as the malicious payloads are passed to the wrapping oracle. Based on these inputs, the wrapping algorithm creates a list of

possibilities. Each possibility is a different XSW attack message, i.e. a wrapped SAML message, as described in Section 2.5.1 about XML signature wrapping.

Finally, the Attack Object is updated with the resulting XSW possibility (the i-th XSW attack message) and passed down to the next step in the pipeline.

4.4.3 Strategy Manager

The next module is the strategy manager. It is responsible for the message exchange flow as defined in the sections about IdP initiated and SP initiated protocol flows (see 2.4.3.2 and 2.4.3.1 respectively). Both protocols require sophisticated message handling, that is realized by the message exchange component. Unfortunately, each service provider can have a different message exchange flow due to multiple redirects and session management. To address this circumstance, the strategy component performs extensive redirect management, which is performed by an included redirect manager module (not shown in the Figure). Moreover, the subsequent components may throw errors. For example, if the host is not responding or the SSL connection failed, the HTTP manager module throws exceptions upwards in the pieline to the strategy manager. To address this, the strategy manager is also responsible for comprehensive exception handling.

The message encoding module is next in the pipeline and resides within the strategy manager. Usually, HTTP POST binding is used, but HTTP Redirect binding is also implemented. Each binding specifies a particular message encoding. The encoding is a transformation of a SAML message in XML representation (and some protocol states like the relay state) into a list of encoded parameters. The implementation of this functionality is within the message encoding module. Further bindings as well as protocol flows can be added easily to the strategy manager.

4.4.4 HTTP Manager

Finally, the HTTP manager column is an illustration for the HTTP client implementation. This module accepts encoded parameters from the message encoding module. It is responsible for message transmission via HTTP and handles the HTTP GET and HTTP POST transmission to the service provider endpoints. Its key tasks are to prepare those parameters for HTTP transmission and manage the HTTP client configuration.

Additionally, it is responsible for headers and cookies. The Apache HTTP client implementation is used as default. In addition, the manager provides a history of all HTTP requests and responses which have been sent during the penetration process, so that an analysis of the HTTP message exchange can be done manually.

4.5 Configuration

The configuration management consists of multiple levels, whereas the main manager processes the highest level in this hierarchy. The creation of a configuration file can be an extensive task. To reduce the effort, a hierarchical concept is implemented in the test tool. It is flexible and dynamic. The design idea is as follows: The modules which have settings to be saved provide an interface to get and set a setting object. The main manager makes use of these methods to obtain the settings of those modules. Each module collocates the settings objects of its components.

The final setting is a collection of configuration objects. The task of the each module is to compound all individual settings into one settings object. Finally, the main manager stores the collection of settings into a text file. As a result, the properties and attribute values are represented in an XML structure.

4.6 Verification Manager

The verification module is intended to fulfill the evaluation task. The evaluation includes the assessment of whether an attack has been successful or not. This decision must be made just on the basis of incoming response messages from the service provider. From a theoretical perspective, this is of course only possible if the response messages are significantly different depending on the authentication status of the provider. For example, a standardized HTTP response that contains a random URL pointing to a resource which represents a successful or rejected state is not distinguishable. Fortunately, following the redirect should solve the problem.

Again, the validator is a complex algorithm for classification. Its goal is to distinguish text messages which are HTTP responses incoming from the SP as a result to SAML requests. This decision process is divided into two classification steps. Figure 4.6 is a

state diagram which illustrates the fundamental decision tree of the validator. Step one is marked red and step two is colored blue.

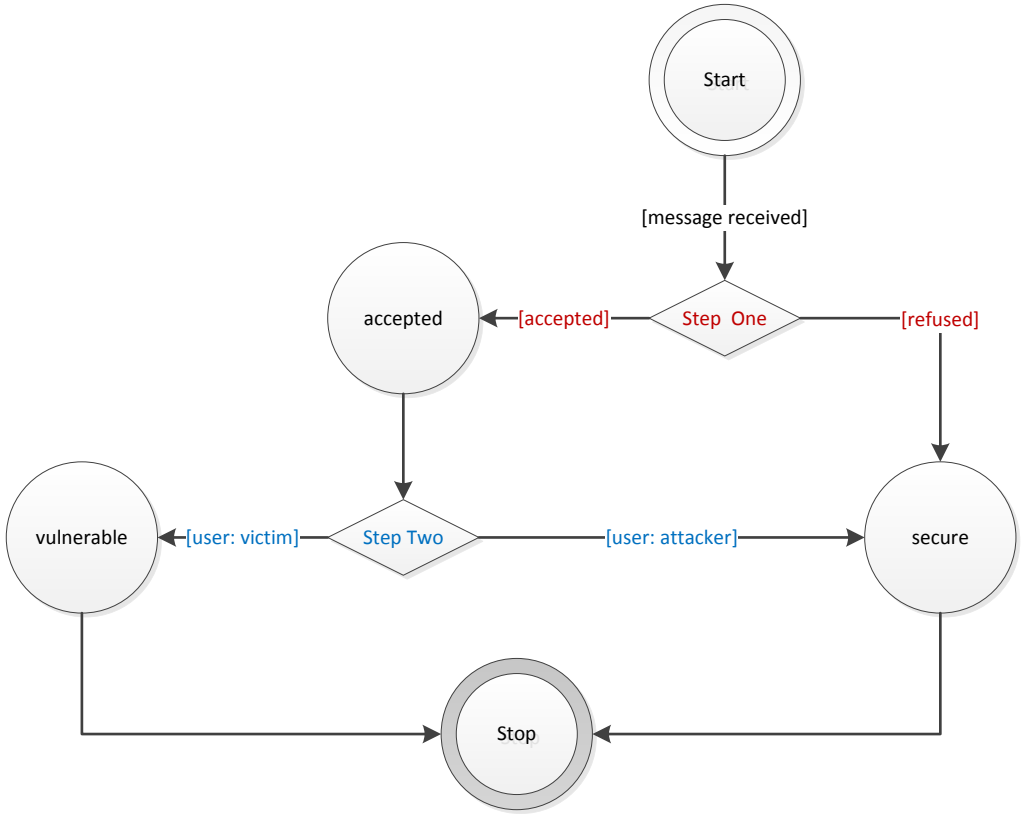


FIGURE 4.6: Validator: Two Steps of Classification

First, the start state is left if an HTTP message (or a sequence of messages because of redirects) is received. This message is passed to a classifier which can distinguish HTTP messages that represent the rejection of the service provider (e.g. an error message) from messages that represent a successful authentication state (e.g. redirect URL to protected resource).

Step two is entered only when state "accepted" was achieved. If this is the case, the distinction is made which account is accepted by the service provider. So, the second step of the decision process can be challenging for the classification algorithm. In essence, the distinguisher of step two must be able to recognize characteristics in a successful HTTP response that identifies a particular user. In some cases, this might only be possible if multiple redirects were followed. Finally, an attack vector, i.e. a manipulated SAML response, is labeled successful if state "vulnerable" is achieved.

Thus two decisions must be made: one in the first step, and another one in the second step. Both decision problems should be decidable using existing methods for natural language processing as explained in chapter 22 of [24]. There are different candidates that exist in theory for a solution of such a decision problem. To identify a sufficient candidate, an overview is created of several methods and practical concepts to solve a classification problem are explained and evaluated in the following subsections. Each of those concepts can be used to implement a distinguisher module that could enable the validator component to classify incoming HTTP responses and therefore "walk" through the decision tree. It is important to note that accuracy and correctness of artificial learning systems is perhaps not as high as necessary.

4.6.1 Classification based on Rules

In most cases, the first approach to design a distinguisher is the definition of rules that are implemented one by one by the programmer. Most programmers nowadays solve decision tasks using rules because it is very simple and each programming language has commands for it. A formal introduction to rule based decision making and information processing is written in German language by Prof. Lunze [9, pp. 91-129].

