Nils Gruschka

Schutz von Web Services durch erweiterte und effiziente Nachrichtenvalidierung

Cuvillier Verlag Göttingen

Schutz von Web Services durch erweiterte und effiziente Nachrichtenvalidierung

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)

der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Nils Gruschka

Kiel, 2008

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.
1. Aufl. - Göttingen : Cuvillier, 2008
Zugl.: Kiel, Univ., Diss., 2008

978-3-86727-674-0

Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21
www.cuvillier.de

1. Auflage, 2008
Gedruckt auf säurefreiem Papier

978-3-86727-674-0

Vorwort

Netzwerkdienste sind einer Vielzahl von Angriffen ausgesetzt. Insbesondere die sogenannten Denial-of-Service-Angriffe (kurz: DoS-Angriffe) stellen eine große Gefahr dar. Bei Web Services sind derartige Angriffe auch noch deutlich einfacher durchzuführen als gegen „klassische“ Dienste (wie dem WWW). Experimentelle Untersuchungen haben gezeigt, dass es problemlos möglich ist, die Verfügbarkeit mit wenigen oder sogar nur einer einzelnen Nachricht erheblich zu beeinträchtigen. Dabei wurde auch festgestellt, dass viele Angriffe auf der Abweichung vom korrekten Protokoll basieren. Einer der wichtigsten Gegenmaßnahme gegen DoS-Angriffs ist daher auch die Überprüfung der Konformität von Nachrichten bezüglich der Protokolldefinition. Allerdings sind die meisten heutigen Netzwerkschutzsysteme zur Analyse der Web-Service-Nachrichten (SOAP-Nachrichten) und damit zur Erkennung von Angriffen auf Web Services ungeeignet. Dies hat verschiedene Gründe. Zunächst erfordern die XML-basierten SOAP-Nachrichten eine grundlegend andere Verarbeitungsmethode als „klassische“ Protokollnachrichten. Weiterhin definiert SOAP nur eine generische Nachrichtenhülle. Die Definition für die konkreten Web-Service-Nachrichten ergibt sich erst aus einer Reihe von Metadaten, die für jeden Web Service unterschiedlich sind.

Zur Lösung dieser Probleme werden in dieser Arbeit Methoden zur Überprüfung von Web-Service-Nachrichten und zum Schutz von Web-Service-Systemen vor Angriffen entwickelt. Den Kern dieser Gegenmaßnahmen stellt dabei die sogenannte *erweiterte* und *effiziente* Nachrichtenvalidierung dar.

Unter erweiterter Validierung wird dabei die Untersuchung von Nachrichten auf Konformität zu sämtlichen beteiligten Definitionen verstanden. Dies umschließt sowohl die Web-Service-Standards, die Nachrichtenformate definieren, als auch die Metadaten, die jedem Web Service zu eigen sind und die an andere Web-Service-Systeme propagiert werden. Dabei hat sich gezeigt, dass diese Art der Validierung nicht nur gegen Denial-of-Service-Angriffe wirksam ist.

Das größte Problem der erweiterten Validierung ist die Realisierung der Nachrichtenverarbeitung. Zur Validierung ist eine Nachrichtenverarbeitung notwendig, die robust gegenüber Angriffsnachrichten ist. Die in heutigen Web-Service-Systemen überwiegend genutzte *baumbasierte* XML-Verarbeitung ist zur Analyse von potentiellen Angriffsnachrichten ungeeignet. Daher werden in dieser Arbeit Algorithmen zur Verarbeitung und Validierung von Web-Service-Nachrichten entwickelt, die vollständig auf der sogenannten *ereignisbasierten* Verarbeitungsmethodik aufbauen.

Die darauf basierenden Validierungsmechanismen und Schutzmaßnahmen wurden in Form einer Web-Service-Firewall realisiert, welche zum Schutz von herkömmlichenen Web-Service-Systemen genutzt werden kann. Die Evaluation der Implementierung hat die hergeleitete Laufzeit- und Speicherkomplexität und damit die theoretische Schutzwirkung bestätigt. Zusätzlich hat die Web-Service-Firewall ihre Schutzwirkung gegen konkrete Angriffe und geringen Ressourcenbedarf unter Beweis gestellt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung

Die vorliegende Arbeit beschäftigt sich mit der Validierung von Web-Service-Nachrichten. Es wird gezeigt, wie diese Validierung wirkungsvoll zur Abwehr von Angriffen gegen Web-Service-Server eingesetzt werden kann.

Dienste, vor allem solche, die in öffentlichen Netzen angeboten werden, sind einer Vielzahl von Angriffen ausgesetzt. Insbesondere Angriffe, die die Verfügbarkeit von Netzwerk-Systemen reduzieren oder sogar vollständig eliminieren, die sogenannten Denial-of-Service-Angriffe (kurz: DoS-Angriffe, [115]), stellen eine große Gefahr dar. Denial-of-Service-Angriffe gegen weitverbreitete Dienste, wie das *World Wide Web* oder E-Mail, haben über die letzten Jahre stark an Häufigkeit und Auswirkungen zugenommen [138]. Eine der bekanntesten Angriffe dieser Art wurde im Mai 2007 mehrere Wochen lang gegen zahlreiche Webserver in Estland geführt [144]. Dabei wurden neben Webportalen von Zeitungen, Banken und anderer Unternehmen insbesondere auch viele Server der estnischen Regierung in Mitleidenschafft gezogen. Dadurch und durch Gerüchte, die Angriffe würden von Russland ausgehen, erhielt dieser Zwischenfall auch eine starke politische Brisanz. Dies wurde auch durch die Verwendung von Begriffe wie „Cyber-Krieg“ und „Cyber-Terrorismus“ verdeutlicht.

Technisch handelte es sich um einen verteilten DoS-Angriff, bei dem die Webserver von einer großen Anzahl von Rechnern aus mit extrem vielen regulären Anfragen überflutet wurden.

DoS-Angriffe auf derartige „klassische“ Dienste sind heutzutage in der Regel nur mit hohem technischen Aufwand durchführbar, beispielsweise durch Verwendung von *Bot-Netzen* [138]. Bei Web Services hingegen sind DoS-Angriffe deutlich einfacher durchzuführen. Wie in dieser Arbeit anhand von experimentellen Untersuchungen verschiedener Angriffstypen gezeigt wird, ist es problemlos möglich, die Verfügbarkeit mit wenigen oder sogar nur einer einzelnen Nachricht erheblich zu beeinträchtigen.

> *„A single XML Message can cause damage to a naive or misconfigured system.“*
>
> Mark O’Neill, Autor des Buches „Web Services Security“ [129]

Eine der wichtigsten Gegenmaßnahmen gegen derartige Angriffe ist die Untersuchung von eingehenden Nachrichten daraufhin, ob sie potentiell Teil eines Angriffs sind. Dabei gibt es in der Regel zwei Arten von Analysekriterien. Zum einen wird die Konformität der Nachricht bezüglich der Protokolldefinition überprüft. Da bei vielen Angriffsarten der Angriffseffekt durch die Abweichung vom korrekten Protokoll erreicht wird, können durch diese Überprüfung derartige

Angriffe erkannt werden. Zum anderen werden zur Erkennung von Angriffen, die innerhalb der Protokolldefinitionen bleiben, Nachrichten nach empirisch gewonnenen Angriffsmustern durchsucht.

Realisiert sind diese Gegenmaßnahmen typischerweise in *Firewalls* oder *Intrusion-Detection-Systemen*. Allerdings sind die meisten derartigen Systeme heutzutage zur Analyse der Web-Service-Nachrichten (SOAP-Nachrichten) und damit zur Erkennung von Angriffen auf Web Services ungeeignet.

> *SOAP goes through firewalls like a knife through butter.*
>
> Tim Bray, Verfasser der XML-Spezifikation [26]

Dies hat verschiedene Gründe. Zunächst erfordern die XML-basierten SOAP-Nachrichten eine grundlegend andere Verarbeitungsmethodik als „klassische“ Protokollnachrichten. Weiterhin definiert SOAP nur eine generische Nachrichtenhülle. Die Definition für die konkreten Web-Service-Nachrichten ergibt sich erst aus einer Reihe von Metadaten, die jeder Web Service verwendet und die auch für jeden Web Service unterschiedlich sind. Und schließlich lassen sich auch die Methoden zur Erkennung von Angriffsmustern in Nachrichtenströmen nicht auf XML-Strukturen anwenden.

Zur Lösung dieses Problems werden in dieser Arbeit Methoden zur Überprüfung von Web-Service-Nachrichten und damit zum Schutz von Web-Service-Systemen vor Angriffen entwickelt. Den Kern dieser Gegenmaßnahmen stellt dabei die sogenannte *erweiterte* und *effiziente* Nachrichtenvalidierung dar.

Unter erweiterter Validierung wird dabei die Untersuchung von Nachrichten auf Konformität zu sämtlichen relevanten Definitionen verstanden. Dies umschließt sowohl die Web-Service-Standards, die Nachrichtenformate definieren (SOAP, WS-Security, WS-Addressing usw.), als auch die Metadaten, die jedem Web Service zu eigen sind und die an andere Web-Service-Systeme propagiert werden (Web-Service-Beschreibung gemäß WSDL, Sicherheits-Policy gemäß WS-SecurityPolicy, Nachrichtenreihenfolge gemäß BPEL usw.). Weiterhin wird auch die Überprüfung gegen die Zugriffskontroll-Policy in diese Validierung mit einbezogen. Die Protokolldefinitionen, die sich aus all diesen Teilvorschriften ergeben, werden dabei zusätzlich derart modifiziert, dass sie Nachrichten mit bekannten Angriffsmustern ausschließen.

Dabei hat sich gezeigt, dass diese Art der Validierung nicht nur Denial-of-Service-Angriffe erkennen kann, sondern auch gegen andere Angriffe (beispielsweise die bekannten *XML-Rewriting*-Angriffe [110]) wirksam ist. Schließlich ist eine vollständige Protokollüberprüfung auch ein wichtiger Teil der Erfüllung von Dienstabsprachen (engl. *Compliance*)[1].

Das größte Problem der erweiterten Validierung ist die Realisierung der Nachrichtenverarbeitung. Die verbreitetste Art der XML-Verarbeitung (die sogenannten *baumbasierten* Methoden), die auch von den meisten Web-Service-Systemen verwendet wird, ist zur Analyse von potentiellen Angriffsnachrichten ungeeignet. Daher werden in dieser Arbeit Algorithmen zur Verarbeitung und Validierung von Web-Service-Nachrichten entwickelt, die vollständig auf der sogenannten *ereignisbasierten* Verarbeitungsmethodik aufbauen.

Darauf basierend wurden die Validierungsmechanismen implementiert und die Schutzmaßnahmen als Web-Service-Firewall realisiert. Die Evaluation der Implementierung hat die hergeleitete Laufzeit- und Speicherkomplexität und damit die theoretische Schutzwirkung bestätigt. Zusätzlich hat die Web-Service-Firewall ihre Schutzwirkung gegen konkrete Angriffe und ihren für praktische Einsätze verwendbaren Ressourcenbedarf bewiesen.

[1] Dies wird im Folgenden nicht weiter diskutiert werden.

1.2 Stand der Technik

In diesem Kapitel wird eine kurze Einführung in die Themengebiete gegeben, die dieser Arbeit zugrunde liegen. Diese sind Computersicherheit und Angriffe, Web Service und Web-Service-Sicherheit. Dabei werden auch bestehende Maßnahmen zur Sicherung von Web Services vorgestellt, die sich zumeist um die wichtigsten Web-Service-Sicherheits-Standards rangen: WS-Security und WS-SecurityPolicy.

1.2.1 Computersicherheit

Informationstechnische Systeme sind einer Vielzahl von Gefährdungen ausgesetzt. Dabei wird zwischen *Bedrohung* (engl. *Threat*) und *Angriff* (engl. *Attack*) unterschieden [142]. Eine Bedrohung ist eine Möglichkeit für eine Verletzung der Sicherheit, während ein Angriff ein konkreter Anschlag auf die Sicherheit eines Systems ist.

Nach klassischer Lehre sind die drei Hauptaspekte von Computersicherheit: *Vertraulichkeit* (engl. *Confidentiality*), *Integrität* (engl. *Integrity*) und *Verfügbarkeit* (engl. *Availability*); im Englischen auch oft (scherzhaft) als „CIA“ abgekürzt [18].

Vertraulichkeit

Vertrauliche Daten sind solche, deren Kenntnis nicht jeder erlangen darf. Zur Erreichung dieses Sicherheitsziels können beispielsweise kryptographische Verschlüsselung von Daten (bei der Speicherung und der Übertragung) und Zugriffskontrolle für die Dienste, die auf diesen Daten operieren, eingesetzt werden.

Integrität

Integrität beschreibt die Anforderung, dass Daten „unverfälscht“ sind. Dabei kann zwischen *Daten-Integrität* (Integrität des Inhalts) und *Ursprungs-Integrität* (Integrität der Herkunft, auch *Authentizität* genannt) unterschieden werden. Der Prozess der Feststellung der Authentizität wird *Authentifizierung* genannt. Zur Gewährleistung der Integrität kommen kryptographische Methoden wie *Hash-Bildung* (auch *Hashing* genannt) oder *digitale Signaturen* und wiederum Zugriffskontrollen zum Einsatz.

Verfügbarkeit

Verfügbarkeit bezeichnet die Möglichkeit, auf eine Ressource zuzugreifen. Obwohl diese Eigenschaft vielfach nicht als sicherheitsrelevant angesehen wird, ist sie für ein sicheres Systems doch zwingend erforderlich. Ein klassisches Beispiel ist ein Alarmdienst, der über einen Einbruch in einen Tresor informiert [126]. Bei diesem Dienst ist Vertraulichkeit und Integrität zweitrangig. Wenn aber die Verfügbarkeit nicht gegeben ist, ist die Sicherheit des Gesamtsystems nicht mehr gewährleistet.

Neben diesen „Axiomen“ der IT-Sicherheit werden teilweise noch weitere Sicherheitsanforderungen genannt, diese lassen sich aber meist auf die oben genannten zurückführen. Häufige Forderungen sind dabei

- Verlässlichkeit (engl. *Reliability*)
- Nicht-Abstreitbarkeit (engl. *Non-Repudiation*)
- Privatheit (engl. *Privacy*)

Die Aufgabe von Computer-Sicherheit ist die Analyse von Bedrohungen, mit dem Ziel Angriffe zu verhindern (engl. *Prevention*) oder zumindest zu erkennen (engl. *Detection*), sowie ein System nach einem Angriff wiederherzustellen (engl. *Recovery*). Ein Mechanismus, der dieses leistet, wird als *Sicherheits-Mechanismus* (engl. *Security Mechanism*) bezeichnet [145].

Angriffe lassen sich nach den durch sie gefährdeten Sicherheitsaspekten klassifizieren. Angriffe gegen die Verfügbarkeit werden als *Denial-of-Service-Angriffe*, im Deutschen teilweise auch als *Sabotageangriffe* bezeichnet. Dabei gibt es natürlich auch Angriffe, die gegen mehr als einen Sicherheitsaspekt wirken oder wirken können. So ist es beispielsweise mit einen *SQL-Injection*-Angriff [4] möglich, nicht intendierte SQL-Anfragen an die Datenbank eines Dienstes zu stellen. Diese Anfragen können dabei sowohl zur Reduktion der Verfügbarkeit als auch zum Erlangen von vertraulichen Informationen missbraucht werden.

Zur Sicherung der Verfügbarkeit sind in der Vergangenheit einige allgemeine Methoden entwickelt worden. Die wichtigsten sind die Folgenden:

- Überprüfung der Eingabe
- Beschränkung des Zugriffs
- Beschränkung des Ressourcenverbrauchs

1.2.2 Web Services

Das Konzept der *dienstorientierten Architektur* [52] (engl. *Service-Oriented Architecture*, SOA) stellt ein wichtigen Schritt in der Geschichte der Software-Entwicklung dar. Die Hauptidee von SOA besteht darin, monolithische Applikationen in kleinere Komponenten aufzubrechen und diese plattformunabhängig als Netzwerkdienst wiederverwendbar anzubieten. Die Erstellung komplexerer Applikationen erfolgt dann durch *Komposition* dieser Dienste [152]. Der bekannteste Standard zur Definition solcher Kompositionen ist die *Business Process Execution Language* (BPEL) [3].

Web Services [19, 2] sind die am weitesten verbreitete Realisierung der Kommunikation in einer dienstorientierten Architektur. Web Services werden oft als *Middleware* für verteilte Anwendungen betrachtet [98]. Im Gegensatz zu *Middleware*-Systemen wie DCOM oder RMI sind Web Services weder an bestimmte Betriebssysteme noch Programmiersprachen gebunden und sollen damit eine leichte Zusammenführung von verteilten Applikationen ermöglichen (beispielsweise im Rahmen der sogenannten *Enterprise Application Integration*).

Die Spezifikationen im Web-Service-Umfeld sind grundsätzlich sehr stark auf Flexibilität und Erweiterbarkeit ausgelegt. Zusätzlich sind Details teilweise nicht präzise oder eindeutig definiert. Da diese beiden Eigenschaften zu Problemen bei der Entwicklung entsprechender Systeme und deren Interoperabilität führen, hat die *Web Service Interoperability Organization* (WS-I) Empfehlungen herausgegeben, in denen die betroffenen Spezifikationen präzisiert oder eingeschränkt werden [9, 111]. Alle wichtigen Web-Service-Implementierungen berücksichtigen diese Empfehlungen inzwischen. In dieser Arbeit werden daher nur Web Services betrachtet, die dieser Empfehlung folgen.

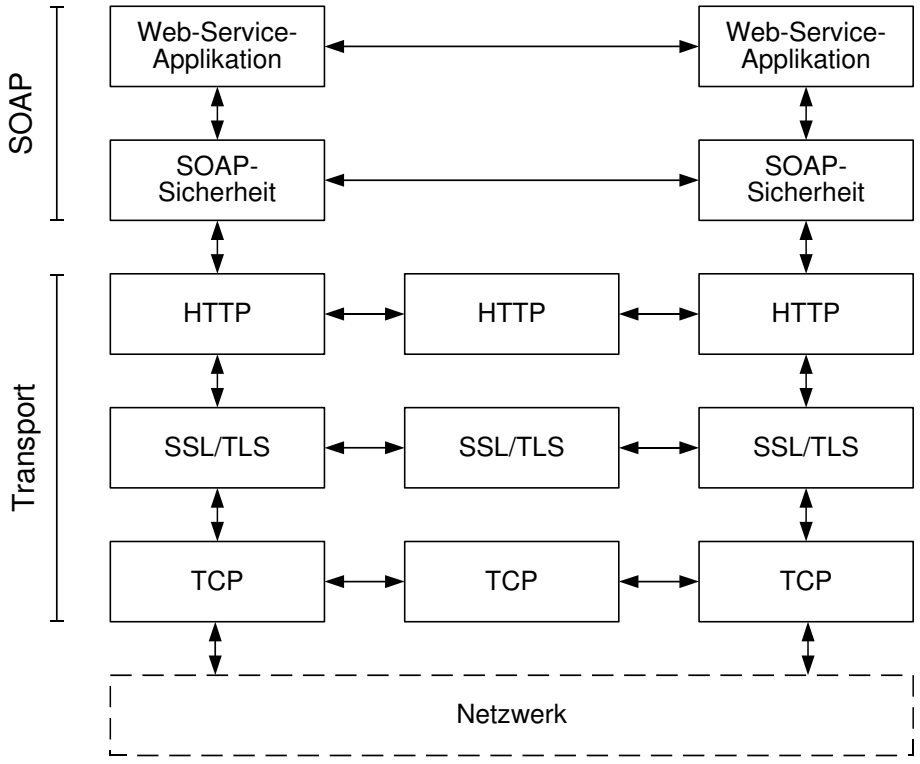


Abbildung 1.1: Protokollschichtung für Web Services (inklusive Sicherheitsprotokollen)

Der wichtigste designierte Anwendungsfall von Web Services ist die Kommunikation zwischen Unternehmen auch über offene Netze hinweg. Auch aus dieser Überlegung heraus ergibt sich die Notwendigkeit zur Sicherung von Web Services.

1.2.3 Web-Service-Sicherheit

Zur Sicherung von Vertraulichkeit, Integrität und Authentizität und zur Durchführung von Authentifizierung bei Netzwerkdiensten existieren einige weitverbreitete Standards wie z. B. *Secure Socket Layer* (SSL bzw. TLS) [59, 44] oder IPsec [46]. Diese Sicherheitsmechanismen sind an ein bestimmtes Transportprotokoll gebunden und realisieren damit *transportorientierte* Sicherheit. Solche Sicherheitsprotokolle sind für Web Services nur bedingt geeignet [40].

Eine der grundlegenden Ideen von Web Services ist die Unabhängigkeit vom verwendeten Transportprotokoll. Auch wenn für SOAP-Nachrichten de facto nur der Transport über HTTP verwendet wird, so sind grundsätzlich auch andere Transportprotokolle denkbar. Weiterhin wird selbst bei einem festen Transportmechanismus der Transport nicht immer Ende-zu-Ende bezüglich des Web Services (d. h. vom Nachrichtenersteller zum -empfänger) durchgeführt. Ein Beispiel sind SOAP-Zwischensysteme, die SOAP-Nachrichten weiterleiten und dabei auch Modifikationen durchführen, die typischerweise als Endpunkt des Transportprotokolls implementiert sind.

Zur Sicherung von SOAP-Nachrichten werden daher *nachrichtenorientierte* Mechanismen verwendet [80, 136]. Abbildung 1.1 zeigt die Einordnung dieser Mechanismen im Verhältnis zu anderen Internetprotokollen. Der wichtigste Standard für SOAP-Sicherheit ist WS-Security [124, 76].

WS-Security definiert folgende Sicherheitsmechanismen:

- Verschlüsselung von SOAP-Fragmenten
- Signierung von SOAP-Fragmenten
- Sicherheits-Token
- Zeitstempel

WS-Security ist bei Systemen für Web-Service-Sicherheit heutzutage weit verbreitet [27]. Viele Web-Service-Frameworks bieten WS-Security-Unterstützung, beispielsweise WSE [89] für Microsoft .NET oder Rampart [55] für Apache Axis2.

Aufgrund der Komplexität von WS-Security und insbesondere der komplexen Referenzierung zwischen den WS-Security-Elementen, wird die WS-Security-Verarbeitung typischerweise baumbasiert realisiert. Es existieren aber auch Ansätze für ereignis- bzw. strombasierte Lösungen. So wird in [108] ein Verfahren vorgestellt, welches SOAP-Nachrichten mittels vordefinierter Schablonen (engl. *Templates*) für WS-Security-Elemente verarbeitet. Dieses System funktioniert aber nur für ein festes WS-Security-Framework und lässt sich nicht verallgemeinern.

Eine ereignisbasierte Verifizierung von Signaturen in SOAP-Nachrichten wird in [107] präsentiert. Die dort vorgestellte Methode ist allerdings nicht geeignet, Signaturen mit Rückwärtsreferenzen zu behandeln, und berücksichtigt die Anforderung des WS-I [111] nicht. Außerdem wird die Problematik der Schlüsselreferenzierung und -erlangung nicht betrachtet.

Ein Ansatz zur ereignisbasierten Ver- und Entschlüsselung von XML-Dokumenten wird in [85] gezeigt. Die Arbeit beschäftigt sich aber nicht mit der Einbindung in SOAP-Nachrichten und insbesondere nicht mit der Referenzierung von Schlüsseln und verschlüsselten Schlüsseln.

WS-Security definiert generische Sicherheitselemente und erlaubt damit eine flexible Anwendung dieser Elemente auf SOAP-Nachrichten. Dies führt zu der Notwendigkeit, dass Web-Service-Partner sich gegenseitig über ihre Sicherheitsanforderungen und damit über die geforderten WS-Security-Elemente informieren müssen. WS-SecurityPolicy [95] erlaubt die Spezifikation solcher Anforderungen.

Policys werden typischerweise von einem Web-Service-Server angeboten und beschreiben seine Forderungen an eingehende Nachrichten sowie die Eigenschaften der ausgehenden Nachrichten. Ein Web-Service-Client ist damit in der Lage, seine ausgehenden Nachrichten entsprechend zu konstruieren bzw. seine eingehenden Nachrichten korrekt zu interpretieren.

Die Anbindung einer Policy an eine Web-Service-Beschreibung wird in WS-PolicyAttachment [48] beschrieben. Dies erlaubt die Referenzierung aus verschiedenen Teilen der Web-Service-Beschreibung auf mehrere Policys. So können dem Endpunkt, der Operation und der Nachricht jeweils eigene Teil-Policys zugewiesen werden. Diese heißen dann entsprechend *Endpunkt-Policy*, *Operations-Policy* und *Nachrichten-Policy*. Die *effektive* Policy für eine Nachricht ergibt sich aus der Vereinigung der Forderungen der (bis zu 4) zugehörigen Policys.

Das größte Problem bei der Benutzung von WS-SecurityPolicy ist die Erstellung von Policys für die gewünschten Sicherheitsanforderungen. Dabei ergibt sich zum einem das Problem der Umsetzung der abstrakten Sicherheitsanforderungen (z. B. „Kreditkartennummern dürfen nicht gelesen oder modifiziert werden.“) in einer konkreten Policy. In [146] und [87] werden Muster für die praktische Realisierung (engl. *Best Practice Pattern*) von Sicherheitsanforderungen vorgestellt.

Ein weiteres Problem ist, dass die einzelnen Teilanforderungen einer Sicherheits-Policy in komplexem Zusammenhang stehen. Dies führt dazu, dass Sicherheits-Policys oftmals Sicherheitsschwächen enthalten, die von Menschen schwer erkennbar sind. Ein einfaches Beispiel ist die Forderung von Integrität des SOAP-Bodys und Schutz gegen *Replay*-Angriffe. Die direkte Umsetzung in eine Sicherheits-Policy würde die Signierung des SOAP-Bodys, die Benutzung eines Zeitstempels und die Signierung des Zeitstempels erfordern. Eine Nachricht, die dieser Sicherheits-Policy entspricht, ist aber anfällig für einen einfachen *XML-Rewriting*-Angriff: der separat signierte Zeitstempel kann von einer anderen (aktuellen) Nachricht in die alte Nachricht kopiert werden. Zum Schutz vor solchen Angriffen gibt es eine große Anzahl von Arbeiten, die sich mit der Analyse von Sicherheits-Policys bzgl. derartiger Schwachstellen beschäftigen [14, 16, 130].

Für die praktische Realisierung hat WS-SecurityPolicy noch einige weitere Schwächen. So ist neben der komplexen Struktur mit Alternativenbildung und Verteilung der Anforderungen auf Teil-Policys das Fehlen von konkreten Identitäten ein Problem. Damit ist WS-SecurityPolicy allein nicht geeignet zur Definition von Zugriffskontroll-Policys. Einige Systeme lösen dies durch proprietäre Erweiterungen von WS-SecurityPolicy um konkrete Identitäten ([5], Apache Rampart 1.1). Damit sind derartige Policys aber nicht mehr zwischen Web-Service-Partnern austauschbar und widersprechen der Intention von WS-SecurityPolicy. In [68] wird eine Erweiterung um Identitäten vorgestellt, bei der die Austauschbarkeit der Policys erhalten bleibt. Sicherlich nicht zuletzt aufgrund dieser Eigenschaften verwenden Web-Service-Frameworks (Microsoft WSE [89], Apache Rampart 1.0) und Web-Service-Firewalls häufig nicht WS-SecurityPolicy, sondern eigene Policy-Sprachen für ihre Sicherheits-Policys.

Alle hier genannten Arbeiten und Spezifikationen zum Thema Web-Service-Sicherheit beschäftigen sich mit der Sicherung der Integrität und Vertraulichkeit bei Web Services. Die Sicherung der Verfügbarkeit bzw. der Schutz vor Denial-of-Service-Angriffen ist im Kontext von Web Services ein wenig beachtetes Thema. Dabei zeigen Untersuchungen [143, 104, 92], dass Web Services sogar noch verwundbarer für DoS-Angriffe sind als nicht-Web-Service-basierte Dienste.

Ein erhöhtes Bewusstsein für die Problematik der Verfügbarkeit lässt sich an der Entwicklung der Web-Service-Frameworks ablesen. So ließ sich unter Microsoft .NET Version 1.0 ein Web-Service-System noch problemlos durch eine übergroße SOAP-Nachricht zum völligen Stillstand bringen, während dies bei .NET Version 2.0 nicht mehr möglich ist.

Allerdings lösen die verwendeten Maßnahmen die Probleme oftmals nur teilweise oder führen zu neuen Schwachstellen. So wurde die oben genannte Verbesserung dadurch erreicht, dass der .NET-Service alle Nachrichten ablehnt, die eine bestimmte serialisierte Größe überschreiten, in der Standardeinstellung 4 MByte. Eine derartige Maßnahme ist aber nicht unproblematisch. Zunächst kann ein Web Service damit keine sehr großen legitimen Nachrichten verarbeiten. Weiterhin ist die Bewertung einer XML-basierten Nachricht an Hand der serialisierten Größe grundsätzlich ungenau, da ein und derselbe XML-Baum abhängig von der Serialisierung unterschiedliche Dokumentgrößen ergeben kann. So lehnt ein .NET-Service auch Nachrichten ab, die aus 4 MByte Leerzeichen bestehen. Schließlich zeigen die in dieser Arbeit präsentierten Angriffe, dass bei Web Services auch Nachrichten kleiner als 4 MByte zum Denial-of-Service führen können.

Eine weitere Maßnahme, die von vielen Systemen benutzt wird, um die Laufzeit und den Speicherverbrauch während der Verarbeitung zu reduzieren, ist die *tolerante* (engl. *lax*) Verarbeitung von Nachrichten. Dabei wird die Nachricht nicht auf genaue Konformität zum Protokoll überprüft. Bei der Web-Service-Verarbeitung wird beispielsweise keine Schema-Validierung

durchgeführt, was entsprechende Verarbeitungsressourcen einspart. Dies kann dann aber andere Angriffe ermöglichen, wie beispielsweise *XML Rewriting* oder *XML Injection*.

1.3 Beitrag der Arbeit

Der Beitrag der hier vorliegenden Arbeit ist eine Methodik zur Erkennung von Angriffen auf Web Services. Dies wird durch strenge Validierung von eingehenden Nachrichten gegen die Definitionen des konkreten Web Services erreicht. Diese Validierung besteht dabei aus folgenden Teilschritten:

Einschränkung der Nachrichtendefinitionen

Aus den Spezifikationen und Metadaten des Web Services werden die Definitionen für die erlaubten Nachrichten dieses Web Services generiert. Diese werden analysiert und dabei werden Vorschriften, die potentielle Angriffsnachrichten erlauben, entsprechend modifiziert. Dazu gehören beispielsweise unbeschränkte Listen in der Web-Service-Beschreibung oder zu schwache Sicherheitsforderungen in der Sicherheits-Policy.

Validierung von eingehenden Nachrichten

Eingehende Nachrichten werden gegen die generierten Definitionen überprüft. Dies umschließt die folgenden Teilvalidierungen:

- Validierung gegen das XML-Schema [54], das durch die Web-Service-Beschreibung definiert ist
- Entschlüsselung von verschlüsselten Dokumentteilen [86]
- Verifizierung der digitalen Signaturen [10]
- Validierung der Sicherheitseigenschaft gegen die Sicherheits-Policy [95]
- Überprüfung der Erfüllung der Zugriffs-Policy
- Validierung der korrekten Nachrichtenreihenfolge, die sich aus der BPEL-Beschreibung [3] ergibt

Durch diese Validierung werden die allgemeinen DoS-Gegenmaßnahmen „Überprüfung der Eingabe“ und „Beschränkung des Zugriffs“ realisiert. Gleichzeitig werden damit, insbesondere durch die Validierung gegen die (verschärfte) Sicherheits-Policy, Angriffe gegen die Sicherheitsziele Integrität und Vertraulichkeit verhindert oder zumindest erschwert.

Realisierung der Validierung

Die oben geschilderte Verarbeitung und Analyse von Nachrichten erfordert natürlich gewisse Ressourcen. Damit ist die Validierung selbst ein potentielles Ziel für Denial-of-Service-Angriffe, die auf Ressourcenerschöpfung abzielen. Ein solcher Angriff ist bei den üblichen baumbasierten XML-Verarbeitungsverfahren sogar sehr einfach, da diese einen extrem hohen Ressourcenbedarf haben. Insbesondere beim Speicherbedarf sind Faktoren von mindestens fünf im Vergleich zur Dokumentgröße üblich [84].

Das zweite Problem ist, dass hier die Verarbeitung erst beginnen kann, wenn das Dokument vollständig eingelesen wurde. Damit ist leicht einzusehen, dass eine reine Baumverarbeitung in jedem Fall durch ein „unendlich“ großes Dokument angegriffen werden kann. Die DoS-Gegenmaßnahmen „Überprüfung der Eingabe“ und „Beschränkung des Ressourcenverbrauchs“ widersprechen sich hier also offensichtlich. Damit ist eine baumbasierte Validierung von Nachrichten als Schutzmaßnahme für Web Services ungeeignet.

In dieser Arbeit wurde daher für die Durchführung der Validierung eine ereignisbasierte XML-Verarbeitungmethodik gewählt, welche die vorher erwähnten Nachteile nicht aufweist. Durch die direkte Verarbeitung des Eingabestroms wird das Dokument nur maximal bis zu dem Punkt verarbeitet, an dem eine der Validierungsoperationen eine Verletzung der Vorgaben feststellt. Damit lässt sich der Speicherverbrauch bei der Verarbeitung durch Einschränkung der Nachrichtendefinitionen beschränken. Weiterhin ist durch das Fehlen einer komplexen Dokumentrepräsentation auch der absolute Speicherverbrauch deutlich geringer als bei baumbasierten Methoden.

Der Effizienz der ereignisbasierten Verarbeitungsmodelle stehen aber auch einige Nachteile gegenüber. Die Verarbeitung von XML-Dokumenten erfordert typischerweise die Navigation innerhalb des XML-Baums und die Manipulation von Teilbäumen. Dies gilt auch für die Verarbeitung von SOAP-Nachrichten, beispielsweise bei der Referenzierung von signierten oder bei der Entschlüsselung von verschlüsselten Dokumententeilen. Solche Operationen sind bei der ereignisbasierten Verarbeitung nicht direkt durchführbar. Insbesondere das Navigieren zu vorherigen Dokumentteilen oder die Auswertung von XPath-Ausdrücken ist im Allgemeinen gar nicht möglich.

Einer der Hauptbeiträge dieser Arbeit stellt deshalb auch die vollständig ereignisbasierte Validierung von Web-Service-Nachrichten dar. Dafür wurden neue Algorithmen entwickelt, beispielsweise für ereignisbasierte Verarbeitung von WS-Security-Elementen oder für ereignisbasierte Validierung von Sicherheits-Policys.

1.4 Aufbau der Arbeit

Der erste Teil der Arbeit beschäftigt sich mit den Grundlagen für die erweiterte Validierung. Wie bereits motiviert, hat die XML-Verarbeitung großen Einfluss auf die Verletzbarkeit von Web Services und auf die Realisierung der Gegenmaßnahmen. Daher werden in Kapitel 2 die beiden wichtigsten Verarbeitungsmethodiken vorgestellt, abstrakt modelliert und in Bezug auf Speicherverbrauch und Laufzeit bewertet. Im nachfolgenden Kapitel wird eine Einführung in die wichtigsten Web-Service-Standards gegeben, die für das Verständnis diese Arbeit notwendig sind. Kapitel 4 präsentiert die für diese Arbeit wichtigste Klasse von Angriffen, die Denial-of-Service-Angriffe. Neben einem historischen Überblick und einer Vorstellung von Metriken und Gegenmaßnahmen wird ein Klassifizierungssystem für derartige Angriffe entwickelt. In Kapitel 5 werden Angriffe gegen Web Services vorgestellt und gemäß dem vorher genannten Klassifizierungssystem eingeordnet. Die Angriffe sind aus theoretischen Untersuchungen von Schwachstellen entstanden und zum großen Teil in praktischen Experimenten mit verbreiteten Web-Service-Systemen nachgewiesen worden.

