FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Web Service Endpoint Protection (WSEP) mit SAML und XACML

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software and Information Engineering

eingereicht von

Martin Unger
Matrikelnummer 0726109

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dr. Wolfgang Kastner
Mitwirkung: Projektass. DI. Markus Jung

Wien, 31.12.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Martin Unger
Alliiertenstraße 5/17, 1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

Simple Object Access Protocol (SOAP)-based Web Services are an essential part of Service-Oriented Architectures. The basic SOAP-standard does not describe access control.

A possible solution is the combined use of Security Assertion Markup Language (SAML)-tokens, eXtensible Access Control Markup Language (XACML)-requests and XACML-responses, to handle control on the Web Service Layer. The main part is a generic protection intermediary, which allows or forbids access of Web Service clients.

This thesis discusses a standards-compliant realization of this concept using a proof-of-concept implementation which is assessed regarding its security.

Kurzfassung

Simple Object Access Protocol (SOAP)-basierte Web Services sind ein wichtiger Bestandteil Service-orientierter Architekturen. Der Basis-SOAP-Standard beschreibt jedoch nicht, wie die Zugriffskontrolle abgewickelt werden soll.

Eine mögliche Lösung ist der kombinierte Einsatz von Security Assertion Markup Language (SAML)-Token, eXtensible Access Control Markup Language (XACML)-Requests und XACML-Responses, um die Kontrolle auf dem Web Service Layer abzuwickeln. Die Hauptrolle spielt dabei ein generischer Protection Intermediary, der die Zugriffe von Web Service-Clients anhand des von ihnen mitgelieferten WS-Security-Headers erlaubt oder verbietet.

In dieser Arbeit wird die standardkonforme Umsetzung dieses Konzepts anhand einer Proof-of-Concept Implementierung besprochen und bezüglich Security evaluiert.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1 Motivation

Die WS-* Standards versprechen einfache Benutzbarkeit und Interoperabilität im Bereich SOAP-basierter Web Services. Die WS-Security Spezifikation beschäftigt sich (zusammen mit ihren Profilen) mit Sicherheitsfeatures, wie zum Beispiel der Signatur oder Verschlüsselung von Nachrichten und der Übertragung sicherheitsrelevanter Informationen in Form von Token. Wie WS-Security Features konkret genutzt werden sollen, um die Anforderungen selbst entwickelter Applikationen zu erfüllen, lässt die Spezifikation großteils offen.

Diese Arbeit beschreibt eine Möglichkeit, die Zugriffskontrolle auf Web Services standardkonform abzuwickeln. Die Funktion der Zugriffskontrolle besteht darin, für einen gegebenen Web Service Client den Zugriff auf eine einzelne Web Service Operation entweder zu erlauben oder zu verbieten. Dazu kommen Security Assertion Markup Language (SAML)-Token in Kombination mit eXtensible Access Control Markup Language (XACML-)Requests und XACML-Responses zum Einsatz.

Wenn im Folgenden von „Zugriffskontrolle“ die Rede ist, so ist ein Mechanismus gemeint, der es erlaubt, den Zugriff auf Web Services zu erlauben oder zu verbieten. Ist von „Web Service Endpoint Protection (WSEP)“ oder von „der implementierten Zugriffskontrolle“ die Rede, so ist die Proof-of-Concept Implementierung gemeint, die im Rahmen dieser Arbeit entstanden ist.

1.2 Anwendungsbeispiel

Eine Firma hat eine Service-orientierte Architektur aufgebaut, um ihre Serverlandschaft zu verwalten. Es existiert ein Service Provider in Form eines Web Service, mit dem Server in das Backup oder Netzwerk-Monitoring übernommen werden können. Weiters ist es möglich, Service Level Agreements zu verwalten. Die einzelnen Operationen bilden eine thematische Einheit. Deshalb macht es Sinn, sie in einem einzelnen Web Service zu bündeln.

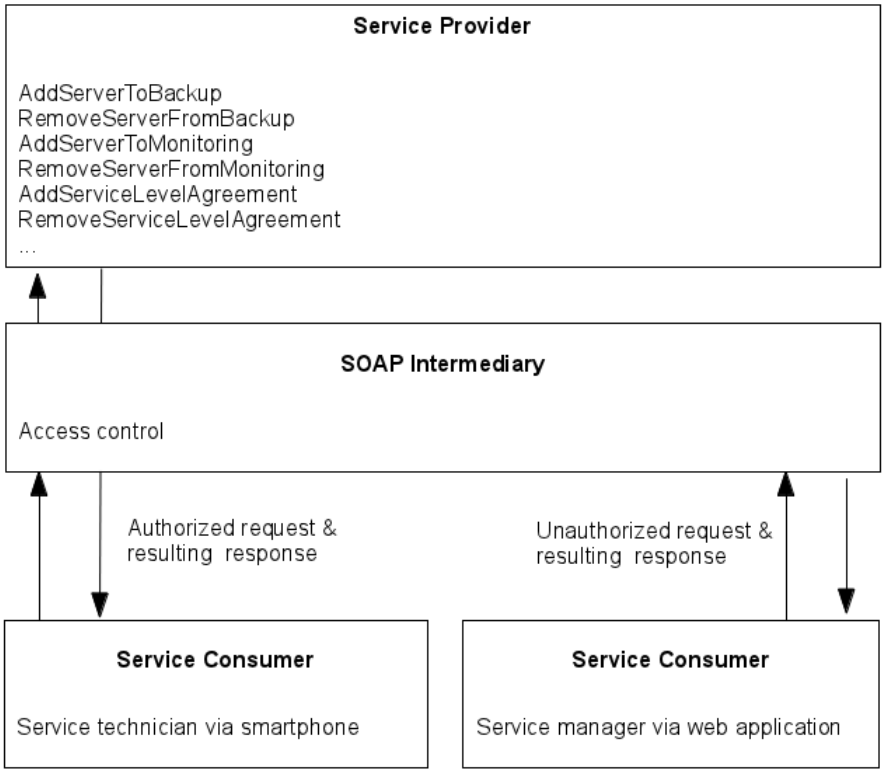

Abbildung 1.1: Anwendungsbeispiel

Viele verschiedene Service Consumer nutzen die Service-orientierte Architektur, aber nicht alle haben die gleichen Rechte. Um zu gewährleisten, dass die Service Consumer nur jene Operationen aufrufen, für die sie autorisiert sind, richten sie ihre Anfragen an einen SOAP Intermediary. Dieser leitet nur Anfragen an den Service Provider weiter, für die eine Berechtigung besteht.

Abbildung 1.1 zeigt ein Beispiel mit zwei Anwendungsfällen. Ein Servicetechniker hat im Serverraum gerade einen Server fertig eingerichtet. Er benutzt eine Applikation auf seinem Smartphone, um den Server in das Backup aufzunehmen. Seine Applikation tritt als Service Consumer auf und wendet sich mit dem Aufruf der SOAP-Operation `AddServerToBackup` an den SOAP Intermediary. Weil eine Autorisierung besteht, wird der Aufruf an den Service Provider weiter geleitet, der die gewünschte Operation durchführt und dem Service Consumer danach direkt antwortet.

Ein Servicemanager will einen Server aus dem Backup entfernen. Da dies eine Aufgabe des Servicetechnikers ist, ist er dazu nicht berechtigt. Allerdings bietet ihm die Webapplikation, die er benutzt, diese Möglichkeit. Sie sendet als Service Consumer einen Aufruf der SOAP-

Operation `RemoveServerFromBackup` an den SOAP Intermediary. Da keine Autorisierung besteht, wird der Aufruf nicht an den Service Provider weiter geleitet. Der Service Consumer wird darüber mit einer Fehlermeldung des SOAP Intermediary informiert.

1.3 Abgrenzung

Der Web Service Client benötigt ein Token, um Zugriff auf die SOAP API des Web Service Providers zu erlangen. Bei dem Token handelt es sich um ein SAML-Token. Es ist also ein XML-Fragment. Das Token wird von einem Token Provider unter der Verwaltung des Web Service Providers abgeholt. Wie in Abbildung 1.2 zu sehen ist, ist der dazu verwendete Mechanismus nicht Teil der für diese Arbeit implementierten Zugriffskontrolle. Denkbar ist zum Beispiel der Transport über HTTP-Requests. Die Funktionstüchtigkeit der Zugriffskontrolle hängt jedenfalls nicht von der Geheimhaltung des Token ab.

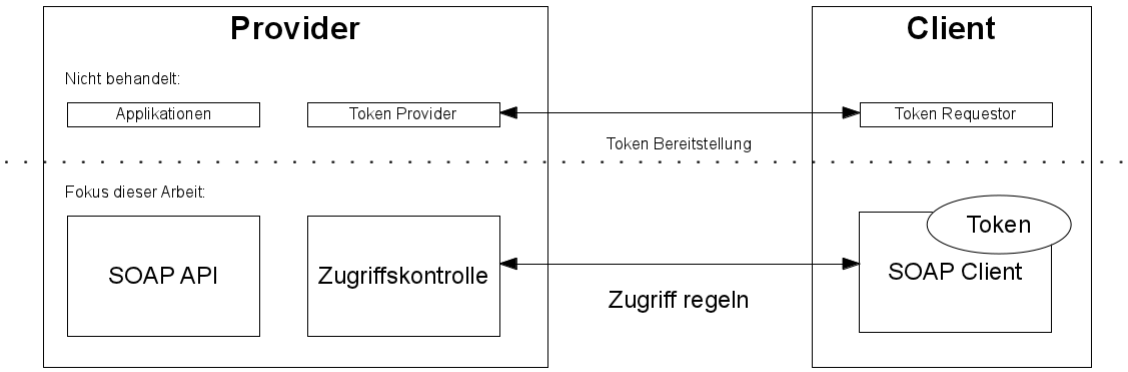

Abbildung 1.2: Abgrenzung

1.4 Funktionsweise

Das Token enthält für ein bestimmtes Subjekt (also für einen durch seinen Public Key identifizierten Web Service Client) die Information, welche Operation der vorhandenen Applikationen über die SOAP API aufgerufen werden dürfen. Eine digitale Signatur sorgt dafür, dass sich potentielle Angreifer nicht durch Ergänzen des Token zusätzliche Rechte verschaffen können.

Für den Web Service Client bleibt WSEP weitgehend transparent und wird durch den Aufruf einer einzigen Funktion aktiviert.

Die implementierte Zugriffskontrolle wird von einem Protection Intermediary abgewickelt. Das ist im Prinzip ein Router auf Software-Basis, welcher Einblick in SOAP-Requests hat. Er sucht in allen SOAP-Requests nach einem Token. Falls er ein Token findet, das den Zugriff auf die gewünschte SOAP-Operation erlaubt, leitet er den SOAP-Request an das tatsächliche Web Service weiter. Andernfalls wird der SOAP-Request verworfen und der Web Service Client

erhält ein SOAP Fault, um ihn zu informieren, dass seine Anfrage verworfen wurde. Das tatsächliche Web Service bekommt davon nichts mit.

Der eigentliche Web Service Provider muss nicht verändert werden, um WSEP zu nutzen. Es muss lediglich verhindert werden, dass Web Service Clients den Protection Intermediary umgehen. Das kann zum Beispiel dadurch geschehen, dass dieser von den Web Service Clients netzwerktechnisch nicht erreichbar ist. Ähnlich wie das eigentliche Web Service, benötigt der Protection Intermediary kein Wissen über die Applikationen oder ihre SOAP APIs. Er kann alleine aufgrund des SOAP-Requests, des darin enthaltenen Token und seiner Policy eine Entscheidung treffen.

1.5 Kategorisierung nach Ort

Da sich die grundlegenden Web Service-Standards nicht mit Sicherheitsaspekten beschäftigen, sind dafür verschiedene Strategien entstanden. Die Möglichkeiten der Zugriffskontrolle für SOAP-Web Services werden im Folgenden beschrieben. Man kann sie anhand des Orts kategorisieren, an dem die Zugriffskontrolle durchgeführt wird.