Algorithm 1 Representation of Knowledge Which is Based on Rules

Require: state
1: **if** state is A **then**
2: actions in response to state A
3: **end if**
4: **if** state is B **then**
5: actions in response to state B
6: **end if**
7: ...

The "state" variable in the pseudo-code of Algorithm 1 can be a simple fact, a condition, or a symbol that represents a complex situation. In case of text classification, indicators might be several key words that identify a class definitely. Other indicators can be text length, number of words, number of characters, language type, or other language specific characteristics.

The following evaluation approach is not adequate for the validation task. On the one hand, rule based classification has relevant advantages: it is fast, simple to implement,

sufficient and quite effective for static scenarios, changes can be applied easily, it is scalable to some extend because new rules can be added, and accuracy is typically high. On the other hand, a rule based classifier has drawbacks: the user or the developer has to create rules, therefore, intensive user interaction is required and even less complex scenarios could require thousands of rules. Even if the number of rules is no problem, the user still remains very important: the user has to formulate the knowledge into a set of rules (knowledge formalization problem [9, pp. 445-463]). In addition, a further disadvantage is that the tolerance is less, i.e. if there exist small unpredictable variances within each class, a rule based approach comes to its limits.

4.6.2 Classification based on Statistical Methods

In general, a vast amount of statistical methods exist to measure numerous effects. A subset of them might be useful for classification (statistical classification) based on indicators like word and character count, frequency analysis, N-gram character models, et cetera. In addition, a probabilistic classifier exists and it is called "Naive Bayes Classifier". It is based on the Bayes' theorem and can do classification tasks.

With focus on our demands, there is one method that is easy to deploy and meets the requirements imposed by the penetration tool: **similarity analysis** of simple character strings. In essence, this design is deduced from the K-nearest neighbors algorithm (see [24, page 738]). It is simple and works well if there is no observation error, which is the case in the validator scenario. Its advantage is, that the classes need not to be linear separable in the input space [24, page 723]. Assuming that the response messages can be manifold, the choice of such an algorithm is quite promising. On the basis of the kNN learning concept, a custom module to classify response messages was designed.

The `simmetrics` library implements a type of algorithms that can evaluate the similarity of two character sequences. It is published under GNU General Public License by the The University of Sheffield, UK, and hosted by Sourceforge.com [4]. Among other algorithms, four statistical methods can be used: Cosine Similarity, Euclidean Distance, Levenshtein, and MongeElkan. Each of these methods compare two character sequences and return a similarity value in the range of 0.0-1.0.

[4] http://sourceforge.net/projects/simmetrics/

To use a similarity value for classification, a concept using this values must be created, that is able to learn from examples. However, there is a problem: A similarity analysis has no 'memory' (a memory is a storage for examples, or the 'experience' in human sense) and, of course, without any 'memory' learning is impossible. To address this, a simple 'memory' was designed: an extensible list of training examples for each category. This storage task can be a List object which stores String objects in Java. On the basis of this design, the classification process may be carried out in different ways:

Maximum Group Similarity An unclassified message is compared to each training message in the memory regardless of in which list the training message is stored. As a result, a list of similarity values is created and then sorted in ascending order. The last entry of the list is the most similar training message with regard to the unclassified message. Finally, the unclassified message is assigned the same category as the training message.

Average Group Similarity An unclassified message is compared to each training message out of one category. As a result, a list of similarity values is created. In contrast to Maximum Group Similarity, the average of all values in the list is calculated. As a result, we get the average similarity of the unclassified message with respect to all messages of one particular category. This procedure is repeated for all lists in the memory. Finally, the highest average similarity value is determined and the corresponding category is returned as the classification result.

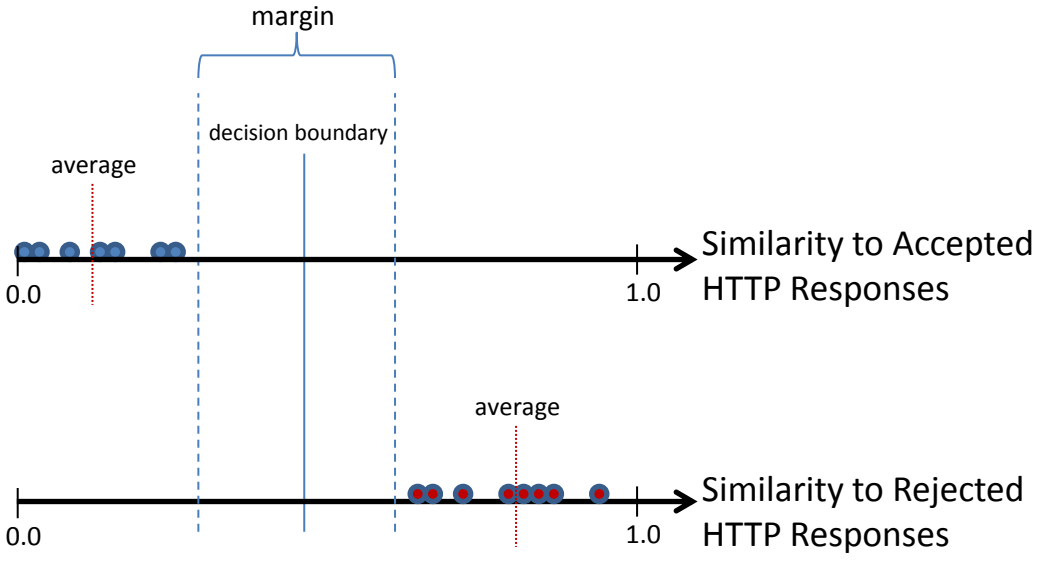


FIGURE 4.7: Classification Example - Two Characteristics

Figure 4.7 is an abstract illustration of the classification process using similarity analysis. Imagine the validator receives a new HTTP message. It is unclassified, so the category is unknown. The classification module of step one must now make a decision whether this HTTP text message represents an "Accepted" or a "Rejected" state. From a previous training phase, some training messages are stored in a list representing "Rejected" and other training messages are stored in a list representing "Accepted" responses from the SP. The similarity of the response message with each training message stored in memory is calculated. Each dot in Figure 4.7 represents a similarity value. To calculate the decision, either Maximum Group Similarity or Average Group Similarity can be used.

However, a similarity based distinguisher has its limits. If the differences between examples that are in the same list of training examples is too high, classification can be wrong. In other words, a high variance within a group results in wrong decisions and hence accuracy is reduced. Also, the similarity of examples within one category must be significantly higher than the similarity to any example of another class.

4.6.3 Classification based on Learning Algorithms

The verification module can be referred to as an agent, as it is used in artificial intelligence. "An agent is learning if it improves its performance on future tasks after making observations about the world" [24, page 693]. Chapter 18 of this book is about "Learning from Examples" and several learning algorithms are explained that could be a solution for the verification task of the penetration test tool.

Several learning algorithms exist for the classification task. For example, artificial neural networks, Bayes networks, or Support Vector Machines [24, pp. 744]. These may be utilized by the verification module. For this reason, the verification manager is implemented modular to facilitate future extensions.

Chapter 5

Implementation

This Chapter deals with the implementation and documentation of the source code and provides a brief insight into it.

For development, the Netbeans Integrated Development Environment (IDE) in version 7.3 has been used together with the Java Development Kit (JDK) version 1.7.0_21. The source code and all other files on the data medium, which is submitted with this thesis, are an integral part of my work.

On the next page, Figure 5.1 shows the dependency graph of the most important packages and classes [1]. To improve the clarity of the Figure, a few elements and their dependencies have been hidden. The overall structure is deduced from the module design, which is illustrated in Figure 4.1 on page 35. In the following, the explanation remains very general, because a detailed description exists as Javadoc and inline comments in the source code itself.