Im zweiten Teil der Arbeit wird die Nachrichtenvalidierung behandelt. Zunächst wird in Kapitel 6 ein Überblick über alle Bestandteile der erweiterten Validierung und deren Schutzwirkung gegeben. Außerdem wird aus den theoretischen Überlegungen zur XML-Verarbeitung die ereignisbasierte Realisierung der Schutzmaßnahmen hergeleitet. In den folgenden Kapiteln wer-

den dann die einzelnen Bestandteile der Validierung beschrieben. Dabei werden jeweils abstrakt die Algorithmen zur Verarbeitung der Metadaten und zur ereignisbasierten Validierung der Nachricht sowie die Schutzwirkung gegen bestimmte Klassen von Angriffen erläutert. Kapitel 7 erläutert die Schema-Generierung aus Web-Service-Beschreibungen und die Validierung von SOAP-Nachrichten gegen solche Schemata. In Kapitel 8 wird die ereignisbasierte Verarbeitung von WS-Security-Elementen behandelt. Damit wird die Schema-Validierung von verschlüsselten Nachrichtenteilen sowie die Validierung gegen die Sicherheits-Policy ermöglicht. Kapitel 9 präsentiert dann die Validierung der Sicherheitselemente in der SOAP-Nachricht gegen die Forderung der Sicherheits-Policy. Anschließend wird in Kapitel 10 gezeigt, wie Zugriffskontrolle (also die Validierung gegen die Zugriffskontroll-Policy) effizient ereignisbasiert durchgesetzt werden kann. Kapitel 11 behandelt die Validierung der korrekten Nachrichtenreihenfolge. Dazu wird ein Automatenmodell vorgestellt, dass es erlaubt, aus BPEL-Prozessbeschreibungen Protokollablaufautomaten zu generieren. Diese Automaten werden verwendet, um zu überprüfen, ob eingehende Nachrichten im aktuellen Zustand des Ablaufprozesses erlaubt sind.

Der dritte Teil der Arbeit behandelt die Realisierung der Nachrichtenvalidierung zum Schutz von Web Services. Die vorgestellten Validierungsmaßnahmen wurden vollständig implementiert und als Web-Service-Firewall realisiert. Diese Implementierung wird in Kapitel 12 vorgestellt. Zum Nachweis der Funktionalität und der auch für praktische Zwecke ausreichenden Performance wurde das System einer Reihe von Evaluationstests unterzogen. Deren Ergebnisse finden sich in Kapitel 13. Die Arbeit schließt in Kapitel 14 mit einer Zusammenfassung und einem Ausblick.

Teil I

Grundlagen

Kapitel 2

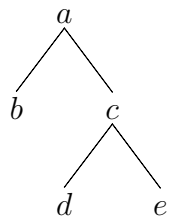
XML

2.1 XML-Dokumentenstruktur

Die *Extensible Markup Language* (XML) [26] ist ein Standard zur Definition von *Auszeichnungssprachen* (engl. *Markup Languages*), die wiederum zur Darstellung hierarchisch strukturierter Daten verwendet werden können. Instanzen einer XML-Sprache repräsentieren baumartige Datenstrukturen wie beispielsweise von XML Information Set [38] oder XPath [36] definiert. Die *serialisierte Form*, d. h. die textuelle Repräsentation des XML-Baums wird als *XML-Dokument* bezeichnet.

In dieser Arbeit wird eine vereinfachte Darstellung des XML-Baums verwendet, die nur die *XML-Elemente* und ihre Beziehung bezüglich der *Kind-Achse* (engl. *Child Axis*) sowie XML-Inhalte (engl. *XML Content*) verwendet [140].

Sei t ein XML-Baum mit den Knoten a, b, c, d und e und der folgenden Struktur:



Dann ist $[t] = a\,b\,\overline{b}\,c\,d\,\overline{d}\,e\,\overline{e}\,\overline{c}\,\overline{a}$ die dazugehörige serialisierte Form, also das XML-Dokument. Dies bedeutet, dass das Ende eines Teilbaums im Dokument nicht direkt von der Teilbaumwurzel aus referenziert, sondern durch ein spezielles Symbol gekennzeichnet wird. Dies ist eine wichtige Eigenschaft, die bei der Verarbeitung von XML-Dokumenten berücksichtigt werden muss. Die serialisierte Form eines Teilbaums wird auch als *XML-Fragment* bezeichnet.

Allgemein gelte: Sei Σ die Menge der XML-Elemente. Dann repräsentiert $a \in \Sigma$ die *Startkennung* (engl. *Opening Tag*). Sei weiterhin $\overline{\Sigma} = \left\{\overline{a} \;\middle|\; a \in \Sigma\right\}$. Dann repräsentiert $\overline{a}$ das *Endkennung* (engl. *Closing Tag*). Die serialisierte Form $[t]$ eines Baums t ist dann induktiv wie folgt definiert: wenn t nur aus dem *Wurzelelement* (engl. *Document Element*) mit der Beschriftung a besteht, dann ist $[t] = a\overline{a}$. Wenn t einen Wurzelknoten mit der Beschriftung a und die Teilbäume $t_1, \ldots, t_n$ besitzt, dann ist $[t] = a[t_1] \ldots [t_n]\overline{a}$.

Elemente können in einem durch eine URN [119] benannten *Namensraum* (engl. *Namespace*) liegen. Dieser kann explizit durch ein `xmlns`-Attribut oder durch Voranstellen eines *Namensraumpräfixes*, das vorher an einen Namensraum gebunden wurde, angezeigt werden. In dieser Arbeit wird für XML-Elemente, sofern der Namensraum im jeweiligen Zusammenhang

Abbildung 2.1: Schema der XML-Verarbeitung

überhaupt eine Rolle spielt, die zweite Variante gewählt. Die Deklaration der in dieser Arbeit verwendeten Namensräume und Präfixe findet sich im Anhang A.

2.2 XML-Verarbeitung

Neben den oben modellierten XML-Elementen enthalten XML-Bäume und XML-Dokumente weitere Knotentypen (hier nach [36]):

- *Wurzelknoten* (engl. *Root Nodes*)
- *Attributknoten* (engl. *Attribute Nodes*)
- *Namensraumknoten* (engl. *Namespace Nodes*)
- *Knoten für Verarbeitungsinstruktionen* (engl. *Processing Instruction Nodes*)
- *Kommentarknoten* (engl. *Comment Nodes*)

Diese Struktur zusammen mit dem System der Baumdarstellung durch öffnende und schließende Kennungen macht das *Parsen* (siehe Glossar) und Navigieren auf dem XML-Dokument sehr kompliziert.

XML-verarbeitende Anwendungen, zu denen auch Web-Service-Frameworks gehören, operieren daher üblicherweise nicht direkt auf XML-Dokumenten. Stattdessen werden Zwischensysteme eingesetzt, die XML-Dokumente zu abstrakteren Darstellungen aufbereiten und der Anwendung eine komfortabelere *Schnittstelle* zum Zugriff auf die XML-Elemente bereitstellen.

Die Verarbeitung von XML-Dokumenten (auch als *Unmarshalling* bezeichnet) erfolgt typischerweise in vier Schritten [1]:

1. Einlesen des XML-Dokuments von der Netzwerkschnittstelle oder von einem Dateisystem
2. Parsen des XML-Dokuments und Überführung in eine abstraktere Form (abhängig von der jeweiligen Verarbeitungsschnittstelle)
3. Konvertierung der Textknoten in typbehaftete Daten
4. Erzeugung einer Objektstruktur (der jeweiligen Programmiersprache) mit den Werten aus dem XML-Dokument

Die ersten beiden Punkte werden auch als *Deserialisierung* [109], die letzten beiden Punkte auch als *Bindung an eine Datenstruktur* (engl. *Data Binding*) [112] bezeichnet. Die folgenden Betrachtungen richten sich primär auf die Schnittstelle zwischen dem Parser und den dahinter liegenden Verarbeitungsinstanzen.

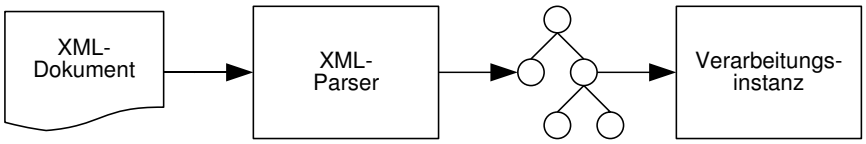


Abbildung 2.2: Baumbasierte Verarbeitung

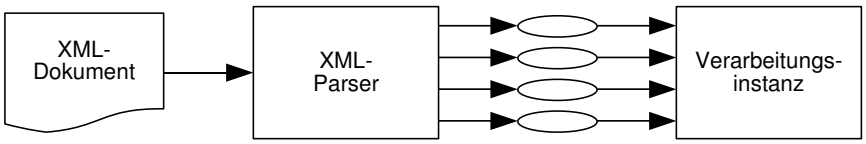


Abbildung 2.3: Ereignisbasierte Verarbeitung

Die beiden wichtigsten Ansätze für diese Schnittstelle sind:

- Baumbasierte Verarbeitung (engl. *Tree-based Processing*)
- Ereignisbasierte Verarbeitung (engl. *Event-based Processing*)

Baumbasierte Verarbeitung

Bei der *baumbasierten* Verarbeitung liest der Parser das komplette XML-Dokument ein und konstruiert eine baumartige Speicherrepräsentation der Informationen aus dem XML-Dokument. Dieser Baum wird dann der nachfolgenden Verarbeitungsinstanz übergeben. Da XML-Dokumente Informationen als Baumstruktur repräsentieren, ist diese Art des Interfaces die *„kanonische Schnittstelle“* zur XML-Verarbeitung [127]. Die bekannteste Realisierung der baumbasierten Verarbeitung ist das *Document Object Model* (DOM) [82].

Der große Vorteil der Baumrepräsentation ist die Möglichkeit des direkten Zugriffs auf alle Dokumentteile und die Unterstützung aller Navigationsoperationen des XPath-Modells. Ein wesentlicher Nachteil dieses Verarbeitungsmodells ist die Tatsache, dass das XML-Dokument vollständig eingelesen, geparst und in den Baum abgebildet werden muss, bevor eine Weiterverarbeitung des XML-Dokuments erfolgen kann.

Einige Implementierungen des baumbasierten Verarbeitungskonzeptes beherrschen die *verzögerte Baumkonstruktion* (engl. *Deferred Tree Building*). Dabei werden nur Teile des XML-Dokuments (z. B. nur der Wurzelknoten mit seinen Kindern) in einen Baum abgebildet. Die übrigen Teile werden bei Bedarf, d. h. sobald ein Zugriff erfolgt, nachträglich in Baumknoten umgewandelt. Dies wird beispielsweise von Apache Xerces unterstützt [131, 148].

Weil das Ende von Teilbäumen in XML-Dokumenten nur durch die Endkennung markiert ist, ist es nicht möglich, Teilbäume des Dokuments zu überspringen [127]. Daher ist es auch bei dieser Variante notwendig, das XML-Dokument vollständig einzulesen und zu parsen, bevor die Weiterverarbeitung beginnen kann. Eine echte partielle baumbasierte Verarbeitung eines XML-Dokuments ist nur möglich, indem in einem Vorverarbeitungsschritt Teile des Dokuments entfernt werden [84].

Ereignisbasierte Verarbeitung

Bei der *ereignisbasierten* oder *strombasierten* Verarbeitung werden die Informationen aus dem XML-Dokument in Datenstrukturen überführt, die Teile des Dokuments repräsentieren (typischerweise einen einzelnen Knoten). Dabei werden im *Callback*-Verfahren vom Parser bestimmte Funktionen in der Verarbeitungsinstanz aufgerufen und die eingelesenen Werte übergeben. Der Parser übergibt das XML-Dokument also als Folge von Ereignissen (engl. *Events*), die von der Verarbeitungsinstanz weiterverarbeitet werden. Die wichtigste Realisierung des ereignisbasierten Verarbeitungskonzeptes ist das *Simple API for XML* (SAX) [149, 29]. Darin werden u. a. Ereignisse für den Beginn eines XML-Elements, für Textknoten und für das Ende eines XML-Elements definiert.

Eine Abwandlung dieser Verarbeitungslogik stellen *pull-basierte* Verfahren dar. Dabei wird das XML-Dokument ebenfalls in kleineren Teilen weitergegeben. Der wesentliche Unterschied zu ereignisbasierten Verfahren ist, dass die Weiterverarbeitung der Dokumentteile nicht vom Parser ausgelöst wird (*Push*- oder *Callback*-Verfahren), sondern diese bei Bedarf von der Verarbeitungsinstanz abgeholt werden. Diese Verarbeitungsmethodik wird beispielsweise vom *Streaming API for XML* (StAX) [147] verwendet.

Abgesehen von der unterschiedlichen Auslöserichtung haben beide Verfahren ähnliche Eigenschaften. So existiert keine Repräsentation des Gesamtdokuments im Speicher des verarbeitenden Systems. Auch können die Dokumentteile nur in genau der Reihenfolge verarbeitet werden, wie sie im XML-Dokument vorkommen. Weiterhin gibt es keine Möglichkeit, einen Dokumentteil ein zweites Mal vom Parser zu erhalten. Daher muss die Verarbeitungsinstanz alle Informationen, die zu einem späteren Zeitpunkt noch benötigt werden, selbst explizit speichern. Damit ist eine Navigation auf dem XML-Baum auch nur durch zusätzlichen Aufwand möglich.

Den genannten Nachteilen gegenüber stehen zwei gewichtige Vorteile dieser Verarbeitungstechniken. Zum einen ist der Speicherverbrauch auf die Elemente beschränkt, die von der Verarbeitungsinstanz explizit gespeichert werden. Zum anderen erfolgt die Verarbeitung des XML-Dokuments durch die Feingranularität der Ereignisse beinahe synchron zum Einlesen des Dokuments. Dies ist insbesondere dann von Vorteil, wenn nicht das gesamte Dokument verarbeitet wird, weil beispielsweise die Verarbeitung bei Erkennung eines ungültigen Dokuments abgebrochen wird. Wenn das Dokument über eine Netzwerkschnittstelle eingelesen wird, kann dies sogar eintreten, bevor das Dokumentende vom Kommunikationspartner verschickt wurde.

Misch- und Sonderformen

Es gibt es auch Systeme, die beide Verarbeitungsparadigmen miteinander vereinigen. Ein Beispiel dafür ist das *Axis Object Model* (AXIOM). Hierbei wird die Deserialisierung des Dokuments StAX-basiert durchgeführt. AXIOM erlaubt dabei einen direkten Zugriff auf die StAX-Schnittstelle. Zusätzlich wird zum Zugriff auf das Dokument auch eine baumartige Schnittstelle angeboten [6].

Weiter gibt es natürlich auch Modelle, die vom Standardvorgehen (wie in Abbildung 2.1 illustriert) abweichen. So wird in [99] ein System präsentiert, bei dem das Erzeugen der Objekte ohne Zwischenrepräsentation direkt aus dem XML-Dokument erfolgt.

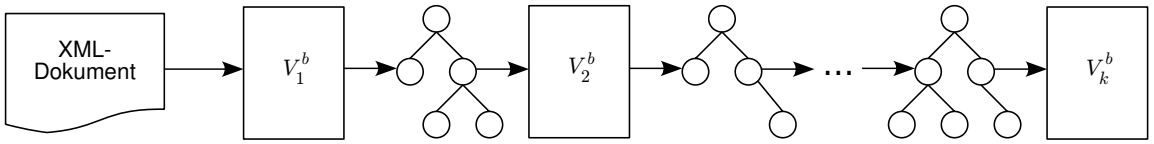


Abbildung 2.4: Verarbeitungskette für baumbasierte Schnittstelle

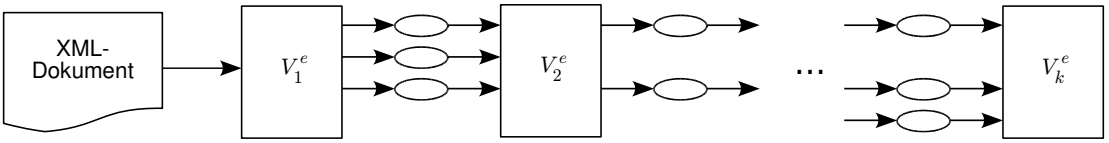


Abbildung 2.5: Verarbeitungskette für ereignisbasierte Schnittstelle

2.3 Modellierung der XML-Verarbeitung

Wie in Abschnitt 4.2.1 erläutert, ist eine wichtige Methode zur Bewertung der Risiken für die Verfügbarkeit von Diensten die Bestimmung des Ressourcenverbrauchs bestimmter Aktionen (z. B. eines Protokollablaufes). Im Folgenden wird eine einfache Ressourcenverbrauchsmetrik für die beiden Hauptmodelle (baum- und ereignisbasiert) der XML-Verarbeitung vorgestellt.

2.3.1 Verarbeitungsmodell

Die für dieses Modell angenommene Verarbeitungsarchitektur ist eine Kette von Verarbeitungsinstanzen V_1^m bis V_k^m, wobei V_1^m immer der XML-Parser ist und $m \in \{b, e\}$ das Verarbeitungsmodell (b = baumbasiert, e = ereignisbasiert) kennzeichnet. Abbildung 2.4 zeigt eine solche Kette für ein baumbasiertes Verarbeitungsmodell, Abbildung 2.5 entsprechend für ein ereignisbasiertes. Eine solche Kette ist typisch für viele Anwendungen (siehe z. B. in [6]).

Instanzen können Informationen ausfiltern oder hinzufügen. Beim baumbasierten Verarbeitungsmodell geschieht dies durch Modifikation des XML-Baums. Beim ereignisbasierten Verarbeitungsmodell bedeutet das, dass Ereignisse herausgefiltert werden bzw. neue Ereignisse generiert werden.

Sei Σ eine Menge von XML-Elementen und T die Menge aller Bäume mit Knoten aus Σ. Dann sei für alle $j \in \{1, \ldots, k\}$

$$V_j^b : T \to T$$

und

$$V_j^e : \Sigma \to \Sigma$$

die Verarbeitung in der Instanz j.

Die Gesamtverarbeitung der Kette ist dann

$$V^b : T \to T, t \mapsto V_k^b(\ldots(V_1^b(t))\ldots)$$

bzw.

$$V^e : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma),$$
$$\{s_1, ..., s_n\} \mapsto \left\{V_k^e(\ldots(V_1^e(s_1))\ldots)\ ,\ \ldots\ ,\ V_k^e(\ldots(V_1^e(s_n))\ldots)\right\}.$$

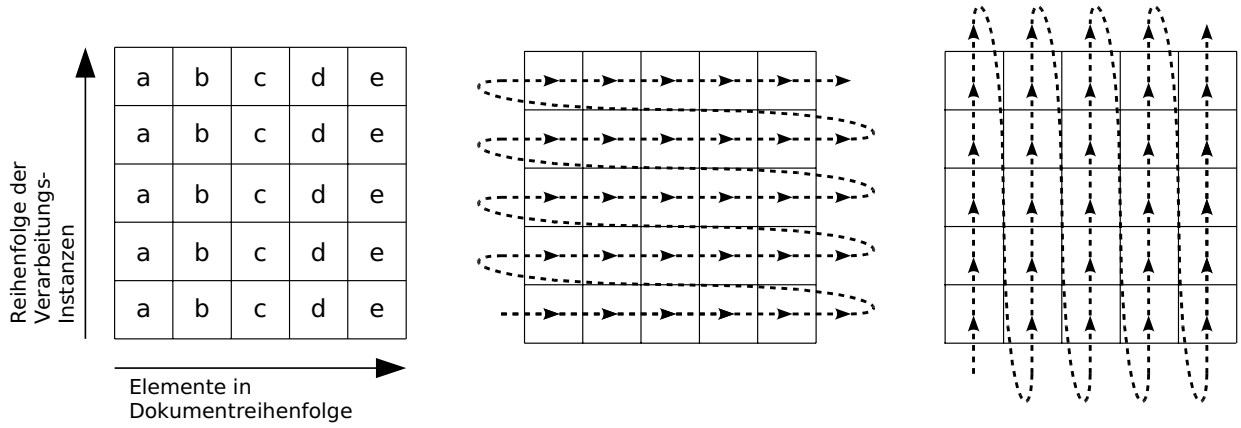


Abbildung 2.6: Graphische Darstellung der Verarbeitungsreihenfolge für baumbasierte (mitte) und ereignisbasierte (rechts) Verarbeitung

Zur besseren Lesbarkeit bezeichne „$\mid$“ die Konkatenation zweier Verarbeitungsoperationen. Es gelte also:

$$\big(V_1(x) \mid V_2(y)\big) = z \quad :\Longleftrightarrow \quad V_1(x) = y \wedge V_2(y) = z.$$

Wird beispielsweise das XML-Dokument aus 2.1 von einem Parser und einer weiteren Instanz verarbeitet, so ergibt sich bei einer baumbasierten Verarbeitung folgende Operationsfolge:

$$V_1^b(a\,b\,\bar{b}\,c\,d\,\bar{d}\,e\,\bar{e}\,\bar{c}\,\bar{a}) \mid V_2^b(a\,b\,\bar{b}\,c\,d\,\bar{d}\,e\,\bar{e}\,\bar{c}\,\bar{a}).$$

Für eine ereignisbasierte Verarbeitung ergibt sich diese Operationsfolge:

$$\begin{aligned} V_1^e(a) \mid V_2^e(a) \mid V_1^e(b) \mid V_2^e(b) \mid V_1^e(\bar{b}) \mid V_2^e(\bar{b}) \mid V_1^e(c) \\ \mid V_2^e(c) \mid V_1^e(d) \mid V_2^e(d) \mid V_1^e(\bar{d}) \mid V_2^e(\bar{d}) \mid V_1^e(e) \mid V_2^e(e) \\ \mid V_1^e(\bar{e}) \mid V_2^e(\bar{e}) \mid V_1^e(\bar{c}) \mid V_2^e(\bar{c}) \mid V_1^e(\bar{a}) \mid V_2^e(\bar{a}). \end{aligned}$$

Graphisch kann man die Reihenfolge der Verarbeitung wie in Abbildung 2.6 darstellen. Jedes Kästchen stellt die Verarbeitung eines XML-Knotens in einer Verarbeitungsinstanz dar. Bei der baumbasierten Verarbeitung wird von jeder Instanz das komplette Dokument verarbeitet, bevor die nächste Instanz startet („horizontale Abarbeitung“, in der Abbildung mitte). Bei der ereignisbasierten Verarbeitung wird ein Element von allen Instanzen verarbeitet, bevor das nächste Element bearbeitet wird („vertikale Abarbeitung“, in der Abbildung rechts).

2.3.2 Ressourcenverbrauchsmodell

Die für die XML-Verarbeitung relevanten Ressourcen sind Speicher und Prozessor. Dabei haben praktische Erfahrungen ergeben, dass der Speicherverbrauch meistens die kritischere Ressource darstellt.

Für die weiteren Betrachtungen wird angenommen, dass in den Verarbeitungsinstanzen keinerlei Filterungen oder Anreicherungen der XML-Informationen erfolgt.

Speicherverbrauchsmodell

Der tatsächliche Speicherverbrauch eines XML-Baums setzt sich aus einem konstanten Anteil, einem linear von der Größe der Textknoten abhängigen Teil und einem linear von der Anzahl n der Knoten abhängigen Teil zusammen. Da für große n nur der letzte Teil relevant ist, liegt der Speicherverbrauch also in $\mathcal{O}(n)$. Für das im Folgenden verwendete Modell wird daher der Speicherverbrauch eines XML-Baums vereinfacht wie folgt angenommen:

$$mem^b(t) = |t| =: n.$$

Wird in der Verarbeitungskette aus Abbildung 2.4 in den Verarbeitungsinstanzen kein zusätzlicher Speicher verbraucht, so beträgt der Gesamtspeicherbedarf also n.

Bei ereignisbasierter Verarbeitung wird zu jedem Zeitpunkt nur ein einzelnes Element im Speicher gehalten, der Speicherbedarf ist also unabhängig von der Größe des Dokuments $\mathcal{O}(1)$ und wird hier als 1 angenommen:

$$mem^e(x) = 1, \text{ für alle } x \in \Sigma.$$

Wird in der letzten Verarbeitungsinstanz das Dokument wieder serialisiert und zwischengespeichert,[1] so beträgt der Gesamtspeicherbedarf auch hier n.

Zu beachten ist, dass dieses Modell eine starke Vereinfachung des Speicherverbrauchs tatsächlicher Implementierungen ist und nur eine untere Schranke für den theoretisch notwendigen Speicherbedarf abbildet. Insbesondere DOM-basierte Implementierungen benötigen deutlich mehr Speicher. So verbraucht bei Apache Xerces bereits ein leerer Baum 80 KByte Speicher. Ein Dokument mit 1000 ineinander geschachtelten Elementen hat 230 KByte Größe, während die entsprechende serialisierte Form nur knapp 10 KByte benötigt.

Der Unterschied zwischen der baum- und der ereignisbasierten Verarbeitung wird deutlich, falls das XML-Dokument nicht vollständig verarbeitet wird. Wird die Gesamtverarbeitung bei einem Element beendet, das in der Dokumentreihenfolge an Position i (mit $i \in \{1, \ldots, n\}$) steht, so beträgt der Speicherverbrauch bei der baumbasierten Verarbeitung immer noch n, bei der ereignisbasierten Verarbeitung aber nur i. Damit gilt folgendes Lemma:

Lemma 1 (Speicherbedarf bei XML-Verarbeitung) *Baumbasierte XML-Verarbeitung benötigt immer mindestens soviel Speicher wie ereignisbasierte Verarbeitung, d. h. für alle $t \in T$ gilt:*

$$mem^b(t) \geq mem^e(t)$$

Bei reiner baumbasierter Verarbeitung ergibt sich aber noch ein weiteres Problem bzgl. des Speicherverbrauchs. Sofern der XML-Parser nicht schon die Verarbeitung abbricht (was nur für nicht wohlgeformte XML-Dokumente der Fall ist), ist der Speicherverbrauch in jedem Fall n und damit vom Erzeuger der Nachricht bestimmbar. Falls der Erzeuger ein Angreifer ist, bietet dies eine Möglichkeit für einen Speicherverbrauchsangriff.

Lemma 2 (Angriff gegen baumbasierte Verarbeitung I) *Bei einer reinen baumbasierten Verarbeitung kann ein Angreifer (mit entsprechend großen Dokumenten) einen beliebig großen Speicherverbrauch auslösen.*

[1] Dies ist für die Verarbeitung, wie sie in späteren Kapiteln postuliert wird, notwendig.

Bei älteren Implementierungen war dies auch möglich bzw. der tatsächliche Speicherverbrauch war nur durch die Speichergröße des angegriffenen Systems bzw. der virtuellen Maschine beschränkt (siehe [70] und Kapitel 5.2.1).

Da die Speichermenge nur von der Dokumentposition k abhängt, bei der die Verarbeitung beendet wird, ergibt sich folgende Grundregel für das ereignisbasierte Finden von ungültigen Nachrichten, die potentiell Teil eines DoS-Angriffs sind:

> *Die Nachrichtenverarbeitung sollte so erfolgen, dass bei ungültigen Nachrichten die Verarbeitung* ***möglichst früh*** *(bezogen auf die Dokumentreihenfolge) abgebrochen wird.*

Prozessornutzung

Für die Prozessornutzung gelten kompliziertere Zusammenhänge als für die Speichernutzung. Im Idealfall liegt die Prozessornutzung einer Verarbeitungskomponente aber in $\mathcal{O}(n)$, wobei $n := |t|$ die Anzahl der Knoten des XML-Baums ist.[2] Vereinfacht ist also die Prozessornutzung für eine Komponente $j \in \{1, \ldots, k\}$ sowohl für baumbasierte als auch ereignisbasierte Verarbeitung[3]:

$$cpu^b(t, j) = n \qquad cpu^e(t, j) = n.$$

Damit ist die Prozessornutzung für die vollständige Gesamtverarbeitung:

$$cpu^b(t) = n \cdot k \qquad cpu^e(t) = n \cdot k.$$

Unterschiede in der Prozessornutzung zeigen sich für den Fall, dass die Verarbeitung nicht vollständig durchgeführt wird (weil beispielsweise vorher ein Fehler im Dokument gefunden wird). Wird die Verarbeitung von Komponente $j \in \{1, \ldots, k\}$ bei dem Knoten abgebrochen, der in der Dokumentreihenfolge an Position $i \in \{1, \ldots, n\}$ steht, so beträgt die jeweilige Prozessornutzung:

$$cpu^b(t) = n \cdot (j - 1) + i \qquad cpu^e(t) = k \cdot (i - 1) + j.$$

Das bedeudet, dass die Prozessornutzung bei der baumbasierten Verarbeitung auch im Abbruchsfall von der Gesamtgröße des Dokuments abhängt (Ausnahme: Abbruch bei der ersten Komponente). Damit ergibt sich folgende Aussage:

Lemma 3 (Angriff gegen baumbasierte Verarbeitung II) *Bei einer reinen baumbasierten Verarbeitung kann ein Angreifer (mit entsprechend großen Dokumenten) eine beliebig große Laufzeit der Verarbeitung auslösen.*

Weiterhin lässt sich für ereignisbasierte Verarbeitung ablesen, dass die Laufzeit im Abbruchsfall stärker von der Position i im Dokument abhängt als von der Position j der Komponente. Dies bedeutet, dass die Anordnung der Komponenten in der Verarbeitungskette nur geringen Einfluss auf die Laufzeit hat. Diese Eigenschaft erlaubt es, die Komponenten rein nach funktionalen Gesichtspunkten zu ordnen, ohne die Laufzeit wesentlich zu erhöhen.

---

[2] Die später in dieser Arbeit vorgestellten Komponenten erfüllen diese Bedingung.

[3] Die Einheit für die Prozessornutzung ist für diese abstrakte Betrachtung nicht relevant. In der Praxis könnte diese beispielsweise Anteil an der Gesamtprozessorleistung, Anzahl der Prozessorzyklen, Laufzeit in ms (bei 100% Prozessorauslastung) o. ä. sein.

Kapitel 3

Web Services

In diesem Kapitel werden die grundlegenden Spezifikationen aus dem Umfeld von Web Services vorgestellt, die für das Verständnis dieser Arbeit notwendig sind.

3.1 WSDL

Die *Web Service Description Language* (WSDL) [34] erlaubt die formale Spezifikation der Schnittstelle eines Web Services. Jeder Web Service bietet dafür eine in der WSDL geschriebene *Web-Service-Beschreibung* (engl. *Web Service Description*) an, in der der Dienst in maschinenlesbarer Form beschrieben wird. Diese Beschreibung enthält (vereinfacht):

- die Netzwerkadresse des Dienstes
- die geforderte Nachrichten-Kodierung und das Transportprotokoll
- die angebotenen Operationen des Dienstes
- die Datentypen der Ein- und Ausgabeparameter der Operationen

Die beiden letztgenannten Teile beschreiben den Dienst *abstrakt*, wie es auch der Schnittstelle (engl. *Interface*) einer Klasse in einer objektorientierten Programmiersprache entspricht. Die ersten beiden Teile beschreiben die *konkrete* Realisierung des Dienstes über das Netzwerk.

Die Nachrichten, die von einer Web-Service-Beschreibung definiert werden, sind folgendermaßen hierarchisch angeordnet: ein *Dienst* (engl. *Service*, definiert durch `wsdl:service`) enthält einen oder mehrere *Endpunkte* (engl. *Endpoint*, definiert durch `wsdl:port`, `wsdl:binding`, `wsdl:portType`), diese enthalten eine oder mehrere *Operationen* (engl. *Operation*, definiert durch `wsdl:operation`). Eine Operation enthält ihrerseits eine *Eingangsnachricht* (engl. *Input Message*, definiert durch `wsdl:input`), ferner optional eine *Ausgangsnachricht* (engl. *Output Message*, definiert durch `wsdl:output`) und eine *Fehlernachricht* (engl. *Fault Message*, definiert durch `wsdl:fault`).

Die vom WS-I empfohlene Nachrichten-Kodierung ist SOAP ([9], R2401), transportiert mittels des HTTP-Protokolls ([9], R2702).

3.2 SOAP

SOAP definiert ein XML-basiertes Nachrichtenformat zum Transport von XML-Fragmenten. Die beiden aktuellen Versionen sind 1.1 [21] (dort ursprünglich als *Simple Object Access Protocol*

bezeichnet) und 1.2 [72]. Beide Versionen unterscheiden sich nur geringfügig. In dieser Arbeit wird nur dort explizit auf die SOAP-Version eingegangen, wo es relevante Unterschiede zwischen den beiden Versionen gibt.

Eine SOAP-Nachricht, auch SOAP-Umschlag (engl. *SOAP Envelope*) genannt, enthält zwei Teile: den SOAP-Header und den SOAP-Body.

Der SOAP-Header enthält die sogenannten *nicht-funktionalen* Bestandteile der Nachricht. Dies sind zusätzliche Informationen, die nicht für den Web-Service-Aufruf im engeren Sinne notwendig sind. Beispiele dafür sind Routing-Informationen, Transaktionsdaten, Signaturen und kryptographische Schlüssel. Der SOAP-Header kann sowohl an SOAP-Zwischenknoten (engl. *SOAP Intermediaries*) als auch an den SOAP-Endpunkt (engl. *Ultimate Receiver*) gerichtet sein.

Der SOAP-Body enthält die sogenannten *funktionalen* Bestandteile der Nachricht, die Nutzdaten des Web-Service-Aufrufs. Dies ist typischerweise eine Operation mit ihren Parametern. Der SOAP-Body sollte nur am SOAP-Endpunkt verarbeitet werden.

Das folgende Beispiel zeigt eine einfache SOAP-Nachricht mit Routing-Informationen gemäß WS-Addressing [73] im Header.

```
<env:Envelope>
  <env:Header>
    <wsa:To>http://example.org/user</wsa:To>
    <wsa:Action>http://example.org/operation</wsa:Action>
  </env:Header>
  <env:Body>
    <ns1:operation>
      <param1>12</param1>
      <param2>23</param2>
    </ns1:operation>
  </env:Body>
</env:Envelope>
```

Die Bindung an das HTTP-Protokoll [57] hat einige Auswirkungen auf die Benutzung von SOAP, die für die Verarbeitung in Web-Service-Servern relevant sind.

Web-Service-Endpunkt

Jeder Web-Service-Endpunkt besitzt (definiert durch `/wsdl:definitions/wsdl:service/wsdl:port/soap:address/@location`) eine eindeutige Netzwerkadresse in Form einer HTTP-URL [13]. Dies bedeutet, dass eine eingehende SOAP-Nachricht auf Grundlage des HTTP-Headers einem Web-Service-Endpunkt zugeordnet werden kann.

HTTP-Header `SOAPAction`

Die Web-Service-Beschreibung definiert (bei Verwendung von SOAP mit HTTP-Bindung) für jede Operation einen Bezeichner, der beim Aufruf dieser Operation im HTTP-Request-Header als Wert des Felds `SOAPAction` übergeben wird. Dieser soll HTTP-basierten Systemen wie z. B. Firewalls erlauben, die Web-Service-Operation zu bestimmen, ohne die SOAP-Nachricht zu analysieren.

Abbildung der Web-Service-Kommunikationsmuster

Die beiden vom WS-I empfohlenen Kommunikationsmuster für Web Services *Request-Reponse* und *One-Way* werden wie folgt in das HTTP-Protokoll eingebunden.

Beim Request-Reponse-Muster wird die SOAP-Request-Nachricht als Nutzlast der HTTP-Request-Nachricht verschickt. Die HTTP-URL und das `SOAPAction`-HTTP-Feld werden wie oben beschrieben festgelegt. Die SOAP-Response-Nachricht wird dann als Nutzlast der dazugehörigen HTTP-Response-Nachricht zurückgeschickt. Der HTTP-Status-Code ist dabei „200“ für Ausgangsnachrichten und „500“ für Fehlernachrichten.