Transport-Level

Sofern der Web Service Client eine öffentliche IP-Adresse verwendet, kann der Zugriff auf alle Operationen für einzelne IP-Adressen erlaubt oder verboten werden. Der Vorteil dieser Lösung besteht in ihrer relativ simplen Umsetzbarkeit. Es gibt allerdings einige Nachteile:

- IP-Address-Spoofing ist einfach durchführbar. Allerdings ist es durch das Fälschen einer IP-Adresse kaum möglich, unerlaubten Zugriff auf ein Web Service zu erlangen, weil keine ebenso simple Möglichkeit besteht, die Antworten zu erhalten.

- Es handelt sich um einen grobkörnigen Ansatz. Ein Web Service-Client bekommt entweder Zugriff auf alle Web Service-Operationen oder auf keine.

Applikation

Hier kümmert sich die Applikation selbst um die Zugriffskontrolle. Rechte können für einzelne Web Service-Operationen vergeben werden. Die Entwicklung einer eigenen Zugriffskontrolle für jedes Web Service stellt einen erheblichen Aufwand dar und lässt Raum für sicherheitskritische Fehler. Ein möglicher Ansatz besteht darin, eine Login-Operation zu implementieren, die Benutzername und Passwort entgegen nimmt und ein für einen begrenzten Zeitraum gültiges Token zurück gibt, welches bei allen weiteren Aufrufen als Parameter mitgegeben wird. Für applikationsbasierte Zugriffskontrolle existieren vorgefertigte, stabile Frameworks, wie zum Beispiel Spring Security[1].

[1] http://static.springsource.org/spring-security/site/

Web Service

Hier wird der WS-Security Standard zur Abwicklung der Zugriffskontrolle genutzt. Informationen werden im SOAP-Header transportiert und die Abwicklung der Zugriffskontrolle ist für den Web Service Client transparent. Das stellt den Hauptvorteil dieser Variante dar. Der WS-Security Standard alleine ermöglicht jedoch keine Rechtevergabe für einzelne Web Service-Operationen. Man kann ihn allerdings mit Standards wie XACML [8] um diese Funktion erweitern.

Eine sichere Zugriffskontrolle kann nicht erfolgen, ohne die Integrität der ausgetauschten Nachrichten zu garantieren. Abhängig von der Sensitivität der übertragenen Informationen kann auch die Geheimhaltung der ausgetauschten Nachrichten erforderlich sein.

1.6 Kategorisierung nach Verschlüsselung

Möglichkeiten der Zugriffskontrolle für SOAP-Web Services können auch anhand ihrer Verschlüsselung kategorisiert werden, sofern eine solche verwendet wird.

Transport-Level

Die SOAP-Kommunikation erfolgt über HTTPS. Die Nachricht wird auf der Transportschicht mit TLS (Transport Layer Security) verschlüsselt. Diese Lösung ist beliebt, weil HTTPS sehr etabliert ist und der Web Service Client mit dem Serverzertifikat die Identität des Applikationsservers verifizieren kann. Probleme ergeben sich, wenn die SOAP-Nachrichten auch über die Endpunkte der HTTPS-Verbindung hinaus geheim oder zumindest integer bleiben soll.

Message-Level

Die SOAP-Nachricht wird mit XML-Security Mechanismen verschlüsselt und dann innerhalb eines unverschlüsselten Transportmediums (z. B. HTTP) transportiert. Diese Lösung ist Message-Level Security insofern überlegen, als dass die Nachricht über die HTTP-Endpunkte hinaus geheim bleibt und auch über Intermediaries transportiert werden kann, denen man nicht uneingeschränkt vertraut.

1.7 Eigenschaften

Bei WSEP erfolgt die Zugriffskontrolle auf Web Service-Ebene. Sie nutzt keine Verschlüsselung, aber man könnte sie sofort auf Transport-Level Security umstellen. Durch digitale Signaturen wird die Integrität (aber nicht die Geheimhaltung) auf dem Message-Level beibehalten. Sie stützt sich zusätzlich auf den XACML-Standard, um Rechte auf einzelne Web Service-Operationen vergeben zu können.

Weitere grundlegende Eigenschaften sind:

- Standards: Es existieren offene Standards, die die Funktionsweise klar beschreiben. Falls es für eine gegebene Plattform keine Implementierung der verwendeten Bibliotheken gibt, könnte man sie selbst implementieren.

- Assertions: Web Service Clients und Web Service Provider wissen wenig voneinander. Eine dritte Partei trifft Aussagen über den Web Service Client und stellt diese dem Web Service Provider in Form von Assertions zur Verfügung.
- Policy: Es existiert eine Policy-Datei, anhand der entschieden wird, welche Zugriffe erlaubt sind und welche nicht.

KAPITEL 2

Konzept

Im Folgenden wird das Grundkonzept von WSEP erklärt. Die Proof-of-Concept Implementierung (beschrieben in Kapitel 3) verwendet Java. Da alle ausgetauschten Nachrichten auf XML-Standards basieren, ist es grundsätzlich möglich, das Konzept in anderen Programmiersprachen umzusetzen. Unabhängig von der konkreten Implementierung muss es einen Web Service Provider, einen Protection Intermediary und einen Web Service Client geben. Abbildung 2.1 zeigt, wie die Komponenten interagieren.

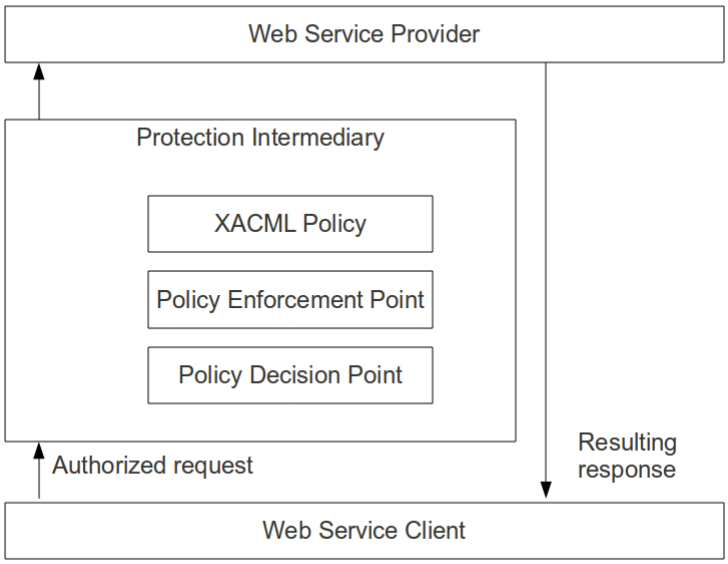

Abbildung 2.1: Interaktion

2.1 Web Service Provider

Beim Web Service Provider handelt es sich in Zusammenhang mit WSEP um einen Server, der SOAP-basierte Web Services bereit stellt.

> „SOAP Version 1.2 ist ein leichtgewichtiges Protokoll mit der Absicht, strukturierte Information in einer dezentralisierten verteilten Umgebung, auszutauschen. Es verwendet XML-Technologie, um ein erweiterbares Nachrichten-Framework zu definieren, welches Nachrichtenkonstrukte bereit stellt, die über eine Vielzahl darunterliegender Protokolle ausgetauscht werden können.“ [3, Seite 1]

WSEP stellt keine spezifischen Forderungen an den konkreten Web Service Provider. Es muss lediglich möglich sein, seine Operationen, wie in der SOAP-Spezifikation [3] beschrieben, aufrufen zu können.

Listing 2.1: SOAP-Beispiel

```
1  <env:Envelope xmlns:env="...">
2    <env:Header>
3     <m:some-data/>
4    </env:Header>
5    <env:Body>
6      <m:meet xmlns:m="...">
7        <m:person>Kathi</m:person>
8      </m:meet>
9    </env:Body>
10 </env:Envelope>
```

Die Beispiel-SOAP-Nachricht in Listing 2.1 enthält einen Header (Zeile 2-4), welcher Metainformationen transportiert. WSEP nutzt ihn, um Daten für die Zugriffskontrolle zu transportieren. Der Body (Zeile 5-9) enthält applikationsspezifische Informationen („Payload“) und wird von WSEP nicht verändert. Das Beispiel zeigt den Aufruf der Operation `meet` mit einem `person`-Parameter (Zeile 6-8).

2.2 Protection Intermediary

Nicht der Web Service Provider sondern der Protection Intermediary nimmt die SOAP-Requests der Web Service Clients entgegen. Ihm obliegt die Entscheidung, ob der Aufruf einer Operation erlaubt ist oder nicht. Dafür wertet er den SOAP-Header jedes Requests aus. Es muss festgelegt sein, welche Informationen an welcher Stelle innerhalb des Headers transportiert werden und was diese bedeuten. Anstatt das selbst festzulegen, verwendet WSEP die Security Assertion Markup Language (SAML).

Die SAML-Spezifikation [1] definiert Syntax und Verarbeitungssemantik von Aussagen über ein Subjekt. Wenn aus der SAML-Assertion im SOAP-Header hervor geht, dass der Web Service Client die im SOAP-Body aufgerufene Operation ausführen darf, so leitet der Protection

Intermediary den SOAP-Request an den Web Service Provider weiter. Falls nicht, antwortet er mit einem SOAP-Fault.

Listing 2.2 illustriert, wie eine von WSEP verwendete SAML-Assertion aussieht.

Listing 2.2: SAML-Assertion

```
1  <saml2:Assertion xmlns:saml2="..." ID="Assertion-1"
2    IssueInstant="2011-10-16T07:40:59.698Z" Version="2.0">
3    <saml2:Issuer Format="...">...</saml2:Issuer>
4      <ds:Signature xmlns:ds="...">...</ds:Signature>
5      <saml2:Subject>
6        <saml2:NameID Format="...">a6188...</saml2:NameID>
7      </saml2:Subject>
8      <saml2:AttributeStatement>
9        <saml2:Attribute FriendlyName="..."
10         Name="EnabledSoapOperation" NameFormat="...">
11           <saml2:AttributeValue>meet</saml2:AttributeValue>
12       </saml2:Attribute>
13     </saml2:AttributeStatement>
14 </saml2:Assertion>
```

Der Aussteller (Zeile 3) trifft eine Aussage über die Eigenschaften (Zeile 8-13) eines Subjekts (Zeile 5-7). Der Aussteller („Token Provider“) ist der Besitzer eines vertrauenswürdigen Schlüsselpaares. Das Subjekt wird durch den SHA-1-Fingerprint[1] seines öffentlichen Schlüssels identifiziert. Für WSEP ist nur eine einzige Eigenschaft von Subjekten interessant, nämlich welche SOAP-Operationen ihnen erlaubt sind (Zeile 9-12).

XACML Policy

Was noch fehlt, ist eine Möglichkeit, über die getroffen Aussagen zu einer Entscheidung für die Zugriffskontrolle (konkret: Erlauben oder Verbieten) zu kommen. Es wäre möglich gewesen, den Ablauf für WSEP willkürlich festzulegen. Allerdings wird, um Interoperabilität zu gewährleisten, der XACML-Standard [8] verwendet.

Einen wesentlichen Teil von XACML stellen sogenannte Policies dar. Listing 2.3 zeigt die von der Proof-of-Concept Implementierung verwendete Policy. Sie kann vom Betreiber des Protection Intermediary ergänzt werden, um dessen spezielle Anforderungen abzubilden. Das ist ein konkreter Vorteil, der sich aus der Verwendung von XACML ergibt.