Figures 4.1 (module hierarchy) and 5.1 (dependencies, packages and classes) should be used as a map for the interested Java developer who wants to study the code.

[1] The graph is created with "ispace" http://ispace.stribor.de/.

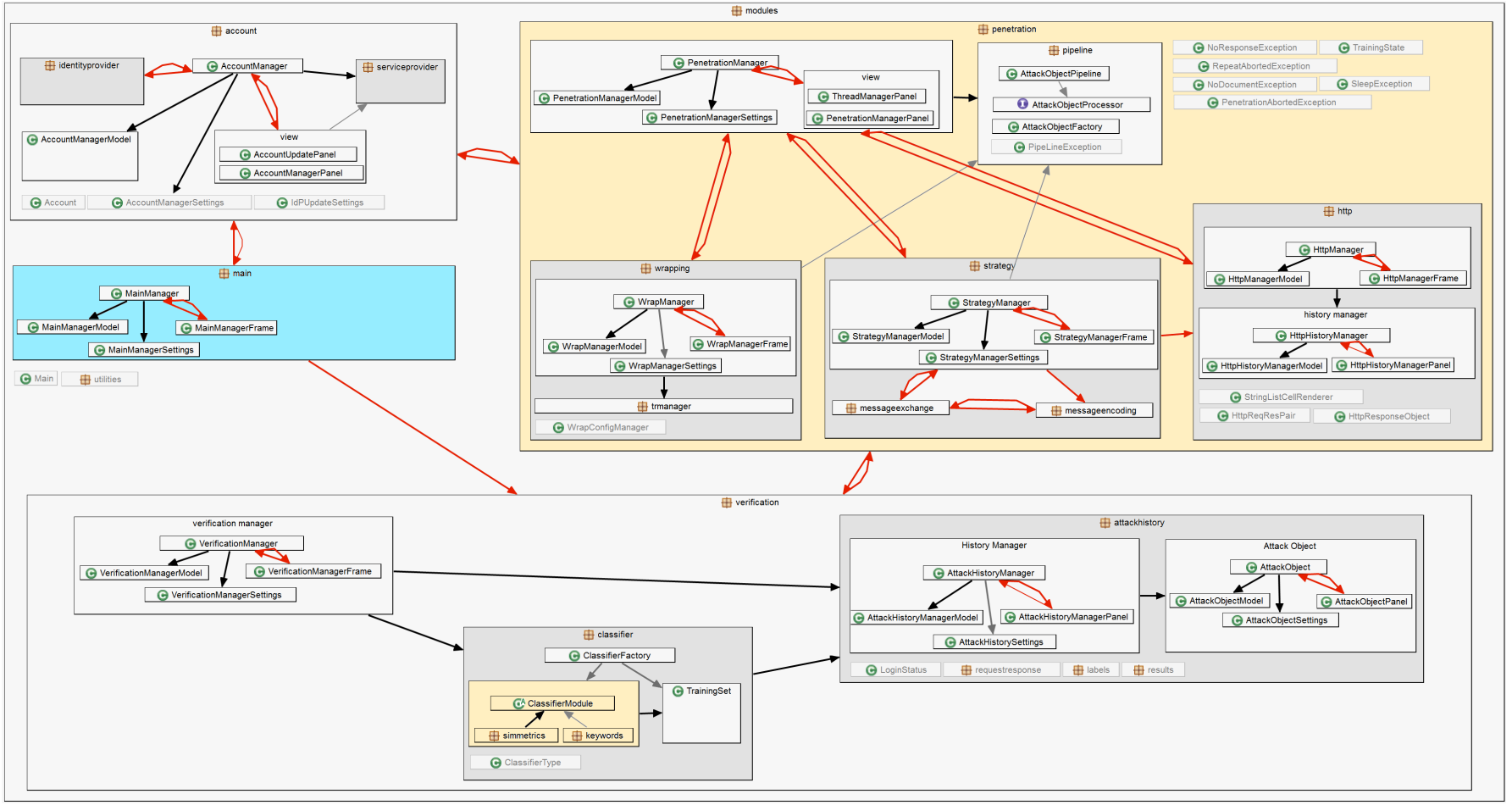


FIGURE 5.1: Dependency Graph - Overview

As expected, the code has much in common with the module hierarchy introduced in Chapter 4. On the left, the main manager is shown and colored in blue. It has dependencies to the `account` and `verification` packages, because it starts the construction process of both. However, is is mostly in contact with the `account` package. Inside the package, there a four classes which implement the model-view-controller concept (MVC compound pattern) with a dedicated settings class. Most of the managers in this Java program are structured in such way. In contrast to the traditional MVC pattern (see [5, pp. 529]), this program implements the MVC pattern differently. In this case, the controller is a mediator and controls all information exchange between the model and the view [2].

Right above the main package, there is the account package, which contains the account manager classes. The MVC compound pattern is represented in the following classes: `AccountManagerModel` (model), `AccountManager` (mediating controller), and two view panels (`AccountUpdatePanel`, `AccountManagerPanel`). Furthermore, the `account` package has two sub-packages: `identityprovider` and `serviceprovider`.

The `identityprovider` package includes many classes and packages, because the identity manager and all subordinate components are located in it. These are the decorator manager, credential provider, policy manager and several others, which all are hidden in the overview for clarity.

The `serviceprovider` package is right to the `AccountManager` class and represents the service manager. Its inner structure is less complex in comparison to the `identitymanager` package because it is not a service provider at all. It only handles several values about the service.

On the right, the `penetration` package is depicted. It contains a `pipeline` package which is a composition of some constitutive classes which implement the pipeline management of the penetration manager. There is a slight difference between the pipeline design and the actual implementation: due to strong coupling of the `PenetrationManager` class on the one side, and the packages `wrapping`, `strategy`, and `http` on the other side, the dependencies between these components of the pipeline are indirect. In other words, based on the pipeline concept, there should be many dependencies between `wrapping`

[2] see mediator based MVC pattern: http://developer.apple.com/library/mac/#documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html

and `strategy` as well as `strategy` and `http`. However, there is a strict hierarchy within the `penetration` package and therefore, the `PenetrationManager` class is always in control over the pipeline process. In addition, all managers in the `penetration` package implement the MVC compound pattern and the strict module hierarchy is maintained.

At the bottom, the `verification` package is depicted. The constructor of its classes is called by the main manager, which shows the red arrow pointing from the left to the package. Another arrow is directed from the `penetration` package. It illustrates the flow of `AttackObject` events (property change events). The `VerificationManager` class is listening for these events from the `PenetrationManager` class and utilizes a model class (`VerificationManagerModel`) and a view frame (`VerificationManagerFrame`). The verification management handles the attack history and the classification process. These functions are grouped together in two other packages: A `attackhistory` package is responsible to store the history of incoming `AttackObject` instances. It contains a history manager and the `AttackObject` implementation itself. These attack objects contain pairs of request and response messages. These pairs are the input for the classification process, which is implemented in several classes within the `classifier` package.

5.1 Third Party Libraries

The following list is a selection of important libraries that are used by the test tool. The libraries are sorted in ascending order by their importance. Most important are OpenSAML and the XSW library:

OpenSAML "OpenSAML is a set of open source C++ & Java libraries meant to support developers working with the Security Assertion Markup Language (SAML)." In addition, "the OpenSAML libraries do not provide a complete SAML identity or service provider." [3]. Therefore, we decided to utilize this library to develop the rudimentary identity provider (within the `identityprovider` package). Classes of internal decorator manager are strongly coupled with methods of this library.

[3] https://wiki.shibboleth.net/confluence/display/OpenSAML/Home

WS-Attacker Signature Wrapping Library This library has been developed by Christian Mainka. For this work, the signature manager and the wrapping oracle implementation is used. For more details see [10], Section 2.5.1 on page 22, and Section 4.4.2 on page 41.