Beim One-Way-Muster wird die SOAP-Request-Nachricht wie oben beschrieben verschickt. Die HTTP-Request-Nachricht wird mit einer HTTP-Response-Nachricht ohne Nutzlast und mit dem HTTP-Status-Code „202“ beantwortet.

3.3 BPEL

BPEL ist ursprünglich als Standard zur Modellierung und Definition von Geschäftsprozessen entwickelt worden [153]. Ein solcher BPEL-Geschäftsprozess kann dabei eine Vielzahl von externen Web Services aufrufen und die aggregierte Funktionalität wiederum als Web Service anbieten. Aus diesem Grund wird BPEL heutzutage auch primär zur Komposition von Web Services verwendet. BPEL erlaubt die Definition von komplexen Prozessabläufen mit Parallelverarbeitung, Schleifen, bedingter Ausführung, Wartekonstrukten, Prozessvariablen, Fehlerbehandlung usw. und wird als die „vollständigste Ablaufbeschreibungsprache“ (engl. *„most complete workflow description language“*) [39] angesehen.

BPEL dient nicht nur zur abstrakten Definition von Geschäftsprozessen, sondern erlaubt auch die automatische Instanzierung und Ausführung des Prozesses. Dazu wird das BPEL-Dokument auf eine *BPEL-Ablaufumgebung* (engl. *BPEL Engine*) geladen. Diese erzeugt die definierten Web-Service-Endpunkte, verarbeitet einkommende SOAP-Nachrichten, verfolgt den Ablauffluss und ruft externe Web Services auf.

Ein BPEL-Dokument definiert Metadaten des Prozesses (z. B. Prozessvariablen) und sogenannte *Aktivitäten*. Die *Strukturaktivitäten* bilden eine Baumstruktur zur Definition der Ausführungsreihenfolge der *Basisaktivitäten*. Die wichtigsten Basisaktivitäten sind die *Kommunikationsaktivitäten*, die eingehende und ausgehende Web-Service-Aufrufe repräsentieren. In Rahmen dieser Arbeit werden die folgenden (wichtigsten) BPEL-Aktivitäten betrachtet.

`receive` definiert eine eingehende Nachricht und kann die BPEL-Ablaufumgebung zur Erzeugung eines Web-Service-Endpunkts veranlassen. In Attributen des `receive`-Elements kann festgelegt werden, von welchem Partner der Aufruf erwartet wird. Außerdem wird hier der WSDL-Port-Typ (engl. *Port Type*) und die WSDL-Operation der eingehenden SOAP-Nachricht festgelegt. Weiterhin kann spezifiziert werden, in welche Prozessvariablen die eingehenden Daten geschrieben werden. Schließlich legt das Attribut `createInstance` fest, ob durch den Aufruf dieser Operation eine neue Prozessinstanz erzeugt wird.

`reply` ist das Gegenstück zu `receive` und stellt eine ausgehende Nachricht als Antwort auf eine in einem `receive` eingegangene Nachricht dar. Das `receive`-`reply`-Paar stellt damit die Abstraktion des *Request-Response*-Musters aus WSDL und wird dementsprechend von der BPEL-Ablaufumgebung realisiert. Auch hier können Partner, Port-Typ, Operation und zugehörige Prozessvariablen festgelegt werden. Daten aus diesen Variablen

werden in die ausgehende Nachricht übernommen. Dabei müssen alle Werte außer den Prozessvariablen mit denen des dazugehörigen `receive` übereinstimmen.

`invoke` repräsentiert eine ausgehende Nachricht. Bei Erreichen eines solchen Konstrukts wird ein externer Web Service aufgerufen. Ebenso wie bei den beiden obigen Kommunikationsaktivitäten können Partner, Nachrichtenformat und Prozessvariablen festgelegt werden.

`pick` ist sowohl eine Kommunikations- als auch eine Strukturaktivität. Es enthält ein oder mehrere `onMessage`-Elemente, die wie eine `receive`-Aktivität funktionieren, und ein oder mehrere `onAlarm`-Elemente, die einen absoluten oder relativen Zeitwert enthalten. Es wird genau eines dieser Kindelemente ausgeführt, nämlich dasjeninge, dessen Ereignis (d. h. einkommende Nachricht oder Ablauf der Zeitspanne) als erstes erfüllt ist.

`sequence` definiert die Ausführung der Kindaktivitäten in genau der Dokumentreihenfolge.

`while` definiert eine Schleife. Die Kindaktivität wird solange ausgeführt bis eine angegebene Bedingung erfüllt ist.

`switch` enthält eine bedingungsgesteuerte Ablaufsteuerung. Jede Kindaktivität enthält in einem `case`-Element eine Bedingung. Die Kindelemente werden in der Dokumentreihenfolge ausgewertet und das erste, dessen Bedingung erfüllt ist, wird ausgeführt. Alle anderen Kindaktivitäten werden nicht ausgeführt.

`assign` enthält Anweisungen zur Zuweisung von Variablen. Als Wertequelle können dabei Konstanten oder der Inhalt bzw. Teilinhalt anderer Variablen dienen.

Jedem BPEL-Prozess wird ein eindeutiger WSDL-Endpunkt zugeordnet. Damit kann der BPEL-Prozess anhand der HTTP-URL einer Nachricht erkannt werden. Die Operationen der Kommunikationsaktivitäten müssen eindeutig durch die WSDL-Operation identifizierbar sein, also beispielsweise durch das `SOAPAction`-Feld im HTTP-Header.

Die Zuordnung zu einer Prozessinstanz erfolgt mittels der *Korrelationsgruppen* (engl. *Correlation Sets*), die als Metadaten in einer Prozessbeschreibung definiert werden. Diese enthalten eine Menge von *Korrelationseigenschaften* (engl. *Correlation Set Properties*), die Verweise auf Werte in Nachrichten darstellen[1] und mit Kommunikationsaktivitäten verbunden werden. Für den entsprechenden Kommunikationsvorgang dienen diese Werte dann als Identifikator für eine Prozessinstanz. Eine eingehende Nachricht m mit der Operation o wird also genau der Prozessinstanz i zugeordnet, für die gilt: für alle Korrelationseigenschaften c der Korrelationsgruppe, die o zugeordnet ist, gilt: der Wert von c in m ist gleich dem Wert von c in i.

[1] Genauer: auf Schema-Elemente in einer der zugehörigen Web-Service-Beschreibungen.

```
<xenc:EncryptedData @Id>
  <xenc:EncryptionMethod @Algorithm/>
  <ds:KeyInfo>
    ...
  </ds:KeyInfo>?
  <xenc:CipherData>
    <xenc:CipherValue>?
    <xenc:CipherReference @URI?>?
  </xenc:CipherData>
</xenc:EncryptedData>
```

```
<xenc:EncryptedKey @Id>
  <xenc:EncryptionMethod @Algorithm/>
  <ds:KeyInfo>
    ...
  </ds:KeyInfo>?
  <xenc:CipherData>
    <xenc:CipherValue>?
    <xenc:CipherReference @URI?>?
  </xenc:CipherData>
  <xenc:ReferenceList>
   <xenc:DataReference @URI?>?
   <xenc:KeyReference @URI?>?
  </xenc:ReferenceList>
</xenc:EncryptedData>
```

Abbildung 3.1: Schema für den Aufbau eines verschlüsselten XML-Fragments

3.4 WS-Security

Wie bereits in der Einleitung beschrieben, definiert WS-Security zur Sicherung von SOAP-Nachrichten folgende Sicherheitsmechanismen:

- Verschlüsselung von SOAP-Fragmenten
- Signierung von SOAP-Fragmenten
- Sicherheits-Token
- Zeitstempel

Diese Mechanismen werden nachstehend detailliert erläutert.

Verschlüsselung

Zur Sicherung der Vertraulichkeit sensibler Daten können Teile der SOAP-Nachricht verschlüsselt werden. Dazu werden die Mechanismen aus dem XML-Encryption-Standard [86] mit einigen kleineren Modifikationen verwendet. Verschlüsselt werden können (fast) beliebige Fragmente aus dem SOAP-Header oder dem SOAP-Body[2]. Das entsprechende Fragment wird serialisiert und kryptographisch verschlüsselt. Der Chiffretext wird in *Base64-codierter* (siehe Glossar) [58] Form in ein `xenc:EncryptedData`-Element eingefügt (siehe auch Abbildung 3.1). Dieses Element wird im ursprünglichen Dokument an Stelle des verschlüsselten Elements eingefügt.

XML-Encryption erlaubt auch die Realisierung von *hybrider* Verschlüsselung. Dabei wird das XML-Fragment mit einem zufällig generierten (symmetrischen) Schlüssel verschlüsselt, der wiederum in (asymetrisch) verschlüsselter Form in die SOAP-Nachricht eingefügt wird. Der verschlüsselte Schlüssel wird in einem besonderen XML-Encryption-Element,

[2]XML-Encryption erlaubt neben der Verschlüsselung eines Elements auch das Verschlüsseln des *Inhalts* eines Elements. Dabei werden alle Kindknoten verschlüsselt, das Element selbst bleibt erhalten. Dieses wird im Folgenden nicht weiter betrachtet.

```
<ds:Signature>
   <ds:SignedInfo>
      <ds:CanonicalizationMethod @Algorithm/>
      <ds:SignatureMethod @Algorithm/>
      <ds:Reference @URI>
         <ds:Transforms>...</Transforms>?
         <ds:DigestMethod @Algorithm/>
         <ds:DigestValue>...</DigestValue>
      </ds:Reference> +
   </ds:SignedInfo>
   <ds:SignatureValue>...</SignatureValue>
   <ds:KeyInfo>...</KeyInfo>?
</ds:Signature>
```

Abbildung 3.2: Schema für den Aufbau einer Signatur gemäß XML Signature

dem `xenc:EncryptedKey` abgelegt und in den SOAP-Header eingefügt. Die Struktur des `xenc:EncryptedKey`-Elements ist weitgehend identisch mit der des `xenc:EncryptedData`-Elements, enthält aber zusätzlich ein `xenc:ReferenceList`-Element zur Referenzierung des verschlüsselten Blocks, der diesen Schlüssel verwendet.

Signierung

Zur Sicherung der Integrität und zum Nachweis der Authentizität können Teile einer SOAP-Nachricht digital signiert werden. Dazu werden in WS-Security die Mechanismen von XML Signature [10] verwendet.

Abbildung 3.2 zeigt das Schema einer Signatur gemäß XML Signature. Die Erstellung einer solchen Signatur funktioniert wie folgt: Jedes Fragment der SOAP-Nachricht, das von der Signatur umfasst werden soll, wird zunächst *normalisiert* (engl. *canonicalized*) [22]. Aus diesem normalisiertem XML-Fragment wird mittels einer *Hash-Funktion* der sogenannte *Digest* berechnet. Dieser wird Base64-codiert in das `ds:DigestValue`-Element des `ds:Reference`-Elements eingefügt. Das `ds:SignedInfo`-Element, das alle `ds:Reference`-Elemente umschließt, wird normalisiert und signiert. Eine Besonderheit bei der Anwendung von XML Signature auf SOAP-Nachricht ist, dass das `ds:Signature`-Element in den `wsse:Security`-Header eingefügt werden muss.

Das `ds:KeyInfo`-Element enthält Hinweise auf den kryptographischen Schlüssel, der bei der Signierung benutzt wird.

Sicherheits-Token

Zum Transport von digitalen Identitäten und kryptographischen Schlüsseln definiert WS-Security *Sicherheits-Token*: `wsse:UsernameToken` und `wsse:BinarySecurityToken`. Im Zusammenhang mit Signierung und Verschlüsselung ist nur der zweite Typ von Interesse. Das `wsse:BinarySecurityToken` kann verschiedene Arten von Sicherheits-Token transportieren, wie X.509-Zertifikate [83], SAML-Zusicherungen [30] oder Kerberos-Token [132].

Zeitstempel

WS-Security definiert weiterhin einen Mechanismus, um SOAP-Nachrichten mit einem *Zeitstempel* (engl. *Timestamp*) zu versehen. Ein solcher Zeitstempel besitzt einen Erzeugungs- und einen Ablauf-Zeitpunkt und kann beispielsweise verwendet werden, um die *Frische* (engl. *Freshness*) einer Nachricht zu gewährleisten und damit *Wiedereinspiel-Angriffe* (engl. *Replay Attacks*) zu verhindern.

3.5 WS-SecurityPolicy

WS-SecurityPolicy basiert auf WS-Policy [49], einem allgemeinen Framework zur Beschreibung von *nicht-funktionalen Anforderungen* (engl. *non-functional requirements*) von Web Services. Darin können nicht-funktionale Anforderungen und Eigenschaften des Web Services definiert werden. Beispiele dafür sind Sicherheitsanforderungen, Datenschutzerklärungen (engl. *Privacy*), Zugriffsbeschränkungen und Qualitätsaussagen (engl. *Quality of Service*).

Innerhalb einer Policy werden diese Eigenschaften in sogenannten *Zusicherungen* (engl. *Assertions*) deklariert. Die Zusicherungen können mittels der folgenden Operatoren kombiniert werden. Der Operator `wsp:All`[3] fordert eine Erfüllung aller Optionen; dies entspricht einem logischen AND. Der Operator `wsp:ExactlyOne` fordert die Erfüllung genau einer Option; dies entspricht einem logischen XOR.

WS-Policy selbst definiert keinerlei Zusicherungen, diese werden in ergänzenden Spezifikationen (z. B. WS-RM Policy [41]) festgelegt. So definiert WS-SecurityPolicy Zusicherungen, mit denen sich Sicherheits-Anforderungen formulieren lassen. Dies sind im Einzelnen:

1. Integrität (d. h. welche Elemente der SOAP-Nachricht sollen signiert sein): `sp:SignedParts`, `sp:SignedElements`

2. Vertraulichkeit (d. h. welche Elemente der SOAP-Nachricht sollen verschlüsselt sein): `sp:EncryptedParts`, `sp:EncryptedElements`

3. Sicherheits-Token (d. h. welches Sicherheits-Token soll für Verschlüsselung und Signierung verwendet werden): z. B. `sp:X509Token`

4. Algorithmen (d. h. welche Algorithmen sollen für Verschlüsselung und Signierung verwendet werden): `sp:AlgorithmSuite`

5. Layout des `wsse:Security`-Headers: `sp:Layout`

6. Verwendung eines Zeitstempels: `sp:IncludeTimestamp`

7. Reihenfolge von Signierung und Verschlüsselung: z. B. `sp:SignBeforeEncrypting`

8. Integrität für Sicherheits-Token und Signaturen durch zusätzliche Sicherheits-Token: `sp:SignedSupportingTokens`, `sp:EndorsingSupportingTokens`, `sp:SignedEndorsing-SupportingTokens`.

[3] Dieser Operator kann auch als `wsp:Policy` geschrieben werden.

Diese Forderungen verteilen sich auf die Endpunkt-Policy (3., 4., 5., 6., 7., 8.), Operations-Policy (8.) und Nachrichten-Policy (1., 2., 8.). In der Endpunkt-Policy können die o. g. Forderungen entweder für beide Nachrichtenrichtungen (Sender → Empfänger, Empfänger → Sender) separat (`sp:AsymmetricBinding`) oder gemeinsam (`sp:SymmetricBinding`) definiert werden.

Für die Referenzierung von zu schützenden Dokumententeilen gibt es zwei Möglichkeiten. In den `sp:*EncryptedParts`- und `sp:*SignedParts`-Elementen können vordefinierte Bezeicher für den SOAP-Body und für Teile des SOAP-Headers verwendet werden. In den `sp:*Elements`-Elementen können beliebige XPath-Ausdrücke angegeben werden.

Die Forderung der verwendeten Algorithmen erfolgt durch *Algorithmen-Gruppen* (engl. *Algorithm Suites*). Eine solche Gruppe definiert für jede mögliche Verwendung (z. B. „digitale Signatur mit asymetrischem Schlüssel“ oder „Verschlüsselung“) einen kryptographischen Algorithmus.

Die Policy kann eine beliebige Anzahl von Token innerhalb der `sp:*SupportingTokens`-Elemente fordern, aber nur eines innerhalb des (einzigen) `sp:*Binding`-Elements. Die ersteren werden *Unterstützungs-Token*, das zweite wird *Haupttoken* und die damit erstellte Signatur *Hauptsignatur* (engl. *Message Signature*) genannt. Dem Haupttoken kommt eine besondere Rolle zu, da es den Nachrichtenerzeuger identifiziert.

Kapitel 4

Angriffe

4.1 Verfügbarkeit und Denial-of-Service

4.1.1 Historie

Die ersten Arbeiten über Denial-of-Service stammen aus den 80er und 90er Jahren des letzten Jahrhunderts von Gligor und Yu [62, 156] und Millen [116, 115]. Diese definieren DoS im Kontext von *Multi-User*-Betriebssystemen. Dabei wurden als Ressourcen Hardware-Komponenten (z. B. der Prozessor) und als Dienste Betriebssystem-Komponenten, die den Zugriff auf diese Ressourcen ermöglichen (z. B. der Prozess-*Scheduler*), betrachtet. Diese Modelle wurden auf Netzwerke erweitert [63], wobei der Netzwerkdienst einfach als weiterer Benutzer betrachtet wurde, der um die lokalen Ressourcen konkurriert.

Bei diesen klassischen Modellen wird das Denial-of-Service-Problem auf fehlerhafte Belegung der Systemressourcen zurückgeführt. Weiterhin wird davon ausgegangen, dass jeder Dienst eine *Service Sharing Policy*, auch *User Agreement* genannt, besitzt und publiziert. In dieser Policy legt ein Dienst u.a. fest, wieviele (autorisierte) Benutzer diesen Dienst gleichzeitig benutzen dürfen, in welcher Reihenfolge die Operationen aufgerufen werden dürfen und welche *maximale Wartezeit* (engl. *Maximum Waiting Time, MWT*) dieser Dienst in diesem Fall garantiert. Wird die MWT nicht eingehalten, so wird das als Denial-of-Service angesehen. Die Basis der Gegenmaßnahmen ist eine *DoS Protection Base (DPB)*, die z. B. auf einer *Trusted Computing Base (TCB)* aufbaut.

Diesen Modellen liegt die Annahme zugrunde, dass genügend Ressourcen für alle Benutzer zur Verfügung stehen. Damit muss „nur“ die Einhaltung der Vorschriften für die Zuteilung der Ressourcen (engl. *Service Sharing Policies*) erzwungen werden, um die korrekte Belegung der Ressourcen zu garantieren. Das verhindert dann beispielsweise DoS-Situationen wie Überbelegung (d. h. für einen Benutzer stehen keine Ressourcen mehr zur Verfügung) oder Zugriffsverklemmung (d. h. zwei belegte Ressourcen warten gegenseitig auf das Freiwerden der anderen Ressource). Diese Modelle lassen sich aber nur schlecht auf offene verteilte Netze wie das Internet übertragen. Zum einen steht keine globale Kontrollinstanz zur Verfügung, die alle Dienstaufrufe kontrolliert. Zum anderen ist es bei der heutigen Größe des Internets nicht möglich, einen Dienst (inklusive aller von ihm verwendeten Dienste, z. B. wie Netzwerktransport) so auszulegen, dass er für jede denkbare Menge und Kombination von Dienstaufrufen genügend Ressourcen besitzt.

Ein weiteres Problem ist die Beschränkung auf autorisierte Benutzer. Zum einen werden die Benutzer in vielen Netzwerk-Szenarien gar nicht oder nur schwach authentifiziert, womit auch

die Überprüfung ihrer Autorisierung entfällt; zum anderen kann eine starke Authentifizierung, wie Meadows [113] zeigt, selbst wiederum für Denial-of-Service-Angriffe ausgenutzt werden.

4.2 Klassifizierung von Angriffen

Moderne Denial-of-Service-Angriffe lassen sich nach der Art, wie sie ausgeführt werden und wie sie auf das angegriffene System wirken, in verschiedene Klassen einteilen. Diese Einteilung wird in der Literatur vielfach „eindimensional“ durchgeführt, d. h. DoS-Angriffe werden mithilfe nur eines einzigen Kriteriums klassifiziert.

Eine solche Einteilung findet sich beispielsweise in [121]. Dort werden DoS-Angriffe in *Logic Attacks* und *Resource Attacks* unterteilt. Zur ersten Gruppe gehören Angriffe, die Software-Fehler ausnutzen, um einen Service außer Gefecht zu setzen. Zur zweiten Gruppe gehören Angriffe, die bei einem Service versuchen, große Mengen an Ressourcen zu belegen. Diese Einteilung ist zum einen nicht fein genug, zum anderen lassen sich bestimmte Angriffe nicht nur einer der beiden Klassen zuordnen. So gibt es Angriffe, die durch einen Software-Fehler Ressourcen belegen (also zu beiden Gruppen gehören). Weiterhin kann ein Denial-of-Service auch ohne Ressourcenbelegung oder Softwarefehler erzeugt werden (beispielsweise durch Eindringen in ein System und Abschalten eines Dienstes).

In [138] werden DoS-Angriffe in die Klassen *Ressourcenzerstörung* und *Ressourcenerschöpfung* unterteilt und diese dann wiederrum in unterschiedliche Unterklassen. Dabei enthält beispielsweise nur die Klasse Ressourcenzerstörung die Unterklasse „Abweichung von der korrekten Protokollausführung“. Allerdings erlaubt eine solche Abweichung nicht nur eine Ressourcenzerstörung, sondern auch eine Ressourcenerschöpfung.

Das im Folgenden vorgestellte Modell fasst die in der Literatur üblichen Einteilungen zusammen. Es werden Klassifizierungskriterien benutzt, die orthogonal zueinander sind. Jeder DoS-Angriff lässt sich damit als n-Tupel der Kriterien der jeweiligen Klasse darstellen. Ein ähnliches – wenn auch deutlich einfacheres – Modell findet sich in [141].

Auch wenn diese Klassifizierung mit dem Fokus auf Denial-of-Service-Angriffe entwickelt wurde, kann sie ebenso zur Analyse von anderen Angriffen verwendet werden. Dabei entfallen dann die beiden Kriterien *Effekt auf angegriffene Ressource* und *Art der angegriffenen Ressource*.

Protokoll-Schicht (K1)

Für den Zugriff auf Netzwerk-Dienste werden Protokolle auf unterschiedlichen Schichten (z. B. Schichten des ISO/OSI-Referenzmodells [158]) verwendet. Dies gilt für legitime Zugriffe aber selbstverständlich auch für Angriffe. Angriffe lassen sich danach einteilen, welche Protokoll-Schicht sie für den Angriff verwenden bzw. welche Protokoll-Instanz sie angreifen.

Typische Protokolle sind dabei beispielsweise IP, TCP, HTTP, SMTP, SOAP. Beispiele für Angriffe, die die zugehörigen Netzwerk-Schichten benutzen, sind entsprechend: *Ping-of-Death* [88], *TCP SYN Flooding* [139], *Cookie Poisoning* [75], *E-Mail Bombing* [12] und *Coercive Parsing* [104].

Desweiteren ergeben sich durch die Verarbeitung einer Nachricht innerhalb des Dienstes weitere „Schichten“, die für Angriffe verwendet bzw. angegriffen werden können. Ein Beispiel ist der *SQL-Injection*-Angriff, mit dem die Datenbank, die von der Dienstapplikation benutzt wird, indirekt angegriffen wird.

Abweichung vom Protokoll (K2)

Angriffe können das verwendete Protokoll auf verschiedene Weise benutzen oder missbrauchen. Dabei gibt es folgende Möglichkeiten (nach [101]):

- *Normale Protokollausführung* (engl. *Normal protocol execution only*): Dabei wird das Protokoll bezüglich der Protokoll-Definition vollständig korrekt ausgeführt.

- *Abweichung von der Nachrichtenabfolge* (engl. *Deviation from protocol message sequence*): Bei dieser Art von Angriff weicht der Angreifer von der korrekten Reihenfolge der Protokollablaufs ab. Ein Beispiel für diese Art von Angriff ist das *TCP SYN Flooding*. Bei diesem Angriff schickt der Angreifer nach der initialen SYN-Nachricht und dem Empfang der SYN-ACK-Nachricht keine ACK-Nachricht und weicht damit von der Abfolge des TCP-Protokolls [51] ab.

- *Abweichung von der Nachrichtensyntax* (engl. *Deviation from protocol message syntax*): Hierbei werden für den Angriff Nachrichten verwendet, die syntaktisch die Protokolldefinition nicht erfüllen. Ein Beispiel ist der *Ping-of-Death*-Angriff. Bei diesem wird ein ICMP-ECHO-Paket mit einer Gesamtgröße größer als 65536 Bytes verwendet. Dieses widerspricht aber der IP-Spezifikation [50], die 65535 Bytes als obere Grenze für die Paketgröße angibt. Eine solches Ping-Paket hat in vielen früheren TCP/IP-Implementierungen Fehler verursacht.

- *Abweichung von der Nachrichtensemantik* (engl. *Deviation from protocol message semantics*): Dabei füllt der Angreifer Protokollfelder mit (syntaktisch korrekten) gefälschten Werten. So werden beispielsweise beim *Address Spoofing* [77] Absender- oder auch Empfänger-Adressen gefälscht, um einen Angriff zu verschleiern. Das *TCP SYN Flooding* wird z. B. oftmals mit *IP Spoofing* kombiniert.

Dabei sind im Allgemeinen Angriffe der zweiten Art am einfachsten, Angriffe der ersten und letzten Art am schwersten zu erkennen.

Effekt auf angegriffene Ressource (K3)

Als Effekt auf die angegriffene Ressource kann man unterscheiden [138]:

- *Ressourcenzerstörung* (engl. *Resource Destruction*): Hierbei wird die angegriffene Ressource dauerhaft außer Funktion gesetzt. Beispiele für derartige Angriffe sind das Eindringen in ein System plus zerstörender Manipulation oder das Ausnutzen vom Implementierungsschwächen, z. B. *Pufferüberlauf* (engl. *Buffer Overflow*). Dies wird beispielsweise oftmals von *Würmern* (engl. *Worms*) ausgenutzt [120].

- *Ressourcenerschöpfung* (engl. *Resource Exhaustion*): Diese Art von Angriffen zielt darauf ab, eine oder mehrere Ressourcen des angegriffen Systems zu belegen und somit die Verfügbarkeit temporär teilweise oder vollständig zu eliminieren. Ein Beispiel ist das bereits erwähnte *TCP SYN Flooding*, bei dem der Speicherbereich für halb-offene TCP-Verbindungen erschöpft wird.

Art der angegriffenen Ressource (K4)

Die folgenden Ressourcen können Ziel eines *Denial-of-Service*-Angriffs werden [141].

- *(Haupt-)Speicher* (engl. *Memory*): Hierbei werden auf dem angegriffenen System große Menge des Hauptspeichers belegt. Die Folgen können Ablehnung weiterer (legitimer) Zugriffe, Verlangsamung des Systems durch Auslagerung von Speicherseiten (engl. *Swapping*) oder Absturz von Prozessen durch Speicherüberlauf sein.

- *Verarbeitungszeit* (engl. *Processing*) bzw. *Prozessornutzung* (engl. *CPU Usage*): Auf dem angegriffen System werden aufwendige Berechnungen hervorgerufen. Dieses können beispielsweise rechenintensive kryptographische Operationen im Verlauf von kryptographischen Protokollen sein.

- *Persistenter Speicher* (engl. *Storage*): Dabei werden große Mengen persistenter Speicher beispielsweise auf Festplatten oder in Datenbanken belegt. Ein Beispiel für einen solchen Angriff ist das *E-Mail Bombing*.

- *Zustand* (engl. *State*): Ein solcher Angriff versucht, den aktuellen Zustand eines *zustandsbehafteten* (engl. *stateful*) Protokolls oder Systems zu modifizieren, so dass ein weiterer Zugriff nicht mehr möglich ist. Ein Beispiel für einen solchen Angriff ist das *FIN/RST Spoofing* [94].

- *Variablen* (engl. *Variables*): Hierbei werden für die Kommunikation notwendige Parameter, die nicht zum Zustand des Protokolls gehören, verändert. Dies wird beispielsweise beim *ARP Poisoning* [151] eingesetzt.

- *Bandbreite* (engl. *Bandwidth*): Dabei wird versucht, in einem Netz die verfügbare Übertragungskapazität möglichst vollständig zu erschöpfen. Im einfachsten Fall wird dies durch direktes Senden einer großen Datenmenge (*Dump Flooding*) erreicht.

Es gibt natürlich auch Angriffe, die gegen mehr als eine Ressource wirken. Außerdem hat der Verbrauch einer Ressource mitunter auch Auswirkungen auf eine andere Ressource. Ein einfaches Beispiel ist ein Angriff, der den Hauptspeicher belegt. Dies führt bei vielen Systemen zu einem Auslagern von Teilen des Hauptspeichers auf die Festplatte (*Swapping*) und damit zu einer Belastung der Festplatte und der CPU.

Angriffs-Maßstab (K5)

Angriffe lassen sich nach der Anzahl der Angriffsquellen unterscheiden.

- *Einzelner Angriff*: Dabei geht der (unmittelbare) Angriff nur von einer Quelle aus.

- *Verteilter Angriff*: Der Angriff geht von einer großen Anzahl von Quellen aus. Diese Art von Angriffen wird auch *Distributed Denial-of-Service (DDoS)* Angriff genannt [118]. Die angreifenden Systeme können dabei entweder freiwillig an diesem Angriff teilnehmen oder werden von dem eigentlichen Angreifer nur missbraucht. In diesem Fall sind die direkten Angreifer, (*Slaves* genannt) mit einem Schadprogramm infiziert, das die Fernsteuerung des Systems ermöglicht. Zur Verschleierung seiner Identität kann der Angreifer die Fernsteuerung der *Slaves* zusätzlich über weitere Zwischenstationen (*Master* genannt) leiten [138]. Diese Methode wird bei sogenannten *Bot-Netzen* verwendet.

Verteilte Angriffe können weiter nach dem Maßstab ihrer Auswirkung in *mittlere* (engl. *medium-scale*) und *große* (engl. *large-scale*) DDoS-Angriffe unterteilt werden [155]. Dabei sind große DDoS-Angriffe leichter zu detektieren, da diese stärkere Auswirkungen auf das gesamte Netz haben und damit bereits im Netzwerk, beispielsweise vom ISP, erkannt werden können.

Ziel des Angriffs (K6)

Weiterhin lassen sich Angriffe nach dem Ziel innerhalb des Netzwerkes klassifizieren [125]. Angriffe können entweder gegen den *Server*, auf dem der Dienst angeboten wird, gerichtet sein, gegen ein Transitsystem im *Netzwerk*, das Nachrichten von oder zu diesem Dienst passieren, oder auch gegen den *Client*, der den Dienst aufruft, gerichtet sein.

Abstraktionslevel (K7)

Angriffe lassen sich nach der Art der *Schwachstelle* unterscheiden, die vom Angriff ausgenutzt wird. Die beiden prinzipiellen Möglichkeiten sind dabei:

- *Implementationsabhängiger Angriff*: Dabei basiert der Angriff auf einer Schwachstelle in einer bestimmten Implementierung. Der Angriff ist in der Form dann nur gegen diese Implementierung erfolgreich. Ein Beispiel dafür ist der *Ping-of-Death*-Angriff, mit dem es möglich ist, bestimmte Implementierungen zum sofortigen Absturz zu bringen.

- *Protokollspezifischer Angriff*: Das sind Angriffe, die auf einer grundsätzlichen Designschwäche des zugrunde liegenden Protokolls basieren. Dies bedeutet allerdings nicht, dass der Angriff gegen alle Implementierungen gleich erfolgreich ist.

  So beschränken viele Protokolle Größe oder Anzahl von Nachrichten nicht. Dies ist eine Schwäche, die für Protokoll-spezifische Angriffe ausgenutzt werden kann (z. B. *E-Mail Bombing*).

Beispiel

Die folgende Tabelle zeigt die Anwendung dieses Klassifizierungssystems auf den *TCP SYN Flooding*-Angriff.

K1 (Protokoll)	TCP
K2 (Abweichung)	Abweichung von der Nachrichtenreihenfolge
K3 (Effekt)	Ressourcenerschöpfung
K4 (Ressource)	Speicher
K5 (Quelle)	einzeln oder verteilt
K6 (Ziel)	Server
K7 (Abstraktion)	protokollspezifisch

Weitere Kategorien

Neben den oben beschriebenen Kategorien gibt es noch viele weitere Einteilungen von Angriffen. Beispielsweise nach *absichtlich* (engl. *intentional*) und *unabsichtlich* (engl. *accidential*), *von innen* (engl. *internal*) und *von außen* (engl. *external*) oder danach, ob ein Denial-of-Service durch eine Person (engl. *human*) oder durch höhere Gewalt (engl. *physical*) ausgelöst wurde [102]. Diese Kriterien werden im Folgenden nicht berücksichtigt.

4.2.1 Metriken für Verfügbarkeit und Denial-of-Service

Für Vertraulichkeit und Integrität gibt es allgemein anerkannte Modelle, die diese Begriffe innerhalb des jeweiligen Sicherheits- und Angreifer-Kontextes definieren. Eines der wichtigsten Modelle dieser Art stammt dabei von Dolev und Yao [45].

Um die Wirkung von DoS-Angriffen quantitativ zu beschreiben, benötigt man Metriken. Dabei gibt es zwei unterschiedliche Ansätze zur Definition solcher Metriken. Der erste Ansatz geht von der quantitativen Beschreibung der Verfügbarkeit eines Systems aus, während der zweite Ansatz versucht, die Belegung von Ressourcen quantitativ zu erfassen.

Metriken für Verfügbarkeit

Die einfachste Metrik für Verfügbarkeit beschreibt den Zustand eines Dienstes zu einem bestimmten Zeitpunkt binär mit „verfügbar“ oder „nicht verfügbar“ (oder aus dem englischen: „up“ und „down“) [28]. Einfache Erweiterungen, die beispielsweise auch für *Dienstgütevereinbarungen* (engl. *Service Level Agreements, SLA*) [97] verwendet werden, berechnen den Anteil eines bestimmten Zeitraums (bei SLAs typischerweise eines Monats), zu dem der jeweilige Dienst verfügbar war. Solcherlei Definition haben zwei Nachteile.

Zum einen ist eine prozentuale Angabe für die Verfügbarkeit als praktisches Gütemerkmal nur wenig aussagekräftig. So ist es für die meisten Dienste eher akzeptabel, wenn sie pro Stunde 2 Minuten nicht verfügbar sind, als pro Monat einen Tag, was aber beides einem Ausfall von ca. 3,3% entspricht.

Zum anderen stellt auch die Abbildung des aktuellen Zustands eines Dienstes auf ein binäres Maß eine zu starke Vereinfachung der Realität dar. Die Frage, ob ein Dienst verfügbar ist oder nicht, lässt sich nämlich nicht trivial beantworten. Gligor und Yu definieren die Verfügbarkeit über die *Wartezeit*, die ein Benutzer auf die Bearbeitung seiner Dienstanfrage warten muss. Dabei gibt es zwei Möglichkeiten: Entweder ist die maximale Wartezeit (MWT) explizit durch den Dienst gegeben oder es wird implizit eine *endliche Wartezeit* (engl. *Finite Waiting Time, FWT*) angenommen. Beide Metriken sind für praktische Untersuchungen ungünstig. Die Forderung nach einer endlichen Wartzeit ist in der Praxis fast immer erfüllt und damit viel zu schwach. Eine MWT wird von den meisten Netzwerkdiensten nicht vorgegeben, so dass diese zusätzlich definiert werden muss. Um die reale Benutzung abzubilden, muss diese MWT dabei weiterhin auch die impliziten und expliziten *Zeitbeschränkungen* (engl. *Timeout*) der Netzwerkprotokolle und Client-Implementierungen berücksichtigen.