Listing 2.3: XACML-Policy

```
1  <Policy xmlns="..." xmlns:xsi="..." xsi:schemaLocation="..."
2    PolicyId="..."
3    RuleCombiningAlgId="...first-applicable">
4    <Description>
5      Policy for PAP.
```

[1] http://tools.ietf.org/html/rfc3174

```
6    </Description>
7    <Target />
8    <Rule RuleId="Rule1" Effect="Permit">
9      <Description>
10     The requested SOAP-operation is in the list of enabled SOAP-
11     operations.
12     </Description>
13     <Condition>
14       <Apply FunctionId="...string-is-in">
15         <Apply FunctionId="...string-one-and-only">
16           <ResourceAttributeDesignator
17             AttributeId="RequestedSoapOperation"
18             DataType="...string" />
19         </Apply>
20         <SubjectAttributeDesignator
21           AttributeId="EnabledSoapOperation"
22           DataType="...string" />
23       </Apply>
24     </Condition>
25   </Rule>
26   <Rule RuleId="Rule2" Effect="Deny">
27     <Description>
28     Everything else is not allowed.
29     </Description>
30   </Rule>
31 </Policy>
```

Listing 2.3 besteht aus zwei Regeln. Die zweite (Zeile 26-30) existiert lediglich aus Sicherheitsgründen, um Zugriff zu verbieten, falls die erste Regel (Zeile 8-25) nicht anwendbar ist. Die erste Regel ist anwendbar, falls ihre Bedingung (Zeile 13-24) erfüllt ist und führt zum Erlauben des Zugriffs. Die Bedingung ist erfüllt, wenn die nachgefragte SOAP-Operation in jenen SOAP-Operationen enthalten ist, die laut SAML-Token erlaubt sind.

Policy Enforcement- und Policy Decision Point

Der Protection Intermediary fungiert als Policy Enforcement Point (PEP). Er sorgt dafür, dass umgesetzt wird, was die XACML-Policy besagt. Der Intermediary prüft:

- Ob es sich um gültiges XML handelt.
- Ob der Counter in der Nachricht höher als alle Counter in den bisherigen Nachrichten ist.
- Ob das SAML-Token gültig (z.B. nicht abgelaufen) ist.

- Ob das SAML-Token noch genau jenes ist, das der Web Service Client vom Token Provider erhalten hat. Zu diesem Zweck wurde es mit dem privaten Schlüssel des Token Providers signiert und der Intermediary kann diese Signatur prüfen.

- Ob der gesamte SOAP-Request unverändert ist und von jenem Web Service Client stammt, für den das SAML-Token ausgestellt wurde. Deshalb wird jeder Request mit dem privaten Schlüssel des Web Service Client signiert.

Schlägt eine der Prüfungen fehl, verbietet der Intermediary die SOAP-Operation. Ansonsten schickt er einen Decision Request an den Policy Decision Point (PDP), der ebenfalls Teil des Protection Intermediary ist, aber beispielsweise als eigener Prozess umgesetzt sein kann.

Listing 2.4: XACML Decision Request

```
1  <Request xmlns="...">
2    <Subject>
3      <Attribute AttributeId="...X509SubjectName"
4        DataType="...string">
5          <AttributeValue>a6188...</AttributeValue>
6      </Attribute>
7      <Attribute AttributeId="EnabledSoapOperation"
8        DataType="...string">
9          <AttributeValue>piecewiseinterpolation</AttributeValue>
10         <AttributeValue>linearinterpolation</AttributeValue>
11     </Attribute>
12   </Subject>
13   <Resource>
14     <Attribute AttributeId="RequestedSoapOperation"
15       DataType="...string">
16         <AttributeValue>linearinterpolation</AttributeValue>
17     </Attribute>
18   </Resource>
19   <Action>
20     <Attribute AttributeId="SoapAction" DataType="...string">
21       <AttributeValue>Execute</AttributeValue>
22     </Attribute>
23   </Action>
24   <Environment/>
25 </Request>
```

Listing 2.4 zeigt einen solchen Decision Request. Dabei fragt ein Subjekt (Zeile 2-12) an, ob es eine bestimmte Aktion (Zeile 19-23) auf eine bestimmte Ressource (Zeile 13-18) durchführen darf.

WSEP übermittelt den SHA-1-Fingerprint des öffentlichen Schlüssels des Subjekts (Zeile 5) und die dem Subjekt erlaubten Operationen (Zeile 9-10). Bei der Ressource handelt es sich um

die SOAP-Operation (Zeile 16), die das Subjekt aufgerufen hat. WSEP verwendet lediglich die Aktion `Execute` (Zeile 21).

Der PDP benutzt die Policy und den Decision Request, um eine Entscheidung zu treffen. Er übermittelt diese als Decision Response an den PEP, wie in Listing 2.5 gezeigt.

Listing 2.5: XACML Decision Response

```
1 <Response xmlns="..." xmlns:ns0="..." ns0:schemaLocation="...">
2   <Result>
3     <Decision>Permit</Decision>
4     <Status>
5       <StatusCode Value="...ok"/>
6     </Status>
7   </Result>
8 </Response>
```

2.3 Web Service Client

Der Web Service Client sendet SOAP-Nachrichten. Er ergänzt den SOAP-Header dieser Nachrichten, wie in Listing 2.6 gezeigt. Ein Security-Element wird benutzt, um das SAML-Token des Client zu transportieren (Zeile 3-7). Zusätzlich werden die SOAP-Requests des Web Service Client mit einer Sequenznummer versehen (Zeile 8), um Replay-Attacken zu verhindern. Damit der Protection Intermediary die Integrität der Client-Nachrichten überprüfen kann, signiert der Web Service Client den gesamten SOAP-Envelope (Zeile 9), sobald er ihn fertig aufgebaut hat.

Listing 2.6: XACML Decision Response

```
1  <S:Envelope xmlns:S="..." xmlns:T="..." T:id="...">
2    <S:Header>
3      <wsse:Security xmlns:wsse="...">
4        <saml2:Assertion>
5            ...
6        </saml2:Assertion>
7      </wsse:Security>
8      <U:Sequence xmlns:U="...">3</U:Sequence>
9      <Signature xmlns="...">...</Signature>
10   </S:Header>
11   <S:Body>
12     ...
13   </S:Body>
14 </S:Envelope>
```

2.4 Zusammenspiel

Wie bereits ausgeführt, bedient sich WSEP einer Kombination diverser etablierter Standards, die teilweise ineinander verschachtelt werden. Abbildung 2.2 fasst noch einmal zusammen, wie die verwendeten Standards zusammen spielen.

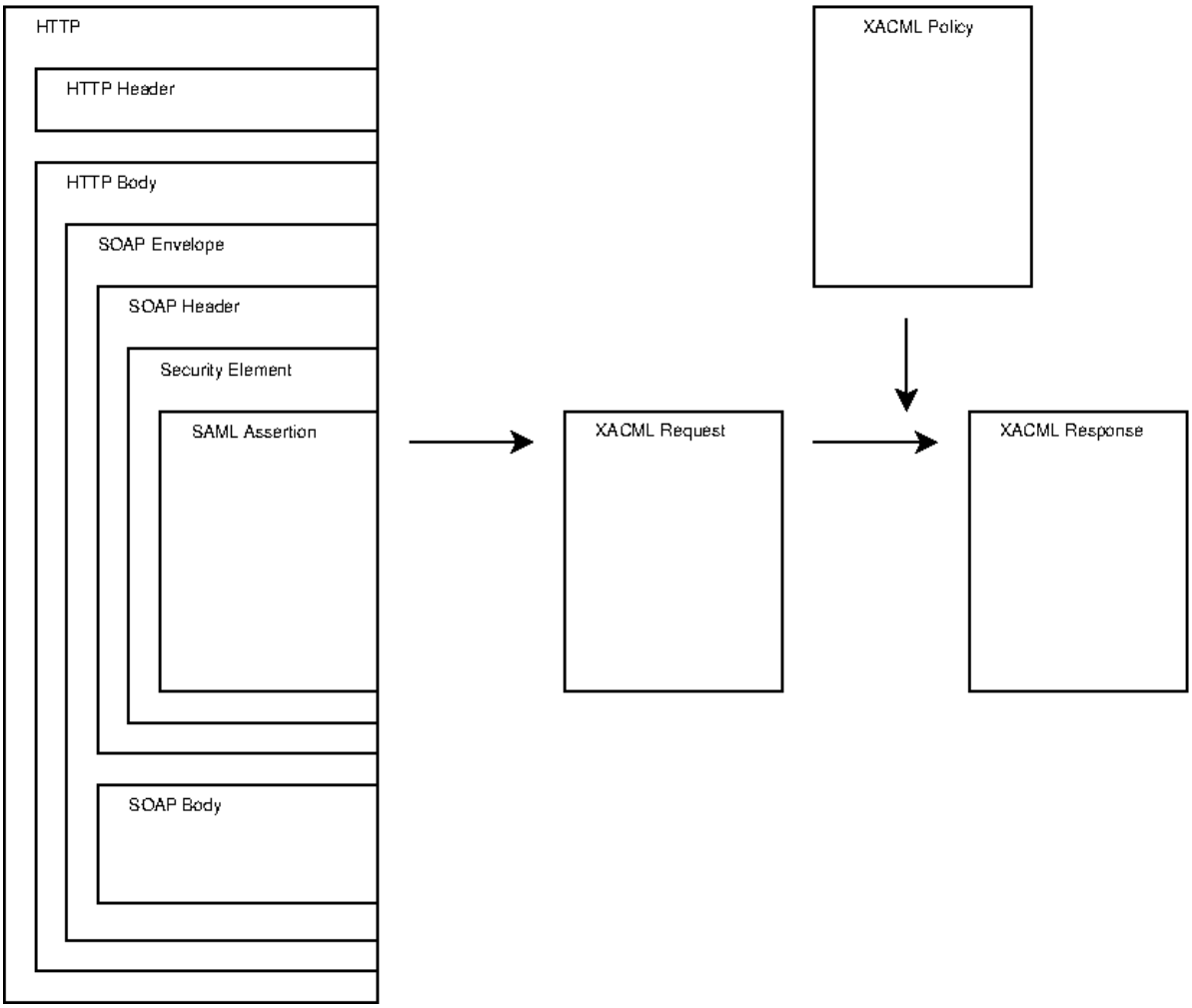

Abbildung 2.2: Zusammenspiel der Standards

Hyper Text Transfer Protocol (HTTP) dient als Transportmedium. Im HTTP-Body wird eine SOAP-Nachricht transportiert. Von dieser Hierachieebene an findet die gesamte Kommunikation in Form von Extensible Markup Language (XML) statt. SOAP-Nachrichten weisen ebenfalls einen Header und einen Body auf. Der Body dient im Kontext von Web Services dazu, bestimmte Web Service Operation aufzurufen. Er bleibt unverändert, damit die Zugriffskontrolle für etwaige Clientapplikationen transparent bleibt. Alle dafür relevanten Informationen werden im SOAP-Header transportiert.

Der Header enthält ein Security-Element, bestehend aus einen Counter, einer digitalen Signatur und einer Security Assertion Markup Language (SAML) Assertion. Zuerst muss die Assertion extrahiert werden. Dann kann sie in einen eXtensible Access Control Markup Language (XACML) Request umgewandelt werden. Nun kann anhand einer XACML-Policy ein XACML-Response generiert werden. Diese trifft eine Aussage darüber, ob die nachgefragte Web Service Operation erlaubt ist.

KAPITEL 3

Umsetzung

Im Zuge dieser Arbeit ist eine Proof-of-Concept Implementierung in Java entstanden[1], deren Sourcecode zur freien Verfügung steht. Abbildung 3.1 liefert eine Übersicht der Systemkomponenten.