Simmetrics "SimMetrics is a Similarity Metric Library, e.g. from edit distance's (Levenshtein, Gotoh, Jaro etc) to other metrics, (e.g Soundex, Chapman). Work provided by UK Sheffield University funded by (AKT) an IRC sponsored by EPSRC, grant number GR/N15764/01" [4]. This project is used to build the message classifier, which can learn and distinguish HTTP responses.

XStream XML Library "XStream is a simple library to serialize [Java] objects to XML and back again." [5]. It is used to create XML-based configuration files. The advantage is that it does not require any user interaction to load and store Java instances of almost any kind. Hence, it is quite automatic, that's what we want. But it also meets the requirements with regard to flexibility, which we need for the IdP implementation, especially for the decorator pattern. As a result, if the decorator is extended, no additional work is necessary to adapt the implementation of the configuration.

Apache HTTP Client This HTTP client implementation is used to send and receive HTTP messages over TLS. It can manage TLS as well as cookies (using the `CookieStore` class) and it has an easy interface. The corresponding class name is `org.apache.http.impl.client.DefaultHttpClient`.

Bouncy Castle Crypto Provider It is utilized by the `X509CredentialProvider` class, which is responsible for certificate and key pair creation. The credentials can be created, stored or loaded to Privacy Enhanced Mail (PEM) files. To load and store these files, the `PEMReader` and `PEMWriter` classes, both classified in `org.bouncycastle.openssl`, are used. The PEM format is defined in RFC's 1421 through 1424.

[4] http://sourceforge.net/projects/simmetrics/
[5] http://xstream.codehaus.org/

Chapter 6

Penetration Test Results

Chapter four deals with the test results. The penetration program has been applied to several service providers to test their security. The next section introduces the experimental setup.

6.1 Experimental Setup

It is a real world scenario, and because of that, no laboratory limitations exist, which could distort the results. Penetration testing is performed using official URLs and endpoints.

Two Firefox plugins were used to gather information about the services. They are the SAML tracer plugin [1] and the HTTP fox plugin for Firefox [2].

Of course, the principles of ethical hacking were applied only: all tested accounts belong to me, I never tried to access any foreign accounts. If any weakness is detected, all relevant people and companies are informed at first and without publishing critical information to the public.

Because hundreds of erroneous messages are sent to the SP and this looks like a DoS attack, it is possible that the IPs are blocked by the Intrusion Detection System (IDS)

[1] The SAML Tracer plugin version 0.2 by Olav Morken was used. See https://addons.mozilla.org/de/firefox/addon/saml-tracer/.

[2] The HttpFox plugin version 0.8.11 by Martin Theimer was used. See https://addons.mozilla.org/de/firefox/addon/httpfox/.

of the service provider. If this provider is Google.com for example, this would result in serious problems for the IPs that are blocked. To address this risk, a consumer DSL line instead of the University's gateway was used to protect the official IPs. In addition, all attacks were executed with adequate rest period (mostly 5 seconds) between each request. If the request frequency is to high, this could trigger mechanisms to prevent DoS attacks and this would distort the results. The service would refuse any request and thus no analysis of the authentication procedure is possible.

To test a service provider, two valid accounts are created. Then, the test tool is configured to log in to the first account. Next, the malicious payload must be set up and, therefore, the issuer value and the user value, i.e. eMail address or user ID, are replaced. Finally, wrapping is enabled and the software is sending all attack vectors to the ACS URL.

In general, the test tool is very flexible and is able to create numerous SAML responses. Only a fraction of those have been tested, but for each provider I chose the best configuration based on my knowledge. Hence, for those classes of attack vectors I chose, the service provider can be declared as secure.

6.2 Measurements

The penetration test tool was applied to several providers to check if the tool meets the requirements for this work, and to verify the security of those services. The description for each experiment is always the same: First, a company and its service is described shortly. Second, the authentication process which is specific for this provider is explained. Third, a detailed analysis is written down to describe the results and to give a threat assessment.[3]

6.2.1 Google Apps

Google Apps is a service owned by the American company "Google Inc.". It is a platform that provides a collection of several web applications from the portfolio which is offered by Google. The focus of Google Apps is on small business needs. In essence, it is

[3] First, the tests are described in detail and in subsequent Sections these descriptions are kept short.

a collaboration platform for companies with a few employees who predominantly use Google applications like Gmail, Calendar, Drive, Docs, Sites, DoubleClick, Ad Sense, Analytics and others. Therefore, companies with high data privacy needs should check thoroughly if they can trust Google to process their business information. A requirement for registration is to specify a domain the administrator has access to. In contrast to most of Google web applications, this service is not free. But there is a validation period of 30 days.

6.2.1.1 Authentication Process

The authentication process is based on the SP initiated message exchange protocol (see Section 2.4.3.1 on page 16). It uses HTTP Redirect binding to send the authentication request and HTTP POST binding for the response. Therefore, the SAML response consumer service (i.e. the Assertion Consumer Service (ACS)) expects the SAML response and the RelayState parameter encoded in the body of an HTTP POST message. The test scenario and the protocol flow is illustrated in Figures 6.1 and 6.2. In the following, each step of the sequence diagram is described.

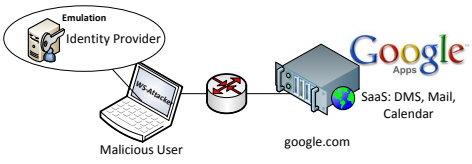


FIGURE 6.1: Google Apps Attack Scenario

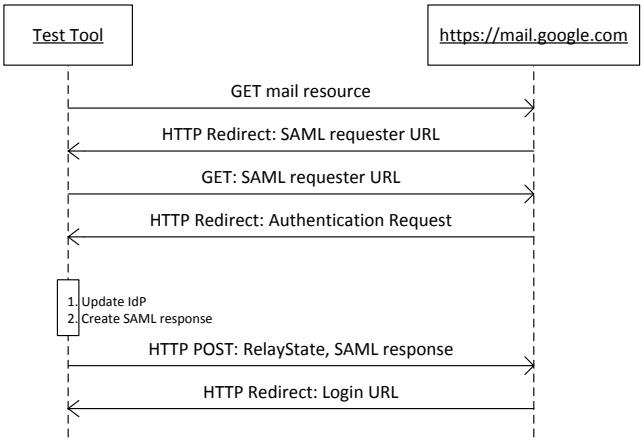


FIGURE 6.2: Google Apps Attack Sequence

1. **Access Message** Google Mail is one of the apps that can be accessed via SAML authentication. Therefore, the URL `https://mail.google.com/a/company/` is chosen. A simple GET message is send to this URL to request access to the mail resource. Because the test tool is not logged in to the service, it is redirected to a SAML requester endpoint, which differs from the service URL.

2. **Authentication Request** The SAML requester endpoint responds by sending an HTTP Redirect. The redirect URL contains the actual SAML request, which is URL encoded into the location header (HTTP Redirect binding).

3. **IdP** The `<AuthRequest>` message is processed by the test tool. First, the IdP configuration is updated by the information given in the request message. The decorator object of the IdP is updated with attributes of the `<AuthnRequest>` message. These values are an `InResponseTo` ID and a URL that is the assertion consumer service endpoint. Also, the requested HTTP method is read and the penetration manager configuration is adapted automatically. Then, a fresh SAML response message is build and signed.

4. **Send SAML response** Finally, the SAML response message is accompanied with the RelayState parameter, that has been received with the authentication request. Both messages are sent to the SP using simple HTTP POST. The XML message representing the SAML response is base64 encoded. The RelayState URL as well as the encoded XML representation are URL encoded and transmitted within the HTTP POST body.

In addition, Google uses session cookies to track a user who is logged in. If the authentication is successful in step 4, the response message contains a redirect URL to merge the authentication session, so that the user is allowed to access the requested resource. The test tool is able to process session cookies and follow redirects if the penetration tester enables this function. For this SP, it was not necessary to follow these redirects and process the session cookies, because no attack possibility was successful and thus, step 2 of the decision process (4.6) was never entered.