Weitere typische Metriken für Verfügbarkeit [33] benutzen *Reaktions-Latenz* (siehe Glossar) (engl. *Response Latency*), *Anfragen-Durchsatz* (siehe Glossar) (engl. *Request Throughput*) und *Wiederherstellungs-Geschwindigkeit* (siehe Glossar) (engl. *Time to Recover*).

Allen oben genannten Metriken verwenden ausschließlich *Leistungsdaten* (engl. *Performance*) des Dienstes als Kriterium für die Verfügbarkeit. Die Verfügbarkeit wird mit solchen Metriken typischerweise derart ermittelt, dass ein Dienst angegriffen wird und die Verfügbarkeit des Dienstes vor, während und nach dem Angriff gemessen wird. Dieses Verfahren lässt sich auch zum Testen von Gegenmaßnahmen gegen DoS-Angriffe verwenden, indem die Verfügbarkeit während des Angriffs ohne und mit aktivierter Gegenmaßnahme gemessen wird. Ein solches Verfahren wird beispielsweise in [139] zur Bewertung des Tools `synkill` zur Abwehr von *TCP SYN Flooding*-Angriffen eingesetzt.

Diese Leistungsmetriken sind für die Bewertung der praktischen Relevanz der Angriffe von großer Bedeutung, sind aber stark vom Szenario abhängig, in dem der Testangriff durchgeführt wird; sie eignen sich daher nur beschränkt für abstrakte Betrachtungen.

Metriken für Ressourcen-Belegung

Für die Bewertung von Angriffen oder die Messung der Angreifbarkeit von Protokollen oder Implementierungen ist ein direktes Maß für die Verfügbarkeit nicht immer zwingend notwendig. Bei Angriffen, die auf Ressourcenerschöpfung zielen, kann auch die Wirkung auf die Ressourcen als Metrik für den Angriff bzw. die Verwundbarkeit des angegriffenen Systems verwendet werden. Dies kann auf zweierlei Weisen geschehen: Zum einen kann in einem Experiment die Menge der verbrauchten Ressourcen während eines Angriffs direkt gemessen und als Maß für die Gefährlichkeit des Angriffs betrachtet werden. Zum anderen kann in theoretischen Betrachtungen von Protokollen oder Verarbeitungsalgorithmen festgestellt werden, wieviele Ressourcen im Durchschnitt oder im schlechtesten Fall verbraucht werden.

Als Anwendung der zweiten Möglichkeit wird beispielsweise in [79] die Anzahl der Berechnungs-Operationen eines Schlüssel-Austauschprotokolls als Maß für die Gefährdung durch DoS-Angriffe verwendet. Derartige Metriken sind natürlich nur dann aussagekräftig, wenn bereits praktische Erfahrungswerte vorliegen, durch welche Art von Ressourcen-Belegung der Denial-of-Service ausgelöst wird. So haben beispielsweise Dean und Stubblefield [42] RSA-Berechnungen als Ressourcen-kritische Operation erkannt und versucht, zum Schutz vor DoS-Angriffen die Anzahl dieser Operationen zu beschränken.

Desweiteren sind solche Metriken hilfreich, um die Gefährdung unterschiedlicher Systeme bezüglich eines Angriffs zu vergleichen, wie es in [79] beispielsweise für verschiedene Schlüsselaustauschprotokolle durchgeführt wurde.

4.2.2 Maßnahmen gegen Denial-of-Service-Angriffe

Während zur Sicherung von Integrität und Vertraulichkeit verschiedene Standardmechanismen existieren, ist dies bei der Sicherung der Verfügbarkeit nicht der Fall. Viele Mechanismen, die in diesem Umfeld entwickelt wurden, dienen dem Schutz eines bestimmten Protokolls oder der Abwehr eines bestimmten Angriffs. Dennoch haben sich einige allgemeine Richtlinien zum Schutz der Verfügbarkeit herausgebildet.

Grundsätzlich kann Verfügbarkeit zum einen durch Maßnahmen gegen Denial-of-Service-Angriffe und zum anderen durch eine Erhöhung der Leistung (und damit der *Performance*-Metrik) erreicht werden. Zu dem zweiten Fall gehören Maßnahmen wie leistungsstärkere Hardware, Lastverteilung (engl. *Load Balancing*), unterbrechungsfreie Stromversorgung usw. Derartige Maßnahmen werden im Folgenden nicht weiter betrachtet.

Von größerem wissenschaftlichen Interesse sind Maßnahmen gegen Denial-of-Service-Angriffe. Derartige Maßnahmen lassen sich in drei Kategorien einteilen [134, 138]:

1. Vorbeugende Maßnahmen zur Eliminierung der Angriffsmöglichkeit
2. Erkennung von Angriffen und Abschwächung der Angriffswirkung
3. Gezielte Abwehrmaßnahmen:
   - Akute Angriffsabwehr
   - Rückverfolgung des Angriffs (engl. *Traceback*); u. a. zur Abschreckung des Angreifers
4. Wiederherstellung der Ressource

Wie bei allen Sicherungsmaßnahmen (auch in Nicht-IT-Systemen) sollten aus ökonomischen Gründen die Kosten (bzgl. monetärer oder anderer Ressourcen), die eine Gegenmaßnahme erfordert, in Relation zu dem Risiko (d. h. der Wahrscheinlichkeit und der Tragweite) des entsprechenden Angriffs gesetzt werden [102]. Bei DoS-Schutzmaßnahmen, die zur Laufzeit des Angriffs wirken, ist die Menge der erforderlichen Ressourcen aber auch von theoretischem Interesse. So erzeugt jede Maßnahme, die zusätzliche Ressourcen bindet, eine neue Möglichkeit für einen Angriff, der genau diese Maßnahme für einen Ressourcen-Erschöpfungs-Angriff ausnutzt. Dies ist ein wichtiger Unterschied zu Maßnahmen zur Sicherung von Integrität und Vertraulichkeit.

Im Folgenden werden grundsätzliche Realisierungen der Punkte 1. und 2. vorgestellt. Diese lassen sich allerdings nicht immer trennen. Viele Maßnahmen haben sowohl eine vorbeugende Komponente (z. B. Installation eines Schutzsystems) als auch eine Laufzeitkomponente (z. B. Überprüfung des Netzwerkverkehrs in diesem Schutzsystem).

Überprüfung der Eingabe

Alle eingehenden Nachrichten müssen als potenziell gefährlich angesehen werden (*„never trust your input“* [129]) und einer möglichst genauen Überprüfung unterzogen werden. Diese Überprüfung kann in zwei Klassen unterteilt werden.

Zum einen können Nachrichten auf Konformität bezüglich des jeweiligen Protokolls untersucht werden (engl. *Protocol Conformance*). Damit können Angriffe erkannt und abgewehrt werden, bei denen von der korrekten Protokollausführung abgewichen wird (sei es bezüglich Nachrichtensyntax oder -folge). Ein großer Vorteil solcher Überprüfungen ist, dass das Prüfkriterium durch die Protokolldefinition bereits vorgegeben ist.

Zum anderen müssen auch Nachrichten, die innerhalb der Protokoll-Spezifikation bleiben, auf ihre Gefährlichkeit untersucht werden. Zur Erkennung von derartigen Angriffen müssen die Prüfkriterien, die oben durch die Protokolldefinition gegeben waren, zunächst identifiziert werden. Die Suche und Bestimmung von Angriffsmustern (engl. *Patterns*) ist ein wichtiger Teil von *Intrusion Detection and Prevention Systems* (IDS) [43]. Die Muster können dabei entweder durch Analyse bekannter Angriffe gewonnen und statisch konfiguriert (*Signature-based* bzw. *Knowledge-based Detection*) oder aber zur Laufzeit adaptiv mittels statistischer Methoden durch Abgleich des aktuellen Netzwerkverkehrs mit einem definierten Normalprofil (*Anomaly-Based Detection*) gewonnen werden [137].

Wie bereits vorher für allgemeine Schutzmaßnahmen erläutert, müssen Systeme zur Überprüfung des eingehenden Netzwerkverkehrs sehr sorgfältig bezüglich des Ressourcenverbrauchs entwickelt werden, um nicht selbst Ziel eines Angriffs zu werden.

Beschränkung des Zugriffs

Ein wichtige Methode, um Angriffe auf Ressourcen zu verhindern, ist die Beschränkung des Zugriffs auf diese Ressource. *Zugriffskontrolle* (engl. *Access Control*) besteht dabei aus der Feststellung des Urhebers des Ressourcenzugriffs, der *Authentifizierung* (engl. *Authentification*) oder auch *Herkunftsprüfung* und der Überprüfung anhand einer *Zugriffs-Policy*, ob diesem Urheber der Zugriff auf diese Ressource erlaubt ist, die Autorisierung (engl. *Authorization*). Zugriffskontrolle kann dabei auf unterschiedlichen Netzwerkschichten, mit unterschiedlich starken *Berechtigungsnachweisen* (engl. *Credentials*) und an unterschiedlichen Orten im Netzwerk durchgeführt werden. Einige typische Mechanismen sind:

Gateways: Gateways werden üblicherweise an Grenzen verschiedener *Sicherheitsbereiche* (engl. *Security Domain*) eingesetzt, um (primär) Zugriffe aus dem unsicheren Bereich auf Ressourcen innerhalb des sicheren Bereichs zu beschränken. Diese Ressourcen werden meistens über IP-Adresse, TCP-Port und HTTP-URL des Dienstes festgelegt.

IP-Filterung: Eine Filterung nach IP-Adressen kann an beliebigen Stellen im Netzwerk erfolgen und verwendet IP-Adressen als Berechtigungsnachweis. Da diese leicht zu fälschen sind, ist dies nur eine schwache Zugriffskontrolle.

Kryptographische Authentifizierung: Hierbei werden kryptographische Mechanismen wie beispielsweise digitale Signaturen zur Authentifizierung verwendet. Dies erlaubt eine sehr starke Authentifizierung, wenn man sowohl die kryptographischen Protokolle als auch die zugrunde liegenden elementaren kryptographischen Operationen als sicher annimmt.

Ein großes Problem dieser Zugriffskontrolle ist aber die hohe CPU-Nutzung der kryptographischen Operationen, die bei vielen Authentifizierungssystemen Ressourcen-Angriffe ermöglichen. In einigen Arbeiten wurden daher bereits Anstrengungen unternommen, DoS-sichere Systeme zur Authentifizierung zu entwickeln [79, 113]. Die dabei verwendeten Methoden zur Beschränkung oder Kontrolle der verbrauchten Ressourcen sind nachfolgend aufgeführt.

Beschränkung des Ressourcenverbrauchs

Eine naheliegende Maßnahme gegen Ressourcenerschöpfungs-Angriffe ist, den Ressourcenverbrauch der Dienstzugriffe zu beschränken. Dies ist aber nicht trivial und teilweise sogar gar nicht möglich.

So stehen einem Dienst, der über das Internet angeboten wird, viel mehr potentielle Clients gegenüber als irgendein System bedienen kann. Verteilte Angriffe sind daher eine grundsätzliche Bedrohung, der auch große Firmen mit sehr leistungsstarker Infrastruktur oder sogar die gesamte Internet-Architektur ausgesetzt sind. Beispiele dafür sind Angriffe auf Yahoo und Amazon im Februar 2000 [32] und ein DDoS-Angriff auf die DNS-Rootserver im Februar 2007 [53].

Weiterhin besteht bei der Einrichtung einer neuen Kontrollinstanz immer die Gefahr, dass diese selbst zum Opfer eines DoS-Angriffs wird. Insbesondere wenn die Durchsetzung der Ressourcenbeschränkung selbst Ressourcen in der gleichen Größenordnung wie das zu schützende System verbraucht, ist ein solches Schutzsystem wirkungslos.

Hinzu kommt, dass die Parameter für Nachrichtenbeschränkungen sich am tatsächlichen Ressourcenverbrauch für die Verarbeitung dieser Nachricht orientieren sollten [114]. Allerdings ist dies für eine Kontrollinstanz nur schwer zu berechnen. Ein von einer *Datenmengenkontrolle* (engl. *Traffic Shaping*) stark reduzierter Datenstrom kann beispielsweise trotzdem aufwendige kryptographische Operationen oder großen Speicherverbrauch (z. B. bei einer *ZIP-Bombe* (siehe Glossar)) im Dienst auslösen.

Ein weiteres Problem bei der Durchführung solcher Beschränkungen ist, dass sie oftmals der Spezifikation der verwendeten Protokolle widersprechen. Bei praktisch allen Netzwerk-Protokollen sind zumindest für einige Parameter beliebig große oder extrem große Werte erlaubt: Die Anzahl der offenen TCP-Verbindungen ist nur durch den Raum der Port-Nummern beschränkt, E-Mails können beliebig groß sein, ein HTTP-Header kann beliebig viele Einträge haben usw. Wenn man zur Bekämpfung von Ressourcenerschöpfungs-Angriffen nun für diese Parameter Schranken definiert, so riskiert man die Ablehnung von legitimen Anfragen. Die meisten Protokolle besitzen auch keine Mechanismen, um solche Beschränkungen einem Client

mitzuteilen. Die einzige Möglichkeit ist dann, dies in separaten Abkommen über die Dienstgüte (engl. *Quality of Service*) bekannt zu geben.

Beispiele für Maßnahmen zur Beschränkung des Ressourcenverbrauch eines Dienstes sind die folgenden:

Ratenkontrolle: Dabei werden bei Überschreitung einer festgelegten Dienstauslastung (z. B. Verkehrsmenge, Anzahl der offenen Verbindungen) bestimmte ressourcenkonsumierende Protokollabläufe reduziert (z. B. Reduzierung des eingehenden Verkehrs) oder sogar abgeschaltet (z. B. Ablehnung weiterer Verbindungsaufbauwünsche). Solche Maßnahmen funktionieren nur auf der Netzwerkebene und sind bei vielen Angriffen auf Applikationsebene wirkungslos.

Zustandlose Protokolle: Einige Angriffe (z. B. *TCP SYN Flooding*) beruhen darauf, Ressourcen zu erschöpfen, die zur Speicherung des Zustands eines Protokolls benötigt werden. Solche Angriffe können durch Verwendung von zustandslosen Protokollen abgewehrt werden [7]. Dabei können beispielsweise Zustandsinformationen, die ein Server ansonsten zwischenspeichern müsste, als Protokollparameter zum Client zurückgegeben werden. Diese Parameter müssen vom Client zur Fortsetzung der Protokoll-Sitzung benutzt werden. Die Zustandsinformationen müssen sorgfältig gegen Manipulation gesichert werden, um Angriffe gegen die Konsistenz des Protokolls zu verhindern (wie z. B. beim *Cookie Poisoning* (siehe Glossar) [61]).

Client Puzzle: Eine weitere Möglichkeit, den Ressourcenverbrauch eines Dienstes zu beschränken, stellen *Client Puzzles* dar. Dabei muss ein Client, der einen Dienst nutzen will, zunächst ein „Rätsel“ lösen, d. h. eine mehr oder weniger aufwendige Berechnung durchführen. Damit kann ein Server durch Variation des Berechnungsaufwands die Senderate der Clients steuern. Clients mit geringen Zugriffsraten erhalten einfache Rätsel, Clients mit hohen Zugriffsraten (also potentielle Angreifer) erhalten komplexe Rätsel. Eine wichtige Eigenschaft von *Client Puzzles* ist, dass der Server zu Erstellung und Überprüfung der Rätsel nur geringen Berechnungsaufwand benötigt und keinerlei Zustandsinformationen speichern muss. *Client Puzzles* wurden ursprünglich zur Abwehr von E-Mail-Spam entwickelt [47] und später auch zum Schutz von kryptographischen Protokollen vor DoS-Angriffen eingesetzt [8].

Kapitel 5

Angriffe gegen Web Services

In diesem Kapitel werden einige typische Angriffe gegen Web Services beschrieben. Die Angriffe werden bezüglich des Klassifizierungssystems aus Abschnitt 4.2 eingeordnet. Anschließend wird der Angriff, die dabei ausgenutzte Schwachstelle und die Wirkung auf das angegriffene System abstrakt beschrieben. Für die meisten Angriffe folgt ein Beispiel für die tatsächliche Ausführung dieses Angriffs gegen ein reales System. Details zu der Konfiguration der Testsysteme (System A bis D) finden sich am Ende dieses Kapitels.

Die installierten Programme wurden bei den beschriebenen Experimenten soweit wie möglich in der Standardkonfiguration betrieben. Einige Angriffe sind in anderen Konfigurationen möglicherweise schwieriger oder vielleicht sogar gar nicht durchzuführen. Dies ändert aber nichts an der Gefährdung durch die dargestellten Schwachstellen. Zum einen werden in der Praxis viele Systeme mit Standardkonfigurationen betrieben. Zum anderen ist wie bei allen Angriffen die Existenz einer angreifbaren Konfiguration ausreichend, um die entsprechende Schwachstelle ernst zu nehmen. Weitere Angriffe gegen Web Services finden sich in [104] und [92].

5.1 Metriken für Angriffe auf Web Services

Wie bereits im Abschnitt 4.2.1 erläutert, ist die direkte Messung der Verfügbarkeit während eines Angriffs und damit des „Erfolgs“ des Angriffs nur schwer möglich. Zur Bewertung der Beispiel-Angriffe, die allesamt Ressourcen-Erschöpfungs-Angriffe sind, wurde stattdessen die Wirkung auf diese Ressourcen gemessen. Dies sind hier im wesentlichen Rechenleistung und Hauptspeicher.

Die Speicherverbrauchsdaten beziehen sich dabei auf den Verbrauch des Prozesses, in dem der Web Service läuft. Bei Java-basierten Web-Service-Frameworks ist dies also die Java-VM, bei .NET-Web-Services der ASPNET-Prozess. Zu beachten ist, dass diese Prozesse aufgrund von Caching-Mechanismen u. ä. mehr Speicher allozieren als für die Verarbeitung notwendig ist. Dies bedeutet, dass bei einer eingehenden Nachricht der Speicherzuwachs des Web-Service-Prozesses nicht genau dem Speicherverbrauch für die Verarbeitung der Nachricht entspricht. Da der allozierte Speicher dem Betriebssystem für andere Prozesse nicht mehr zur Verfügung steht, stellt dieser Wert aber trotzdem ein sinnvolles Maß für die Belastung des Serversystems dar.

Für einen Angreifer ist die Größe und Anzahl der gesendeten Angriffsnachrichten ein kritisches Maß. Ein Angriff ist umso einfacher, je weniger Bytes benötigt werden. Zum einen verbraucht das Senden großer Nachrichtenmengen Ressourcen, zum anderen steigt mit der

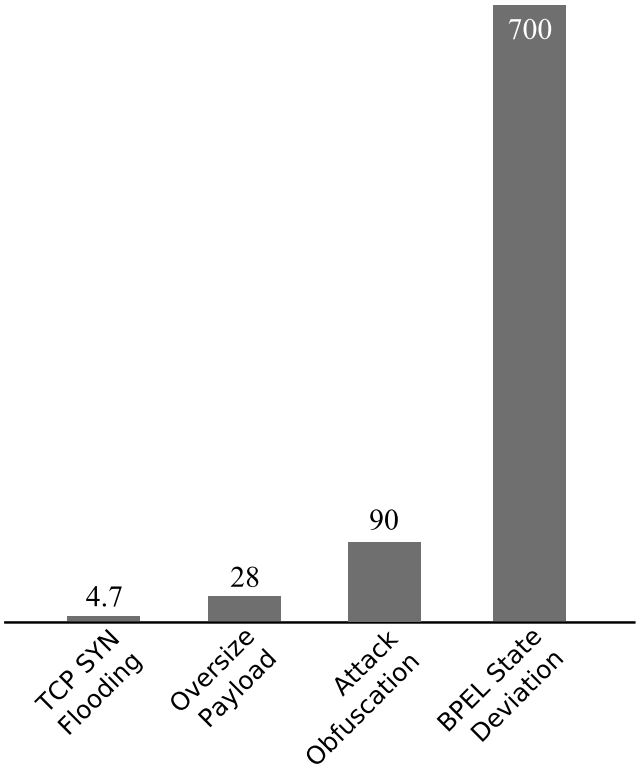


Abbildung 5.1: Verstärkungsfaktor α für verschiedene Angriffe

Anzahl und Größe der Nachrichten die Wahrscheinlichkeit, dass der Angriff von Netzwerkelementen erkannt wird. Ein wichtiges Maß für die Gefährlichkeit eines Angriff ist also der *Verstärkungsfaktor*, das Verhältnis zwischem dem Ressourcenverbrauch beim Opfer und dem Ressourcenverbrauch zur Erstellung der entsprechenden Angriffsnachricht [93]. Im Folgenden wird der Verstärkungsfaktor für den Speicherverbrauch betrachtet. Zur Vereinfachung wird der Verstärkungsfaktor α als das Verhältnis von gemessenen Speicherverbrauch mem auf dem angegriffenen System und der Gesamtgröße der Angriffsnachricht(en) $mesg$ ermittelt:

$$\alpha = \frac{mem}{mesg}$$

Zum Vergleich mit einem nicht-Web-Service-Angriff ist der Verstärkungsfaktor für den SYN-Flooding-Angriff nach [139]:

$$\alpha = \frac{280 \text{ Byte}}{60 \text{ Byte}} \approx 4.7$$

Abbildung 5.1 zeigt den Verstärkungsfaktor für einige Web-Service-Angriffe, die in diesem Kapitel vorgestellt werden, im Vergleich zum Verstärkungsfaktor für den SYN-Flut-Angriff. Darin zeigt sich die hohe Gefährdung von Web Services, die mit ansteigender Komplexität der Web-Service-Technologie (einfacher Web Service → Web Service mit WS-Security → Web-Service-Komposition) noch deutlich zunimmt.

5.2 Denial-of-Service-Angriffe

5.2.1 *Oversized SOAP Message*

Kurzbeschreibung

Angriffe mit sehr großen SOAP-Nachrichten

Klassifizierung

K1 (Protokoll)	Web Service
K2 (Abweichung)	Normale Protokollausführung *oder* Abweichung von der Nachrichtensyntax
K3 (Effekt)	Ressourcenerschöpfung
K4 (Ressource)	Speicher
K5 (Quelle)	einzeln
K6 (Ziel)	Server
K7 (Abstraktion)	protokollspezifisch

Beschreibung

Der *Oversized Payload*-Angriff versucht die Hauptspeicher-Ressourcen eines Web-Service-Servers mittels einer sehr großen SOAP-Nachricht zu erschöpfen. Ähnliche Angriffe sind grundsätzlich auch gegen alle anderen Arten von Diensten möglich.

Die Wirkung des *Oversize Payload*-Angriffs basiert darauf, dass bei vielen Implementierungen (inbesondere bei solchen mit baumbasiertem Verarbeitungsmodell) der Speicherbedarf während der Verarbeitung einer Anfrage deutlich größer ist als die Anfragenachricht selbst. Der Angreifer erreicht also eine Art *Multiplikationseffekt*. Der Steigerungsfaktor liegt für gängige Web-Service-Implementierungen bei 2 bis 30. Dabei ist der Steigerungsfaktor größer, wenn die Nachricht viele Elemente mit kleinen oder gar keinen Textinhalten enthält als wenn sie wenige Elemente mit großen Textinhalten enthält.

Dieser Angriff kann mit protokollkonformen Nachrichten durchgeführt werden, wenn der Web Service unbeschränkt große Nachrichten erlaubt. Dies ist bei den meisten Web Services der Fall. Zum einen sind viele Datentypen in *XML Schema* nicht in der Länge beschränkt. Sobald eine WSDL also einen Datentyp wie `xsd:string` oder `anyURI` enthält, lassen sich valide Nachrichten mit unbeschränkt großen Textknoten konstruieren. Wie bereits oben angesprochen, stellen Nachrichten mit einer großen Anzahl von XML-Elementen eine noch viel größere Gefahr dar. Aber auch solche Nachrichten sind bei vielen Web Services valide. Ein wichtiges Beispiel dafür sind unbeschränkte Listen von Elementen. Diese werden im WSDL-Schema durch ein Konstrukt der Art `<xsd:element maxOccurs="unbounded"...>` dargestellt und erlauben eine beliebige Wiederholung dieses Elements.

Zum anderen ist der *Oversized Payload* auch oftmals erfolgreich, wenn die Web-Service-Definition nur Nachrichten beschränkter Größe erlaubt. Dies liegt daran, dass viele Implementierungen eingehende SOAP-Nachrichten gar nicht oder nur unzureichend auf syntaktische Korrektheit überprüfen.

Beispielangriff

Ein Web Service (System A) wurde mit einer validen SOAP-Nachricht aufgerufen, die eine große Liste von Parameterelementen enthielt:

```
<env:Envelope>
  <env:Body>
    <arrayLength>
      <field>
        <item>Hello</item>
        <item>Hello</item>
        <item>Hello</item>
      ...
      </field>
    </arrayLength>
  </env:Body>
</env:Envelope>
```

Bei der Verarbeitung einer Nachricht mit einer Liste mit 100.000 Elementen (entspricht einer serialisierten Größen von ca. 1,8 MByte) war das System für ca. 55 Sekunden zu 100% ausgelastet und benötigte ca. 50 MByte Hauptspeicher. Damit ergibt sich also:

$$\alpha = \frac{50 \text{ MByte}}{1.8 \text{ MByte}} \approx 28$$

Bei einer Nachricht mit 110.000 Elementen (entspricht einer serialisierten Größe von ca. 2,0 MByte) war das System nicht mehr in der Lage, die Anfrage auszuführen und gab eine `java.lang.OutOfMemoryError`-Fehlermeldung zurück.

5.2.2 *Coercive Parsing*

Kurzbeschreibung

Angriffe gegen den XML-Parsing-Vorgang

Klassifizierung

K1 (Protokoll)	XML
K2 (Abweichung)	Normale Protokollausführung *oder* Abweichung von der Nachrichtensyntax
K3 (Effekt)	Ressourcenerschöpfung
K4 (Ressource)	Speicher
K5 (Quelle)	einzeln
K6 (Ziel)	Server
K7 (Abstraktion)	protokollspezifisch

Beschreibung

Diese Art der Angriffe ist gegen den XML-Parser gerichtet und versucht, diesen durch spezielle XML-Konstrukte zu speicheraufwendigen Operationen zu zwingen. Dafür gibt es mehrere

Möglichkeiten. Eine einfache Möglichkeit ist die Verwendung eines XML-Dokuments mit sehr tief geschachtelter Baumstruktur (siehe Beispiel unten). Bei den meisten Web Services widerspricht ein solches SOAP-Dokument der Web-Service-Beschreibung, der Angriff fällt also in die Kategorie *Abweichung von der Nachrichtensyntax*.

Gefährlicher, da schwieriger zu erkennen, sind Angriffe mit XML-Dokumenten, die unabhängig von dem Web Service immer protokollkonform sind. Diese benutzen Konstrukte, die laut XML-Spezifikation in jedem XML-Dokument erlaubt sind. So sind für Namensraum-Präfixe weder die Länge der Präfixe oder der Namensraum-URLs noch die Anzahl der Präfix-Definitionen pro XML-Element beschränkt [25]. Außerdem muss der XML-Parser alle Präfix-Definitionen zwischenspeichern, auch solche, die im Dokument gar nicht verwendet werden.

Weitere Angriffe ähnlicher Art ergeben sich aus der Möglichkeit, beliebig lange Element- oder Attributnamen zu verwenden. Weiterhin kann ein Dokument Verweise auf (möglicherweise sehr große) externe DTDs [26] enthalten. Diese werden vom Parser nachgeladen, was Speicher und Rechenzeit verbraucht (*Entity Expansion*). Auch wenn DTDs vom WS-I für Web Services explizit ausgeschlossen werden, unterstützen XML-Parser immer noch DTDs.

Beispielangriff

Ein Web Service (System A) wurde mit einer SOAP-Nachricht angegriffen, die eine nichtabbrechende Folge von Startkennungen enthielt:

```
<x>
  <x>
    <x>
      ...
```

Als Resultat des Angriffs verbrauchte der Web-Service-Server 100% der Prozessorleistung. Dadurch war die Verfügbarkeit des Systems so stark eingeschränkt, dass die Angriffsnachricht nur noch mit einer Rate von 150 Byte/s empfangen wurde.[1] Bei diesem Angriff hat der Web-Service-Server die Verbindung nicht beendet. Der Angriff könnte also offenbar beliebig lange durchgeführt werden.

5.2.3 *Attack Obfuscation*

Kurzbeschreibung

Verschleiern von Angriffen durch Verschlüsselung

Klassifizierung

K1 (Protokoll)	SOAP-Sicherheit
K2 (Abweichung)	*unterschiedlich*
K3 (Effekt)	Ressourcenerschöpfung
K4 (Ressource)	Speicher + Prozessornutzung
K5 (Quelle)	einzeln
K6 (Ziel)	Server
K7 (Abstraktion)	protokollspezifisch

[1] Dies bedeutet umgekehrt, dass dieser Angriff selbst über eine extrem schmalbandige Verbindung ausführbar wäre.

Beschreibung

WS-Security erlaubt die Verschlüsselung von Teilen der SOAP-Nachricht. Wenn dieser Teil *bösartigen* (engl. *malicious*) Inhalt enthält, so kann dies erst nach der Entschlüsselung detektiert werden. Dies führt zu zweierlei Problemen. Zum einen können Systeme, die XML-Encryption nicht beherrschen (z. B. einfache XML-Firewalls) oder aber die notwendigen (geheimen) Schlüssel nicht zur Verfügung haben, derartige Angriffe nicht erkennen. Zum anderen müssen zur Detektion des ungültigen Inhalts in jedem Fall bereits Ressourcen für die Entschlüsselung aufgewendet werden. Die Verarbeitung von verschlüsselten Nachrichten im baumbasierten Modell sieht beispielhaft wie folgt aus (hier angenommen: SOAP-Body enthält `Operation`, welche verschlüsselt wurde):

$$
\begin{aligned}
&\mathit{Parsing}(\ldots\ \texttt{Body EncData}\ \ldots\ \texttt{qX7f2csGdf7a}\ \ldots \overline{\texttt{EncData}}\ \overline{\texttt{Body}}\ \ldots) \mid \\
&\mathit{Validate}(\ldots\ \texttt{Body EncData}\ \ldots\ \texttt{qX7f2csGdf7a}\ \ldots \overline{\texttt{EncData}}\ \overline{\texttt{Body}}\ \ldots) \mid \\
&\mathit{Decrypt}(\ldots\ \texttt{Body EncData}\ \ldots\ \texttt{qX7f2csGdf7a}\ \ldots \overline{\texttt{EncData}}\ \overline{\texttt{Body}}\ \ldots) \mid \\
&\mathit{Validate}(\ldots\ \texttt{Body Operation IntParam xyz}\ \overline{\texttt{IntParam}}\ \overline{\texttt{Operation}}\ \overline{\texttt{Body}}\ \ldots)
\end{aligned}
$$

Man erkennt, dass Verarbeitungsschritte, die vor dem Entschlüsseln stattfinden (hier: *Parsing* und das erstes *Validate*), für das gesamte Dokument angewendet werden, und ebenso das gesamte Dokument enschlüsselt wird (*Decrypt*), bevor der ungültige Dokumententeil erkannt werden kann (hier: ab dem zweiten *Validate*). All dies zusammen verbraucht sowohl Speicher- als auch Rechenleistungsressourcen.

Beispielangriff

Eine Nachricht mit einer verschlüsselten Operation im SOAP-Body und einer Gesamtgröße von 1 MByte wurde an einen Web Service (System B) gesendet. Zur Verarbeitung dieser Nachricht (inklusive Entschlüsselung) benötigte der Server 23 Sekunden und ca. 90 MByte zusätzlichen Hauptspeicher. Damit ergibt sich:

$$\alpha = \frac{90\ \text{MByte}}{1\ \text{MByte}} = 90$$

Zum Vergleich dazu wurde eine ähnliche Nachricht mit 20 MByte Größe, aber ohne Verschlüsselung, von demselben System in weniger als einer Sekunde verarbeitet.

5.2.4 *Oversized Cryptography*

Kurzbeschreibung

Erzwingen von kryptographischen Operationen

Klassifizierung

K1 (Protokoll)	SOAP-Sicherheit
K2 (Abweichung)	Normale Protokollausführung
K3 (Effekt)	Ressourcenerschöpfung
K4 (Ressource)	Prozessornutzung (z. T. auch Speicher)
K5 (Quelle)	einzeln
K6 (Ziel)	Server
K7 (Abstraktion)	protokollspezifisch

Beschreibung

Bei dieser Art von Angriffen werden beim Web-Service-Server (aufwändige) kryptographische Operationen ausgelöst. Dies führt zur Belastung des Prozessors und damit zu reduzierter Verfügbarkeit.

Ein Beispiel für einen solchen Angriff ist das Versenden einer großen Anzahl von Nachrichten, deren Verarbeitung kryptographische Operationen erfordern, beispielsweise eine Authentifizierung mittels Signaturen. Dabei kann ein Angreifer u. a. ausnutzen, dass bei DSA die Generierung einer Signatur schneller ist als die Verifizierung [106] und dadurch beim angegriffenen System mehr Ressourcen gebunden werden als er selbst aufwendet.

Durch die flexible Anwendbarkeit von WS-Security bieten sich aber auch mit nur einer Nachricht vielfältige Möglichkeiten für „kryptographische“ Angriffe. Ein Beispiel dafür ist eine Kette von verschlüsselten Schlüsseln. Dabei wird ein Teil der Nachricht mit einem generierten Schlüssel verschlüsselt, dieser Schlüssel wird wiederum mit einem generierten Schlüssel verschlüsselt usw.:

$$\texttt{EncKey}_{key_0}(key_1), \texttt{EncKey}_{key_1}(key_2), \ldots, \texttt{EncKey}_{key_{n-1}}(key_n), \texttt{EncData}_{key_n}(data)$$

Zum Entschlüsseln der verschlüsselten Daten ist es nun notwendig, alle verschlüsselten Schlüssel zu entschlüsseln, was eine erhebliche Prozessornutzung erfordert.

Eine ähnlicher Angriff verwendet mehrfach ineinander geschachtelte verschlüsselte Blöcke (wie eine Matroschka-Puppe):

$$\texttt{EncData}(\texttt{EncData}(\ldots(\texttt{EncData}(data))\ldots))$$

Zusätzlich zur Prozessornutzung verbraucht die Verarbeitung einer solchen Nachricht auch große Speichermengen, falls jeweils sowohl die verschlüsselten als auch die entschlüsselten Daten zwischengespeichert werden (was sehr üblich ist).

5.2.5 *Policy Violation*

Kurzbeschreibung

Angriffe mit nicht Policy-konformen Nachrichten

Klassifizierung

K1 (Protokoll)	SOAP-Sicherheit
K2 (Abweichung)	Abweichung von der Nachrichtensyntax
K3 (Effekt)	Ressourcenerschöpfung
K4 (Ressource)	Speicher + Prozessornutzung
K5 (Quelle)	einzeln
K6 (Ziel)	Server
K7 (Abstraktion)	protokollspezifisch

Beschreibung

Bei dieser Art von Angriffen werden Nachrichten verwendet, die nicht der Sicherheits-Policy entsprechen. Mit solchen Nachrichten lassen sich verschiedene Effekte erzielen. So können mit Nachrichten, die viele (zusätzlich zu denen in der Policy geforderten) Sicherheitselemente enthalten, Ressourcen zum Zwischenspeichern und für die Durchführung von kryptographischen Operationen verbraucht werden. Beispiele für zusätzliche Sicherheitselemente sind:

- eingebettete Sicherheits-Token
- externe Sicherheits-Token (hierbei werden zusätzlich Ressourcen für das Erlangen des Tokens benötigt)
- Signaturen
- verschlüsselte Blöcke

Desweiteren lassen sich auch andere Angriffe, als solche gegen die Verfügbarkeit, mit nicht konformen Nachrichen durchführen (siehe Abschnitt 5.3).