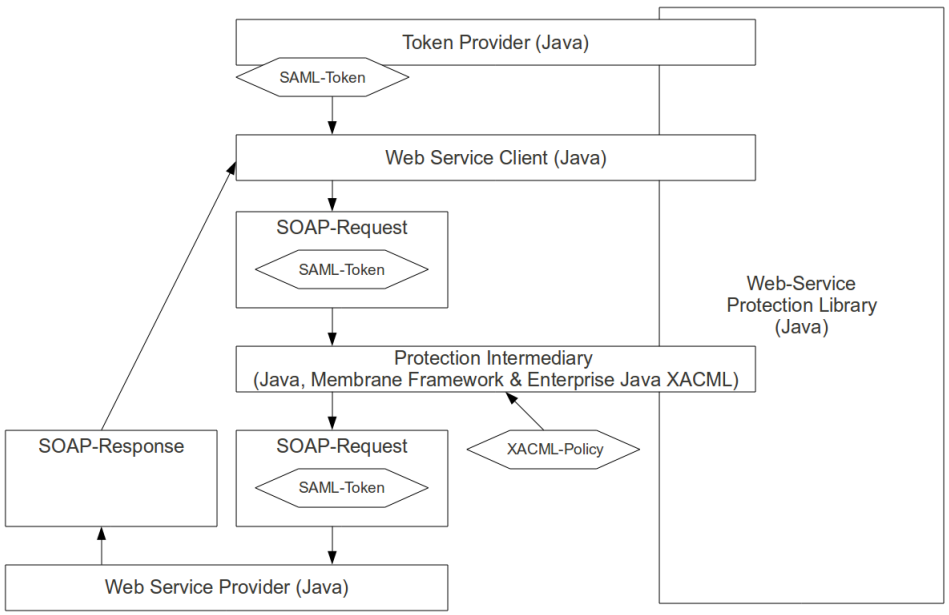

Abbildung 3.1: Systemkomponenten

[1] http://code.google.com/p/wsep2011/

Ein Web Service Client, der auf WSEP aufsetzt, benötigt ein SAML-Token, das er vom Token Provider bekommt. Er hängt es an alle SOAP-Requests an, die er an den Protection Intermediary richtet. Dieser prüft anhand seiner XACML-Policy, ob er den SOAP-Request erlauben soll. Falls der Zugriff nicht erlaubt werden soll, liefert er eine SOAP-Fault an den Web Service Client zurück. Andernfalls leitet er den SOAP-Request an den eigentlichen Web Service Provider weiter. Dieser führt die gewünschte Operation aus und antwortet dem Web Service Client direkt.

Im Folgenden wird die Implementierung im Detail beschrieben. Sie lehnt sich an das englischsprachige Tutorial zur Benutzung von WSEP[2] vom gleichen Autor an.

3.1 Web Service Provider

Man muss ein Web Service entwickeln und dieses auf einem Server ausrollen. Ab JDK 6 gestaltet sich das Verfassen eines Web Services denkbar einfach - man muss lediglich passende Annotations hinzufügen. Dieser Schritt ist nicht WSEP-spezifisch und kann für anders gesicherte Web Services identisch aussehen. Listing A.1 im Anhang liefert ein Beispiel für ein solches Web Service.

Das Web Service bekommt eine Liste aus Fließkommazahlen und interpoliert alle Listenelemente, die `null` sind. Interpoliert man zum Beispiel zwischen $\{1.0, null, null, 13.3, null, 50.9\}$ bekommt man:

- $\{1.0, 1.0, 1.0, 13.3, 13.3, 50.9\}$ stückweise und
- $\{1.0, 5.1, 9.200001, 13.3, 32.100002, 50.9\}$ linear.

Ein Server wird benötigt, um das Web Service auszuliefern. Für diesen Zweck gibt es den in Listing 3.1 gezeigten Service Provider.

Listing 3.1: Beispiel Service Provider

```
1  package at.ac.tuwien.simpleprovider;
2
3  import javax.xml.ws.Endpoint;
4
5  public class Main {
6
7          public final static String ENDPOINT_DEFAULT =
8                  "http://localhost:8088/interpolator";
9
10         public static void main(String args[]) {
11
12                 String endpoint = ENDPOINT_DEFAULT;
13
14                 if(args.length > 0) {
```

[2] http://code.google.com/p/wsep2011/wiki/TutorialDevelopProtectedApplication

```
15                          endpoint = args[0];
16                  }
17
18                  Interpolation server = new Interpolation();
19                  Endpoint.publish(endpoint, server);
20
21          }
22  }
```

Zu beachten ist, dass sich der Web Service Client später nicht auf die hier angegebene Adresse, sondern auf die des Intermediary verbinden wird. Durch direktes Verbinden würde man die komplette Zugriffskontrolle umgehen. Derzeit unternimmt WSEP keine Maßnahmen, um das zu verhindern.

3.2 Protection Intermediary

Beim Protection Intermediary handelt es sich um einen Serverprozess, der SOAP-Requests entgegennimmt und analysiert. Die Implementierung basiert auf dem Membrane Framework[3], einer Java-Library, die Proxy-Funktionen anbietet und SOAP-Nachrichten analysieren kann.

Weiters stützt sich die Implementierung auf die Enterprise Java XACML Implementation[4]. Das ist eine Java-Library, welche die XACML-Spezifikation implementiert.

3.3 Web Service Client

Die Verwendung von WSEP ergibt für die Entwicklung des Web Service Client kaum Abweichungen vom Standardablauf. Zuerst generiert man die Client-Stubs, wie in Listing 3.2 dargestellt. Die so erhaltenen Klassen bindet man in den Web Service Client ein.

Listing 3.2: Client-Stubs generieren

```
1  wsimport -keep http://localhost:8088/interpolator?wsdl
```

Listing A.2 im Anhang zeigt einen beispielhaften Web Service Client. Die Aktivierung von WSEP geschieht in Zeile 37. Dieser Methodenaufruf verändert den Web Service Stack derart, dass bei jedem SOAP-Request das SOAP-Security Element verändert wird:

- Unverändertes Einfügen des erhaltenen SAML-Token.
- Signieren der gesamten SOAP-Nachricht mit dem privaten Schlüssel des Clients.

Das geschieht für den Web Service Client in weiterer Folge vollkommen transparent.

[3] http://www.membrane-soa.org/soap-monitor/
[4] http://code.google.com/p/enterprise-java-xacml/

Man kann eigene SAML-Token generieren, indem man zuerst die Einstellungen anpasst und dann einen Ant[5]-Task aufruft. Zuletzt muss man das generierte Token noch in das Web Service Client-Verzeichnis kopieren, wie Listing 3.3 zeigt.

Listing 3.3: Ein SAML-Token generieren

```
1 vi WSProtectionLib/build.xml
2 ant tp
3 cp WSProtectionLib/bin/saml.xml SimpleClient
```

Vor der Erstellung des Token kann man folgende Einstellungen treffen:

- Dateiname des Token.
- Name des Ausstellers.
- Keystore mit dem privatem Schlüssel des Ausstellers.
- Passwort, um diesen Keystore zu öffnen.
- Name dieses privaten Schlüssels.
- Passwort dieses privaten Schlüssels.
- Keystore mit dem öffentlichen Schlüssel der Applikation, für die das Token ausgestellt wird.
- Passwort, um diesen Keystore zu öffnen.
- Name des öffentlichen Schlüssels.
- Erlaubte Web Service Operationen.

3.4 Zusammenspiel

Die Proof-of-Concept Implementierung wird über Ant gesteuert. Listing 3.4 zeigt, wie (in dieser Reihenfolge) der Web Service Provider, der Protection Intermediary und der Web Service Client gestartet werden.

Listing 3.4: Ant Befehle zum Starten der Komponenten

```
1 ant vi
2 ant pi
3 ant si
```

Mit den Standardeinstellungen (Listing A.3 im Anhang), sendet der Web Service Client seine SOAP-Nachrichten an `http://localhost:7077/interpolator`. Dort wird die Nachricht vom Protection Intermediary empfangen und - falls sie korrekt ist - an den Web Service Provider unter `http://localhost:8088/interpolator` weitergeleitet.

[5] http://ant.apache.org/

KAPITEL 4

Security Analyse

4.1 Angriffspotential

WSEP setzt sich aus zahlreichen verschiedenen Technologien zusammen. Dementsprechend gibt es vielfältige Möglichkeiten, das System anzugreifen. Es existiert allerdings keine Möglichkeit, eine Zugriffskontrolle auf Web Service-Ebene zu implementieren, dabei auf etablierte Standards zu setzen, und die Angriffsmöglichkeiten auf sehr wenige - am besten eine - Technologien zu begrenzen. Abbildung 4.1 liefert einen Überblick der angreifbaren Technologien.

Weil WSEP eine Zugriffskontrolle für beliebige Metro-Web Services ist, kann das konkrete, abgesicherte Web Service nicht evaluiert werden. Metro, Glassfish und HTTP werden ebenfalls nicht berücksichtigt, weil deren Sicherheit ein sehr breites Thema ist und konzeptionelle Schwachstellen darin bei weitem mehr als nur WSEP betreffen.

4.2 Web of Trust / Public Key Infrastructure

Eine Public Key Infrastructure (PKI) und ein Web of Trust (WOT) können zusammen mit WSEP genutzt werden. Wenn man ein Token erstellt, signiert man den Hash des öffentlichen Schlüssels einer Anwendung mit dem privaten Schlüssel der Zugriffskontrolle. Damit erteilt man dem Besitzer des privaten Schlüssels der Anwendung die Erlaubnis, die gewählten Web Service Operationen aufzurufen.

Folgende Szenarien sind denkbar:

- Der private Schlüssel der Zugriffskontrolle wird kompromittiert: Man kann den verwendeten privaten Schlüssel im Keystore der Zugriffskontrolle durch einen neuen ersetzen. Dadurch verlieren zwar alle ausgestellten Token ihre Gültigkeit, aber das Problem ist behoben.

- Der private Schlüssel einer Anwendung wird kompromittiert: Man kann - wie vorher - den privaten Schlüssel im Keystore der Zugriffskontrolle durch einen neuen ersetzen.

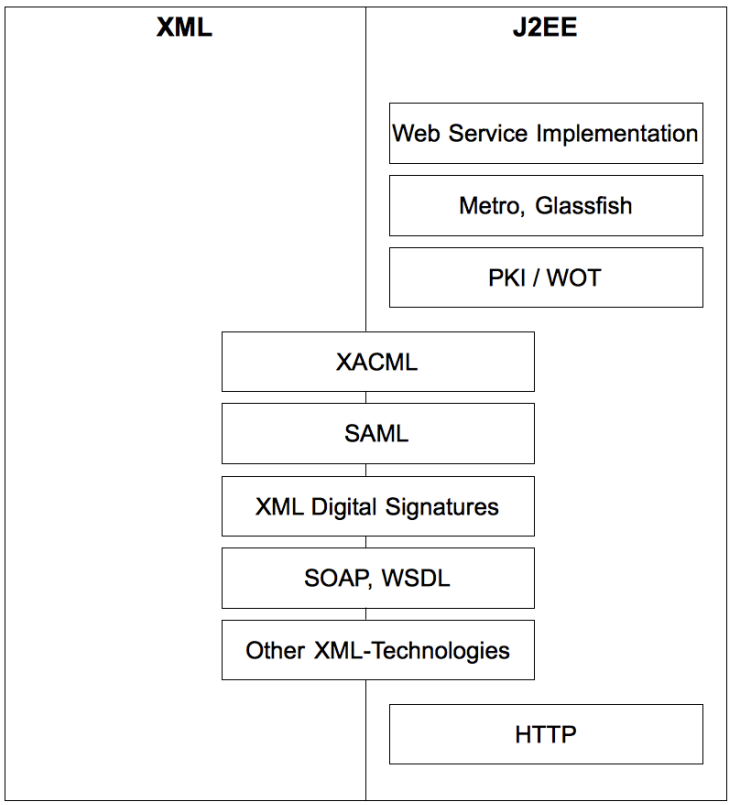

Abbildung 4.1: Angriffspotential

Momentan ist dafür keine saubere Lösung implementiert. Prinzipiell gibt es dazu zwei Möglichkeiten:

  - Man begrenzt den Zeitraum, in dem SAML-Token gültig sind. In Listing 4.1 wird dieser Ansatz illustriert.
  - Der Intermediary wendet sich jedes Mal, wenn er einen Request verarbeitet, an eine Validation Authority, die bestätigt, dass der Schlüssel nicht zurückgerufen wurde und deshalb nach wie vor gültig ist.