6.2.1.2 Configuration

Two Google Apps accounts were created: one for the attacker and another for the victim. For both accounts, SAML authentication was enabled in the web frontend, which can be accessed via the web URL `https://www.google.com/a/COMPANY/`. The SAML authentication is located in "extended tools" → "authentication" → "Single Sign-On configuration". The certificate (created by the test tool) is uploaded to the service. Each account, the victim and the attacker, has its own certificate and private keys. Both

accounts could be successfully accessed with the tool and therefore, authentication using the test tool is valid. Consequently, we can be sure that the configuration is correct for genuine SAML responses before the attack process is started.

Based on my knowledge about the authentication process and how to create malicious payloads, the following configuration and manipulations were applied:

The timestamps are always fresh, so that there is no way to declare the SAML message out of date. A typical Google Apps SAML response has four identifiers: response's `ID`, `InResponseTo` ID, assertion's `ID`, and a `SessionIndex` attribute. All but the `InResponseTo` identifier is created randomly for each SAML message. In addition, the tool tracks all IDs and thus can avoid all identifiers that are sent to the service previously. If set to "SPinit", the test tool uses the ID attribute of the `<AuthnRequest>` to update the `InResponseTo` value.
There are two issuer values in the message: the response issuer and the assertion issuer. Google does not accept a response from an unknown issuer although the embedded assertion is correct and signed. To address this, the issuer value of the `<Response>` element is manipulated, so that it is the same as in the victim's response. The `Destination` attribute is checked because Google uses unique ACS URL's for each customer based on this pattern: `https://www.google.com/a/COMPANY/acs`. For each account, a unique destination exists and can be attacked. Both, the victim's ACS URL and the attacker's ACS URL were tested.
Finally, the `Recipient` attribute value located in the `<SubjectConfirmationData>` element is manipulated, so that it is the same as in the victim's response.

6.2.1.3 Findings

Google's ACS endpoints respond with four different error messages depending on which kind of message has been sent. Hence, there is some state information the adversary can get.

- "This account cannot be accessed because the login credentials could not be verified."
- "This account cannot be accessed because we could not parse the login request."

- "Moved Temporarily"
- "The required response parameter RelayState was missing."

The `RelayState` parameter must be present, but the value itself can be any character string. The change of the transmission protocol from Hypertext Transfer Protocol Secure (HTTPS) to HTTP was rejected by the ACS endpoint. The wrapping oracle found 258 different XSW permutations. No attack possibility was successful for any configuration that was tested. As a result, Google's SAML authentication process can be declared **secure** against the attack possibilities that were checked for this work.

6.2.2 Salesforce

SalesForce.com Inc. is a US-American company. A typical customer of SalesForce is a small or mid-sized business that can buy access to the services. This scenario is called Software as a Service (SaaS). The service portfolio has a strong focus on marketing and sales, therefore it is specialized on CRM and Business to Business (B2B) communication needs. A business manager can fill leads and opportunities, create reports automatically, chat with colleagues or manage contacts and customer information.

6.2.2.1 Authentication Process

The authentication process is based on the IdP initiated message exchange protocol (see section 2.4.3.2 on page 17). No authentication request is transmitted previously. The ACS URL is equal to the service URL and accepts the SAML response as encoded character sequence in the body of an HTTP POST message (HTTP POST binding). A `RelayState` parameter is not used. The scenario as well as the message exchange are illustrated in Figures 6.3 and 6.4.

6.2.2.2 Configuration

The target domain is `https://login.salesforce.com` resolved to IP 204.14.234.101. The HTTP connection is secured by TLS version 1.0 [4].

[4]The network tool 'Wireshark' was used to analyze the protocol `http://www.wireshark.org/`.

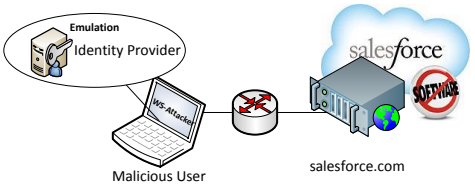


FIGURE 6.3: SalesForce.com Attack Scenario

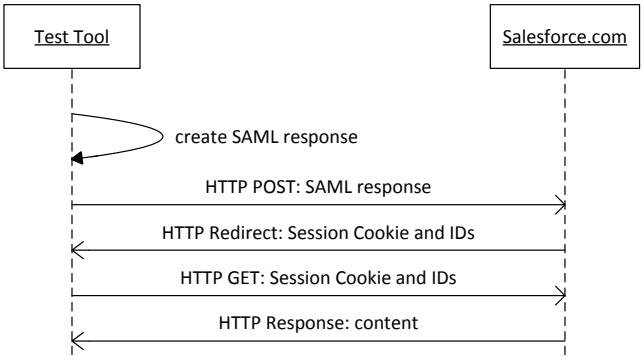


FIGURE 6.4: SalesForce.com Attack Sequence

Salesforce.com provides a SAML response validator (see "Setup" → "Security Configuration" → "Single Sign-On Settings" → "SAML Response Validator"). This can be helpful to understand the verification process (and to find weaknesses).

Several different configurations for the test tool were tested. Timestamps can be set to old, fresh or future values. Furthermore, the issuer value and destination of the response can be set to the victim's value (or deleted). The `<Assertion>` has several values which may be altered: these are the issuer value, recipient, session IDs or an email address, for instance.

6.2.2.3 Findings

The response issuer, the destination attribute, and `NotOnOrAfter` (`<Conditions>` element) can be omitted (standard compliant behaviour) and the service accepts the same response ID multiple times. However, the `IssueInstant` attribute cannot be omitted and it must be fresh. The `AuthnStatement` tag cannot be omitted, but an empty `<AuthnStatement>` element is accepted. An adversary does not need to set the values for `AuthnInstant`, `SessionIndex`, or the `<AuthnContext>` child node. The `AttributeStatement` element can be omitted (standard compliant behaviour). If there is more than one assertion element, the message is not rejected (the response validator displays a warning). Furthermore, differing IDs in the assertion and signature reference are not allowed. The `<Conditions>` element cannot be omitted (not standard compliant behaviour) and the `NotBefore` attribute cannot be omitted. The `<AudienceRestriction>` element cannot be omitted and it must be `https://saml.salesforce.com`. Finally, the `Recipient` attribute (within `<SubjectConfirmation>`)

cannot be omitted. A change to the HTTP protocol was rejected by the `http://login.salesforce.com` endpoint, however it was responding with a SAML error.

The wrapping oracle found 258 different XSW permutations. No attack possibility was successful for any configuration that was tested. As a result, the SAML response validation procedure could not be tricked.

6.2.3 Samanage

Samanage is a company located at North America, Israel, and the Netherlands [5]. It provides SaaS, which is hosted by their cloud service. The application portfolio contains asset management, license management, contract management, risk management, and an enterprise IT service desk, for example. It is mainly funded by Israeli venture capital funds.

6.2.3.1 Authentication Process

Similar to Google Apps, this service uses SP initiated SSO. To access a resource, the HTTP GET method is used.