5.2.6 *BPEL State Deviation*

Kurzbeschreibung

Angriffe mit zustandsinvaliden Nachrichten

Klassifizierung

K1 (Protokoll)	BPEL
K2 (Abweichung)	Abweichung von der Nachrichtenfolge
K3 (Effekt)	Ressourcenerschöpfung
K4 (Ressource)	Speicher + Prozessornutzung
K5 (Quelle)	einzeln
K6 (Ziel)	Server
K7 (Abstraktion)	protokollspezifisch

Beschreibung

Wird ein BPEL-Prozess auf einer BPEL-Ablaufumgebung veröffentlicht, so wird für jede BPEL-Aktivität, die eine eingehende Nachricht beschreibt, ein Web-Service-Endpunkt erzeugt. In dem

BPEL-Prozess ist festgelegt, in welcher Reihenfolge diese Web-Services aufgerufen werden *sollen*. Allerdings können die Web-Service-Endpunkte zu jedem Zeitpunkt des Protokolls aufgerufen werden. Dies liegt daran, dass ein BPEL-Prozess mehrere Instanzen haben kann. Wird also ein Web-Service-Endpunkt aufgerufen, muss diese Nachricht verarbeitet werden. Dabei muss überprüft werden:

1. Zu welcher Prozessinstanz gehört diese Nachricht?
2. Ist diese Prozessinstanz gerade in einen Zustand, in dem der Aufruf dieser Operation erlaubt ist?

Gehört die Nachricht zu keiner Prozessinstanz oder nicht zum aktuellen Zustand der Instanz, so wird die Nachricht verworfen. Um dieses zu erkennen, müssen aber Speicher- und Rechenressourcen aufgewendet werden. Neben der bereits (als teuer) bekannten XML-Verarbeitung kommen bei BPEL-Servern typischerweise noch Datenbankzugriffe hinzu, da aufgrund der großen Anzahl und der Langlebigkeit der Prozessinstanzen deren Zustandsinformationen in Datenbanken abgelegt werden. Diese Eigenschaften kann ein Angreifer ausnutzen, um mit zustandsinvaliden Nachrichten die Ressourcen der BPEL-Ablaufumgebung zu erschöpfen.

Beispielangriff

Der folgende Angriff wurde gegen eine BPEL-Ablaufumgebung (System D) mit einem BPEL-Prozess durchgeführt. Dieser Prozess enthielt zwei `bpel:receive`-Aktivitäten first und second, wobei nur first eine neue Prozessinstanz erzeugt und second nach first aufgerufen werden soll. Der Prozess enthielt weiterhin einige Korrelations-Eigenschaften zur Instanz-Identifizierung. Der Angriff verwendete Nachrichtenaufrufe an die Operation second und enthielt Korrelations-Eigenschaften, die zu keiner Prozessinstanz gehörten. Insgesamt wurden 1000 dieser Nachrichten (mit jeweils einer Größe von 0,5 KByte) an die BPEL-Ablaufumgebung geschickt. Die Nachrichten wurden als ungültig erkannt und verworfen. Für die Verarbeitung benötigte die BPEL-Ablaufumgebung aber zusätzlich ca. 350 MByte Hauptspeicher. Für den Verstärkungsfaktor α ergibt sich also:

$$\alpha = \frac{350 \text{ MByte}}{0.5 \text{ MByte}} = 700$$

5.3 Andere Angriffe

In diesem Abschnitt werden Angriffe gegen Web Services vorgestellt, die nicht direkt zur Reduktion der Verfügbarkeit führen. Die geschilderten Schwachstellen erlauben die Verletzung von Integrität oder Vertraulichkeit, können dabei aber auch in bestimmten Szenarien für DoS-Angriffe ausgenutzt werden.

Die in Abschnitt 4.2 entwickelte Klassifizierung kann auf solche Angriffe ebenso angewendet werden, es entfallen aber die Kategorien *Effekt auf angegriffene Ressource* und *Art der angegriffenen Ressource*.

5.3.1 *SOAP Action Spoofing*

Kurzbeschreibung

Angriffe mit gefälschtem `SOAPAction`-HTTP-Header

Klassifizierung

K1 (Protokoll)	Web Service + HTTP
K2 (Abweichung)	Abweichung von der Nachrichtensemantik
K5 (Quelle)	einzeln
K6 (Ziel)	Server
K7 (Abstraktion)	protokollspezifisch

Beschreibung

Die Web-Service-Operation ist sowohl durch das `SOAPAction`-Feld als auch durch die SOAP-Nachricht selbst (genauer: durch das erste Kindelement des SOAP-Bodys) identifiziert. Obwohl der `SOAPAction`-Wert nur als Hinweis auf die tatsächliche Operation zu verstehen ist, wird dieser teilweise auch von Web-Service-Frameworks als (einziger) Identifikator für die Operation benutzt. Umgekehrt wird von anderen Frameworks der `SOAPAction`-Wert vollständig ignoriert. Diese Eigenschaften können für Angriffe ausgenutzt werden.

Beispielangriff 1

Für dieses Experiment hatte der angegriffene Web Service (System C) zwei Operationen: `op1(string s)` und `op2(int x)`, mit entsprechend benanntem `SOAPAction`-Bezeichner und Nachrichtenelementen. Das angenommene Szenario ist das folgende: Ein Web-Service-Client ruft die Operation `op1` mit folgender SOAP-Nachricht auf.

```
POST /Service.asmx HTTP/1.1
SOAPAction: "op1"

<soap:Envelope>
  <soap:Body>
    <op1>
      <s>Hello</s>
    </op1>
  </soap:Body>
</soap:Envelope>
```

Ein *Janus-Angreifer* (engl. *Man-in-the-Middle*) fängt die Nachricht ab, modifiziert nur das `SOAPAction`-Feld und leitet die Nachricht ansonsten unverändert an den Web Service weiter:

```
POST /Service.asmx HTTP/1.1
SOAPAction: "op2"

<soap:Envelope>
  <soap:Body>
    <op1>
      <s>Hello</s>
    </op1>
  </soap:Body>
</soap:Envelope>
```

Die durch die Nachricht vom Web Service tatsächliche ausgelöste Operation war: `op2(0)`. Hier wurde die Operation also ausschließlich durch das `SOAPAction`-Feld bestimmt und sogar mit den inkompatiblen Parameternamen und -typen ausgeführt.

So kann ein Angreifer also den Aufruf eines Clients nur durch Modifikation des HTTP-Headers auf eine andere Operation umleiten. Dies ist insbesondere dann interessant, wenn die SOAP-Nachricht durch WS-Security gegen Modifikationen gesichert ist. Die Daten im HTTP-Header können dagegen beispielsweise von einem SOAP-Zwischenknoten leicht verändert werden.

Beispielangriff 2

Bei diesem Experiment wurde ein Web Service (System B) mit zwei Operationen `hidden` und `visible` mit gleichnamigen `SOAPAction`-Bezeichnern und Nachrichtenelementen angegriffen. Der Angreifer schickt folgende Nachricht an den Web Service.

```
POST /axis2/testService HTTP/1.1
SOAPAction: "visible"

<soap:Envelope>
  <soap:Body>
    <hidden />
  </soap:Body>
</soap:Envelope>
```

Die ausgeführte Operation war `hidden`. Der Web Service bestimmt also korrekterweise die Operation aus dem Inhalt der SOAP-Nachricht. Dabei wird aber das inkorrekte `SOAPAction`-Feld ignoriert. Damit ist es beispielsweise möglich, die Zugangskontrolle einer Firewall, die Zugriffe auf `hidden` verhindert und auf `visible` zuläßt, zu umgehen. Da Firewalls häufig nur auf dem HTTP-Header operieren, würde die obige Nachricht fälschlicherweise durchgelassen werden.

5.3.2 *XML Injection*

Kurzbeschreibung

Einfügen von XML-Elementen in Textknoten

Klassifizierung

K1 (Protokoll)	XML
K2 (Abweichung)	Abweichung von der Nachrichtensyntax
K5 (Quelle)	einzeln
K6 (Ziel)	Server
K7 (Abstraktion)	protokollspezifisch

Beschreibung

Enthält ein XML-Textknoten die Zeichen „<“ oder „>“ so müssen diese im XML-Dokument durch „`<`“ und „`>`“ ersetzt werden, um nicht mit der Markierung für XML-Kennungen

verwechselt zu werden. XML-Parser wandeln diese Darstellung beim Deserialisieren wieder in die ursprüngliche Form um.

Wird diese Umsetzung nicht durchgeführt, ergibt sich die Möglichkeit nur durch Änderung eines Textknotens die Struktur des Dokuments zu ändern. Das folgende Beispiel illustriert, wie dies bei einem Web-Service-Aufruf ausgenutzt werden kann.

Beispielangriff

Dieses Experiment wurde mit einem Web Service (System C) mit einer Operation `Hello World(int a, int b)` durchgeführt. Dieser wurde mit der folgenden SOAP-Nachricht aufgerufen.

```
<soap:Envelope>
  <soap:Body>
    <HelloWorld>
      <a> <b>1</b> </a>
      <b> 2 </b>
    </HelloWorld>
  </soap:Body>
</soap:Envelope>
```

Diese Nachricht könnte das Ergebnis eines *XML-Injection*-Angriffs sein: In die syntaktisch korrekte Nachricht wurde `<b>1</b>` als Parameter von `a` eingefügt, ohne die spitzen Klammern zu ersetzen. Da die obige Nachricht der Web-Service-Beschreibung widerspricht, sollte sie vom Web Service verworfen werden. Dies war im Experiment aber nicht der Fall. Beachtenswert ist auch die Belegung der Parameter in der Web-Service-Applikation: `a` = 1, `b` = 0. Durch Modifikation des Parameters `a` kann ein Angreifer also die Parameter `a` **und** `b` beeinflussen. Ein Szenario, in dem ein solcher Angriff stattfinden könnte, ist der eines *vertrauenswürdigen* (engl. *trusted*) Web-Service-Client, bei dem ein (nicht vertrauenswürdiger) Benutzer nur den Wert von `a` eingeben kann.

5.3.3 *XML Rewriting*

Kurzbeschreibung

Angriffe, bei denen abgefangene XML-Nachrichten modifiziert und weitergeleitet werden

Klassifizierung

K1 (Protokoll)	Web Service + SOAP-Sicherheit
K2 (Abweichung)	Abweichung von der Nachrichtensemantik
K5 (Quelle)	einzeln
K6 (Ziel)	Server
K7 (Abstraktion)	protokollspezifisch

Beschreibung

Bei dieser Art von Angriffen fängt ein Angreifer eine XML-basierte Nachricht (hier: SOAP-Nachricht) ab, verändert Teile der Nachricht und schickt diese weiter. Durch die Modifikationen

versucht der Angreifer typischerweise Operationen, auf die nur der ursprüngliche Sender Zugriff hat, so aufzurufen, dass der Angreifer dadurch Vorteile hat. Ein Beispiel wäre das Umbiegen einer Geldüberweisung auf das Konto des Angreifers. Natürlich werden derartige sicherheitskritische Nachrichten meist durch kryptographische Maßnahmen geschützt. Allerdings ermöglicht die generische Struktur von SOAP (insbesondere des SOAP-Headers) und die Anwendung von WS-Security auf Teile der SOAP-Nachricht derartige Angriffe trotz Signierung. Einige Beispielen dafür finden sich in [110] und [133].

Beispielangriff

Ein Sender verschickt eine Bestellung, die gegen Veränderung durch Signierung des SOAP-Bodys geschützt ist:

```
<soap:Envelope>
  <soap:Header>
    <wsse:Security>
      <ds:Signature>
        ...
        <ds:Reference URI="#myBody">
          ...
        </ds:Reference>
       ...
      </ds:Signature>
    </wsse:Security>
  </soap:Header>
  <soap:Body wsu:Id="myBody">
    <order>
      <amount>1</amount>
    </order>
  </soap:Body>
</soap:Envelope>
```

Durch die Signatur würde eine Veränderung des SOAP-Bodys durch den Angreifer an der Empfänger-Seite auffallen. Allerdings kann der Angreifer durch geschicktes Verschieben des SOAP-Bodys und Einfügen eines neuen SOAP-Bodys trotzdem beliebige Werte in die SOAP-Nachricht einbringen:

```
<soap:Envelope>
  <soap:Header>
    <wsse:Security>
      <ds:Signature>
        ...
        <ds:Reference URI="#myBody">
          ...
        </ds:Reference>
       ...
      </ds:Signature>
    </wsse:Security>
```

```
    <DummyHeader>
      <soap:Body wsu:Id="myBody">
        <order>
          <amount>1</amount>
        </order>
      </soap:Body>
    </DummyHeader>
  </soap:Header>
  <soap:Body>
    <order>
      <amount>5000</amount>
    </order>
  </soap:Body>
</soap:Envelope>
```

Auf diese Weise ist die Signatur weiterhin gültig, die Bestellung (von 5000 Einheiten) würde beim Empfänger fälschlicherweise als vom ursprünglichen Sender autorisiert angesehen werden.

Die Entdeckung der Modifikation kann dadurch noch weiter erschwert werden, dass dem `DummyHeader` die SOAP-Rolle `none` zugewiesen wird. Weitere Möglichkeiten zum „Überschreiben“ der ursprünglichen Elemente sind das Einfügen eines weiteren `wsse:Security`-Headers oder eines zweiten SOAP-Bodys. Damit bleiben sogar XPath-Referenzen (innerhalb der Nachricht und in einer Sicherheits-Policy) in der modifizierten Nachricht gültig.

5.3.4 *WSDL Scanning*

Kurzbeschreibung

Aufrufen von versteckten Web-Service-Operationen

Klassifizierung

K1 (Protokoll)	WSDL
K2 (Abweichung)	Abweichung von der Nachrichtensemantik
K5 (Quelle)	einzeln
K6 (Ziel)	Server
K7 (Abstraktion)	protokollspezifisch

Beschreibung

Web Services, die innerhalb und außerhalb eines lokalen Netzes einen Dienst erbringen, bieten oftmals auf demselben Endpunkt verschiedene Operationen für die interne und externe Dienstnutzung an. So benötigt beispielsweise ein Online-Shop Operationen, mit denen Kunden Aufträge erteilen können, sowie Operationen zum Verwalten der Aufträge. Natürlich sind nur die ersteren für den externen Zugriff vorgesehen. Da der Web-Service-Endpunkt aber erreichbar ist, kann ein Angreifer auch die internen Operationen aufrufen und beispielsweise versuchen, vertrauliche Daten zu erlangen.

Selbst wenn nur die externen Operationen in der Web-Service-Beschreibung vorhanden sind, so kann der Angreifer mittels der Informationen in der Web-Service-Beschreibung versuchen, die

Daten der ausgeblendeten Operationen zu erraten. Diese Art des Angriffs wird *WSDL Scanning* genannt.

Zur Vermeidung solcher Angriffe sollten interne und externe Operationen nicht auf demselben Web-Service-Endpunkt zur Verfügung gestellt werden. Wenn dies nicht möglich oder praktikabel ist, so müssen die Zugriffe auf die internen Operationen kontrolliert und bei externen Aufrufen abgewiesen werden. Da Zugriffe auf Operationen desselben Endpunkts aber die gleiche Ziel-IP-Adresse, Ziel-TCP-Port und sogar HTTP-URL besitzen, sind herkömmliche Kontrollsysteme, die nicht die SOAP-Nachricht selbst analysieren, nicht in der Lage, Zugriffe auf interne und externe Operationen zu unterscheiden. Die einzige Möglichkeit, auf den Transportschichten Operationen zu erkennen, ist das HTTP-Header-Feld `SOAPAction`. Die Benutzung dieses Eintrags ist aber nicht zu empfehlen und führt zu anderen Möglichkeiten für Angriffe (siehe Abschnitt 5.3.1).

5.4 Konfigurationen

Die Systeme, die in den obigen Beispielen angegriffen wurden, hatten folgende Hardware- und Software-Konfigurationen:

System A Apache Axis 1.4, Apache Tomcat 5.5.20, Java 1.5, Debian Linux 2.6.16, Pentium III, 1 GHz, 512 MB RAM

System B Apache Axis2 1.0, Rampart 1.0, Apache Tomcat 5.5.20, Java 1.5, Debian Linux 2.6.16, Pentium III, 1 GHz, 512 MB RAM

System C Microsoft .NET 2.0, IIS 5.1, Windows XP SP2, Athlon 64 3500+, 2 GB RAM

System D Oracle BPEL Process Manager Server 10.1, Java 1.5, Windows XP SP2, Athlon 64 3500+, 2 GB RAM

Teil II

Erweiterte und effiziente Valdierung

Kapitel 6

Validierung als Sicherheitsmaßnahme

Das vorangegangene Kapitel hat die Verwundbarkeit von Web Services gezeigt. Die verwendeten Technologien und Protokolle führen zum einen zu neuartigen Angriffen gegen Integrität und Vertraulichkeit, zum anderen werden Denial-of-Service-Angriffe möglich, bei denen bereits kleine Nachrichten verheerende Wirkungen auf die Verfügbarkeit der Web Services haben.

Andererseits eröffnen genau diese Technologien auch neue Möglichkeiten zum Schutz vor Angriffen. So ist jeder Web Service durch mehrere Metadaten formal definiert, welche die Funktionsweise, die Nachrichtenformate, die Sicherheitseigenschaften und die Nachrichtenabfolge beschreiben. Diese formalen Beschreibungen können genutzt werden, um die Sicherheit von Web Services zu verbessern und diese vor Angriffen zu schützen. Im ersten Teil dieses Kapitels wird gezeigt, wie sich daraus Sicherheitsmaßnahmen für Web Services ableiten lassen. Speziell bei der Abwehr von DoS-Angriffen spielt der Ressourcenbedarf der Nachrichtenverarbeitung eine entscheidende Rolle. Im zweiten Teil wird daher eine besonders effiziente Realisierung eines Schutzsystem präsentiert.

6.1 Erweiterte Validierung

Wie bereits in vorherigen Kapiteln angeführt, basieren viele Angriffe auf der Verwendung von Nachrichten, die nicht genau dem intendierten Protokoll entsprechen. Bei Web Services wird das Protokoll (im weiteren Sinne) nicht nur durch die Spezifikationen, sondern auch durch einige Metadaten, die jeder Web Service erzeugt und anbietet, explizit und formal definiert. Im Einzelnen sind dies u. a. die folgenden Metadaten:

- Web-Service-Beschreibung
- Sicherheits-Policy
- Nachrichtenfolge-Beschreibung
- Zugriffs-Policy

Dadurch ergibt sich folgende Möglichkeit zum Schutz von Web Services: alle Nachrichten werden gegen die Definitionen validiert, die sich aus diesen Metadaten ergeben. Nur Nachrichten, die den formalen Spezifikationen entsprechen, werden an den Web Service weitergeleitet, alle anderen werden abgelehnt oder verworfen. Dies realisiert die allgemeinen Gegenmaßnahmen „Überprüfung der Eingabe“ und „Beschränkung des Zugriffs“ aus Kapitel 4.2.2.

Konkret kann dies durch die folgenden Validierungsschritte umgesetzt werden.

Schema-Validierung

Die Web-Service-Beschreibung definiert u. a. eine Grammatik, die die gültigen Nachrichten für diesen Web Service syntaktisch beschreibt. Diese Grammatik ist in *XML Schema* [54] verfasst und kann zur Schema-Validierung aller eingehenden (und evtl. ausgehenden) SOAP-Nachrichten verwendet werden.

Mit dieser Maßnahme werden Angriffe abgewehrt, die auf der Abweichung von der Protokollsyntax auf Web-Service-Protokollebene basieren.

Nachrichten-Entschlüsselung

Um verschlüsselte Nachrichtenteile auf Protokollkonformität zu überprüfen, müssen diese entschlüsselt und gegen das Nachrichtenschema validiert werden. Damit werden alle Angriffe, gegen die die Schema-Validierung wirkt (siehe oben), auch dann abgewehrt, falls der Angriff durch Verschlüsselung verschleiert wurde.

Validierung der HTTP-Bindung

Zusätzlich definiert die Web-Service-Beschreibung die Bindung an das HTTP-Protokoll. Auch diese Definition wird bei allen Nachrichten validiert. Dies bedeutet konkret, dass die Endpunkt-URL und der `SOAPAction`-Wert konform zur SOAP-Nachricht sein müssen. Damit werden Angriffe mittels Abweichung von der Protokollsemantik auf der HTTP-Protokollschicht erschwert.

Sicherheits-Policy-Validierung

Alle eingehenden (und evtl. ausgehenden) Nachrichten werden daraufhin überprüft, ob sie der Sicherheits-Policy entsprechen. Mit dieser Maßnahmen werden Angriffe abgewehrt, die auf der Abweichung von der Protokollsyntax auf SOAP-Sicherheits-Protokollebene basieren.

Wesentlicher Bestandteil einer Sicherheits-Policy sind Anforderungen an kryptographische Elemente wie digitale Signatur und Verschlüsselung. Zur Überprüfung der Sicherheits-Policy müssen diese vollständig verarbeitet und überprüft werden.

Validierung der Nachrichtenreihenfolge

Komplexe Web-Service-Applikationen, wie beispielsweise BPEL-Prozesse, definieren zustandsbehaftete Kommunikationsprotokolle. Damit ist nicht jede (syntaktisch korrekte) Web-Service-Nachricht zu jedem Zeitpunkt des Protokolls erlaubt. Solcherlei Nachrichten können zu unerwünschtem Verhalten des Web Services führen und müssen daher erkannt und abgelehnt werden.

Alle ein- und ausgehenden Nachrichten werden daraufhin überprüft, ob die Nachricht in diesem Zustand des Web-Service-Protokolls erlaubt ist. Damit können Angriffe abgewehrt werden, die auf der Abweichung von der Nachrichtenreihenfolge basieren.

Zugriffsbeschränkung

Für jede eingehende Nachricht wird überprüft, ob der Benutzer auf die angeforderte Ressource zugreifen darf. Formal wird also die Erfüllung der Definitionen aus der Zugriffskontroll-Policy validiert.

Zugriffsbeschränkung ist eine wichtige Maßnahme, die grundsätzlich gegen alle Arten von Angriffen wirksam ist. Allerdings ist Zugriffsbeschränkung wirkungslos gegen Angriffe, die ihre Wirkung entfalten, *bevor* die Zugriffskontrolle durchgeführt wird. Dies ist bei Web Services besonders problematisch, da die Zugriffskontrolle typischerweise nicht in einem separaten Protokollschritt durchgeführt wird. Im Allgemeinen ist zur Festellung der Zugriffsberechtigung die Bearbeitung der gesamten Nachricht notwendig.

Ein zweites Problem ist, dass die Zugriffsparameter *Authentifizierungssubjekt* (d. h. der Benutzer) und *Authentifizierungsobjekt* (d. h. die Ressource) für Web Services nicht standardisiert sind.

Diese beiden Probleme müssen bei der Entwicklung eines Zugriffsbeschränkungssystems berücksichtigt werden.

Mit den oben beschriebenen Maßnahmen lassen sich im wesentlichen Angriffe erkennen, bei denen die Angriffsnachrichten in einem Merkmal von der Protokolldefinition abweichen. Aber auch zur Erkennung von Angriffen, bei denen dies nicht der Fall ist, läßt sich die Validierung gegen die formalen Beschreibungen eines Web Service verwenden. Die Grundidee ist dabei, die Metadaten auf Konstrukte zu untersuchen, die bekannterweise Angriffsnachrichten erlauben. Diese Konstrukte werden dann modifiziert bzw. aus den Metadaten entfernt. Die Validierung von Nachrichten erfolgt dann gegen diese beschränkten Definitionen. Damit können bekannte Angriffstypen vom Typ „Normale Protokollausführung“ erkannt werden.

Diese Art der Erkennung hat einige Vorteile gegenüber einem klassischen IDS-System. Zunächst sind viele Angriffe dieses Typs gegen Web-Services auf der Byte-Ebene nicht zu detektieren; sie erfordern eine XML-Verarbeitung der Nachricht. Weiterhin findet die Überprüfung auf bekannte Angriffsmuster bei der oben beschriebenen Methode automatisch durch die (in jedem Fall durchgeführte) Validierung statt. Dabei muss natürlich gewährleistet sein, dass die Validierung gegen die modifizierten Metadaten nicht ressourcenintensiver ist als gegen die ursprünglichen Daten. Und schließlich *können* die Modifikationen auch in die Metadaten aufgenommen werden, die dem Client mitgeteilt werden. Dadurch erfolgt eine eventuelle Ablehnung nur für Nachrichten, die der (allen Parteien bekannten) Protokollbeschreibung widersprechen und (aus Sicht des Web-Service-Clients) nicht willkürlich erscheinen.

Die wichtigste Maßnahme dieser Art ist die Modifikation des Nachrichtenschemas in der Web-Service-Beschreibung. Weiterhin können Sicherheits-Policys nach zu schwachen Forderungen, die Angriffe mit Verletzung von Integrität und Vertraulichkeit erlauben, durchsucht werden. Ein solches Verfahren wird beispielsweise in [16] beschrieben.

Schema-Beschränkung

Durch Modifikation der Nachrichten-Schemata (zusammen mit der Durchsetzung der Schema-Validität) können weitere Angriffstypen verhindert werden.

Zum einen können die Schemata so modifiziert werden, dass sie nur Nachrichten einer bestimmten Größe erlauben. Damit wird der Ressourcenverbrauch bei der Verarbeitung limitiert und damit Angriffe, die auf Ressourcenerschöpfung abzielen, erschwert.

Eine zweite Anwendung der Schema-Beschränkung ist die Abwehr von bekannten Angriffsmustern. Sofern sich diese Muster in der Nachrichtengrammatik ausdrücken lassen, kann die Schema-Definition entsprechend modifiziert werden. Ein einfaches Beispiel ist der Ausschluss bestimmter Zeichenfolge bei Zeichenketten-Datentypen.

Schließlich können Operationen, auf welche kein Zugriff erfolgen soll, aus dem Schema

entfernt werden. Damit setzt die Schema-Validierung automatisch die (generelle) Zugriffsbeschränkung auf diese Operationen durch.

Mit dieser Maßnahme wird u. a. auch die allgemeine DoS-Gegenmaßnahme „Beschränkung des Ressourcenverbrauchs“ umgesetzt. Um den Ressourcenverbrauch durch kryptographische Elemente zu beschränken, ist für die Sicherheits-Policy eine striktere Interpretation erforderlich als die, die Spezifikation vorsieht.

Strikte Policy-Interpretation

Es werden nur Nachrichten akzeptiert, deren Sicherheitselemente *genau* der Sicherheits-Policy entsprechen. Nachrichten mit überflüssigen Sicherheitselementen werden also abgelehnt. Dadurch kann der Ressourcenverbrauch für die Verarbeitung der Sicherheitselemente beschränkt und damit Ressourcenangriffe auf der SOAP-Sicherheits-Protokollebene erschwert werden.

Die strenge Interpretation der Sicherheits-Policy stellt dabei keine Einschränkung für die praktische Verwendung der SOAP-Sicherheit dar. Dies wird später diskutiert werden.

6.2 Effiziente Validierung

6.2.1 Verarbeitungsmodell

Wie bereits erwähnt, spielt für die Wirksamkeit einer Gegenmaßnahme gegen Denial-of-Service-Angriffe die Umsetzung dieser Gegenmaßnahme eine entscheidende Rolle. Diese muss so ausgeführt werden, dass das Gegenmaßnahmensystem nicht selbst zum Opfer eines DoS-Angriffs wird. Die wichtigste Voraussetzung dafür ist, dass der Ressourcenverbrauch des Gegenmaßnahmensystems dieses nicht selbst für Ressourcenverbrauchsangriffe anfällig macht. Daraus ergibt sich die folgende (ideale) Forderung an ein System zur Abwehr von Denial-of-Service-Angriffen gegen Web Services:

> *Die Laufzeit und der Speicherbedarf für die Überprüfung einer Nachricht sind durch Werte beschränkt, die unabhängig von der Nachricht – und damit vom Angreifer – sind.*

Dies soll für die in dieser Arbeit entwickelten Realisierung der o. g. Gegenmaßnahmen als Ziel angesetzt werden. Das bedeutet, dass Laufzeit R und Speicherbedarf M beschränkt sein sollen, also $R \in \mathcal{O}(1)$ und $M \in \mathcal{O}(1)$. Zusätzlich soll für Nachrichten, die als gültig erkannt werden, die Laufzeit in $\mathcal{O}(n)$ liegen (wobei n die Größe der Nachricht darstellt). Ein weiteres Ziel für die Implementierung der Gegenmaßnahmen ist, nicht nur die komplexitätttheoretischen, sondern auch die tatsächlichen Laufzeiten klein zu halten.

Für die Abwehr von Denial-of-Service-Angriffen sind natürlich noch weitere Bedingungen zu erfüllen. So ist zum Schutz vor Angriffen mit Ressourcenzerstörung beispielsweise eine korrekte Implementierung und ein Laufzeitsystem ohne Sicherheitslücken notwendig; das wird im Folgenden nicht weiter betrachtet. Weiterhin berücksichtigen die oben genannten Forderungen nur eine einzelne Nachricht. Angriffe, die auf dem sogenannten *Fluten* mit einer großen Anzahl von Nachrichten basieren, werden in dieser Arbeit nicht explizit untersucht.

Das Erreichen der Forderungen an Laufzeit und Speicherbedarf ist mit einer baumbasierten Verarbeitungsmethode nicht möglich. In Kapitel 2.3.2 wurde gezeigt, dass ein Angreifer theoretisch beliebige Verarbeitungslaufzeiten und beliebigen Speicherbedarf erzwingen kann. Dieser

Überlegung folgend muss ein System zur Realisierung der Gegenmaßnahmen eine ereignisbasierte Verarbeitungsmethodik verwenden.

Neben dieser theoretischen Überlegung hat ereignisbasierte XML-Verarbeitung natürlich auch in der praktischen Verwendung deutliche Vorteile bezüglich des Ressourcenverbrauchs, insbesondere bezogen auf den Speicherverbrauch [84].

6.2.2 Verarbeitungskomponenten

Im Folgenden werden die Komponenten vorgestellt, die die im vorherigen Abschnitt entwickelten Schutzmaßnahmen realisieren. Alle Komponenten basieren dabei auf dem ereignisbasierten Verarbeitungsmodell.

Schema-Validator

Diese Komponente realisiert die Maßnahmen, die auf Schema-Validierung basieren. Dazu werden aus den Web-Service-Beschreibungen Schemata generiert und zum Ausschluss von bösartigen Nachrichten verschärft. Die daraus resultierenden Schemata fließen in die Web-Service-Beschreibungen ein, die dann publiziert werden. Außerdem werden eingehende Nachricht gegen diese Schemata validiert.

Sicherheitskomponte

Die Sicherheitskomponente enthält drei Teilkomponenten:

Die erste Teilkomponente ist für die Verarbeitung der WS-Security-Elemente zuständig. Konkret bedeutet dies die Überprüfung der WS-Security-Elemente auf syntaktische und semantische Korrektheit (beispielsweise: Gültigkeit der Zeitstempel, Gültigkeit der Signaturen usw.) und die Entschlüsselung von verschlüsselten Elementen. Mit den ersten beiden Maßnahmen wird die Policy-Validierung vorbereitet. Die letzte Maßnahme ist notwendig, um verschlüsselte Inhalte auf Schema-Validität zu überprüfen.

Die zweite Teilkomponente ist für die Überprüfung der Konformität zur Sicherheits-Policy nach WS-SecurityPolicy zuständig. Sie basiert auf der Verarbeitung der WS-Security-Elemente durch die WS-Security-Komponente.

Die dritte Teilkomponente dient der Zugriffskontrolle. Dabei wird mittels Zugriffskontroll-Policy geprüft, ob der Zugriff erlaubt ist. Da der Benutzer durch kryptographische Methoden authentifiziert wird, benötigt diese Komponente die Ergebnisse der ersten beiden Teilkomponenten.

Zustandsverfolgung

Diese Komponente überprüft, ob eingehende Nachrichten konform zum aktuellen Zustand des Web-Service-Protokolls sind. Als Protokollbeschreibung wird dabei BPEL verwendet. Eine BPEL-Prozess-Beschreibung legt u. a. ein zustandsbehaftetes Web-Service-Protokoll fest. Entspricht eine eingehende Nachricht dem aktuellen Zustand, so wird sie akzeptiert und der Protokollzustand aktualisiert.

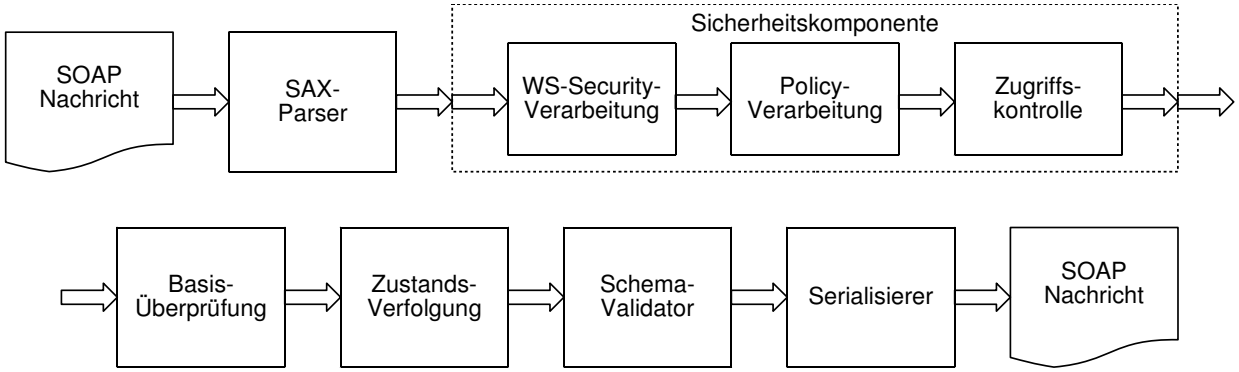


Abbildung 6.1: Architektur für Web-Service-Schutzsystem

Basis-Überprüfung

In dieser Komponente wird die Konformität des HTTP-Protokolls überprüft. Zum einen wird die Korrektheit des `SOAPAction`-Felds getestet, d. h. die Übereinstimmung der im `SOAPAction`-Feld deklarierten Operation mit der tatsächlichen Operation in der SOAP-Nachricht. Die tatsächliche Operation ergibt sich aus dem Element

```
/env:Envelope/env:Body/*
```

in der SOAP-Nachricht. Der dazugehörige Sollwert für `SOAPAction` ergibt sich aus dem Attribut

```
/wsdl:definitions/wsdl:binding/wsdl:operation/
/soap:operation/@soapAction
```

Dieser Wert wird mit dem im HTTP-Header angegebenen Wert verglichen.

Mit dieser Basis-Überprüfung können bereits *SOAP-Action-Spoofing*-Angriffe erkannt und damit abgewehrt werden. Diese (einfache) Überprüfung wird im Folgenden nicht weiter behandelt.

6.2.3 Gesamtarchitektur

Die im vorherigen Abschnitt vorgestellten Komponenten lassen sich zu einem Gesamtschutzsystem für Web-Services kombinieren. Aufgrund der ereignisbasierten Verarbeitung können die Komponenten innerhalb der Verarbeitungskette ohne Einfluss auf die Gefährdung durch Ressourcenangriffe verschoben werden (siehe dazu auch Kapitel 2).

Aus funktionalen Gesichtspunkten ergibt sich die Reihenfolge, wie sie Abbildung 6.1 zeigt. Man erkennt, dass die Basis-Überprüfung, die Zustandsverfolgung und der Schema-Validator nach der Sicherheitskomponente angeordnet sind. Das ist darin begründet, dass diese auf auf den unverschlüsselten Elementen operieren, welche von der WS-Security-Komponente erzeugt werden.