- Eine Anwendung teilt ihr legitim erworbenes Token (und ihren Schlüssel) mit anderen Anwendungen. Das kann nie gänzlich verhindert werden. Um dieses Verhalten zumindest teilweise zu unterbinden, könnte man erlaubte Ursprungsadressen der SOAP-Requests in das SAML-Zertifikat aufnehmen.

Listing 4.1: Gültigkeit eines SAML-Token begrenzen

```
1  <saml:Conditions
2        NotBefore="2011-12-31T17:17:17Z"
3        NotOnOrAfter="2014-08-09T09:18:15Z" />
```

4.3 eXtensible Access Control Markup Language

Momentan laufen die einzelnen, für XACML notwendigen Komponenten innerhalb des selben Prozesses. Falls später die Aufteilung in eigenständige Prozesse auf möglicherweise unterschiedlichen Servern erfolgen soll, so ist das relativ einfach möglich, weil es definierte Interfaces gibt, zwischen denen nur XML-Dokumente transportiert werden.

Weil momentan keine XACML Decision Requests oder XACML Authorization Decisions den Intermediary verlassen, sind die nun folgenden Sicherheitsbedenken rein hypothetisch.

Die XACML-Spezifikation [8] beinhaltet ein Kapitel mit einer nicht-normativen Abhandlung zur Sicherheit von Systemen, die XACML verwenden. Im Folgenden werden die darin wiedergegebenen Anforderungen betrachtet.

> „XACML spezifiziert keinen inhärenten Mechanismus, um die Vertraulichkeit der zwischen den Teilnehmern ausgetauschten Nachrichten zu gewährleisten. Deshalb könnte ein Gegenspieler die Nachrichten betrachten, während sie übertragen werden.“ [8, Seite 90]

> „Bei einer Message Replay Attacke nimmt der Gegenspieler legitime Nachrichten zwischen den XACML-Teilnehmern auf und spielt sie anschließend ab. Diese Attacke kann zu Denial of Service, der Benutzung von veralteter Information oder zu einer Identitätsübernahme führen.“ [8, Seite 90]

Man kann die Vertraulichkeit der ausgetauschten Nachrichten gewährleisten, indem man alle Nachrichten über HTTPS transportiert.

> „Bei einer Message Insertion Attacke fügt der Gegenspieler Nachrichten in die Folge der Nachrichten zwischen den Teilnehmern ein.“ [8, Seite 90]

Ein Teil der Lösung kann hier wieder HTTPS sein, sofern man für die gegenseitige Authentifizierung der Teilnehmer sorgt. Außerdem muss sichergestellt werden, dass das Senden der Nachricht für einen gegebenen Teilnehmer auch erlaubt ist.

> „Bei einer Message Deletion Attacke löscht der Gegenspieler Nachrichten aus der Folge der Nachrichten zwischen den Teilnehmern. Message Deletion kann zu Denial of Service führen. Ein richtig gestaltetes XACML System sollte keine falschen Autorisierungsentscheidungen aufgrund von Message Deletion Attacken treffen.“ [8, Seite 90]

Eine mögliche Lösung ist es, den Zähler von vorhin zu verwenden und abzubrechen, sobald Zahlen übersprungen werden.

> „Wenn ein Gegenspieler eine Nachricht abfangen und ihren Inhalt verändern kann, könnte es möglich sein, eine Autorisierungsentscheidung zu verändern. Eine Schutzmaßnahme für die Nachrichtenintegrität verhindert eine erfolgreiche Message Modification Attacke.“ [8, Seite 91]

Auch diese Attacke wird durch die Verwendung von HTTPS verhindert.
Die folgenden Sicherheitsbedenken sind für WSEP nicht rein hypothetisch.

> „Das Ergebnis NotApplicable bedeutet, dass der PDP keine Policy finden konnte, deren Target mit der Information im Decision Request übereinstimmt. Für den Normalfall wird stark empfohlen, eine Policy mit einem Deny Effekt zu verwenden, damit, wenn der PDP NotApplicable zurück geben würde, stattdessen Deny zurückgegeben wird.“ [8, Seite 91]

WSEPs Intermediary verwendet standardmäßig eine Policy, die das berücksichtigt. Den relevante Ausschnitt sieht man in Listing 4.2. Wenn man die Policy anpasst, muss man die genannte Problematik allerdings im Hinterkopf behalten.

Listing 4.2: Deny-Effekt

```
1  <Policy>
2     <!-- ... -->
3      <Rule
4            RuleId="Rule2"
5            Effect="Deny">
6          <Description>
7             Everything else is not allowed.
8          </Description>
9      </Rule>
10 </Policy>
```

Ein zweiter Punkt ist zu beachten, wenn man eine eigene Policy erstellt.

> „Eine Negative Rule basiert darauf, dass ein Prädikat nicht wahr ist. Unvorsichtig genutzt, können Negative Rules zu einer Falschbehandlung der Policy führen.“ [8, Seite 91]

Negative Rules werden oft verwendet, um Zugriff für eine Gruppe zu erlauben und Zugriff für einzelne, wenige Mitglieder der Gruppe zu verbieten. Für dieses Anwendungsszenario empfiehlt es sich, stattdessen eine neue Gruppe zu definieren.

4.4 Security Assertion Markup Language

Der SAML-Standard enthält ein eigenes, nicht normatives Dokument zur Systemsicherheit [5]. Im Folgenden werden ausgewählte, daraus entnommene, Punkte besprochen.

Die Privatsphäre der Teilnehmer muss geschützt werden. Dazu gehören die Geheimhaltung der Nachrichten und die Anonymität der Teilnehmer und ihres Verhaltens. WSEP trifft in seinen SAML-Statements momentan keine Vorkehrungen, um die Privatsphäre der Teilnehmer zu schützen. Wiederum könnte man HTTPS nutzen, um Geheimhaltung zu erreichen. Das ist allerdings noch keine hinreichende Lösung:

> „Ein Benutzer, der jeden Dienstag um 21:00 auf eine Datenbank zugreift, welche die Länge der Finger mit der Lebenserwartung gegenüber stellt, hört auf, anonym zu sein. Abhängig vom sonstigen Verhalten des Benutzers könnte er oder sie verfolgbar werden, indem es möglich wird, weitere identifizierende Information zu sammeln.“ [5, Seite 8]

Die Sicherheit wird teilweise durch dieselben Attacken bedroht, die schon auf XACML ausgeführt wurden. Allerdings transportiert WSEP SAML tatsächlich über ein (potentiell unsicheres) Netzwerk.

- Replay Attacks: Die SOAP Security-Elemente enthalten in der aktuellen Implementierung bereits einen Counter. Wurde eine Nachricht einmal vom Intermediary empfangen, kann sie von einem Widersacher nicht erneut verwendet werden.

- Message Insertion: „Eine gefälschte Anfrage oder Antwort wird in den Nachrichtenstrom eingefügt. Eine falsche Antwort, wie etwa ein gefälschtes Ja als Antwort auf eine Authorization Decision Query, oder die Rückgabe von falscher Attributinformation als Antwort auf eine Attibute Query, können in einer unangemessenen Reaktion des Empfängers resultieren.“ [5, Seite 18] Message Insertion ist in WSEP nicht möglich, sofern keine privaten Schlüssel kompromittiert sind.

- Message Deletion: „Im Allgemeinen kann eine Korrelation zwischen Anfrage- und Antwort-Nachrichten eine solche Attacke verhindern.“ [5, Seite 18] WSEP ist davon nicht betroffen, weil keine Anfragen mit darauf folgenden Antworten geschickt werden.

- Message Modification: WSEP signiert komplette SOAP-Requests und ist deshalb von dieser Attacke nicht direkt betroffen. Die SOAP-Responses werden allerdings nicht signiert und sind daher betroffen.

4.5 XML Digital Signatures

Hinsichtlich der von WSEP verwendeten Algorithmen und Schlüssellängen ergeben sich aktuell keine Bedenken. Die vielfältigen Verwendungsmöglichkeiten von XML Signaturen machen allerdings vielfältige Fehlanwendungen möglich.

Listing 4.3: Fehlanwendung einer XML-Signatur - Teil 1

```
1  <order>
2    <item>
3        <name>Box of pencils</name>
4        <price Id="p1">$1.50</price>
5        <quantity>1</quantity>
6    </item>
7    <item>
8      <name>Laptop</name>
9      <price Id="p2">$2500.00</price>
10     <quantity>100</quantity>
11   </item>
12 </order>
13 <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
14   <SignedInfo> . . .
15     <Reference URI="#xpointer(id('p1'))">. . .</Reference>
16     <Reference URI="#xpointer(id('p2'))">. . .</Reference>
17   </SignedInfo>
18   <SignatureValue>. . .</SignatureValue>
19   <KeyInfo>. . .</KeyInfo>
20 </Signature>
```

Listing 4.3 [4, Seite 119] zeigt ein Beispiel für die falsche Anwendung von XML Signaturen. In Zeile 15 wird der Preis aus Zeile 4 signiert und in Zeile 16 jener aus Zeile 9. Sehr oberflächlich betrachtet, schaut das sicher aus - schließlich wird der Preis signiert. Man kann das XML-Fragment aber manipulieren, ohne eine ungültige Signatur zu verursachen, wie Listing 4.4 [4, Seite 119] zeigt. WSEP ist von derartigen Attacken nicht betroffen.

Listing 4.4: Fehlanwendung einer XML-Signatur - Teil 2

```
1  <order>
2    <item>
3      <name>Box of pencils</name>
4      <price Id="p2">$2500.00</price>
5      <quantity>1</quantity>
6    </item>
7    <item>
8      <name>Laptop</name>
9      <price Id="p1">$1.50</price>
10     <quantity>100</quantity>
11   </item>
12 </order>
13 <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
14   <SignedInfo> . . .
15     <Reference URI="#xpointer(id('p1'))">. . .</Reference>
```

```
16      <Reference URI="#xpointer(id('p2'))">. . .</Reference>
17    </SignedInfo>
18    <SignatureValue>. . .</SignatureValue>
19    <KeyInfo>. . .</KeyInfo>
20  </Signature>
```

4.6 Web Service Description Language

> „WSDL beschreibt logische und konkrete Details von Web Services. Ein WSDL-Dokument enthält Informationen zur Parameter- und Methodenbenutzung. Durch Auslesen des WSDL-Dokuments kann ein Angreifer an sensible Informationen, wie zum Beispiel Typen, Nachrichten, Operationen, Porttypen und Bindings kommen und andere Methodennamen erraten.“ [7, Seite 11]

Im Falle von WSEP kommt ein potentieller Angreifer spätestens dann auf die Idee, den Protection Intermediary zu umgehen, wenn er das zugehörige WSDL-Dokument analysiert. Als Abhilfe ist es erforderlich, den Web Service Provider nur vom Protection Intermediary aus erreichbar zu machen.

4.7 Simple Object Access Protocol

In einem XML-Dokument kann man grundsätzlich Document Type Definitions (DTDs)[1] verwenden. DTDs ermöglichen es unter anderem, Entities zu verwenden. Ein Beispiel dafür ist `<`, das in < umgewandelt wird. Benutzerdefinierte Entities sind ebenfalls vorgesehen. Die Verschachtelung benutzerdefinierter Entities ist möglich. Dies ermöglicht eine Denial of Service (DOS) Attacke, die „Many Laughs Attack“ genannt wird.