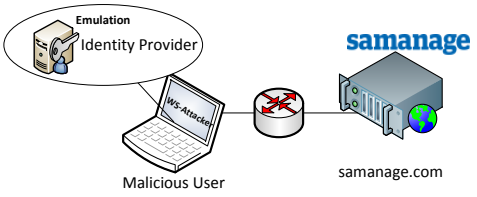


FIGURE 6.5: SAManage Attack Scenario

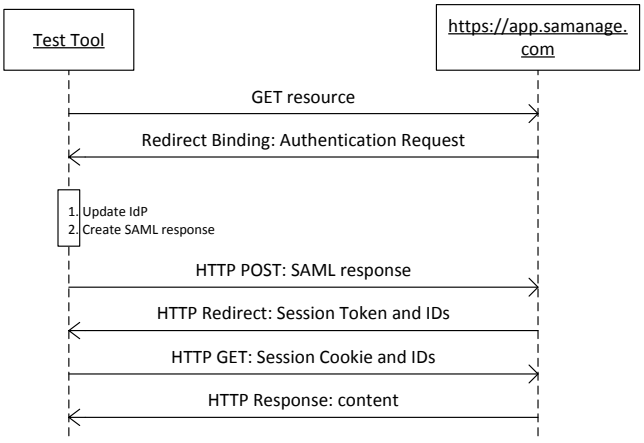


FIGURE 6.6: SAManage Attack Sequence

The service URL endpoint immediately responds with the `<AuthnRequest>` and it does not contain a binding URL (in contrast to Google Apps). If the authentication is successful, an HTTP Redirect message is send to the user. The location is set to `https://app.samanage.com/` and the security token is a cookie that contains an

[5] `http://www.samanage.com/`

"auth_token" identifier and a session ID. The "auth_token" is a unique and persistent character sequence identifying the user account. The session ID is always different and the final secret to get access.

6.2.3.2 Configuration

The configuration for the test tool is almost the same as described in 6.2.1.2. The pattern for the ACS URL is `https://app.samanage.com/saml/ACCOUNTNAME`. Because Samanage as well as Google Apps use a unique ACS URL for each account, both endpoints were tested. Again, multiple configuration sets for the test tool were applied: manipulation of just the user name (i.e. the email address of the account) or the adoption of the `Destination`, `Recipient`, and `Issuer` fields of the `<Assertion>` element were tested.

6.2.3.3 Findings

Samanages ACS endpoints respond with four different error messages depending on which kind of message has been sent. Hence, there is some state information the adversary can get.

- "You are being redirected" redirect to error URL that is specified in the account settings.
- "You have successfully authenticated with SAML, but your user-id USERNAME does not exist in Samanage. Please contact your Samanage account administrator and request to add or modify your user-id."
- 500 Internal Server Error, "Something went wrong"
- Sometimes an SSL error occurred ("Peer not authenticated") and no correct HTTP response were transmitted.

In conclusion, the authentication service can be declared as **secure** against all attack possibilities that were tested for this work.

6.2.4 Clarizen

Clarizen is a service that provides "collaborative project execution software" [6]. Their target group of customers are project managers, sales persons, and financial officers, for example. A user of their services can plan projects, set milestones, upload reports of any kind as well as documents, notes or emails. Also, expenses, hours of work and other resources of the company can be tracked.

6.2.4.1 Authentication Process

This authentication process is based on IdP initiated SSO. The scenario as well as the message exchange are illustrated in Figures 6.7 and 6.8. The steps of the sequence diagram are quite similar to those of SalesForce.com.

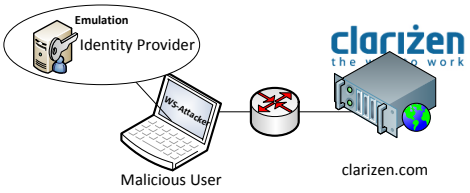


FIGURE 6.7: Clarizen Attack Scenario

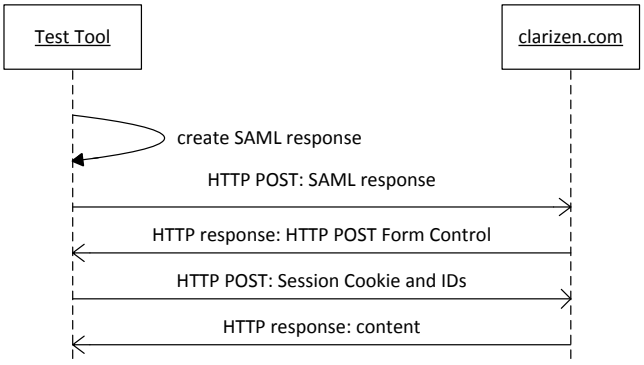


FIGURE 6.8: Clarizen Attack Sequence

6.2.4.2 Configuration

The configuration settings are quite similar to that in 6.2.2.2 for SalesForce.com.

6.2.4.3 Findings

The only value that is checked within the `<Response>` element is the `IssueInstant` attribute. This time stamp can be 59 minutes in the past and the SAML response is still accepted as valid. The `<Issuer>` and the `<Status>` element can be omitted and their values are neglected. If the `<StatusCode>` is set to "Authentication Failed",

[6] http://www.clarizen.com/

the response is still accepted by the service. The `<AuthnStatement>` element is not checked for plausibility (any manipulation seems possible). The attribute values of `SessionNotOnOrAfter`, `AuthnInstant`, and `SessionIndex` can be set to any date or ID and these elements can be omitted. The service responds with the following error messages:

- You are not authorized to perform this operation. Please contact your administrator for further information.
- Failed
- Object reference not set to an instance of an object.

Clarizen is **vulnerable** against the XSW attack as declared in 3.4.1 and the message is manipulated as described in Section XSW - type one 2.5.1.1. The service responds with an HTTP message containing a valid session ID and authentication data of the victim. This should be analyzed by Clarizen.

Chapter 7

Discussion

The idea was to create a prototype, so that numerous of service providers can be tested easily. In the scope of this master thesis, only a few service providers could be tested. However, these can be declared as secure against XSW attacks. A weakness which was found in the authentication process of Clarizen. However, this seems to be not critical. Another motivation for this work was to create a tool for developers of SPs and penetration testers. They can use this tool to analyze their application and measure the vulnerability against XSW easily.

The design and development of an automatic test tool for SAML based SSO frameworks is not trivial and creative solutions are required. At this stage of the development process, the test tool has some limitations with regard to its automatism and classification functionality.

The test tool is not completely automatic: for instance, the penetration tester has to create accounts and to do the configuration of SAML as a service on SP side manually. The automatic configuration management requires some user interaction and inspection. For example, during the training process, setting just the `<StatusCode>` to 'failed' is not sufficient to get negative responses. The automatic configuration of the wrap manager only works, if there are two accounts. In case of other scenarios, the wrapper has to be configured manually.

Furthermore, the verification process can be problematic. In some cases, the accuracy of the classification may be low, so that the automatic analysis cannot be used. In addition,

the prototype has no dynamic process to adjust the parameters of the classifier, because the decision boundaries and thresholds are fixed.

Another limitation is that the test tool is developed for a particular scenario: The penetration tester has access to the SP and is able to create new accounts on the target service. If this scenario changes, which means the attacker cannot create a new account with an own certificate, the applicability of the program is limited. Another scenario would be that the attacker has obtained a SAML response message from a valid account through eavesdropping or interception. He does not have an own account on the target system. At present stage of development the test tool is not capable to load a SAML response as it is into the identity manager. The decoration, build, and sign functions of the identity manager must be disabled and the response builder is extended.

Chapter 8

Conclusion

Today, managing digital identities and credentials is an extensive task for most of the users. An SSO architecture is a suitable solution to solve this problem and, for example, SAML-based authentication can be used to establish user authentication. Unfortunately, the implementation of an effective and robust verification of SAML requests on Service Provider (SP) side is neither a trivial nor a straightforward task. Each time a new SP is created which accepts SAML for authentication, the service could be vulnerable to an XSW attack.

To verify the security of a service provider against XSW attacks, the previous method was to review the implementation manually. However, this is error-prone and time-consuming. To solve this issue, the motivation for this thesis was to implement a test tool in Java, that is able to execute the process of SAML authentication. The test tool has the ability to create malicious SAML responses and to verify the behavior of the SP automatically. Typically, an official IdP is responsible for the creation of SAML responses but in this case, manipulations are restricted. To get rid of these limitations, the goal was to emulate fundamental IdP functionality so that the test tool does not require any external IdP. A requirement is that this implementation should be flexible in a manner that facilitates the program to be used in almost any SAML-based authentication scenario. Another difficult problem, which should be solved, is that the response to malicious SAML messages can be different for each service provider. And additionally, the universal test tool should require as little user interaction as possible.