Die Ereignisse, die vollständig die obige Kette durchlaufen haben, könnten sofort weiterverarbeitet oder (zu einem Teildokument serialisiert) über das Netzwerk verschickt werden. Dies führt aber zu Problemen, falls das SOAP-Dokument während der weiteren Verarbeitung als ungültig erkannt wird. Daher werden alle Ereignisse serialisiert und in einem Puffer zwischengespeichert. Sobald das SOAP-Dokument vollständig überprüft ist, wird der Inhalt des

Puffers weiterverarbeitet, also beispielsweise an den Web-Service-Server geschickt. Die Speicherung des SOAP-Dokuments in serialisierter Form ist dabei sehr speichereffizient, deutlich effizienter beispielsweise als ein DOM-Baum.

Ein solches Schutzsystem kann als eigenständige Web-Service-Firewall eingesetzt werden, aber auch als Vorstufe bzw. Bestandteil eines Web-Service-Frameworks fungieren. Im Folgenden wird vom Szenario einer Web-Service-Firewall ausgegangen.

In den folgenden Kapiteln werden (in der Reihenfolge wie in Abschnitt 6.2.2) diese Komponenten zur ereignisbasierten Überprüfung von SOAP-Nachrichten im Detail beschrieben.

Kapitel 7

Schema-Validierung

7.1 Einführung

Wie im vorherigen Kapitel erläutert, ist die Überprüfung von eingehenden Nachrichten gegen ein Nachrichtenschema (d. h. eine Grammatik) eine wichtige Möglichkeit zur Abwehr von Angriffen. Ein solches Schema für eingehende und ausgehende Nachrichten kann aus der Web-Service-Beschreibung gewonnen werden. Wird die Validierung gegen dieses Schema zur Abwehr gegen Angriffe eingesetzt, so ist damit zunächst nur die Erkennung von Angriffen des Typs „Abweichung von der Nachrichtensyntax“ möglich. Um auch Angriffe erkennen zu können, die mit syntaktisch korrekten Nachrichten arbeiten, dabei aber einem anderen bekannten Angriffsmuster entsprechen, ist es notwendig, die Web-Service-Beschreibung so anzupassen, dass das daraus generierte Schema auch diese Angriffsnachrichten ausschließt.

Für die Gegenmaßnahme „Schema“ werden zwei Komponenten benötigt: der *Schema-Generator* als „Offline“-Komponente und der *Schema-Validator*, der zur Laufzeit eingehende Nachrichten gegen das generierte Schema überprüft. Der Generator erfüllt dabei folgende Teilaufgaben:

- Generierung des Schemas aus der Web-Service-Beschreibung
- Überprüfung des Schemas auf Konstrukte, die bekannte gefährliche Nachrichten erlauben würden oder eine Erkennung solcher Nachrichten verhindern
- Einschränkung des Schemas, um bekannte gefährliche Nachrichten auszuschließen

Als Schemasprache für ein solches Schutzsystem bietet sich aus verschiedenen Gründen *XML Schema* an.

Zunächst ist *XML Schema* die weitverbreiteste Grammatiksprache für XML. Dadurch existiert eine große Zahl von Programmen und Bibliotheken zum Bearbeiten von Schema-Dokumenten und zum Validieren eines XML-Dokuments auf Konformität bezüglich eines Schemas. Desweiteren wird *XML Schema* auch in der WSDL verwendet, um die Elemente und Datentypen der SOAP-Nachrichten des Web Services zu definieren. Das erleichtert die Generierung des Nachrichtenschemas aus einer Web-Service-Beschreibung. Schließlich bietet *XML Schema* vielfältige Möglichkeiten ein Schema einzuschränken und dadurch Nachrichten mit bekannten Angriffsmustern aus der vom Schema induzierten Nachrichtenmenge auszuschließen.

Im Folgenden wird unter *Schema* eine Menge von *XML-Schema*-Dokumenten verstanden, die einen Typ von XML-Dokumenten beschreiben (also beispielsweise die Menge der validen

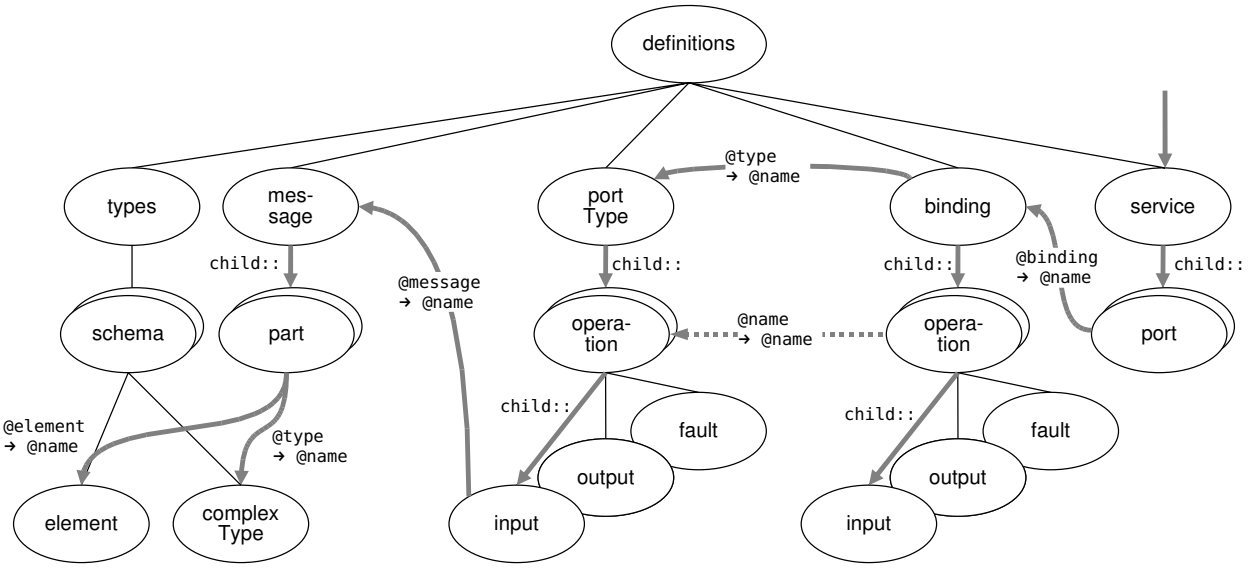


Abbildung 7.1: Durchlauf durch eine WSDL zur Schema-Generierung (angelehnt an [159])

SOAP-Dokumente eines Web Services). Mehr als ein Schema-Dokument wird immer dann benötigt, wenn die Elemente des XML-Dokuments in verschiedenen Namensräumen liegen.

7.2 Schema-Generierung

Eine wichtige Voraussetzung für die praktische Einsetzbarkeit der Schema-Validierung als Schutzsystem ist die automatische oder zumindest semi-automatische Generierung von Schemata. Durch die formale Beschreibung der Web-Service-Schnittstelle durch die Web-Service-Beschreibung ist das bei Web Services möglich.

Die folgenden Ausführungen zur Generierung von Nachrichten-Schemata aus einer Web-Service-Beschreibung beziehen sich auf den Standard WSDL 1.1 [34]. Bei der Verwendung von WSDL 2.0 ist dieses Verfahren geringfügig anders [20].

Die von einem Web Service definierten SOAP-Nachrichten sind im WSDL-Dokument nicht explizit beschrieben, sondern jeweils durch eine Menge von untereinander referenzierten WSDL-Elementen abgebildet. Abbildung 7.1 zeigt die Reihenfolge eines Durchlaufs durch eine Web-Service-Beschreibung zur Sammlung aller Informationen für die definierte SOAP-Nachricht und damit auch für das entsprechende Schema. Die Zeichnung illustriert das Verfahren für eine eingehende Nachricht. Für ausgehende Nachrichten ist das Verfahren analog, zusätzlich sollte das Schema für ausgehende Nachrichten aber auch SOAP-Fehlernachrichten (engl. *SOAPFault*) erlauben. Zur Behandlung von SOAP-Fehlernachrichten siehe auch [157]. Die Beschriftung der fettgedruckten Pfeile beschreibt (in XPath-ähnlicher Notation) die Navigation zum jeweils nächsten Knoten im WSDL-Dokument. Dabei bedeutet `child::` die Iteration über alle Kinder des Ausgangsknotens. Die Schreibweise

$$A \xrightarrow{\texttt{@a1 -> @a2}} B$$

bedeutet die Navigation zu einem Element B, dessen Attribut `a2` den gleichen Wert wie das Attribut `a1` von A hat. Dabei ist zu beachten, dass die Werte von `name`-Attributen vom Typ

`xsd:NCName` sind, also keinen Namensraum tragen. Für diese Attribute ist als Namensraum der *Zielnamensraum* (engl. *Target Namespace*) der Web-Service-Beschreibung anzunehmen.

Beim Durchlauf durch die Web-Service-Beschreibung werden alle Informationen gesammelt, um ein entsprechendes Schema zu konstruieren. Wie in der Zeichnung zu erkennen ist, ist der Startpunkt dabei das `service`-Element, welches eine Menge von `port`-Elementen enthält. Jedem Port ist eine eindeutige HTTP-URL zugeordnet. Daher ist es sinnvoll, pro Port ein Schema zu generieren. Im Folgenden wird nur noch die Generierung eines Schemas für einen Port betrachtet.

Beim Durchlauf durch die Web-Service-Beschreibung werden auch folgende Werte eingelesen, die nicht in der obigen Zeichnung enthalten sind. Diese sind zum Teil für die Schema-Generierung und zum Teil für das Gesamtschutzsystem von Bedeutung.

Netzwerkadresse: Das Attribut `/wsdl:definitions/wsdl:service/wsdl:port/soap:address/@location` enthält die HTTP-URL, unter der der Web Service erreichbar ist.

Nachrichtenstil: Das Attribut `/wsdl:definitions/wsdl:binding/soap:binding/@style` enthält den Nachrichtenstil. Mögliche Werte sind `document` und `rpc` [37].

SOAPAction: Das Attribut `/wsdl:definitions/wsdl:binding/wsdl:operation/soap:operation/@soapAction` enthält einen Identifikator der jeweiligen Operation, der in dem HTTP-Header verwendet wird.

Nachrichtenkodierung: Das Attribut `/wsdl:definitions/wsdl:binding/wsdl:operation/wsdl:input/soap:body/@use` muss den Wert `literal` haben.

Der Nachrichtenstil definiert maßgeblich die Struktur der SOAP-Nachrichten und damit auch der entsprechenden Schemata. Im Folgenden werden die Schemata für die beiden möglichen Stile angegeben.

Nachrichtenstil: `document`

Seien $\mathit{element}_1$ bis $\mathit{element}_k$ die Namen (d. h. der Wert des `name`-Attributs) der `element`-Elemente, die gemäß dem Durchlauf aus Zeichnung 7.1 referenziert werden und $\mathtt{ns}_1$ bis $\mathtt{ns}_k$ Präfixe für die Namensräume, die durch das `targetNamespace`-Attribut des jeweiligen Schemas definiert sind. Abbildung 7.3 zeigt das abstrakte Schema für die SOAP-Nachricht. Dabei stellen rechteckig durchgehend umrahmte Knoten Elementdefinitionen (`xsd:element`) und rechteckig gestrichelt umrahmte Knoten Typdefitionen (`xsd:complexType`) dar.

Nachrichtenstil: `rpc`

Seien $\mathit{operation}_1$ bis $\mathit{operation}_n$ die Namen der `operation`-Elemente, die gemäß dem Durchlauf aus Zeichnung 7.1 referenziert werden, und $\mathtt{ns}_1$ bis $\mathtt{ns}_k$ Präfixe für die Namensräume, die durch das Attribut `/wsdl:definitions/wsdl:binding/wsdl:operation/wsdl:input/soap:body/@namespace` definiert sind. Seien weiterhin zu $i \in \{1, \ldots, n\}$ part_i^1 bis $\mathit{part}_i^{m_i}$ die Namen der `part`-Elemente, die von der Operation $\mathit{operation}_i$ referenziert werden. Sei schließlich für jedes $j \in \{1, \ldots, m_i\}$ type_i^j das `type`-Element, das von part_i^j aus referenziert wird und $\mathtt{ns}_i^j$ ein Präfix für den Namensraum, der durch das `targetNamespace`-Attribut des jeweiligen Schemas definiert ist. Das abstrakte Schema für die SOAP-Nachricht sieht dann wie in Abbildung 7.3 gezeigt aus.

Zu beachten ist, dass die `part`-Elemente im leeren Namensraum liegen.

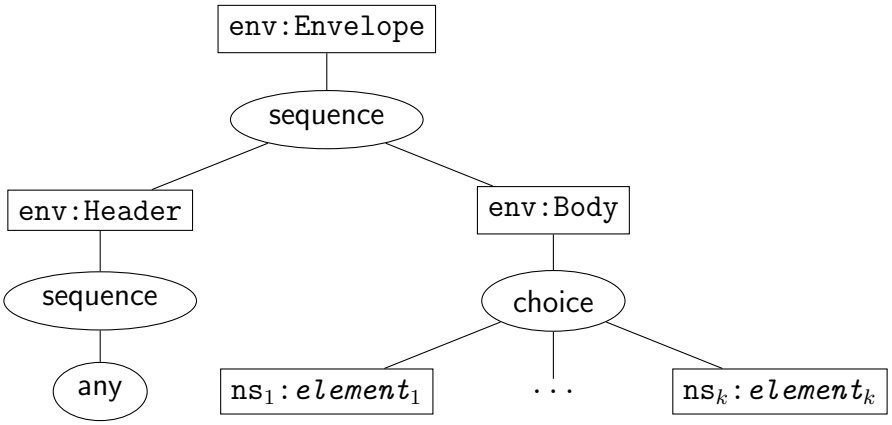


Abbildung 7.2: Struktur des generierten Schemas für den `document`-Nachrichtenstil

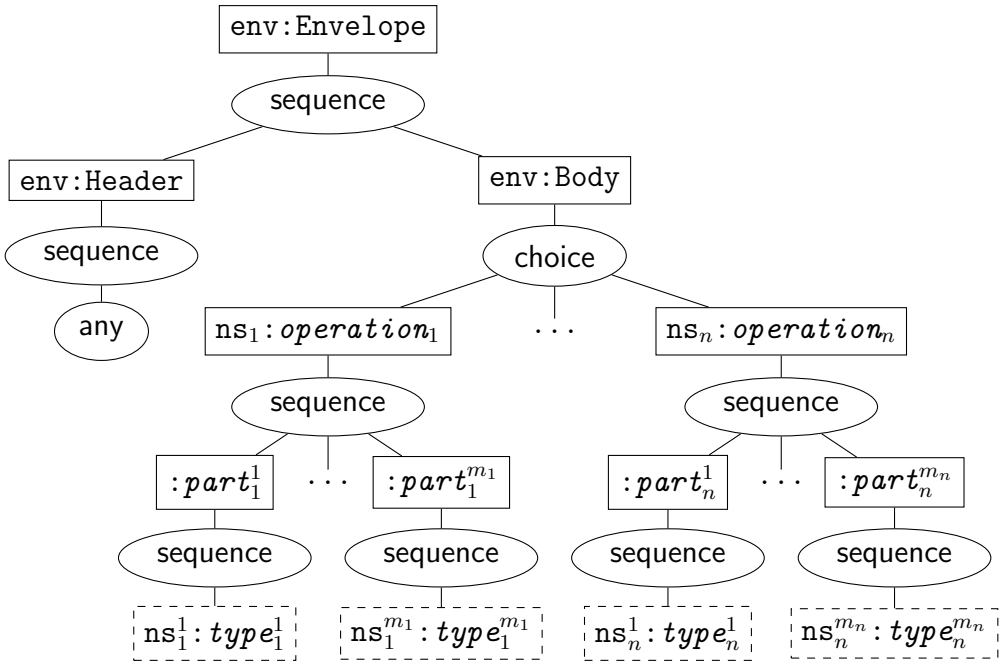


Abbildung 7.3: Struktur des generierten Schemas für den `rpc`-Nachrichtenstil

7.3 Schema-Überprüfung

Damit die Schema-Validierung wirkungsvoll zur Abwehr von Angriffen eingesetzt werden kann, müssen die aus den WSDL-Dokumenten generierten Schemata zwei Anforderungen erfüllen. Zum einen muss das Schema möglichst strikt sein, d. h. die Menge der zu diesem Schema validen XML-Dokumente sollte minimal sein ohne die für die Erbringung des Web Services notwendigen SOAP-Nachrichten auszuschließen. Zum anderen darf zur Vermeidung von Angriffen mit übergroßen Nachrichten das Schema keinerlei unbeschränkte Nachrichtenteile erlauben.

Als zweiten Schritt der Schema-Verarbeitung wird das Schema daher auf kritische Konstrukte bzgl. dieser Eigenschaften überprüft [90]. Diese Konstrukte sind im Einzelnen:

- Benutzung von unspezifischen Schemadefinitionen [150, 117]:
  - `xsd:any` als *Elementdeklaration* (engl. *Element Declaration*)
  - `xsd:anyType` als *komplexe Typdefinition* (engl. *Complex Type Definition*)
  - `xsd:anySimpleType` als *einfache Typdefinition* (engl. *Simple Type Definition*)
- rekursive Definition von Elementen oder Typen
- unbeschränkte Listen durch Verwendung des Attributes `maxOccurs` mit dem Wert `unbounded`
- Benutzung von `simpleType`-Datentypen aus *XML-Schema* [17] ohne zusätzliche Beschränkungen

Unspezifische Definition

Die Verwendung von unspezifischen Element- oder Typdefinitionen widerspricht dem Grundsatz der strikten Nachrichtendefinition. Da an diesen Stellen im XML-Dokument beliebige (oder zumindest sehr frei wählbare) XML-Elemente erlaubt sind, sind bösartige Nachrichten wie beim *XML-Injection*-Angriff (siehe Kapitel 5.3.2) valide bezüglich des Schemas und können nicht erkannt werden. Weiterhin können diese auch *Platzhalter* (engl. *Wildcards*) genannten Konstrukte rekursive und unbeschränkte Definitionen (s. u.) repräsentieren und damit Nachrichten erlauben, die bei *Oversize Payload*-Angriffen (siehe Kapitel 5.2.1) verwendet werden.

Rekursive Definition

Das folgende Beispiel zeigt eine rekursive Definition eines Elements.

```
...
<xsd:element name="x" type="y"/>
...
<xsd:complexType name="y">
  <xsd:choice>
    <xsd:element ref="x"/>
    <xsd:element name="z" type="xsd:string"/>
  </xsd:choice>
</xsd:complexType>
...
```

Dieses Schema erlaubt unbeschränkt große und unbeschränkt tief geschachtelte XML-Dokumente, was in der Praxis noch gefährlicher bezüglich der Ressourcenangriffe ist (siehe *Coercive Parsing*, Kapitel 5.2.2).

Unbeschränkte Listen

Eine Liste, also eine mehrfache Wiederholung eines Elements oder einer Folge von Elementen, kann in *XML Schema* mit den *Kardinalitätsgrenzwerten* (engl. *Occurrence Constraints*) `minOccurs` und `maxOccurs` als Attribut in den Konstrukten `xsd:element`, `xsd:group`, `xsd:sequence` oder `xsd:choice` definiert werden. Dabei kann der obere Grenzwert einen Wert aus $\mathbb{N} \cup \{\texttt{unbounded}\}$ annehmen. Da unbeschränkte Listen sich für *Oversize Payload*-Angriffe missbrauchen lassen, darf im Schema kein `maxOccurs`-Attribut den Wert `unbounded` annehmen.

Dieses wird aber sehr oft verwendet. Von 440 untersuchten WSDL-Dokumenten[1] enthielten ca. 200 Web-Service-Beschreibungen unbeschränkte Listen. Ein Grund für die häufige Verwendung ist, dass WSDL-Dokumente meist automatisch von den jeweiligen Web-Service-Frameworks generiert werden und dabei für Datenfelder (engl. *Arrays*) standardmäßig ein Element mit `minOccurs="0"` und `maxOccurs="unbounded"` erzeugt wird.

Unbeschränkte einfache Typdefinitionen

Mit Ausnahme einiger Zeit-, Datums- und numerischen Datentypen sind fast alle in *XML Schema* vordefinierten (engl. *built-in*) Datentypen nicht in der Größe beschränkt. Enthält ein Schema solche Datentypen ohne weitere Beschränkungen, so sind Nachrichten mit beliebig großem XML-Textknoten valide bezüglich dieses Schemas.

7.4 Schema-Modifikation

Wird in einem Schema eines der oben genannten Konstrukte gefunden, so ist dieses Schema ungeeignet für die Abwehr von Angriffen mittels Schema-Validierung und muss modifiziert werden. Ein großer Teil dieser Konstrukte kann, zum Teil nach Konfiguration mit bestimmten Parametern, automatisch entschärft werden. Das wird im Folgenden beschrieben.

Schemakonstrukte, die unbeschränkte Nachrichtenteile erlauben, können automatisch durch beschränkte ersetzt werden. Dafür ist es allerdings notwendig, die entsprechenden Schranken vorher festzulegen. Diese Grenzwerte sind:

- maximale Listenlänge l
- maximale Größe von XML-Textknoten c
- maximale Schachtelungstiefe pro Element d

Diese Werte können entweder pauschal, pro Web Service, pro Operation oder pro XML-Element bzw. XML-Typ festgelegt werden. Damit können dann folgende Modifikationen am Schema vorgenommen werden.

[1] Quelle: XMethods, Januar 2007.

Listenbeschränkung

Um die maximale Länge von Listen zu beschränken, wird einem unbeschränkten `maxOccurs`-Attribut ein endlicher Wert zugewiesen:

$$\texttt{maxOccurs="unbounded"} \quad \rightsquigarrow \quad \texttt{maxOccurs="}l\texttt{"}$$

Beschränkung der Rekursionstiefe

XML Schema bietet keine Möglichkeit, die Rekursionstiefe direkt zu beschränken. Für eine endliche Zahl d kann eine rekursive Definition wie im Beispiel oben allerdings durch Einfügen zusätzlicher Element- bzw. Typdefinitionen umgewandelt werden in eine Definition, die nur Fragmente mit Schachtelungstiefe $\leq d$ erlaubt.

Für $d = 2$ kann das Schema aus dem Beispiel aus Abschnitt 7.3 wie folgt modifiziert werden:

```
<xsd:element name="x" type="y"/>

<xsd:complexType name="y">
  <xsd:choice>
    <xsd:element name="x" type="z"/>
    <xsd:element name="z" type="xsd:string"/>
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="z">
  <xsd:choice>
    <xsd:element name="z" type="xsd:string"/>
  </xsd:choice>
</xsd:complexType>
```

Dieses Schema erlaubt nur XML-Dokumente mit zwei ineinander geschachtelten `x`-Elementen. In ähnlicher Weise kann auch die maximale Schachtelungstiefe des Gesamtdokuments beschränkt werden.

Beschränkung von einfachen Typdefinitionen

Einfache Datentypen können mittels Schema-Facetten (engl. *facet*) in der Größe beschränkt werden. Die einfachste Einschränkung erlaubt dabei die Facette `maxLength`. So kann beispielsweise der Datentyp `xsd:string` durch folgende Typdefinition ersetzt werden, die nur noch Zeichenketten mit einer beschränkten Länge erlaubt:

```
<xsd:simpleType name="length-restricted-string">
  <xsd:restriction base="xsd:string">
    <xsd:maxLength value="100"/>
  </xsd:restriction>
</xsd:simpleType>
```

Eliminierung unspezifischer Definitionen

Die generierten Schemata in Abschnitt 7.2 zeigen zweierlei Arten von unspezifischen Definitionen. Zum einen können solche Definitionen bereits in der `wsdl:types`-Sektion der Web-Service-Beschreibung vorhanden sein und damit das Schema für den SOAP-Body aufweichen. Diese Art der unspezifischen Definition lässt sich nicht automatisch korrigieren.

Zum anderen wird bei der Schema-Generierung selbst eine `xsd:any`-Elementdefinition im SOAP-Header eingefügt. Dies ist notwendig, da der SOAP-Header beliebige nicht-funktionale Zusatzinformationen enthalten kann. In der Praxis können die möglichen Header-Blöcke aber oftmals problemlos eingeschränkt werden. In diesem Fall wird die `xsd:any`-Elementdefinition durch Definitionen für die erlaubten Elemente ersetzt. So definiert das folgende Schema-Fragment einen SOAP-Header, der nur Elemente aus WS-Security und (teilweise) WS-Addressing enthalten darf.

```
<xsd:import namespace="http://www.w3.org/2005/02/addressing" .../>
<xsd:import namespace="http://docs.oasis-open.org/wss/2004/01/oasis-
                       200401-wss-wssecurity-secext-1.0.xsd" .../>
...
<xsd:element minOccurs="0" name="Header">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element minOccurs="0" ref="wsa:Action"/>
      <xsd:element minOccurs="0" ref="wsa:To"/>
      <xsd:element minOccurs="0" ref="wsa:From"/>
      <xsd:element minOccurs="0" ref="wsa:ReplyTo"/>
      <xsd:element minOccurs="0" ref="wsse:Security"/>
    </xsd:sequence>
    <xsd:anyAttribute namespace="##other"/>
  </xsd:complexType>
</xsd:element>
```

Entfernen versteckter Operationen

Zur Vermeidung von *WSDL-Scanning*-Angriffen (siehe Kapitel 5.3.4) können Operationen, die nicht aufgerufen werden sollen (zumindest nicht von Nachrichten, die durch das Schutzsystem überprüft werden), aus dem Schema entfernt werden. Damit sind Nachrichten, die diese Operation enthalten, nicht valide und werden abgelehnt.

Ausschließen bekannter Angriffsmuster

Desweiteren kann das Schema weitere Einschränkungen von einfachen Typen enthalten, um Angriffe gegen die Web-Service-Applikation oder -Datenbank zu verhindern. Ein Beispiel für einen solchen Angriff ist der *SQL-Injection*-Angriff [4]. Bei einer Ausprägung dieses Angriffs versucht der Angreifer durch Einfügen von *Escaping*-Zeichen, dem SQL-Befehlstrenner „;“ und SQL-Befehlen in Web-Service-Parametern ein nicht intendiertes SQL-Kommando an die Datenbank zu erzeugen und damit Daten zu modifizieren oder auszulesen. Zur Vermeidung solcher Angriffe können Eingabeparameter auf solche Zeichenfolgen untersucht werden. Diese Überprüfung kann

während der Schema-Validierung durchgeführt werden, falls das Schema entsprechend solcher bekannter Angriffsmuster angepasst wird. Zur Erkennung solcher Muster können Zeichenketten mittels des `xsd:pattern`-Konstrukts auf bestimmte reguläre Ausdrücke eingeschränkt werden.

7.5 Ereignisgesteuerte Validierung

Die Laufzeitkomponente der Schema-Schutzkomponente ist ein ereignisgesteuerter Schema-Validator. Dieser wird einmalig mit dem aus der Web-Service-Beschreibung generierten Schema konfiguriert und erzeugt aus dem Schema eine spezielle Art endlicher Automaten [135]. Danach wird jede eingehende SOAP-Nachricht (als SAX-Ereignis-Strom) gegen dieses Schema validiert. Wird eine Verletzung des Schemas festgestellt, so wird die weitere Verarbeitung sofort abgebrochen.

Der eingesetzte Schema-Validator ist dabei insbesondere in der Lage, die durch die Schema-Modifikation erzeugten Kardinalitätsgrenzwerte der Form `maxOccurs="`l`"` effizient zu behandeln. Dies ist bei vielen verfügbaren Implementierungen nicht der Fall.

Konkret hat der Schema-Validator folgende Eigenschaften bezüglich Laufzeit und Speicherverbrauch. Der Speicherverbrauch des Validierungsautomaten hängt nur von dem Schema ab und ist unabhängig von der Größe des zu validierenden Dokuments. Damit ist die Forderung $M \in \mathcal{O}(1)$ erfüllt. Die Laufzeit ist linear abhängig von der Größe des Dokuments $R \in \mathcal{O}(n)$. Durch die Beschränkung des Schemas (siehe Abschnitt 7.4) werden nur beschränkt große Dokumente akzeptiert. Dadurch, dass der Algorithmus abbricht, sobald ein Dokument eine der Beschränkungen verletzt, ist also auch $R \in \mathcal{O}(1)$. Damit ist im Übrigen auch die Laufzeit und der Speicherverbrauch des Serialisierers beschränkt.

7.6 Schutzwirkung gegen Angriffe

Die Schema-Validierung erkennt (und verhindert dadurch) eine Vielzahl von Angriffen, die in Kapitel 5 vorgestellt wurden.

Der *Oversized-SOAP-Message*-Angriff ist durch das beschränkte Schema nicht mehr möglich. Die strenge Schema-Überprüfung erkennt weiterhin *XML-Injection*-Angriffe. Durch das Entfernen versteckter Operationen auf dem Nachrichtenschema, sind Aufrufe dieser Operationen Schema-invalide und werden verworfen. Damit sind *WSDL-Scanning*-Angriffe nicht mehr möglich.

Einige weitere Angriffe können durch die Schema-Validierung nicht vollständig verhindert, aber zumindest erschwert werden.

Coercive-Parsing-Angriffe, die auf Schema-invaliden Nachrichten basieren, werden ebenfalls verhindert. Weiterhin möglich sind Angriffe, die den Parser mit Schema-validen Nachrichten angreifen. Beispiele dafür sind XML-Elemente mit vielen (überflüssigen) Präfix-Definitionen. *XML-Rewriting*-Angriffe können durch strenge Schema-Validierung erschwert werden. Sie verhindert beispielsweise das Einfügen eines zweiten Body-Elements oder einer zusätzlichen Operation. Werden auch für den SOAP-Header Restriktionen in das Schema eingefügt (wie in Abschnitt 7.4 vorgestellt), so ist auch das Einfügen eines neuen Header-Blocks nur noch eingeschränkt möglich.

Schließlich kann die Schema-Validierung auch eingesetzt werden, um Angriffsmuster in Text-Inhalten zu erkennen. So lassen sich beispielsweise *SQL-Injection*-Angriffe verhindern.

Kapitel 8

WS-Security-Verarbeitung

8.1 Einführung

In dieser Komponente werden die WS-Security-Elemente innerhalb der SOAP-Nachricht verarbeitet [71]. Dabei werden zwei Hauptfunktionen wahrgenommen. Zum einen werden die WS-Security-Elemente auf Korrektheit überprüft. Dies umschließt im Wesentlichen die Überprüfung der syntaktischen Korrektheit der WS-Security-Elemente[1] und die Verifizierung von XML-Signaturen. Zum anderen werden verschlüsselte Blöcke entschlüsselt, um deren weitere Validierung zu ermöglichen. Im Gegensatz zu den anderen Komponenten des Schutzsystems operiert die WS-Security-Komponente damit nicht nur als Validator, sondern auch als Transformator. Auf die ereignisbasierte Verarbeitung übertragen bedeutet das, dass die eingehenden Ereignisse verschieden von den ausgehenden Ereignisse sind. So werden Parsingereignisse, die das Auftreten verschlüsselter Blöcke anzeigen, verschluckt und dafür neue Ereignisse durch das Parsen der entschlüsselten Blöcke erzeugt. Zusätzlich werden auch neue Meta-Ereignisse produziert, die keinen Teil der SOAP-Nachricht repräsentieren, sondern Ergebnisse der Verarbeitung an nachfolgende Komponenten transportieren. Ein Beispiel ist das Ereignis signedElement, mit dem kommuniziert wird, dass ein Element in der SOAP-Nachricht signiert war.

Die ereignisbasierte Verarbeitung der komplexen WS-Security-Strukturen wirft allerdings einige Problemen auf. Diese ergeben sich beispielsweise aus der Referenzierung der WS-Security-Elemente untereinander und der Verarbeitung von kryptographischen Schlüsseln. Diese Probleme werden im nächsten Abschnitt untersucht.

8.2 Grundlagen der WS-Security-Verarbeitung

In diesem Abschnitt werden die Grundlagen und die Probleme bezüglich der Verarbeitung von WS-Security-Elementen diskutiert. Dabei liegt der Fokus auf den Anforderungen, die für eine ereignisbasierte Verarbeitung gelten.

8.2.1 Referenzierung zwischen Sicherheitselementen

Ein wichtiger Aspekt bei der ereignisbasierten XML-Verarbeitung sind Referenzen innerhalb des XML-Dokuments und dabei insbesondere Rückwärtsreferenzen. Die Ereignisse des Elements, auf das eine solche Referenz zeigt, sind zum Zeitpunkt der Verarbeitung der Referenz nicht mehr

[1] Diese Überprüfung wird wie in Kapitel 7 beschrieben durchgeführt und hier nicht weiter erläutert.

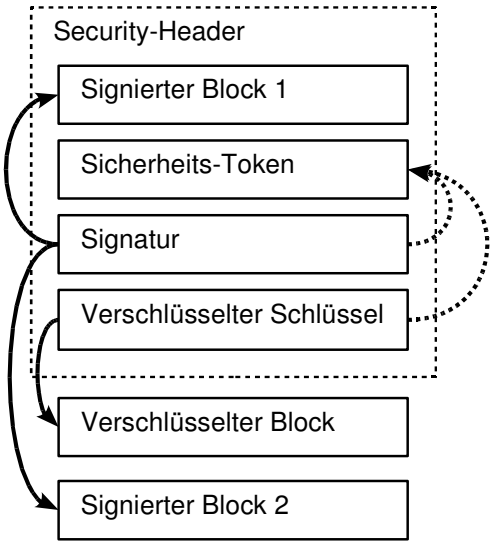


Abbildung 8.1: Typische Referenzierungen bei WS-Security

verfügbar. Sofern das Element nicht explizit gespeichert wurde, kann eine Rückwärtsreferenz also nicht ausgewertet werden. Daher ist die Analyse der Referenzierungsmöglichkeiten innerhalb von SOAP-Nachrichten eine wichtige Voraussetzung für die Realisierung der ereignisbasierten Verarbeitung.

Zwischen den Sicherheitselementen nach WS-Security bestehen zahlreiche Referenzierungsmöglichkeiten innerhalb einer SOAP-Nachricht (und zum Teil auch auf externe Ressourcen). Abbildung 8.1 zeigt schematisch ein typisches Beispiel für die Referenzierungen von WS-Security-Elementen. Im Folgenden werden diese möglichen Referenzierungen im Detail beschrieben.

Signaturen

Die Referenz auf das signierte XML-Fragment geschieht mittels einer *abgekürzten* (engl. *shorthand*) XPointer-Referenz [65] im `URI`-Attribut des `ds:Reference`-Elements. Dabei verweist `URI="#`*`ref`*`"` auf ein Element in derselben Nachricht mit dem Attribut `Id="`*`ref`*`"` ([111], R3001). Eine solche Referenz kann entweder *rückwärts* (d. h. zu einem in Dokumentreihenfolge früheren Element) oder (engl. *forward*) (d. h. zu einem in Dokumentreihenfolge späteren Element) erfolgen. Das folgende Beispiel enthält den Fall einer Vorwärtsreferenz.

```
<env:Envelope>
  <env:Header>
    <wsse:Security>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:CanonicalizationMethod/>
          <ds:SignatureMethod/>
          <ds:Reference URI="#MyBody">
            <ds:Transforms>
              <ds:Transform/>
            </ds:Transforms>
```

```
            <ds:DigestMethod/>
            <ds:DigestValue>...</ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>...</ds:SignatureValue>
        <ds:KeyInfo>
          <wsse:SecurityTokenReference>
            ...
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </env:Header>
  <env:Body wsu:Id="MyBody">
    ...
  </env:Body>
</env:Envelope>
```

Verschlüsselung

Verschlüsselte Elementen können zwei mögliche Verweise enthalten: Referenzen auf Sicherheits-Token, deren Schlüssel zur Entschlüsselung benötigt werden, und Referenzen auf verschlüsselte Blöcke im SOAP-Header.

Jeder verschlüsselte `xenc:EncryptedData`-Block muss im SOAP-Header referenziert werden. Dabei gibt es zwei Möglichkeiten.