Listing 4.5: Many Laughs Attack

```
1  <!DOCTYPE envelope [
2    <!ENTITY a "hahahahaha" >
3    <!ENTITY b "&a;&a;&a;&a;&a;&a;&a;&a;" >
4    <!ENTITY c "&b;&b;&b;&b;&b;&b;&b;&b;" >
5    <!ENTITY d "&c;&c;&c;&c;&c;&c;&c;&c;" >
6    <!ENTITY e "&d;&d;&d;&d;&d;&d;&d;&d;" >
7    <!ENTITY f "&e;&e;&e;&e;&e;&e;&e;&e;" >
8    <!ENTITY g "&f;&f;&f;&f;&f;&f;&f;&f;" >
9    <!ENTITY h "&g;&g;&g;&g;&g;&g;&g;&g;" >
10   <!ENTITY i "&h;&h;&h;&h;&h;&h;&h;&h;" >
11   <!ENTITY j "&i;&i;&i;&i;&i;&i;&i;&i;" >
12   <!ENTITY k "&j;&j;&j;&j;&j;&j;&j;&j;" >
13   <!ENTITY l "&k;&k;&k;&k;&k;&k;&k;&k;" >
```

[1] http://www.w3.org/TR/REC-xml

```
14   <!ENTITY m "&l;&l;&l;&l;&l;&l;&l;" >
15 ]>
16 <envelope> &m; </envelope>
```

Ein Angreifer kann so mit sehr geringem Bandbreiteneinsatz eine sehr große Menge an Daten erzeugen. Listing 4.5 skizziert die Vorgangsweise. Dabei werden zusätzlich $10 * 8^{12}$ Zeichen erzeugt.

Der aktuelle SOAP-Standard [3] schließt Entity Expansion aus. Die von WSEP verwendete Implementierung setzt das korrekt um und ist deshalb von dieser Attacke nicht betroffen.

4.8 Zusammenfassung

Tabelle 4.1 fasst die Ergebnisse dieses Kapitels zusammen. Wird ein privater Schlüssel kompromittiert, stellt das immer ein Sicherheitsrisiko dar. WSEP trifft keine Maßnahmen, um die Vertraulichkeit der SAML-Statements zu gewährleisten. Attacken auf SAML sind möglich, wenn man sich auf SOAP-Responses konzentriert. Diese wurden im Rahmen dieser Arbeit nicht näher betrachtet. Das größte Sicherheitsrisiko stellt allerdings das Umgehen des Protection Intermediary dar, sofern der Service Provider vom Service Client direkt erreichbar ist. WSDL-Dokumentenanalyse kann einem potentiellen Angreifer diese Schwachstelle aufzeigen.

Tabelle 4.1: Evaluierungsergebnisse

Technologie	**Angriffsvektor**	**Gefährdet WSEP?**
WOT / PKI	Schlüsselkompromittierung	Ja
XACML	Vertraulichkeit kompromittierbar	Nein
XACML	Message Replay	Nein
XACML	Message Insertion	Nein
XACML	Message Deletion	Nein
XACML	Message Modification	Nein
XACML	NotApplicable	Nein
XACML	Negative Rules	Nein
SAML	Vertraulichkeit kompromittierbar	Ja
SAML	Message Replay	Ja, SOAP-Responses
SAML	Message Insertion	Ja, SOAP-Responses
SAML	Message Deletion	Ja, SOAP-Responses
SAML	Message Modification	Ja, SOAP-Responses
XML Digital Signatures	Falschreferenzierung	Nein
WSEP	Protection Intermediary umgehen	Ja, sofern Service Provider erreichbar
WSDL	Dokumentanalyse	Ja, sofern Service Provider erreichbar
SOAP	Many Laughs Attacke	Nein

KAPITEL 5

Related Work

5.1 Damiani

In [2] wird erstmals ein feinkörniges Konzept der Zugriffskontrolle präsentiert, welches die Struktur von SOAP-Nachrichten nutzt [6, Seite 4]. Das Grundkonzept ist dem von WSEP ähnlich und wird hier kurz beschrieben.

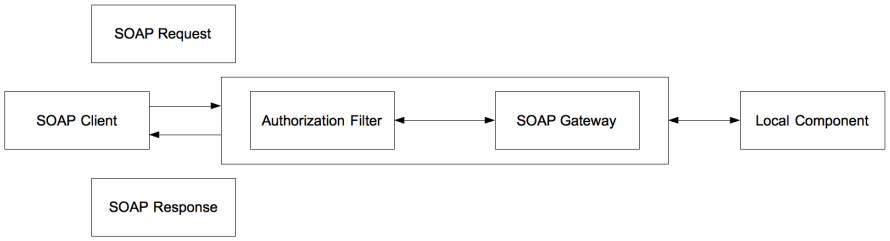

Abbildung 5.1: Funktionsweise Damiani

> „Wie in Abbildung 5.1 beschrieben, basiert die Zugriffskontrolle auf einem Authorization Filter, der jeden an den SOAP Gateway gehenden Aufruf abfängt und ihn mit Autorisierungen vergleicht, welche die Service-Nutzbarkeit festlegen. Basierend auf diesen Autorisierungen kann die Anfrage: 1) abgelehnt werden; 2) unverändert erlaubt werden; oder 3) gefiltert und in einer veränderten Form ausgeführt werden. Die Filterung kann das Eliminieren einiger Parameter zur Folge haben, deren Angabe für den Aufrufer nicht erlaubt ist.“ [2, Seite 4]

Im Gegensatz dazu kann WSEP Anfragen nur komplett ablehnen oder erlauben.

Die beiden Zugriffskontrollen arbeiten jeweils auf Basis von SOAP-Requests. Die SOAP-Responses werden nicht berücksichtigt, könnten aber in ähnlicher Weise behandelt werden. Die

Behandlung von SOAP-Responses durch den SOAP Gateway bzw. Intermediary macht weniger Sinn als die Behandlung von SOAP-Requests, weil der Service Provider nach dem SOAP-Request die nachgefragte Operation bereits durchgeführt hat und dadurch nur die Rückmeldung darüber unterdrückt würde.

> „Im Zusammenhang mit SOAP kann es sein, dass vollständige Schnittstellen nicht ohne weiteres zur Verfügung stehen. Allerdings senden Clients Requests, deren XML-Struktur die durch das Remote-E-Service (Anmerkung: Web Service) angebotene Schnittstelle modellieren. Deshalb, so meinen wir, macht es mehr Sinn, die Requests selbst als Objekte des Autorisierungssystems zu sehen.“ [2, Seite 4]

Beide Zugriffskontrollen sehen die konkreten SOAP-Requests als Autorisierungsobjekte. Ein Beispiel soll das illustrieren: Ein Benutzer A will die Ware B in der Menge C bestellen. Es existiert ein Web Service mit den beiden Operationen

- `bestelleEine(Ware)` und
- `bestelleN(Ware)`.

Dazu schickt er C SOAP-Requests `bestelleEine(B)`. Der SOAP-Gateway / Intermediary erlaubt ihm diese Operation(en), aber würde ihm `bestelleN(Ware)` verbieten, weil das seine Beispiel-Policy so besagt. In der Beispiel-Policy wird nur Bezug auf konkrete SOAP-Requests genommen. Aussagen, wie zum Beispiel “Benutzer A darf Ware B bestellen“, sind nicht vorgesehen und können nicht direkt formuliert werden.

[2] nutzt für die Autorisierung von Subjekten eine Menge vordefinierter Attribute:

- Identität des Benutzers, für den der Web Service Client ausgeführt wird.
- Ort, von dem der SOAP-Request kommt (numerisch oder symbolisch).
- Mitgliedschaft des Benutzers in Benutzergruppen.
- Die wirksame Rolle des Benutzers.

Überraschenderweise entsteht dadurch, dass sowohl Benutzerrollen als auch Benutzergruppen verwendet werden, keine Redundanz:

> „Es sei angemerkt, dass die Unterstützung von sowohl Gruppen als auch Rollen nicht redundant ist, weil sie andere Verhalten an den Tag legen. Im Unterschied zu Gruppen, verhalten sich Rollen dynamisch und können von Benutzern willkürlich aktiviert oder deaktiviert werden.“ [2, Seite 4]

Daraus ergeben sich die Informationen, welche von [2] für das Subjekt in den Header der SOAP-Requests eingebracht werden:

- User-ID

- IP-Address
- Sym-Address
- Role-ID

WSEP basiert auf der Annahme, dass man mit fix definierten Attributen nicht alle Anwendungsfälle abdecken kann. Deshalb ist das Autorisierungssubjekt ein beliebiges Objekt, welches durch den Hash-Wert seines öffentlichen Schlüssels identifiziert wird. Es muss mit seinem privaten Schlüssel für jeden SOAP-Request einen Signaturwert berechnen, um seine Identität zu beweisen. Die Header der SOAP-Requests werden daher für das Subjekt, wie in Listing 5.1 gezeigt, ergänzt.

Listing 5.1: WSEP Subjekt

```
1 <saml2:Subject>
2     <saml2:NameID Format="...:nameid-format:unspecified">
3         ...
4     </saml2:NameID>
5 </saml2:Subject>
```

Beide Zugriffskontrollen benötigen ein Format, um ihre Autorisierungsrichtlinien / Policies auszudrücken. [2] definiert dazu über eine Document Type Definition (DTD) ein eigenes Format. Listing 5.2 [2, Seite 7] zeigt ein Beispiel für eine solche Autorisierungsrichtlinie.

Listing 5.2: Autorisierungsrichtlinie

```
1 <subject>
2   <groupid>Registered_users</groupid>
3 </subject>
4 <object>
5   /SOAP-ENV:Envelope/[SOAP-ENV:Body/ACME:GetQuote]
6 </object>
7 <sign value = "+" />
```

Sie besagt, dass für die Gruppe der registrierten Benutzer (Zeile 2) jeder SOAP-Envelope (Zeile 5) durchgelassen wird (Zeile 7), sofern er einen Body mit dem Kind `ACME:GetQuote` (ebenfalls Zeile 5) enthält. In Zeile 7 könnte auch `<sign value="-">` stehen. Das würde bewirken, dass der SOAP-Envelope gefiltert und somit der ganze SOAP-Request nicht erlaubt wird.

Es sei angemerkt, dass es sich in Zeile 5 um X-Path[1] Syntax handelt und man nicht notwendigerweise einen SOAP-Envelope selektieren muss. Ebenfalls muss nicht unbedingt auf eine spezifische SOAP-Operation abgefragt werden. In den meisten Anwendungsfällen läuft es aber darauf hinaus - schon alleine deshalb, weil der SOAP-Request syntaktisch korrekt bleiben muss.

Autorisierungsrichtlinien können sich auch widersprechen. [2, Seite 7] definiert dafür eigene Vorrangregeln.

[1] http://www.w3.org/TR/xpath/

Tabelle 5.1: Vergleichsergebnisse Damiani

Merkmal	**Damiani** [2]	**WSEP**
Einzeloperationen berechtigbar	Ja	Ja
Parametereliminierung	Ja	Nein
Nur Request-Filterung	Ja	Ja
Autorisierungsobjekte	SOAP-Requests	SOAP-Requests
Autorisierungssubjekte	Vordefinierte Attribute	öffentliche Schlüssel
Autorisierungsrichtlinie	Eigenes Format	XACML
Auswahl Autorisierungsobjekte	X-Path	X-Path $\subseteq$ XACML
Widersprüche	Festgelegter Algorithmus	Anpassbarer Algorithmus

Bei WSEP erfolgt die Spezifizierung der Autorisierungsrichtlinie / Policy mit XACML. Das schließt die Verwendung von X-Path mit ein, bietet aber noch weitere Möglichkeiten. Auch hier sind widersprüchliche Autorisierungsrichtlinien erlaubt. Wie diese behandelt werden, wird durch den Rule Combining Algorithmus fest gelegt. XACML beinhaltet mehrere solcher Algorithmen. Im Bedarfsfall kann man XACML um einen eigenen Rule Combining Algorithmus erweitern.

Tabelle 5.1 liefert eine Zusammenfassung der behandelten Unterschiede.

5.2 Sharifi

In [10] wird ein „Framework für die Zugriffskontrolle vorgestellt, welches die Sicherheit von Web Services durch starke Authentifizierung und Autorisierung ausbaut“ [10, Seite 529]. Das Framework nutzt dazu die Technologien SAML und XACML.