Therefore, such a universal test tool requires intelligent software design and elegant solutions.

To achieve this goal, the following steps have been carried out: First, the process of SAML authentication has been analyzed. This includes the study of the message exchange, data encoding, and the format of SAML messages. Also, the XSW attack is proposed and countermeasures are consolidate. Second, an elegant software architecture has been designed. Some of the design principles are the decorator pattern, class inheritance and type generalization, concurrency, event propagation, a pipeline structure and a hierarchical MVC compound pattern. In addition, basic principles of machine learning are applied in order to increase the degree of automation. The test tool can create responses and manage the authentication process. An adaptive software component, which is capable of learning, is used to evaluate whether the behavior of an SP can be declared as secure.

As a result, an automatic and universal test tool for SAML frameworks exists now, which can be used to measure the vulnerability of a service provider against XSW. To demonstrate the functionality of the program, the following providers have been analyzed as an example: Google Apps, Salesforce, and Samanage which can resist all of the attacks; and Clarizen which has vulnerabilities.

Future work should focus on software development and security audits. The implementation of new automatisms is a useful extension of the source code. For example, enable the tool to automatically process SAML metadata. Moreover, the implementation of new HTTP bindings could be necessary for particular providers. Another useful step of development might be the integration of this tool into the "WS-Attacker" framework. In contrast to software enhancements, it is doubtless that the search for better attack vectors on SAML, e.g. to enhance the XSW attack if possible, is a bit more interesting for security research. As a result, this would lead to an even more sophisticated signature wrapping oracle. Finally, the program should be used to analyze more providers, so that as many services as possible can be declared as secure.

Appendix A

Appendix

A.1 XML Signature Creation Process

Algorithm 2 Creation: The `<Reference>` element [27, section 3.1.1]

Require: Access to all referenced resources, digest algorithm, transform algorithm

1: **Load** resources R to be signed
2: **for all** resource $r \in R$ **do** ▷ Transformation
3: Apply **canonicalization** (Canonical XML 1.0 or 1.1)
4: Apply `<Transforms>` commands
5: **end for**
6: Calculate **hash** over transformation results using specified digest method
7: Eencode resulting binary value with the **base64** method
8: **Create** a `<Reference>` element and collocate these elements as child nodes:
9: **Add** `<Transform>` element
10: **Add** `<DigestMethod>` element
11: **Add** `<DigestValue>` element
12: **Insert** base64 encoded result in `<DigestValue>` element

The first algorithm takes over the task to prepare the resources. The access to all referenced resources as well as the selection which algorithms to use are a precondition. First, the resources to be signed are loaded and the transformation is applied. The binary representation of the resulting character sequence is hashed using the digest method that is defined beforehand. The hash result is base64 encoded and this string is embedded into a newly created `<Reference>` XML element.

Thereafter, the second algorithm is processed during the XML signature creation procedure. It actually signs the referenced element.

Algorithm 3 Creation: Signature value [27, section 3.1.2]

Require: Algorithm 2 was carried out, `<Reference>` element, selection of algorithms

1: **Create** `<SignedInfo>` element and append these as child nodes:
2: **Add** `<CanonicalizationMethod>` element; value is set to the algorithm's URI.
3: **Add** `<SignatureMethod>` element; set to the URI of the selected algorithm.
4: **Add** `<Reference>` element that is the result of Algorithm 2.
5: Apply **canonicalization** method to `<SignedInfo>` element
6: Calculate **hash** over canonicalization result as specified in `<SignedInfo>`
7: **Sign** the hash using the signature algorithm specified in `<SignatureMethod>`
8: **Base64** encode binary result with the base64 method
9: **Create** a `<Signature>` element and collocate these elements as child nodes:
10: **Add** `<SignatureValue>` element
11: **Insert** insert base64-encoded character sequence in `<SignatureValue>`.
12: **Add** `<KeyInfo>` element

First state of Algorithm 3 is the creation and collocation of the `<SignedInfo>` node. The signature method is identified by an URI which can be `http://www.w3.org/2000/09/xmldsig#rsa-sha1`, for example. An `Algorithm` attribute, which is set to the URI, is inserted into the start tag of the `<SignatureMethod>` element. Next, the canonicalization, hash and sign methods are executed. The binary result of the signature method is base64 encoded. After creation of a `<Signature>` and `<SignatureValue>` element, the character sequence is inserted between the start and end tags of the `<SignatureValue>` node. The result of Algorithm 2 is the `<Reference>` element that is appended at state four. As a result, the XML document is finally signed.

A.1.1 XML Signature Validation Process

Algorithm 4 Validation: The `<Reference>` elements [27, section 3.2.1]

Require: Document to validate; Access to all referenced resources

1: **Canonicalize** the `<SignedInfo>` element.
2: **Load** resources R that are referenced by the `<SignedInfo>` element.
3: **for all** resource $r \in R$ **do** ▷ Validation
4: Apply **canonicalization** (Canonical XML 1.0 or 1.1).
5: Apply `<Transforms>` commands.
6: Calculate **hash** over the result as specified in `<DigestMethod>`.
7: **Compare** hash result with the digest of `<DigestValue>`.
8: **if** hash result equals the digest value of `<DigestValue>`. **then**
9: **continue**
10: **else**
11: **return** false ▷ Validation Failure
12: **end if**
13: **end for**
14: **return** true ▷ Validation Successful

The reference verification algorithm is similar to the creation algorithm with the crucial difference that at the end the resulting digest value is compared with the postulated value. Each resource is canonicalized, transformed and digested using the specified methods. If the computed digest value matches the value of the corresponding `<DigestValue>` element, this process is repeated for the next resource until all hash results passed the comparison. Finally, the algorithm returns true, which represents a successful validation of the `<Reference>` element. Otherwise, if any computed digest does not match the expected value, the algorithm immediately returns false and thus the validation has failed.

Next, the second algorithm is processed during the XML signature validation procedure.

Algorithm 5 Validation: Signature value [27, section 3.2.2]

Require: Signed XML document, genuine certificate, Algorithm 4 successful

1: **Load** the `<KeyInfo>` content from the XML document
2: Apply **canonicalization** method to `<SignedInfo>` element
3: Calculate **hash** over the canonical form using `<SignatureMethod>`
4: **Sign** the hash using the signature algorithm also specified by the `<SignatureMethod>` and the key information obtained with the `<KeyInfo>` element.
5: **Base64** encode binary result with the base64 method
6: **if** hash result equals the signature value of `<SignatureValue>` **then**
7: **return** true ▷ Signature Valid
8: **else**
9: **return** false ▷ Signature Invalid
10: **end if**

Algorithm 5 is the last, but it is a very important procedure. It is a precondition that the previous Algorithm 4 is processed with success and the validity of the certificate (or that one stated in the `<KeyInfo>` element) has been verified. Canonicalization is applied to the `<SignedInfo>` element as well as hash and sign is processed. The resulting base64 value is compared to the genuine signature value which is obtained from the `<SignatureValue>` element. Finally, if all checks are successful, the signature verification logic of the SP returns true.

A.2 Enveloping Signature

"The signature is over content found within an Object element of the signature itself. The Object (or its content) is identified via a Reference (via a URI fragment identifier or transform)" [27, section 10]. In other words, the signed XML fragment is a child node of the `<Signature>` element itself. With respect to the XML signature standard, the signed node should be the last child of `<Signature>`. The signed XML element can be referenced by the ID or by position (e.g. with an XPath expression). Figure A.1 depicts the enveloping structure. As it is known, XML content within a grey rectangle is input for the signature, and the white area is not signed. The `<Signature>` element is an imaginary envelope for the signed `<Object>`.