Ist für den `xenc:EncryptedData`-Block ein verschlüsselter Schlüssel verwendet worden, so enthält der entsprechende `xenc:EncryptedKey` ein `xenc:ReferenceList`-Element, das in einem `xenc:DataReference`-Element mittels eines XPointers auf den `xenc:EncryptedData`-Block verweist. Dabei muss der `xenc:EncryptedKey` in Dokumentreihenfolge *vor* dem `xenc:EncryptedData`-Block auftreten ([111], R3208). Der `xenc:EncryptedKey` wiederum enthält ein `ds:KeyInfo`-Element mit einer Referenz auf ein Sicherheits-Token, das einen Hinweis auf den benutzten Schlüssel enthält. Dies wird im folgenden Beispiel illustriert.

```
<env:Envelope>
  <env:Header>
    <wsse:Security>
      <xenc:EncryptedKey>
        <ds:KeyInfo>
          <wsse:SecurityTokenReference>
            ...
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
        <xenc:CipherData>
          <xenc:CipherValue>...</xenc:CipherValue>
        </xenc:CipherData>
        <xenc:ReferenceList>
          <xenc:DataReference URI="#MyBody"/>
        </xenc:ReferenceList>
```

```
      </xenc:EncryptedKey>
    </wsse:Security>
  </env:Header>
  <env:Body>
    <xenc:EncryptedData Id="MyBody">
      <xenc:CipherData>
        <xenc:CipherValue>...</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </env:Body>
</env:Envelope>
```

Ist für den `xenc:EncryptedData`-Block kein verschlüsselter Schlüssel verwendet worden, so wird in den `wsse:Security`-Header ein `xenc:ReferenceList`-Element mit einem Verweis auf das `xenc:EncryptedData` hinzugefügt. Der `xenc:EncryptedData`-Block enthält ein `ds:KeyInfo`-Element mit einer Referenz auf ein Sicherheits-Token, das einen Hinweis auf den benutzten Schlüssel enthält. Das folgende Beispiel zeigt diesen Fall.

```
<env:Envelope>
  <env:Header>
    <wsse:Security>
      <xenc:ReferenceList>
        <xenc:DataReference URI="#MyBody"/>
      </xenc:ReferenceList>
    </wsse:Security>
  </env:Header>
  <env:Body>
    <xenc:EncryptedData Id="MyBody">
      <ds:KeyInfo>
        <wsse:SecurityTokenReference>
          ...
        </wsse:SecurityTokenReference>
      </ds:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>...</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </env:Body>
</env:Envelope>
```

Sicherheits-Token

Das `ds:KeyInfo`-Element enthält ein `wsse:SecurityTokenReference`-Element, welches seinerseits auf den kryptographischen Schlüssel referenziert, welcher zum Signieren bzw. Verschlüsseln verwendet wurde ([111], R5417). Für diese Referenzierung gibt es folgende Möglichkeiten:

1. Direkte Referenz: Referenzierung mittels eines XPointers auf einen Sicherheits-Token innerhalb derselben Nachricht. Dabei muss das referenzierte Token *vor* der Referenz im Dokument auftreten ([111], R5205).

```
<wsse:BinarySecurityToken wsu:Id="MyCert">
   GVIpzSg4V486hHFe7sH...
</wsse:BinarySecurityToken>

<wsse:SecurityTokenReference>
  <wsse:Reference URI="#MyCert">
</wsse:SecurityTokenReference>
```

2. Eingebettete Referenz: Das Sicherheits-Token ist als Kindknoten eingebettet.

```
<wsse:SecurityTokenReference>
  <wsse:BinarySecurityToken>
    GVIpzSg4V486hHFe7sH...
  </wsse:BinarySecurityToken>
</wsse:SecurityTokenReference>
```

3. Schlüssel-Identifikator: Verweis auf ein externes Sicherheits-Token mittels eines Schüssel-Identifikators (engl. *Key Identifier*).

```
<wsse:SecurityTokenReference>
   <wsse:KeyIdentifier>
     MIGfMa0GCSq
  </wsse:KeyIdentifier>
</wsse:SecurityTokenReference>
```

4. Externe Referenz: Verweis auf ein externes Sicherheits-Token mittels einer URI.

```
<wsse:SecurityTokenReference>
  <wsse:Reference URI="http://www.example.org/certs/test.crt" />
</wsse:SecurityTokenReference>
```

8.2.2 WS-Security-Verarbeitung

Im Einzelnen müssen folgende WS-Security-Elemente verarbeitet werden:

- WS-Security-Header
- signierte Blöcke
- verschlüsselte Blöcke

WS-Security-Header

Ein SOAP-Header kann mehrere WS-Security-Header enthalten, allerdings darf für jeden Adressaten nur maximal ein Security-Header existieren. Damit ist für die WS-Security-Komponente eindeutig bestimmbar, welchen Header sie verarbeiten soll. Die wichtigsten Elemente innerhalb des Security-Headers sind dabei:

- Sicherheits-Token
- verschlüsselte Schlüssel
- Signaturen

Sicherheits-Token

Sicherheits-Token transportieren Identitäten und teilweise auch kryptographische Schlüssel bzw. Verweise auf solche. Zur Verarbeitung von Signaturen, signierter oder verschlüsselter Blöcke werden diese Informationen benötigt. Daher wird jedes gelesene Sicherheits-Token gespeichert. Zusätzlich werden die Informationen, auf die ein Sicherheits-Token verweist, eingelesen und ebenfalls gespeichert. Dies können beispielsweise private Schlüssel oder externe Zertifikate sein.

Verschlüsselte Schlüssel

Verschlüsselte Schlüssel transportieren Schlüssel, die für die Entschlüsselung von verschlüsselten Blöcken benötigt werden. Der Schlüssel für die Entschlüsselung des verschlüsselten Schlüssels selbst ergibt sich aus einem Sicherheits-Token. Dieses Sicherheits-Token muss in Dokumentreihenfolge vor dem verschlüsselten Schlüssel auftreten (siehe vorheriger Abschnitt). Damit kann bei der ereignisbasierten Verarbeitung der verschlüsselte Schlüssel sofort entschlüsselt und gespeichert werden.

Signaturen

Der Signatur-Block im WS-Security-Header enthält Referenzen und Hash-Werte aller Blöcke, die die Signatur umfaßt. Zusätzlich enthält der Signatur-Block die eigentliche Signatur über diese Referenzen und Hash-Werte. Diese Signatur kann bei der ereignisbasierten Verarbeitung sofort überprüft werden. Für die Überprüfung der Hash-Werte gibt es zwei mögliche Fälle. Ist die Referenz eine Rückwärtsreferenz, so ist der entsprechende signierte Block bereits verarbeitet und sein Hash-Wert gebildet worden (siehe unten). In diesem Fall können die beiden Hash-Werte (der berechnete und der in der Signatur angegebene) während der Signatur-Verarbeitung verglichen werden. Im Falle einer Vorwärtsreferenz muss der Hash-Wert aus der Signatur zwischengespeichert werden.

Verschlüsselte Blöcke

Verschlüsselte Blöcke können an beliebigen Stellen in der SOAP-Nachricht vorkommen. Sie sind durch das `xenc:EncryptedData`-Element eindeutig indentifizierbar. Sobald ein entsprechendes Ereignis auftritt, kann die Verarbeitung für verschlüsselte Blöcke gestartet werden. Der notwendige Entschlüsselungsschlüssel kann sich aus einem verschlüsselten Schlüssel oder aus einem Sicherheits-Token ergeben. Diese müssen in Dokumentreihenfolge immer vor dem verschlüsselten Block liegen. Damit kann bei der ereignisbasierten Verarbeitung die Entschlüsselung sofort begonnen werden. Die entschlüsselten Daten werden an einen weiteren XML-Parser gegeben, welcher daraus (neue) Ereignisse für die (vorher verschlüsselten) Elemente erzeugt.

Signierte Blöcke

Im Gegensatz zu verschlüsselten Blöcken, sind signierte Blöcke nicht explizit als solche markiert. Diese werden ausschließich durch die Referenz in der Signatur identifiziert.

Ist diese Referenz eine Vorwärtsreferenz (in Abbildung 8.1 der zweite signierte Block), kann der signierte Block durch Vergleichen der gespeicherten Referenzen aus der Signatur mit dem

aktuellen Element gefunden werden. Für diesen Block wird dann die Hash-Bildung durchgeführt und das Ergebnis mit dem gespeicherten Hash-Wert aus der Signatur verglichen.

Signierte Blöcke, die durch eine Rückwärtsreferenz referenziert sind, können auf diese Weise nicht behandelt werden. Da diese erst bei der Verarbeitung der Signatur identifiziert werden können, müsste bei einer simplen Lösung der gesamte SOAP-Header gespeichert werden. Eine geschicktere Lösung nutzt die Tatsache aus, dass die Referenzierung von signierten Fragmenten über XPointer-Referenzen der Form `URI="#id"` erfolgt. Dies bedeutet umgekehrt, dass jedes signierte Element ein Attribut der Form `Id="id"` trägt.[2] Dies wird für die hier vorgestellte Lösung ausgenutzt: Für jedes Element im SOAP-Header, das ein `Id`-Attribut enthält, wird die Hash-Bildung gestartet. Das Ergebnis wird gespeichert und bei der Verarbeitung der Signatur mit dem dort enthaltenen Wert verglichen.

Der konkrete Hash-Algorithmus wird innerhalb der Signatur definiert, ist also zum Zeitpunkt der Verarbeitung des rückwärtsreferenzierten Elements noch nicht bekannt. Daher wird für jeden möglichen Hash-Algorithmus parallel das Hashing durchgeführt. Da der Speicherbedarf pro Instanz nur in der Größenordnung der Länge des Hash-Ergebnisses liegt (z. B. bei SHA1 160 Bit), ist der dafür notwendige Speicheraufwand gering. Da aktuell nur ein einziger Hash-Algorithmus empfohlen und de facto verwendet wird (SHA1; siehe [111], R5420), kann die Anzahl der Hash-Instanzen sogar auf eine reduziert werden.

Ein solch parallele Verarbeitung wird auch beim Normalisieren und Transformieren der potentiell signierten Elemente durchgeführt. Und auch hier gibt es (zum Teil abhängig von dem zu normalisierenden Element) nur wenige mögliche Normalisierungs-/Transformations-Algorithmen (siehe [111], R5423). Für die meisten Elemente existiert nur die *Exclusive-C14N*-Normalisierung [22]. Für `wsse:SecurityTokenReference` auf externe URLs existiert zusätzlich die STR-Transformierung [124]. Für Signatur-Elemente wird zusätzlich die Transformierung für *umschlossene Signaturen* (engl. *enveloped signature*) [10] angewendet.

Die Ergebnisse der Normalisierung werden sofort in die Hashing-Funktion gegeben. Damit ist die Größenordnung des Gesamtspeicherbedarfs für das Normalisieren und Hashing gleich der Anzahl der Normalisierungs-/Transformations-Funktionen mal Anzahl der Hashing-Funktionen mal Größe des Hash-Ergebnisses.

8.2.3 Architektur der WS-Security-Komponente

Abbildung 8.2 zeigt die Gesamtarchitektur der WS-Security-Komponente, in der die zuvor beschriebenen Verarbeitungsschritte ereignisbasiert durchgeführt werden.

Neben den Teilkomponenten zur Verarbeitung des WS-Security-Headers existiert auch ein Dispatcher. Dieser hat die Aufgabe, signierte bzw. verschlüsselte Nachrichtenteile zu erkennen und *dynamisch* eine Instanz der Teilkomponente zur Verarbeitung signierter und verschlüsselter Blöcke in die Verarbeitungskette einzufügen.

[2] Das `Id`-Attribut kann dabei aus verschiedenen Namensräumen stammen, z. B. die lokalen ID-Attribute aus XML Signature und Encryption. Empfohlen wird aber die Verwendung des `wsu:Id`-Attributs ([111], R3005)

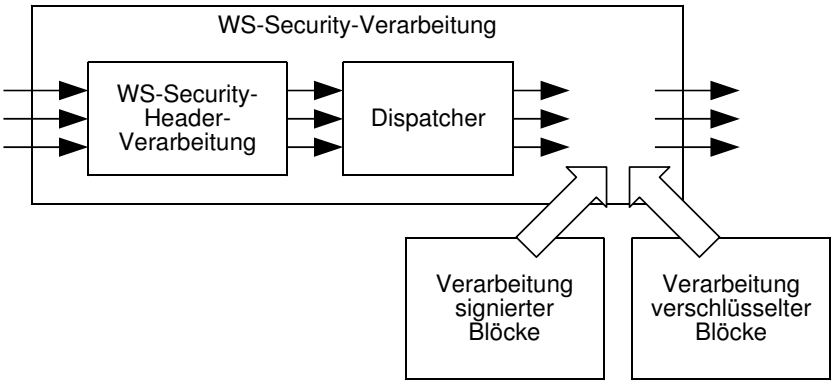


Abbildung 8.2: Architektur der WS-Security-Komponente

8.3 Ereignisgesteuerte WS-Security-Verarbeitung

8.3.1 Vorbemerkungen

Ereignisgesteuerte Verarbeitung wird üblicherweise durch einen *endlichen Zustandsautomaten* (engl. *Finite State Machine*) [81] beschrieben. Die Transitionen werden dabei von den Ereignissen (hier: XML-Ereignisse) ausgelöst. Zur Illustration der Verarbeitung von WS-Security werden daher die entsprechenden (Teil-)Automaten angegeben.

Die abgebildeten Automaten sind dabei *nichtdeterministische endliche Automaten* mit ε-Transitionen (ε-NEA). Diese Darstellung wurde gewählt, da sie zu übersichtlicheren Automaten führt als die entsprechenden *deterministischen endlichen Automaten* (DEA). Das Eingabealphabet besteht aus den XML-Ergeignissen, die von der WS-Security-Komponente aufgenommen werden. Die XML-Ereignisse in den Automaten-Diagrammen orientieren sich dabei am Format von SAX. Die relevanten Ereignisse sind dabei:

- $\mathsf{start}(a, @x = c)$: Startkennung des Elements a, das das Attribut x mit dem Wert c enthält
- $\mathsf{content}(c)$: XML-Inhalt mit dem Wert c
- $\mathsf{end}(a)$: Endkennung des Elements a

Das Ausgabealphabet sind „Methodenaufrufe“, die die weitere Verarbeitung auslösen. Die emittierten XML-Ereignisse sind nicht in den Zeichnungen zu erkennen sondern werden in den dazugehörigen Erläuterungen beschrieben.

Dabei werden auch Ereignisse erzeugt, die nicht Teil der Nachricht sind, sondern Metainformationen transportieren, die von nachfolgenden Komponenten benötigt werden. Die folgenden Ereignisse werden in der WS-Security-Komponente erzeugt:

- $\mathsf{consumedStart}(a)$: die Ursprungsnachricht enthielt an dieser Stelle den Beginn des Elements a.
- $\mathsf{consumedEnd}(a)$: die Ursprungsnachricht enthielt an dieser Stelle das Ende des Elements a.
- $\mathsf{signedElement}(ref, t, a)$: Das Element mit der ID ref ist von dem Token t unter Benutzung der Algorithmen a signiert worden.

Bei der Entschlüsselung werden dann beispielsweise alle eingehenden start(·)-Ereignisse zu consumedStart(·)-Ereignissen und analog alle end(·)-Ereignisse zu consumedEnd(·)-Ereignissen umgewandelt.

8.3.2 Datenstrukturen

Wie bereits im Abschnitt 8.2.2 angedeutet, müssen außer den Zuständen der Automaten einige Daten während der Verarbeitung zwischengespeichert werden. Dabei werden folgende Basismengen verwendet:

- $TRANS$: Menge der Transformationen gemäß XML-Signature
- $BYTE$: Menge aller Bytefolgen
- $STRING$: Menge aller Zeichenketten

Sei $Ref \subseteq STRING \times \{Sig, Enc\}$ eine geordnete Liste von Sicherheits-Referenzen innerhalb des SOAP-Dokuments. Dabei bedeutet $(ref, Sig) \in Ref$, dass das Element mit der ID *ref* von einer Signatur referenziert wird, also signiert wurde, und $(ref, Enc) \in Ref$, dass ein Element `xenc:EncryptedData` mit der ID *ref* von einer `xenc:ReferenceList` referenziert wird.

Sei $CompletedDigest \subseteq STRING \times \mathcal{P}(TRANS \times BYTE)$ die Menge der Nachrichten-*Digests* von potentiell signierten Blöcken. Dabei bedeutet $(ref, D) \in CompletedDigest$, dass der Block mit der ID *ref* transformiert, normalisiert und gehasht wurde und D die Zuordnung der Transformations-Algorithmen zu den daraus resultierenden *Digest*-Werten enthält.

Sei $OpenDigest \subseteq STRING \times TRANS \times BYTE$ die Menge der noch ausstehenden *Digest*-Überprüfungen. Dabei bedeutet $(ref, t, digest) \in OpenDigest$, dass in einer Signatur für die Referenz *ref* die Transformation t und der *Digest*-Wert *digest* angegeben sind.

Sei $Key \subseteq STRING \times BYTE$ die Menge der im Dokument verwendeten kryptographischen Schlüssel. Dabei bedeutet $(ref, key) \in Key$, dass ein `wsse:BinarySecurityToken` mit dem Schlüssel *key* und der ID *ref* existiert.

Sei $EncKey \subseteq STRING \times BYTE$ die Menge der (symmetrischen) kryptographischen Schlüssel aus `xenc:EncryptedKey`-Elementen. Dabei bedeutet $(ref, key) \in EncKey$, dass ein `xenc:EncryptedKey`-Element existiert, welches den Schlüssel *key* (verschlüsselt) und eine Referenz *ref* enthält. Damit muss ein `xenc:EncryptedData`-Block mit der ID *ref* existieren, bei dem *key* zum Verschlüsseln verwendet wurde.

8.3.3 WS-Security-Header-Verarbeitung

Abbildung 8.3 zeigt den Automaten für die Verarbeitung eines WS-Security-Blocks im SOAP-Header. Ein SOAP-Header kann mehr als einen `wsse:Security`-Block enthalten, allerdings nur maximal einen pro Empfänger. Damit wird die Verarbeitung also durch das `wsse:Security`-Element gestartet, das als `env:actor`- bzw. `env12:role`-Attribut die Rolle des Schutzsystems enthält.

Das `wsse:Security`-Element kann folgende Elemente in (fast) beliebiger Reihenfolge enthalten: Sicherheits-Token, verschlüsselte Schlüssel, Zeitstempel, Signaturen und Referenzen auf verschlüsselte Blöcke, die nicht von einem verschlüsselten Schlüssel referenziert werden. Dabei gelten für den Zeitstempel folgende Einschränkungen:

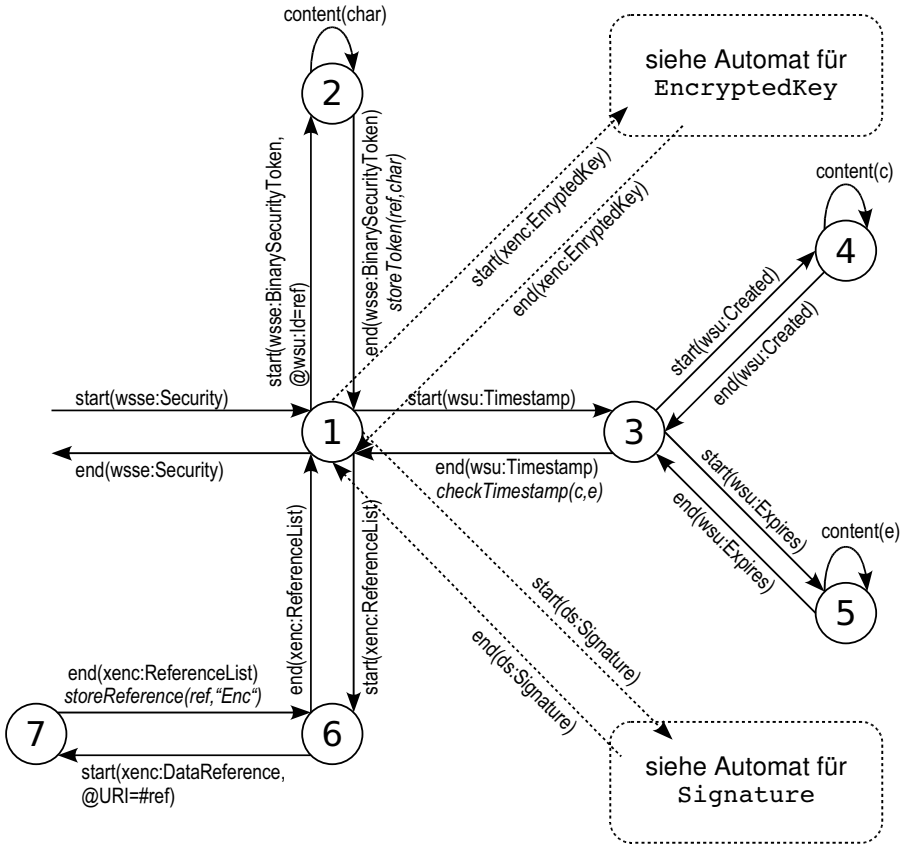


Abbildung 8.3: Automat für die Verarbeitung eines WS-Security-Header-Blocks

- Jeder WS-Security-Header enthält maximal einen Zeitstempel ([111], R3227).
- Jeder Zeitstempel enthält genau ein `wsse:Created`-Element ([111], R3203).
- Jeder Zeitstempel enthält maximal ein `wsse:Expires`-Element, welches dem Erzeugungsdatum folgt ([111], R3224 + R3221).

Während des Einlesens des WS-Security-Headers werden folgende Verarbeitungsschritte ausgeführt:

- storeToken($ref, char$): Der (Roh-)Wert $char$ des Sicherheits-Tokens[3] wird gemäß seines Typs (z. B. X.509) zu einem Schlüssel key dekodiert und zusammen mit dem Identifikator ref in der List Key gespeichert.
- checkTimestamp(c, e): Überprüft, ob der Zeitstempel gültig ist; beispielsweise $e > c$ oder $e \leq t$, wobei t der aktuelle Zeitstempel ist.
- storeReference($ref, type$): Das Paar $(ref, type)$ wird am Ende der Liste der Sicherheits-Referenzen Ref gespeichert.

Alle Ereignisse von Elementen des WS-Security-Headers werden in consumed*-Ereignisse umgewandelt.

Schlüsselinformationen `ds:KeyInfo`

Die `wsse:SecurityTokenReference` im `ds:KeyInfo`-Element verweist entweder auf ein Sicherheits-Token innerhalb derselben Nachricht oder enthält eine externe Referenz zu einem Sicherheits-Token. Ist das Sicherheits-Token in derselben Nachricht enthalten, so muss es vor der Referenzierung oder direkt als Kindelement auftreten. Damit ist also in jedem Fall zum Zeitpunkt des Auftretens des `ds:KeyInfo`-Elements das Sicherheitstoken bekannt oder effizient bestimmbar.

Aus dem Sicherheits-Token ergeben sich auch die kryptographischen Schlüssel, die für Signierung und Verschlüsselung benötigt werden. Dabei gibt es mehrere Möglichkeiten den Schlüssel zu erlangen. Einen typischen Fall stellt die Verwendung eines X.509-Zertifikats als Sicherheits-Token dar. Dieses enthält direkt den öffentlichen Schlüssel für die Überprüfung der Signatur. Der dazugehörige private Schlüssel muss dem Empfänger der Nachricht bekannt sein und kann durch den Identifikator des X.509-Zertifikats identifiziert werden.

Die Ereignisse des `ds:KeyInfo`-Elements werden in consumed*-Ereignisse umgewandelt.

Verschlüsselte Schlüssel

Im Folgenden gelten diese Bezeichnungen: Sei key der Schlüssel, der in verschlüsselter Form innerhalb des `xenc:CipherData`-Elements abgelegt ist. Sei alg der Algorithmus, der für diese Verschlüsselung verwendet wurde, und sei key_{sym} (wenn alg symmetrisch) bzw. key_{pub} (wenn alg asymmetrisch) der Schlüssel, der zur Verschlüsselung von key verwendet wurde. Im Fall der asymmetrischen Verschlüsselung sei key_{priv} der zu key_{pub} gehörige private Schlüssel.

[3] Binärwerte werden in XML-Dokumenten üblicherweise in kodierter Form nach Base64 [58] dargestellt. Die Dekodierung von Base64 wird im Folgenden nicht extra erwähnt.

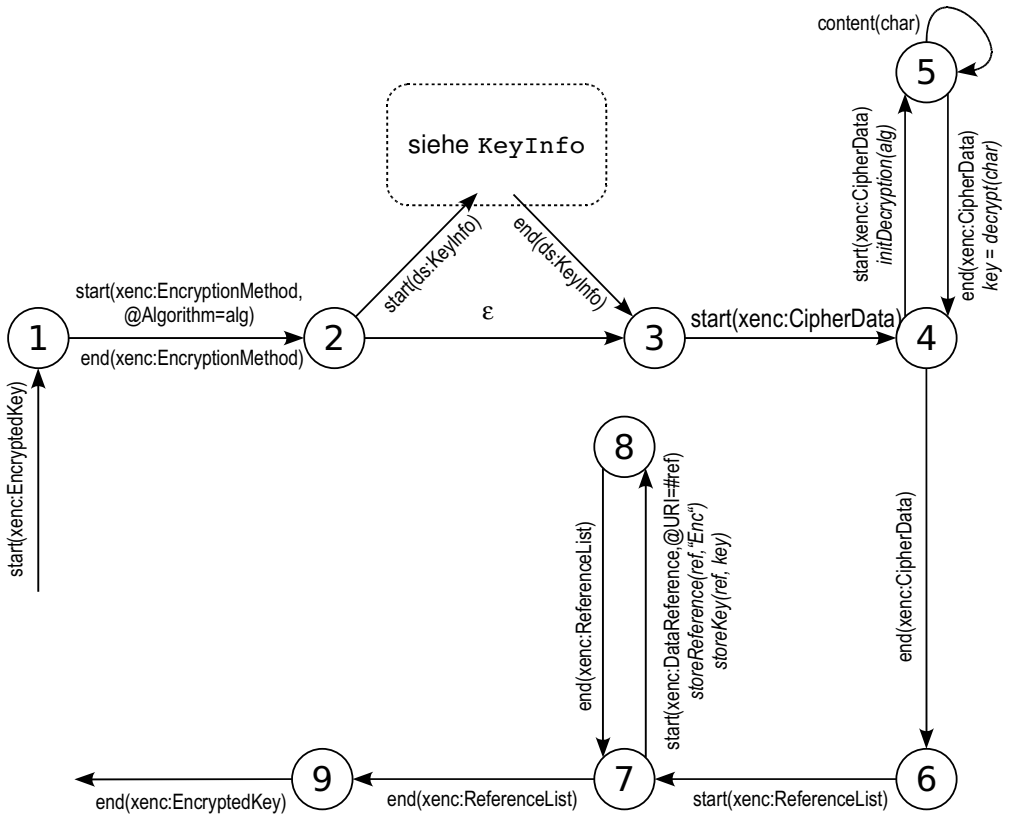


Abbildung 8.4: Automat für die Verarbeitung von verschlüsselten Schlüsseln

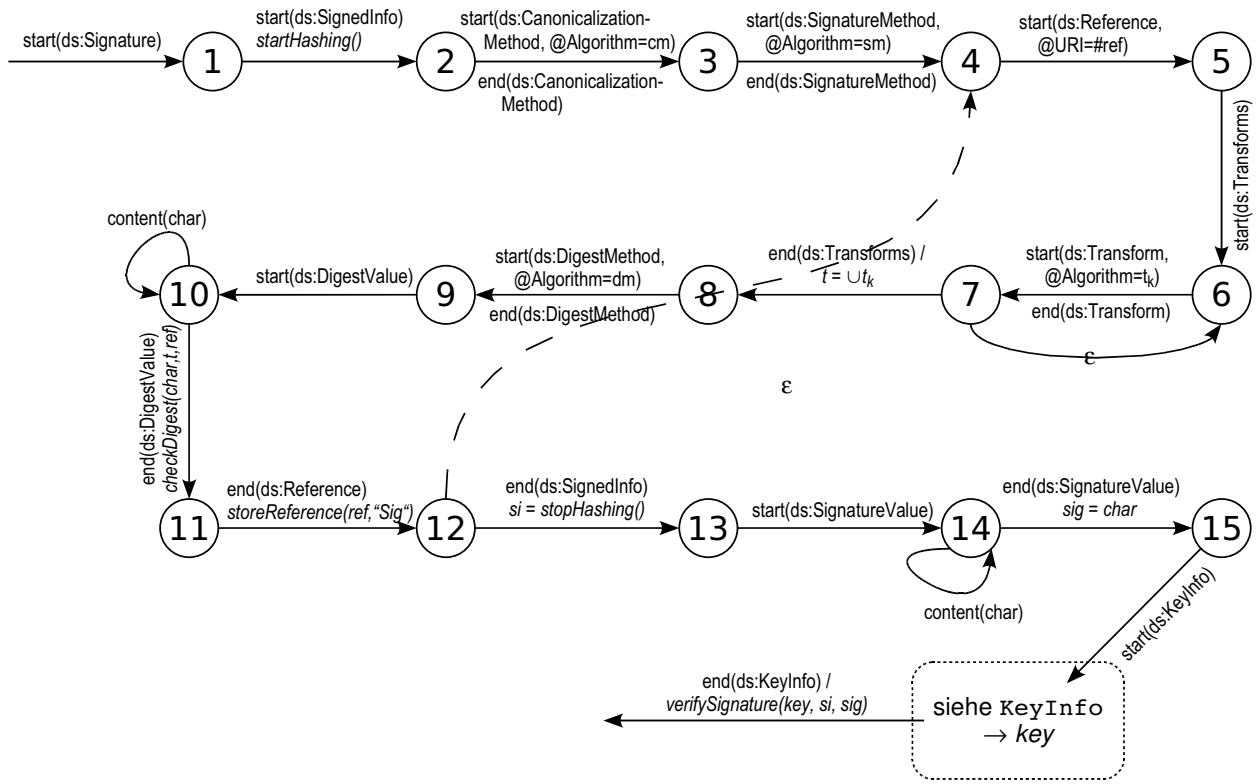


Abbildung 8.5: Automat für die Verarbeitung von Signaturen

Abbildung 8.4 zeigt die Verarbeitung eines verschlüsselten Schlüssels, die durch das Element `xenc:EncryptedKey` ausgelöst wird. Dabei wird zunächst der Verschlüsselungsalgorithmus alg ausgelesen (1).

Ist alg asymmetrisch (was für einen verschlüsselten Schlüssel der typische Fall ist), so muss ein `ds:KeyInfo`-Element (2) angegeben sein (siehe Abschnitt 8.3.3). Dieses liefert dann key_{priv}. Ist alg symmetrisch, so existiert in der Nachricht ein weiterer verschlüsselter Schlüssel, der auf den aktuellen `xenc:EncryptedKey`-Block verweist und *vor* diesem liegt. Damit ist der daraus resultierende Schlüssel key_{sym} bereits bekannt.

Der Schlüssel key_{priv} bzw. key_{sym} wird für die Initialisierung des Entschlüsselungsalgorithmus innerhalb der Funktion initDecryption(alg) verwendet (4). Der Inhalt des `xenc:CipherData`-Elements wird dann mittels dieses Algorithmus entschlüsselt. Die Funktion decrypt($char$) führt also $dec_{key_{priv}}(char)$ bzw. $dec_{key_{sym}}(char)$ aus. Das Ergebnis der Entschlüsselung ist der Schlüssel key, der für weitere kryptographische Operationen verwendet werden kann.

Allen Referenzen des `xenc:EncryptedKey`-Elements wird dieser Schlüssel zugeordnet (7,8): storeKey(ref, key) fügt das Paar (ref, key) zu $EncKey$ hinzu. Außerdem werden die Referenzen in die Liste der Sicherheits-Referenzen eingefügt: storeReference($ref, type$) fügt das Paar $(ref, type)$ ans Ende von Ref ein.

Alle Elemente des `xenc:EncryptedKey`-Blocks werden in consumed*-Ereignisse umgewandelt.

Signaturen

Abbildung 8.5 zeigt den Automaten für die Verarbeitung einer XML Signatur, ausgelöst durch den Beginn des `ds:Signature`-Elements. Während der Abarbeitung werden die folgenden Schritte durchgeführt:

- Zur Überprüfung der Signatur wird der normalisierte und gehashte `ds:SignedInfo`-Block benötigt. Daher wird am Beginn dieses Blocks durch startHashing() die *Exclusive-C14N*-Normalisierung und das Hashing mittels SHA-1 gestartet[4] (1).
- Der Normalisierungs-Algorithmus für den `ds:SignedInfo`-Block wird eingelesen, er muss *Exclusive-C14N* sein (2).
- Der Signatur-Algorithmus (z. B. „RSA with SHA-1“) wird eingelesen (3).
- Die Transformations-Algorithmen für das signierten Fragment werden eingelesen und zu t zusammengefasst (6,7).
- Der Hashing-Algorithmus für das signierte Fragment wird eingelesen, er muss SHA-1 sein (8).
- Der Nachrichten-*Digest* wird eingelesen (10) und checkDigest$(char, t, ref)$ ausgeführt:
  - Falls ein D mit $(ref, D) \in CompletedDigest$ existiert, so ist ref eine Rückwärtsreferenz und das referenzierte Element bereits verarbeitet. Dann ist $digest$ mit $(t, digest) \in D$ der berechnete *Digest* dieses Elements. Falls $char = digest$, stimmen der berechnete und der angegebene *Digest* überein; in diesem Fall wird ein Ereignis signedElement(ref, t, a) erzeugt, wobei a die verwendeten Algorithmen (d. h. Signatur-, Hash-, Transformations- und Normalisierungsalgorithmus) darstellt. Falls nicht, wird die Verarbeitung abgebrochen.
  - Falls ein solches D nicht existiert, so ist ref eine Vorwärtsreferenz und $(ref, t, digest)$ wird zu $OpenDigest$ hinzugefügt.
- An Ende des `ds:Reference`-Blocks wird storeReference$(ref, type)$ ausgeführt (11). Damit wird Paar $(ref, type)$ ans Ende der Liste der Sicherheits-Referenzen Ref angehängt.
- In einem `ds:SignedInfo`-Block können noch beliebig viele weitere `ds:Reference`-Elemente vorkommen (12→4).
- Am Ende des `ds:SignedInfo`-Blocks steht der für die Überprüfung der Signatur notwendige *Digest* dieses Elements fest: $si =$ stopHashing() (12).
- Der Signaturwert sig wird aus dem Element `ds:SignatureValue` eingelesen (14).
- Der für die Signaturüberprüfung notwendige Schlüssel kann durch das `ds:KeyInfo` Element erhalten werden (15).

[4] Weitere Normalisierungs- und Hashing-Algorithmen sind für `ds:SignedInfo` (aktuell) nicht vorgesehen. Für den Fall, dass die Algorithmen nicht eindeutig wären (oder werden), kann eine parallele Abarbeitung wie oben beschrieben verwendet werden.

Am Ende des Signaturelements wird die Überprüfung der Signatur durchgeführt: verifySignature (key, si, sig). Dazu wird die Verifizierungfunktion des Signatur-Algorithmus für den *Digest* si, den Signatur-Wert sig und den Schlüssel key durchgeführt. Für RSA als Signatur-Algorithmus bedeutet das beispielsweise: Berechne $Enc_{key}(Pad(sig))$ und vergleiche das Ergebnis mit si. Dabei ist Pad die *Auffüllfunktion* gemäß PKCS #1 [96].

Alle Elemente des `ds:Signature`-Blocks werden in consumed*-Ereignisse umgewandelt.

8.3.4 Verarbeitung signierter und verschlüsselter Blöcke

Eine SOAP-Nachricht kann verschlüsselte oder signierte Blöcke mehrfach und an beliebiger Stelle im Dokument enthalten. Der Dispatcher erkennt diese Blöcke und fügt dynamisch jeweils eine neue Instanz zur Verarbeitung dieses Blocks in die Verarbeitungskette ein.