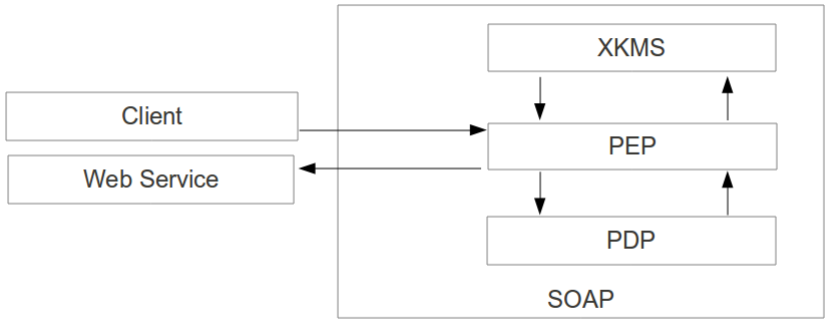

Abbildung 5.2: Funktionsweise Sharifi

Abbildung 5.2 beschreibt die grundlegende Funktionsweise. Der Ablauf beginnt beim Client. Dieser fragt beim PEP um die Nutzung eines Web Service an. Die XML Key Management Spec-

ification (XKMS)[2]-Komponente überprüft die Signatur der Anfrage. Daraufhin wird vom PEP ein XACML Authorization Decision Query an den PDP geschickt.

Der PDP antwortet mit einem XACML Authorization Decision Statement. Darin ist enthalten, ob die Anfrage des Clients erlaubt oder verboten werden soll. In weiterer Folge leitet der PEP die Anfrage des Client entweder an das tatsächliche Web Service weiter, oder er verwirft sie.

Bei [10] handelt es sich um eine konzeptionelle Arbeit. Aus diesem Grund werden keine Details zur Umsetzung besprochen. Im Gegensatz dazu existiert zu WSEP eine Proof-of-Concept Implementierung.

WSEP nutzt einen Protection Intermediary, welcher als SOAP-Intermediary agiert und sich um das Weiterleiten von SOAP-Nachrichten kümmert. Wenn WSEP verwendet wird, verbinden sich Web Service Clients zum Protection Intermediary. Im Gegensatz dazu wendet sich ein Client im Modell von [10] direkt an den PEP. Das Verhalten des PEP beim Weiterleiten (oder Verwerfen) wird nicht näher spezifiziert.

Weiters geht [10] nicht darauf ein, wie die SOAP-Requests, welche der Client an den PEP schickt, im Detail aussehen und welche Maßnahmen die XKMS-Komponente unternimmt, um die Systemsicherheit zu gewährleisten. Außerdem gibt [10] keine Details zur Verwendung von XACML.

Im Modell von [10] kann ein Client mehrere Rollen inne haben. Die Rollenzuordnung wird im Role Repository gespeichert. Es bleibt dem Client überlassen, bestimmte Rollen bei einer Anfrage nicht zu aktivieren.

[10] geht vor allem auf die Funktionsweise des PDP ein. Abbildung 5.3 zeigt, wie dieser aufgebaut ist.

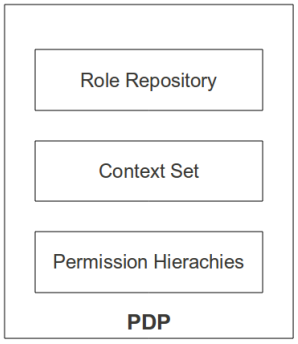

Abbildung 5.3: PDP Sharifi

Der PDP nutzt ein Role Repository, welches vom Sicherheitsadministrator dazu benutzt wird, bestimmten Benutzern bestimmte Rollen zuzuordnen. „Die Mitglieder einer Rolle werden durch die der Rolle zugeordneten Rechte in ihrem Handeln auf eine bestimmte Menge von Aktionen beschränkt.“ [10, Seite 530]

[2] http://www.w3.org/TR/xkms

Das Context Set ist eine weitere Komponente des in [10] beschriebenen PDP. Es enthält Eigenschaften, die alle Clients aufweisen, und Eigenschaften, die nur für bestimmte Clients bekannt sind. Elemente des Context Set können zum Beispiel Quell-IP-Adressen, Datum, Uhrzeit und Firmenzugehörigkeit sein. Unter Verwendung des Context Set kann der Sicherheitsadministrator durch das Formulieren von Regeln eine dynamische Vergabe von Rechten erreichen.

Außerdem verwendet der in [10] beschriebene PDP Permission Hierachies (PH). „Eine PH wird als $PH \subseteq P \times P$ dargestellt. Laut [9] nennt man die partielle Ordnungsrelation $P_{m_1 n_1} \geq P_{m_2 n_2}$ zwischen Permissions genau dann Vererbung, wenn die Permissions von $P_{m_2 n_2}$ zu den Permissions von $P_{m_1 n_1}$ gehören.“ [10, Seite 531]

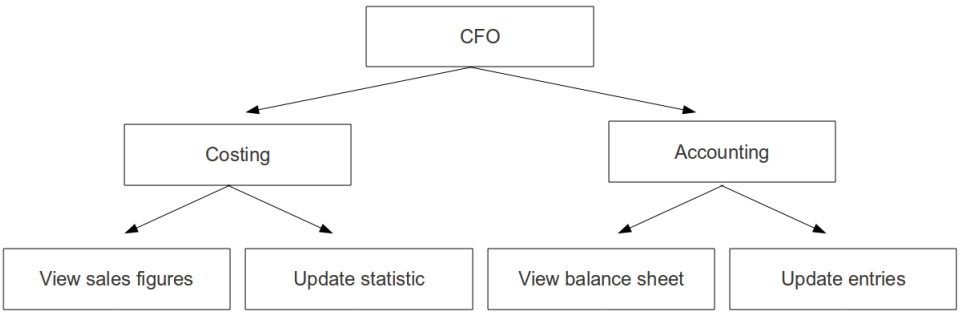

Abbildung 5.4: Permission Hierachy

Abbildung 5.4 zeigt ein Beispiel für eine Permission Hierachy. Vererbung ist mit Pfeilen eingezeichnet. `Costing` beinhaltet `View sales figures` und `Update statistic`. `Accounting` beinhaltet `View balance sheet` und `Update entries`. Weil `CFO` von `Costing` und `Accounting` erbt, gehören dazu auch alle anderen in der Abbildung dargestellten Permissions.

Tabelle 5.2 liefert eine Zusammenfassung der behandelten Unterschiede.

Tabelle 5.2: Vergleichsergebnisse Sharifi

Merkmal	**Sharifi** [10]	**WSEP**
Nutzt SAML und XACML	Ja	Ja
Nutzt SOAP Intermediary	Ja	Ja
Proof-of-Concept existiert	Nein	Ja
Rollen	Ja	Nein
Role Repository	Ja	Nein
Context Set	Ja	Nein
Permission Hierachies	Ja	Nein

KAPITEL 6

Fazit

Web Services sind ein unverzichtbarer Teil jeder Service-Oriented Architecture (SOA). Oft wird dafür SOAP als Nachrichtenprotokoll verwendet. Die Sicherheit ist nicht Teil der eigentlichen SOAP-Spezifikation, sondern wurde mit WS-Security nachgereicht.

> „WS-Security, für sich alleine, garantiert noch keine Sicherheit und stellt auch keine komplette Sicherheitslösung dar. WS-Security ist ein Baustein, der in Verbindung mit anderen Web Service- und applikationsspezifischen Protokollen eine große Menge an Sicherheitsmodellen und Verschlüsselungstechnologien beinhalten kann.“ [7, Seite 4]

Heute existiert eine breite Auswahl an (OASIS-)Standards, die für ein Web Service relevant sein können. Abbildung 6.1 [4, Seite 4] zeigt ein Beispiel.

Man bemerkt, dass viele unterschiedliche Standards die Komplexität erhöhen und die Angriffsfläche vergrößern. Somit wird es auch komplizierter, ein Web Service zu entwickeln, das wirklich sicher ist.

Die Proof-of-Concept Implementierung von WSEP hat gezeigt, dass nicht für alle OASIS-Standards im Web Service-Bereich ausgereifte, sofort nahtlos verwendbare Bibliotheken zur Verfügung stehen. Die WSS4J[1] SAML-Bibliothek ist nicht besonders ausgiebig dokumentiert. Es gibt lediglich den Sourcecode der Testfälle als Dokumentation.

Unter anderem deshalb wurde während der Implementierungsphase auf Open SAML[2] umgestellt. Hier gibt es eine lebendige Community und es ist ausreichend Dokumentation vorhanden. Open SAML ist eher als Tool zur Entwicklung von Sicherheitsprotokollen zu sehen, als als fertige Bibliothek, um mit wenig Aufwand Web Services abzusichern. Die Verwendung von Open SAML ohne Kenntnis der SAML-Spezifikation ist nicht sinnvoll möglich. In diesem Umfeld besteht durchaus noch Bedarf nach Bibliotheken auf einer höheren Abstraktionsebene.

Mit XACML verhält es sich ähnlich wie mit SAML. Im Java-Umfeld existieren zwei Bibliotheken mit jeweils wenig aktiver Community:

[1] http://ws.apache.org/wss4j/

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

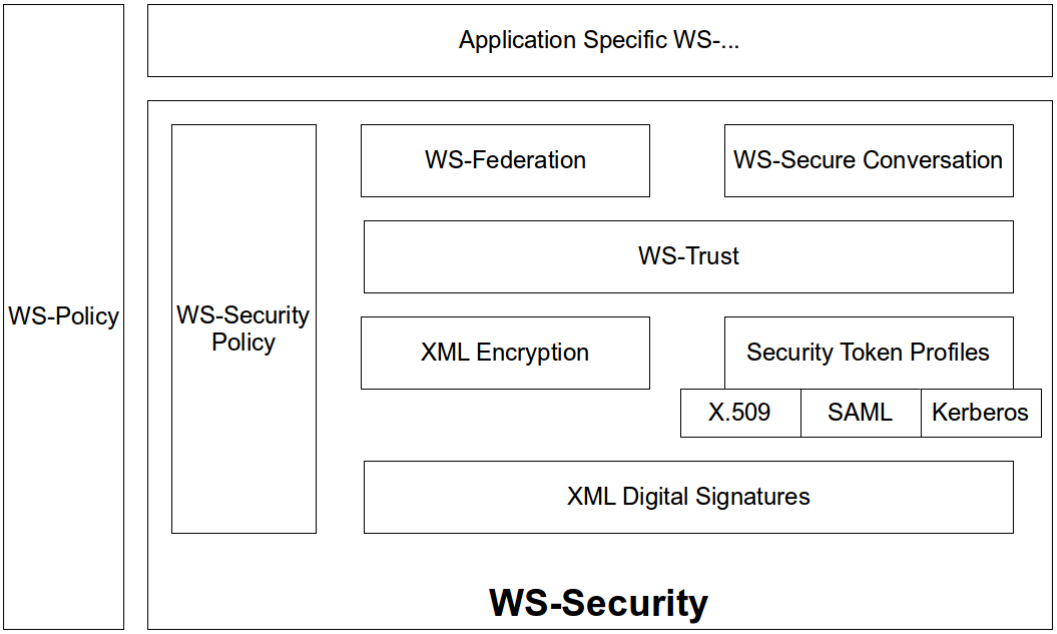

Abbildung 6.1: Relevante Standards

- Enterprise Java XACML Implementation
- Sun's XACML Reference Implementation[3]

Beide Bibliotheken sind ohne Kenntnis der XACML-Spezifikation ebenfalls nicht sinnvoll verwendbar. Auch hier gibt es Raum für eine Bibliothek auf einem höheren Abstraktionsgrad.

WSEP ist ursprünglich entstanden, um den Zugriff auf das SOAP-basierte Web Service eines Digitalstrom Servers abzusichern. Die Auswirkungen des SAML-Token auf die SOAP-Requests ist nicht zu unterschätzen. „Im Durchschnitt erhöht sich die Nachrichtengröße per Aufruf von 750 Bytes auf 14 KBytes“ [6, Seite 7]. Da es sich beim Digitalstrom Server um ein Embedded Device mit begrenzten Ressourcen handelt, ist noch weitere Arbeit erforderlich, um festzustellen, ob man WSEP auf einem Digitalstrom Server verwenden kann.