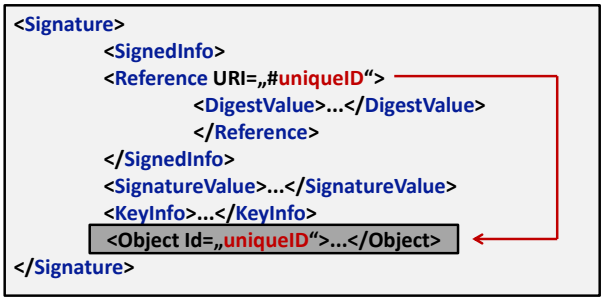


FIGURE A.1: Enveloping Signature

A.3 Detached Signature

This type of signature is illustrated in Figure A.2. The difference in respect to the enveloped signature is minimal in case of "type 1". Again, the signed part is marked with grey and all other parts are white. `<Object>` and `<Signature>` are both child nodes of `<Document>` and thus called siblings. None of them is enclosing the other, in contrast to enveloping and enveloped structures. Referencing the element can be done by using a unique attribute value or an XPath expression. In case of "type 2" the URI attribute points to any resource, e.g. a file or web page, that is located outside the Signature element's hierarchy.

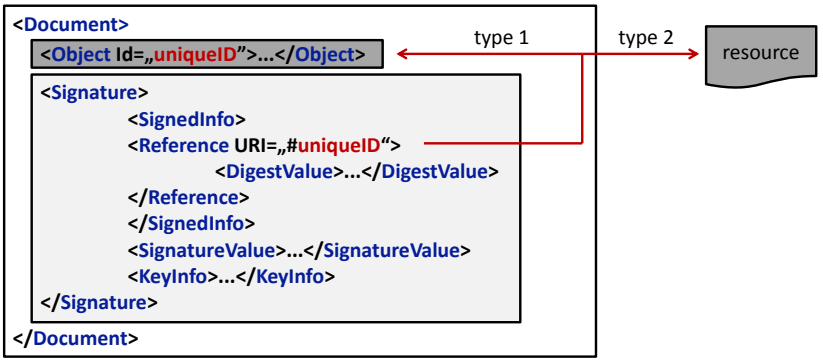


FIGURE A.2: Detached Signature

A.4 SAML Response Example

The following Listing (A.1) illustrates a realistic SAML response. It asserts a statement using an enveloped XML signature which is embedded in the `<Assertion>` node (see also 2.3.1). The example message has all algorithms and method information set to realistic values.

```
<?xml version='1.0'?>
<samlp:Response xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
                xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
                ID="ResponseID"
                Version="2.0"
                IssueInstant="TimeStamp"
                Destination="URL">
  <saml:Issuer>issuer</saml:Issuer>
  <samlp:Status>
    <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success" />
  </samlp:Status>
  <saml:Assertion xmlns:xs="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  Version="2.0"
                  ID="AssertionID"
                  IssueInstant="TimeStamp">
    <saml:Issuer>issuer</saml:Issuer>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <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="#AssertionID">
          <ds:Transforms>
            <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-
    signature"/>
            <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>base64encodedCharacters</ds:DigestValue>
        </ds:Reference>
      </ds:SignedInfo>
      <ds:SignatureValue>base64encodedCharacters</ds:SignatureValue>
      <ds:KeyInfo>
        <ds:X509Data><ds:X509Certificate>base64encodedCharacters</ds:X509Certificate></
    ds:X509Data>
      </ds:KeyInfo>
    </ds:Signature>
    <saml:Subject>
      <saml:NameID Format="urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress">
    username</saml:NameID>
      <saml:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
        <saml:SubjectConfirmationData Recipient="assertionConsumerService"/>
      </saml:SubjectConfirmation>
    </saml:Subject>
    <saml:Conditions>...</saml:Conditions>
    </saml:Assertion>
</samlp:Response>
```

Listing A.1: Basic SAML response example

Bibliography

[1] Armando, A., Carbone, R., Compagna, L., Cuellar, J., and Abad, L. T. Formal analysis of SAML 2.0 web browser single sign-on: Breaking the SAML-based single sign-on for Google Apps. In *6th ACM Workshop on Formal Methods in Security Engineering (FMSE 2008)* (Hilton Alexandria Mark Center, Virginia, USA, 2008), Association for Computing Machinery.

[2] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation, 2008.

[3] Bundesamt für Sicherheit in der Informationstechnik. *XML Spoofing Resistant Electronic Signature (XSpRES) - Sichere Implementierung für XML Signature*. Bonn, Germany, 2012.

[4] Eastlake, D., and Reagle, J. *XML Encryption Syntax and Processing*. W3C Recommendation, 2002.

[5] Freeman, E., Freeman, E., Sierra, K., and Bates, B. *Head First Design Patterns*. O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2004.

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

[7] Gruschka, N., and Iacono, L. L. Vulnerable cloud: SOAP message security validation revisited. *IEEE International Conference on Web Services (ICWS)* (2009), 625 – 631.

[8] Jensen, M., Gruschka, N., and Herkenhoener, R. A survey of attacks on web services. *Computer Science - Research and Development (CSRD) 24*, 4 (2009), 185–197. The original publication is available at www.springerlink.com (May 2009).

[9] Lunze, J. *Künstliche Intelligenz für Ingenieure (Second Edition).* Oldenbourg Wissenschaftsverlag GmbH, Rosenheimer Straße 145, D-81671 München, 2010.

[10] Mainka, C. Automatic Penetration Test Tool for Detection of XML Signature Wrapping Attacks in Web Services. Master's thesis, Ruhr-Univerity Bochum, Bochum, Germany, May 2012.

[11] Mainka, C., Mladenov, V., Somorovsky, J., and Schwenk, J. Penetration test tool for XML-based web services. In *Proceedings of the Doctoral Symposium at the International Symposium on Engineering Secure Software and Systems* (2013), vol. 965, ESSoS 2013.

[12] Maler, E. SAML V2.0 Basics. Tech. rep., Sun Microsystems, Inc., October 2006.

[13] 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), Association for Computing Machinery, pp. 20–27.

[14] OASIS Standard. *Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0*, March 2005.

[15] OASIS Standard. *Authentication Context for the OASIS Security Assertion Markup Language (SAML) V2.0*, March 2005.

[16] OASIS Standard. *Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0*, March 2005.

[17] OASIS Standard. *Conformance Requirements for the OASIS Security Assertion Markup Language (SAML) V2.0*, March 2005.

[18] OASIS Standard. *Glossary for the OASIS Security Assertion Markup Language (SAML) V2.0*, March 2005.

[19] OASIS Standard. *Metadata for the OASIS Security Assertion Markup Language (SAML) V2.0*, March 2005.

[20] OASIS Standard. *Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0*, March 2005.

[21] OASIS Standard. *Security and Privacy Considerations for the OASIS Security Assertion Markup Language (SAML) V2.0*, March 2005.

[22] Raggett, D., Hors, A. L., and Jacobs, I. *HTML 4.01 Specification - Chapter 17 Forms.* W3C Recommendation, 1999.

[23] Reagle, J. *XML Signature Requirements.* W3C Working Draft, 1999.

[24] Russel, S., and Norvig, P. *Artificial Intelligence - A modern approach (Third edition).* Prentice Hall, Upper Saddle River, New Jersey 07458, 2010.

[25] 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. *ACM Cloud Computing Security Workshop (CCSW)* (October 2011).

[26] Somorovsky, J., Mayer, A., Schwenk, J., Kampmann, M., and Jensen, M. On breaking SAML: Be whoever you want to be. In *Proceedings of the 21st USENIX Security Symposium, 2012* (April 2013).

[27] World Wide Web Consortium (W3C). *XML Signature Syntax and Processing (Second Edition)*, June 2008.