Weiters hat die Evaluierung der Proof-of-Concept Implementierung von WSEP gezeigt, dass es noch weiterer Arbeit im Bezug auf Sicherheit bedarf:

- Die Vertraulichkeit der SAML-Statements muss sichergestellt werden.
- Diverse Attacken auf die SOAP-Responses müssen verhindert werden.
- Es muss verhindert werden, dass ein Angreifer den Protection Intermediary umgeht.

[3] http://sunxacml.sourceforge.net/

ANHANG A

Listings

A.1 Beispiel Web Service

Listing A.1: Beispiel Web Service

```
1  package at.ac.tuwien.simpleprovider;
2
3  import javax.jws.WebService;
4  import javax.jws.soap.SOAPBinding;
5  import javax.jws.soap.SOAPBinding.Style;
6
7  @WebService
8  @SOAPBinding
9  public class Interpolation {
10
11   /**
12    * Performs a piecewise interpolation on its input.
13    * @param data Input-array. Null-elements are interpolated.
14    * @return Interpolated array.
15    */
16   public Float[] piecewiseInterpolation(Float[] data) {
17
18     Float last = new Float(0);
19     for(int i = 0; i < data.length ; ++i) {
20       if(data[i] == null) {
21         data[i] = last;
22       }
23       else {
24         last = data[i];
```

```
25          }
26        }
27        return data;
28
29      }
30
31      /**
32       * Performs a linear interpolation on its input.
33       * @param data Input-array. Null-elements are interpolated.
34       * @return Interpolated array.
35       */
36      public Float[] linearInterpolation(Float[] data) {
37
38        int end;
39        for(int i = 0; i < data.length ; ++i) {
40          if(data[i] == null) {
41            if(i >= 1 && i + 1 <= data.length) {
42              end = i + 1;
43              while(end < data.length && data[end] == null ) {
44                ++end;
45              }
46              if(end > data.length - 1) {
47                end = data.length - 1;
48              }
49              if(data[end] == null) {
50                data[end] = new Float(0);
51              }
52              data[i] = data[i - 1] + (data[end] - data[i - 1]) *
53                        1 / (end - (i - 1));
54            }
55            else {
56                data[i] = new Float(0);
57            }
58          }
59        }
60        return data;
61
62      }
63    }
```

A.2 Beispiel Web Service Client

Listing A.2: Beispiel Web Service Client

```
1  package at.ac.tuwien.simpleclient;
2
3  import java.util.Arrays;
4
5  import javax.xml.ws.BindingProvider;
6
7  import net.java.dev.jaxb.array.FloatArray;
8
9  import at.ac.tuwien.simpleprovider.Interpolation;
10 import at.ac.tuwien.simpleprovider.InterpolationService;
11 import at.ac.tuwien.wsprotectionlib.KeysFromKeystore;
12 import at.ac.tuwien.wsprotectionlib.Security;
13
14 /**
15  * Example Web Service client protected by SAML and XACML.
16  */
17 public class Main {
18
19   // Where to connect.
20   public final static String ENDPOINT =
21         "http://localhost:7077/interpolator";
22
23   // SAML-token to use.
24   public final static String TOKEN_FILENAME = "saml.xml";
25
26   // Keystore with client keypair. Put in project-dir.
27   // Use longer constructor-variant for more complicated keystores.
28   public final static KeysFromKeystore CLIENT =
29                              new KeysFromKeystore("client.jks");
30
31   public static void main(String[] args) {
32
33     InterpolationService service = new InterpolationService();
34     Interpolation i = service.getInterpolationPort();
35
36     Security security = new Security();
37     try {
38       security.engage((BindingProvider)i, TOKEN_FILENAME,
39         ENDPOINT, CLIENT);
40     }
```

```
41      catch(Exception e) {
42        System.err.println(e);
43        System.exit(-1);
44      }
45
46      FloatArray piecewise = construct(13.0f, null, 17.5f, null,
47          18.0f, null);
48      piecewise = i.piecewiseInterpolation(piecewise);
49      output(piecewise);
50
51      FloatArray linear = construct(1.0f, null, null, 13.3f,
52          null, 50.9f);
53      linear = i.linearInterpolation(linear);
54      output(linear);
55
56    }
57
58    /**
59     * Print the given array to stdout.
60     * @param output Array.
61     */
62    private static void output(FloatArray output) {
63      for(Float f : output.getItem()) {
64        System.out.println(f);
65      }
66      System.out.println();
67    }
68
69    /**
70     * Converts a list of floats to FloatArray.
71     * Needed to transport Web Service-parameters.
72     * @param input Variable number of floats.
73     * @return FloatArray-object.
74     */
75    private static FloatArray construct(Float... input) {
76      FloatArray fa = new FloatArray();
77      fa.getItem().addAll(Arrays.asList(input));
78      return fa;
79    }
80  }
```

A.3 Ant-Buildfiles

Listing A.3: Ant-Buildfile: Simple Provider

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project name="SimpleProvider" default="compile">
3
4          <property name="lib.dir" value="lib"/>
5          <property name="bin.dir" value="bin"/>
6          <property name="src.dir" value="src"/>
7          <property name="main" value="at.ac.tuwien.simpleprovider.Main"/>
8
9          <path id="classpath">
10                 <fileset dir="${lib.dir}" includes="**/*.jar" />
11                 <pathelement location="${bin.dir}"/>
12                 <fileset dir="${java.home}/lib" includes="**/*.jar"/>
13         </path>
14
15         <fileset id="configfiles" dir="${src.dir}" includes="**/*.xml" />
16         <fileset id="logfiles" dir="." includes="**/*.log" />
17
18         <target name="clean">
19                 <delete dir="${bin.dir}" />
20                 <delete><fileset refid="logfiles"/></delete>
21         </target>
22
23         <target name="compile">
24                 <mkdir dir="${bin.dir}" />
25                 <copy todir="${bin.dir}"><fileset refid="configfiles"/>
26                 </copy>
27                 <javac srcdir="src" destdir="${bin.dir}"
28                   includeantruntime="false"
29                   classpathref="classpath" bootclasspathref="classpath">
30         </javac>
31         </target>
32
33         <target name="run" depends="compile">
34                 <java dir="${bin.dir}" classname="${main}" fork="true"
35                    classpathref="classpath"/>
36         </target>
37
38 </project>
```

Listing A.4: Ant-Buildfile: Protection Intermediary

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project name="ProtectionIntermediary" default="compile">
3
4          <property name="lib.dir" value="lib"/>
5          <property name="bin.dir" value="bin"/>
6          <property name="src.dir" value="src"/>
7          <property name="main"
8            value="at.ac.tuwien.protectionintermediary.Main"/>
9
10         <path id="classpath">
11                 <pathelement location="${bin.dir}"/>
12                 <fileset dir="${lib.dir}" includes="**/*.jar" />
13         </path>
14
15         <fileset id="configfiles" dir="${src.dir}"
16            includes="**/*.jks **/*.xml" />
17
18         <target name="clean">
19                 <delete dir="${bin.dir}" />
20         </target>
21
22         <target name="compile">
23                 <mkdir dir="${bin.dir}" />
24                 <copy todir="${bin.dir}">
25                         <fileset refid="configfiles"/>
26                 </copy>
27                 <javac srcdir="src" destdir="${bin.dir}"
28                  includeantruntime="false" classpathref="classpath"/>
29         </target>
30
31         <target name="run" depends="compile">
32                 <java dir="${bin.dir}" classname="${main}" fork="true"
33                  classpathref="classpath">
34                         <!-- 1st parameter is the listening port -->
35                         <arg value="7077"/>
36                         <!-- 2nd parameter is the endpoint host. -->
37                         <arg value="localhost"/>
38                         <!-- 3rd parameter is the endpoint port. -->
39                         <arg value="8088"/>
40                         <!-- 4th parameter is the Java keystore with
41                              the public key of the token provider. -->
42                         <arg value="provider.jks"/>
```

```
43                          <!-- 5th parameter is the keystore password. -->
44                          <arg value="provider"/>
45                          <!-- 6th parameter is the name of the public
46                               key/certificate in the keystore. -->
47                          <arg value="provider"/>
48                  </java>
49          </target>
50
51  </project>
```

Listing A.5: Ant-Buildfile: Simple Client

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project name="SimpleClient" default="compile">
3
4          <property name="lib.dir" value="lib"/>
5          <property name="bin.dir" value="bin"/>
6          <property name="src.dir" value="src"/>
7          <property name="main" value="at.ac.tuwien.simpleclient.Main"/>
8          <property name="service.endpoint"
9            value="http://localhost:7077/mockdss"/>
10
11         <path id="classpath">
12                 <fileset dir="${lib.dir}" includes="**/*.jar" />
13                 <pathelement location="${bin.dir}"/>
14                 <fileset dir="${java.home}/lib" includes="**/*.jar"/>
15         </path>
16
17         <fileset id="configfiles" dir="${src.dir}"
18           includes="**/*.jks␣**/*.xml" />
19         <fileset id="projectfiles" dir="."
20           includes="**/*.jks␣**/*.xml" />
21
22         <target name="clean">
23                 <delete dir="${bin.dir}" />
24                 <delete file="${seqfile}" />
25         </target>
26
27         <target name="compile">
28                 <mkdir dir="${bin.dir}" />
29                 <copy todir="${bin.dir}">
30                         <fileset refid="configfiles"/>
31                         <fileset refid="projectfiles"/>
32                 </copy>
```

```
33                  <javac srcdir="src" destdir="${bin.dir}"
34                    includeantruntime="false"
35                    classpathref="classpath"
36                    bootclasspathref="classpath">
37                  </javac>
38          </target>
39
40          <target name="run" depends="compile">
41                  <java dir="${bin.dir}" classname="${main}" fork="true"
42                    classpathref="classpath">
43                            <arg value="${service.endpoint}"/>
44                  </java>
45          </target>
46
47  </project>
```

Literaturverzeichnis

[1] S. Cantor, I. Kemp, N. Philpott, and E. Maler. Assertions and protocols for the OASIS security assertion markup language. *OASIS, OASIS Standard, March*, 2005.

[2] E. Damiani, S. di Vimercati, S. Paraboschi, and P. Samarati. Fine grained access control for SOAP E-services. In *Proceedings of the 10th international conference on World Wide Web*, pages 504–513. ACM, 2001.

[3] M. Hadley, N. Mendelsohn, J. Moreau, H. Nielsen, and M. Gudgin. SOAP Version 1.2 Part 1: Messaging Framework. *W3C REC REC-soap12-part1-20030624, June*, pages 240–8491, 2003.

[4] B. Hill. https://www.isecpartners.com, Black Hat Briefings 200717, accessed 2011-12-29.

[5] F. Hirsch, R. Philpott, and E. Maler. Security and Privacy Considerations for the OASIS Security Assertion Markup Language (SAML) v2. 0. *Committee Draft*, 1, 2004.

[6] M. Jung, G. Kienesberger, W. Granzer, M. Unger, and W. Kastner. Privacy enabled Web service access control using SAML and XACML for home automation gateways. 2011.

[7] E. Moradian and A. Håkansson. Possible attacks on XML web services. *Int. J. Computer Science and Network Security (IJCSNS)*, 6(1B):154–170, 2006.

[8] T. Moses. eXtensible Access Control Markup Language (XACML) Version 2.0. *OASIS, OASIS Standard, Feb*, 2005.

[9] C. Shang, Z. Yang, Q. Liu, and C. Zhao. A Context Based Dynamic Access Control Model for Web Service. In *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, volume 2, pages 339–343. IEEE, 2008.

[10] M. Sharifi, H. Movahednejad, S. G. H. Tabatabei, and S. Ibrahim. An Effective Access Control Approach to Support Web Service Security. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '09, pages 529–535, New York, NY, USA, 2009. ACM.

Abbildungsverzeichnis

Tabellenverzeichnis