Reihenfolge von Verschlüsselung und Signierung

Ein allgemeines Problem bei der Verarbeitung von *XML Signature* und *XML Encryption* ist die Erkennung der Reihenfolge in der Signierung- und Verschlüsselungs-Operationen angewendet worden sind. Während die mehrfache Verschlüsselung eines Elements zweifelsfrei erkannt werden kann und bei einer mehrfachen Signierung die Reihenfolge irrelevant ist, so ist die Kombination aus Verschlüsselung und Signierung eines Elements problematisch. In bestimmten Kombinationen kann der Nachricht nicht angesehen werden, ob zuerst die Verschlüsselung (*EncryptBeforeSigning*, [95]) oder zuerst die Signatur (*SignBeforeEncrypting*, [95]) angewendet wurde. Das folgende XML-Fragment zeigt eine Nachricht, in der diese Situation auftritt.

```
<ds:Signature>
  ...
  <ds:Reference URI="#sig-1">
  ...
</ds:Signature>
...
<ns1:a Id="sig-1">
  <xenc:EncryptedData Id="enc-1">
    ...
  </xenc:EncryptedData>
</ns1:a>
```

Das Wissen um die Reihenfolge der Erstellung ist aber notwendig für die korrekte Überprüfung der Signatur. Daher muss der `wsse:Security`-Header die Elemente mit Sicherheits-Referenzen (d. h. `ds:Signature`, `xenc:EncryptedKey` und `xenc:ReferenceList`) in der Reihenfolge enthalten, in der die Abarbeitung der dazugehörigen Operationen erfolgen soll ([111], R3212). Diese Eigenschaft kann für die ereignisbasierte Entschlüsselung und Signaturüberprüfung wie folgt verwendet werden.

Da verschlüsselte Blöcke immer nur vorwärts referenziert werden, kann eine Kombination aus potentiell signierten Blöcken und verschlüsselten Blöcken nicht auftreten. Der Handler für einen potentiell signierten Block kann also immer am Ende der Verarbeitungskette (d. h. direkt vor dem Security Policy Handler) eingefügt werden. Im Folgenden werden also nur noch vorwärts referenzierte signierte Blöcke und verschlüsselte Blöcke betrachtet.

Sei $Ref = \{(ref_1, op_1), \ldots, (ref_n, op_n)\}$ und seien $(H_1, ref_{i_1}), \ldots, (H_k, ref_{i_k})$ mit $i_j \in \{1, \ldots, n\}$ die Handler (mit den dazugehörigen Referenzen) in der aktuellen Verarbeitungskette. Bei Auftreten eines signierten oder verschlüsselten Blocks mit der ID *ref* gilt folgendes: Da *ref* vorwärtsreferenziert ist, ist es in der Liste der Sicherheitsreferenzen vorhanden, d. h. es gibt ein $m \in \{1, \ldots, n\}$ mit $ref = ref_m$. Dann wird die (neue) Instanz des Verarbeitungs-Handlers an die Stelle j (d. h. nach dem Handler H_{j-1}) eingefügt, so dass gilt: $i_{j-1} < m$ und $m < i_j$. Das bedeutet, dass die Reihenfolge der Referenzen der Handler zu jeder Zeit der Reihenfolge der Referenzen in *Ref* entspricht. Diese Vorgehensweise wird durch zwei Beispiele illustriert.

Beispiel 1

Gegeben sei das folgende SOAP-Nachrichten-Fragment (mit stark vereinfachten XML Signaturen):

```
<ds:Signature>
  <ds:Reference URI="#sig-1"/>
</ds:Signature>
...
<ds:Signature>
  <ds:Reference URI="#sig-2"/>
</ds:Signature>
...
<xenc:ReferenceList>
  <xenc:DataReference URI="#enc-1"/>
</xenc:ReferenceList>
...
<env:Body Id="sig-1">
  <ns1:Operation Id="sig-2">
    <xenc:EncryptedData Id="enc-1">
      ...
    </xencEncryptedData>
  </ns1:Operation>
</env:Body>
```

Hier entspricht die Reihenfolge der IDs im SOAP-Body der Reihenfolge der Referenzen im SOAP-Header. Die Handler werden also in folgender Reihenfolge kombiniert: $(H^{Sig}, \texttt{sig-1}), (H^{Sig}, \texttt{sig-2}), (H^{Enc}, \texttt{enc-1})$. Abbildung 8.6 zeigt den daraus resultierenden Durchlauf der XML-Ereignisse[5] durch diese Verarbeitungskette. Offensichtlich ist in diesem Beispiel die Verschlüsselung vor beiden Signierungen durchgeführt worden. Daher wird die Entschlüsselung auch nach der Signatur-Überprüfung durchgeführt.

Beispiel 2

Gegeben sei das folgende SOAP-Nachrichten-Fragment (mit stark vereinfachten XML Signaturen):

```
<ds:Signature>
```

[5] Die consumed*-Ereignisse sind aus Gründen der Übersichtlichkeit nicht dargestellt.

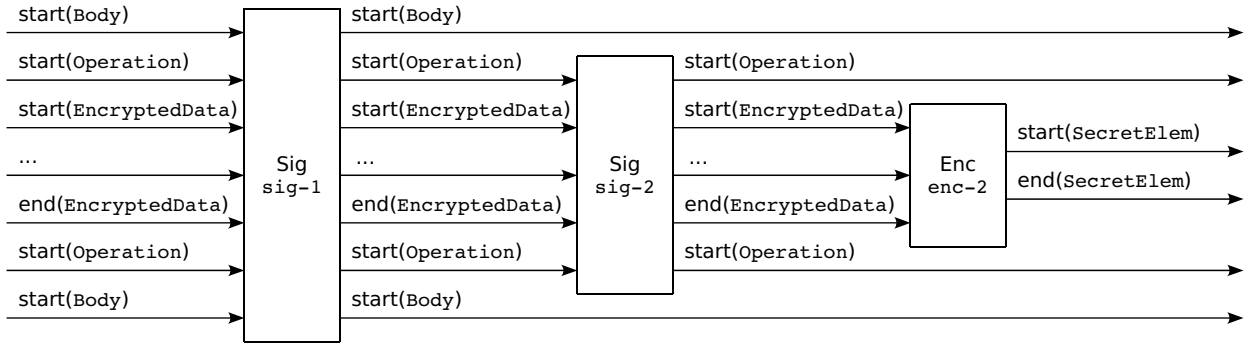


Abbildung 8.6: Signierte und verschlüsselte Blöcke – Beispiel 1

```
  <ds:Reference URI="#sig-1"/>
</ds:Signature>
...
<xenc:ReferenceList>
  <xenc:DataReference URI="#enc-1"/>
</xenc:ReferenceList>
...
<ds:Signature>
  <ds:Reference URI="#sig-2"/>
</ds:Signature>
...
<env:Body Id="sig-1">
  <ns1:Operation Id="sig-2">
    <xenc:EncryptedData Id="enc-1">
      ...
    </EncryptedData>
  </ns1:Operation>
</env:Body>
```

Nach der Bearbeitung von `ns1:Operation` sieht die Verarbeitungskette wie folgt aus: $(H^{Sig}, \texttt{sig-1}), (H^{Sig}, \texttt{sig-2})$. In der Liste der Sicherheitsreferenzen steht die Referenz `enc-1` vor `sig-2`, daher wird der Entschlüsselungs-Handler zwischen die beiden Signatur-Handler eingefügt. Abbildung 8.7 zeigt den Durchlauf der XML-Ereignisse durch diese Verarbeitungskette. In diesem Beispiel ist die Verschlüsselung nach der Signierung von `sig-2` durchgeführt worden, daher muss die Entschlüsselung auch vor der entsprechenden Signatur-Überprüfung stattfinden.

Verschlüsselte Blöcke

Wird während der Verarbeitung der SOAP-Nachricht ein `xenc:EncryptedData`-Element (mit der ID *ref*) gelesen, so wird die Verarbeitung gemäß Abbildung 8.8 gestartet. Dabei wird zunächst der Verschlüsselungs-Algorithmus *alg* eingelesen. Der für die Entschlüsselung notwendige Schlüssel kann durch ein `ds:KeyInfo`-Element referenziert sein. Ist das `ds:KeyInfo`-Element nicht vorhanden, so muss ein *key* mit $(ref, key) \in EncKey$ existieren.

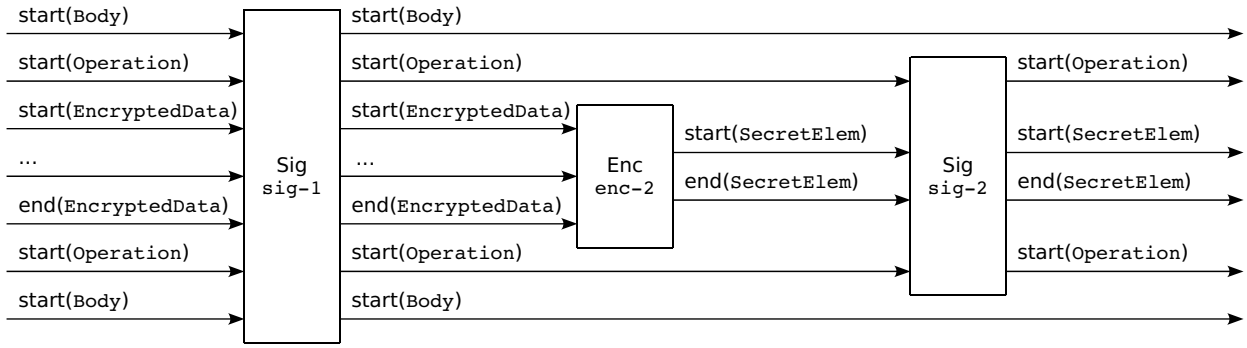


Abbildung 8.7: Signierte und verschlüsselte Blöcke – Beispiel 2

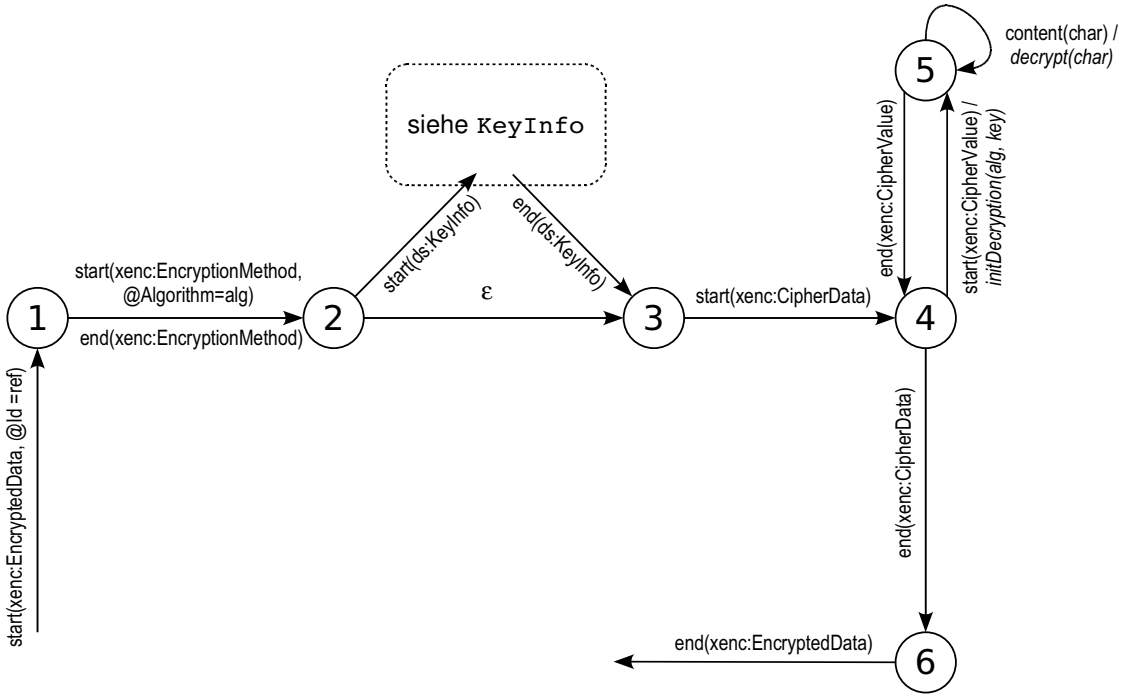


Abbildung 8.8: Automat für die Verarbeitung von verschlüsselten Blöcken

Die Parameter *alg* und *key* werden zur Initialisierung der Entschlüsselungsfunktion verwendet (initDecryption(alg, key)). Mit dieser Funktion wird dann der Inhalt des `xenc:CipherData`-Elements entschlüsselt (decrypt($char$)). Die entschlüsselten Daten stellen die serialisierte Form des verschlüsselten XML-Fragments dar und werden an einen weiteren XML-Parser gegeben, der daraus die XML-Ereignisse dieses Fragments generiert.

Dabei erfolgt die Verarbeitung des verschlüsselten Fragments nicht blockierend bezüglich der nachfolgenden Komponenten. Der Inhalt des `xenc:CipherData`-Elements wird nicht als ein content-Ereignis weitergegeben. Die Größe des Datenfeldes *char* ist beschränkt (in der Regel wenige KByte) und führt dazu, dass der verschlüsselte Inhalt in „kleinen Portionen“ und in mehreren Ereignissen weitergereicht wird. Da alle verwendeten Verschlüsselungsalgorithmen block- oder strombasiert sind, können sie die Teileingaben gleich entschlüsseln.[6] Dieses Ergebnis kann dann auch sofort vom XML-Parser in XML-Ereignisse zerlegt werden. Damit ist eine quasi synchrone Umsetzung des verschlüsselten Inhalts in XML-Ereignisse möglich, und die in Abbildung 2.6 dargestellte „vertikale“ Verarbeitungsmethode kann auch hier eingehalten werden.

Alle Elemente des `xenc:EncryptedData`-Blocks werden in consumed*-Ereignisse umgewandelt. Die vom Parser neu erzeugten XML-Ereignisse, die den (nun entschlüsselten) verschlüsselten Inhalt darstellen, werden an die nachfolgende Komponente weitergegeben.

Signierte Blöcke

Wie oben bereits erwähnt, gibt es für das Erkennen von signierten Blöcken zwei Möglichkeiten. Ist der WS-Security-Header noch nicht verarbeitet worden, müssen alle Elemente, die ein ID-Attribut tragen, als potentiell signiert angesehen werden. Für diese Elemente wird dann die Normalisierung und die Hash-Bildung (ggf. parallel mit mehreren Algorithmen) gestartet. Am Ende des potentiell signierten Blockes werden die Hash-Werte zusammen mit der ID zu *CompletedDigest* hinzugefügt.

Ist der WS-Security-Header bereits verarbeitet, können signierte Blöcke nur über Vorwärtsreferenzen referenziert sein. Damit müssen nur noch IDs beachtet werden, die bereits durch die Signaturverarbeitung bekannt sind. Wird also ein Element mit einem ID-Attribut mit dem Wert *ref* gelesen und existiert ein $t \in TRANS$ und ein $digest \in BYTE$, so dass $(ref, t, digest) \in OpenDigest$ gilt, dann ist dieses Element signiert. In diesem Fall wird eine neue Instanz zur Verarbeitung von signierten Blöcken erzeugt und der signierte Block mit dem Transformations-Algorithmus t transformiert und gehasht.

Am Ende des Elements (also nach der Endkennung) wird das Ergebnis mit *digest* verglichen. Stimmen die Werte nicht überein, wird die Verarbeitung abgebrochen. Ansonsten wird $(ref, t, digest)$ aus *OpenDigest* entfernt, die Teilkomponente aus der Verarbeitungskette entfernt und schließlich das (neue) Ereignis signedElement(ref, t, a) erzeugt, wobei a die bei der Signatur verwendeten Algorithmen darstellt.

In jedem Fall werden bei der Verarbeitung eines signierten Blocks alle eingehenden Ereignisse unverändert wieder ausgegeben.

[6] Gleiches gilt für die vorher notwendige Base64-Dekodierung.

8.3.5 Bewertung der Verarbeitungsmethodik

Einschränkungen

Mit der oben beschriebenen Verarbeitungsmethode lassen sich SOAP-Nachrichten mit WS-Security-Erweiterungen fast vollständig standardkonform verarbeiten. Dabei gibt es allerdings eine kleine Einschränkung. Die Referenzierung bzw. Transformierung von signierten Blöcken mittels XPath-Filter gemäß [23] ist ereignisbasiert im Allgemeinen nicht möglich. Allerdings wird die Benutzung dieser Referenzierungsmethode auch nicht empfohlen ([111], R3001). Dafür gibt es zwei Gründe. Zum einen sind solcherlei Referenzierungen aufwendiger in der Verarbeitung als XPointer. Da der Sender einer Nachricht bei Signaturen beliebig viele Filter aus beliebig komplexen XPath-Ausdrücken konstruieren kann, kann er damit den Server zu ressourcenintensiven Operationen veranlassen.

Zum anderen erschwert die Verwendung der XPath-Filter die Identifikation der effektiv signierten Nachrichtenteile. So lässt sich über einen sicherheitskritischen Block eine Signatur erstellen, die aber Teile dieses Blocks auslässt und damit die Möglichkeit für unbemerkte Modifikationen lässt.

Aus diesen Gründen ist der Ausschluss von XPath-Filtern aus Sicherheitsgesichtspunkten sogar sehr sinnvoll.

Laufzeit und Speicherbedarf

Die Laufzeit des oben beschriebenen Algorithmus ist offensichtlich in $\mathcal{O}(n)$, wobei n die Anzahl der Elemente in der SOAP-Nachricht darstellt. Genauer betrachtet hängt die Laufzeit und der Speicherbedarf von folgenden Eigenschaften der Nachricht ab:

- Anzahl der Signaturen
- Anzahl der verschlüsselten Schlüssel
- Anzahl der Sicherheits-Token

Die Laufzeit hängt zusätzlich von folgenden Eigenschaften ab:

- Größe der verschlüsselten Blöcke
- Größe der signierten Blöcke

Die Anzahl der Sicherheitselemente ist grundsätzlich unbeschränkt. Wie im nächsten Kapitel beschrieben, wird bei der Überprüfung der Sicherheits-Policy aber jede Nachricht abgelehnt, die mehr Sicherheitselemente enthält als in die Policy beschrieben. Also werden von der Policy-Komponente – und damit von allen Komponenten – nur beschränkt viele Sicherheitselemente verarbeitet. Hier zeigt sich ein großer Vorteil der ereignisbasierten Verarbeitungskette.

Die Größe der verschlüsselten oder signierten Blöcke ist weiterhin durch die nachfolgende Schema-Validierung beschränkt. Auch hier erfolgt die WS-Security-Verarbeitung nicht über das Element hinaus, bei dem die Schema-Validierung einen Fehler findet. Damit gilt also schließlich für den Speicherbedarf und die Laufzeit: $M \in \mathcal{O}(1)$ und $R \in \mathcal{O}(1)$.

8.4 Schutzwirkung gegen Angriffe

Die in diesem Kapitel beschriebene WS-Security-Verarbeitung verhindert Angriffe vom Typ *Attack Obfuscation*. Verschlüsselte Blöcke werden entschlüsselt und darin enthaltene Angriffsmuster (beispielsweise Schema-Verletzungen) können erkannt werden. Weiterhin ist die WS-Security-Verarbeitung natürlich für die Überprüfung der Konformität der Nachricht bezüglich der Sicherheits-Policy notwendig. Das wird im nächsten Kapitel beschrieben.

Kapitel 9

Sicherheits-Policy-Verarbeitung

9.1 Einführung

Die Sicherheits-Policy-Komponente hat zwei Aufgaben: die Überprüfung der Konformität einer Nachricht zur Sicherheits-Policy (engl. *Policy Validation*) und die Durchsetzung dieser Policy durch Abweisung von Nachrichten, die dieser Policy nicht entsprechen (engl. *Policy Enforcement*).

Abbildung 9.1 zeigt den Aufbau dieser Komponente. Sie verarbeitet die Ereignisse, die von der WS-Security-Komponente (siehe vorheriges Kapitel) generiert werden. Die Policy-Verarbeitung ist auf drei Teilkomponenten aufgeteilt, deren Aufgaben im folgenden beschrieben werden.

1. Der Policy Handler (Abschnitt 9.4) generiert aus den Parsing-Ereignissen, die vom Security Handler erzeugt bzw. weitergereicht werden, sogenannte *Policy-Ereignisse* (engl. *Policy Events*), die die Sicherheitseigenschaften der Nachricht abstrakt beschreiben. Diese Policy-Events bilden die Eingabe für den Policy-Validator.

2. Der Policy-Validator (Abschnitt 9.5) entscheidet, ob eine Nachricht konform zur Sicherheits-Policy ist. Damit der Policy-Validator diese Entscheidung treffen kann, muß die Security-Policy, wie sie für den konkreten Web Service vorliegt und in einer Notation nach WS-SecurityPolicy-Standard aufgeschrieben ist, zuvor in eine spezielle Normalform überführt werden (Abschnitt 9.2). Wird vom Policy-Validator eine Verletzung der Sicherheits-Policy festgestellt, so fordert er den Policy-Handler auf, die weitere Verarbeitung der Nachricht abzubrechen, was zur Abweisung der Nachricht führt.

3. Eingebettet in den Policy Handler ist der XPath-Checker (Abschnitt 9.3). Dieser überprüft, ob die Referenzen für signierte und verschlüsselte Nachrichtenteile erfüllt sind.

9.2 Aufbereitung der Sicherheits-Policy

Der erste Schritt der Aufbereitung der Sicherheits-Policy ist die Berechnung der Gesamt-Policy aus der Endpunkt-, Nachrichten- und evtl. Operations-Policy. Zur Analyse und Auswertung der (in WS-SecurityPolicy definierten) Sicherheits-Policy, ist es notwendig, das WS-SecurityPolicy-Dokument in eine semantisch äquivalente *Normalform* zu überführen, bei der

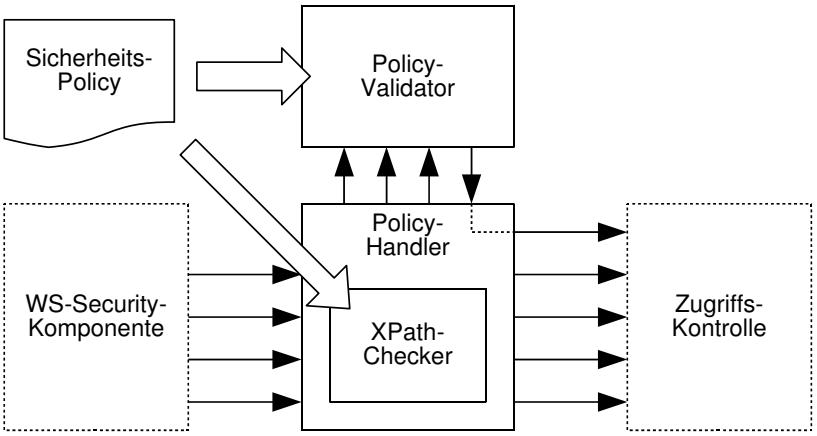


Abbildung 9.1: Architektur der Komponente zur Sicherheits-Policy-Verarbeitung

der `wsp:ExactlyOne`-Operator nur auf der obersten Ebene verkommt. Die dabei entstehende Form ist vergleichbar mit der *Disjunktiven Normalform* von logischen Ausdrücken.

Von WS-Policy wird eine Normalform definiert, die scheinbar diese Forderung erfüllt, ist sie doch wie folgt definiert:

$$ExactlyOne(All(Ass_1, \cdots, Ass_n))$$

wobei Ass_i Policy-Zusicherungen bezeichnen. Allerdings können Policy-Standards, die auf dem WS-Policy-Framework aufsetzen, Zusicherungen definieren, die wiederum `wsp:ExactlyOne`-Operatoren enthalten. So entspricht beispielsweise das folgende WS-SecurityPolicy-Dokument der Normalform gemäß WS-Policy.

```
<wsp:Policy>
  <sp:SymmetricBinding>
    <wsp:Policy>
      <sp:ProtectionToken>
        <wsp:Policy>
          <wsp:ExactlyOne>
            <sp:X509Token>...</sp:X509Token>
            <sp:RelToken>...</sp:RelToken>
          </wsp:ExactlyOne>
        </wsp:Policy>
      </sp:ProtectionToken>
      ...
    </wsp:Policy>
  </sp:SymmetricBinding>
</wsp:Policy>
```

Eine Normalform, bei der der `wsp:ExactlyOne` nur auf der obersten Ebene auftritt, erfordert, dass für Zusicherungen, die Operatoren als Kindelemente enthalten können, Distributivität bezüglich der WS-Policy-Operatoren gilt. Diese Forderung ist für WS-SecurityPolicy erfüllt, d. h. für alle Zusicherungen Ass, Ass_1, Ass_2 aus WS-SecurityPolicy gilt:

$$Ass(ExactlyOne(Ass_1, Ass_2)) \iff ExactlyOne(Ass(Ass_1), Ass(Ass_2))$$

Damit lassen sich alle WS-SecurityPolicy-Dokumente in eine äquivalente Darstellung mit nur einem `wsp:ExactlyOne`-Operator auf der obersten Ebene der Operatoren umformen. Diese Form wird im Folgenden als *Supernormalform* der WS-SecurityPolicy bezeichnet [78]. So sieht die Policy aus dem obigen Beispiel in Supernormalform wie folgt aus:

```
<wsp:Policy>
  <wsp:ExactlyOne>
    <sp:SymmetricBinding>
      <wsp:Policy>
        <sp:ProtectionToken>
          <wsp:Policy>
            <sp:X509Token>...</sp:X509Token>
          </wsp:Policy>
        </sp:ProtectionToken>
        ...
      </wsp:Policy>
    </sp:SymmetricBinding>
    <sp:SymmetricBinding>
      <wsp:Policy>
        <sp:ProtectionToken>
          <wsp:Policy>
            <sp:RelToken>...</sp:RelToken>
          </wsp:Policy>
        </sp:ProtectionToken>
        ...
      </wsp:Policy>
    </sp:SymmetricBinding>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Abbildung 9.2 zeigt die Baumdarstellung einer vollständigen Sicherheits-Policy in Supernormalform. Zum Test, ob eine Nachricht konform zur Policy ist, reicht es nun, die erfüllten Anforderungen zu markieren; sobald ein All-Ast vollständig markiert ist, ist die Policy erfüllt. Im Folgenden wird die Überprüfung der Policy-Konformität nur für eine `wsp:ExactlyOne`-Option dargestellt. Die beschriebenen Verarbeitungsschritte müssen also jeweils für alle Optionen durchgeführt werden. Wird während der Verarbeitung festgestellt, dass alle Optionen nicht erfüllbar sind, so ist die Gesamt-Policy nicht erfüllt. Ist am Ende der Überprüfung eine Option erfüllt, so ist auch die Gesamt-Policy erfüllt.

Jeder Optionen-Zweig enthält allgemeine Policy-Anforderungen und Policy-Anforderungen, die das Signieren und Verschlüsseln von Nachrichtenteilen fordern. Beispiele für die ersteren sind:

- Layout des `wsse:Security`-Headers (`sp:Layout`)
- Benutzung eines Zeitstempels (`sp:IncludeTimestamp`)
- Verschlüsselung von Signaturen (`sp:EncryptSignature`)

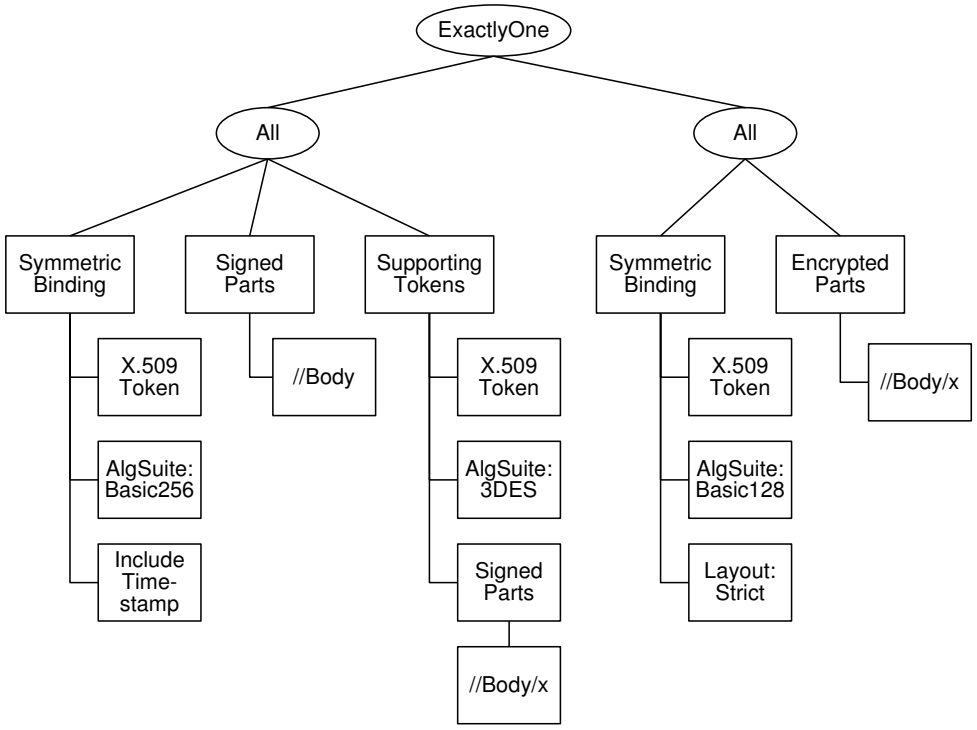


Abbildung 9.2: Struktur einer Sicherheits-Policy (in Supernormalform)

Diese Anforderungen sind einfach zu verifizieren und werden dann entsprechend markiert.

Dies ist für die zweite Art von Anforderungen nicht direkt möglich. Der Grund liegt darin, dass in der Policy keine konkreten Token, sondern nur Tokentypen definiert werden. So werden in der Policy in Abbildung 9.2 vom linken All-Ast zwei X.509-Token gefordert. Wenn nun eine SOAP-Nachricht zwei X.509-Token enthält, so ist es nicht möglich, diese Token eindeutig den Tokentypen in der Policy zuzuordnen. Eindeutig wird die Zuordnung erst durch die Benutzung des Tokens beim Signieren und Verschlüsseln (Ausnahmen: siehe Abschnitt 9.6).

Daher werden die Anforderungen aus der Sicherheits-Policy zu Tripeln der Form $Sig(t, p, a)$ bzw. $Enc(t, p, a)$ zusammengefasst. Dabei ist t ein Tokentyp, p das zu schützende Element und a eine Algorithmen-Gruppe (engl. *Algorithm Suite*). Abbildung 9.3 zeigt die derartig modifizierte Sicherheits-Policy aus dem obigen Beispiel. Formal werden die Tripel wie folgt definiert.

Sei P eine Sicherheits-Policy, beschrieben in WS-SecurityPolicy, sei T^P die Menge der Tokentyp-Forderungen in P, sei R^P die Menge der Referenzen (als XPath-Ausdruck) der zu schützenden Elemente in P und sei A^P die Menge der Algorithmen-Gruppen in P. Dann sei $Sig^P \subseteq T^P \times \left(R^P \cup T^P \cup \{Sig_{MS}\}\right) \times A^P$ die Menge der Signieranforderungen in P. Das bedeutet: Wenn $(t, p, a) \in Sig^P$, dann fordert die Policy P, dass das Element p (entweder ein Sicherheits-Token, durch eine Referenz definiert oder die Hauptsignatur[1]) mit einem Token des Typs t mit den in a definierten Algorithmen signiert ist. Dabei müssen verschiedene t von unterschiedlichen Token erfüllt werden, auch wenn sie denselben Tokentyp fordern (in Abbildung 9.3 durch X.509 und X.509' angedeutet). Sig^P wird aus einer Policy P gemäß den folgenden Regeln konstruiert:

[1] Sig_{MS} von *Message Signature* = Hauptsignatur.

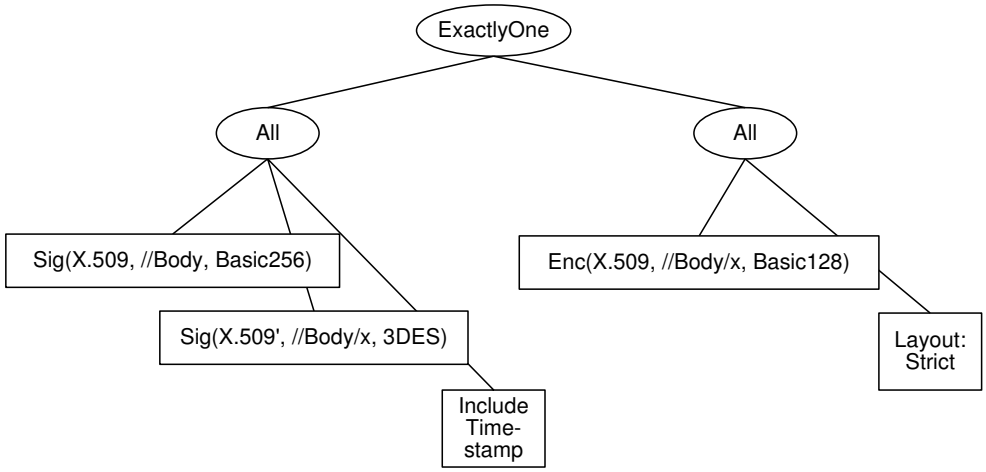


Abbildung 9.3: Struktur einer Sicherheits-Policy mit Anforderungs-Tripeln

- Innerhalb von `sp:SymmetricBinding` oder `sp:AsymmetricBinding` wähle die enthaltene Tokentyp-Forderung für das Haupttoken t_{MS} (aus `sp:*Token`) und die Algorithmen-Forderung a (aus `sp:AlgorithmSuite`). Für jede Referenz r innerhalb eines `sp:SignedParts`-[2] oder `sp:SignedElements`-Elements füge (t_{MS}, r, a) zu Sig^P hinzu.
  Wenn ein Zeitstempel gefordert ist (`sp:IncludeTimestamp`), so füge zusätzlich (t_{MS}, r_{TS}, a) zu Sig^P hinzu, wobei r_{TS} Referenz auf den Zeitstempel ist.[3]

- Innerhalb jeder `sp:*SupportingTokens`-Forderung: für jede enthaltende Tokentyp-Forderung t und Algorithmen-Forderung a (falls nicht vorhanden, wähle a aus der dazugehörigen `sp:*Binding`-Forderung): für jede Referenz r innerhalb eines `sp:SignedParts`- oder `sp:SignedElements`-Elements füge (t, r, a) zu Sig^P hinzu.
  Falls Token-Schutz gefordert ist (`sp:ProtectTokens`), füge zusätzlich (t_{MS}, t, a) zu Sig^P hinzu.

- Innerhalb jeder `sp:Signed*SupportingTokens`-Forderung: für jede enthaltende Tokentyp-Forderung t: füge (t_{MS}, t, a) zu Sig^P hinzu, wobei a die Algorithmen-Forderung und t_{MS} die Tokentyp-Forderung aus der dazugehörigen `sp:*Binding`-Forderung ist.

- Innerhalb jeder `sp:*EndorsingSupportingTokens`-Forderung: für jede enthaltende Tokentyp-Forderung t und die Algorithmen-Forderung a (falls nicht vorhanden, wähle a aus der dazugehörigen `sp:*Binding`-Forderung): füge (t, Sig_{MS}, a) zu Sig^P hinzu.

Analog sei $Enc^P \subseteq T^P \times R^P \times A^P$ die Menge der Verschlüsselungsanforderungen in P. Enc^P wird aus einer Policy P gemäß den folgenden Regeln konstruiert:

- Innerhalb von `sp:SymmetricBinding` oder `sp:AsymmetricBinding` wähle die enthaltene Tokentyp-Forderung t (aus `sp:*Token`) und die Algorithmenforderung a (aus `sp:AlgorithmSuite`). Für jede Referenz r innerhalb eines `sp:EncryptedParts`- oder `sp:EncryptedElements`-Elements füge (t, r, a) zu Enc^P hinzu.

[2] Die vordefinierten Dokumentteile lassen sich leicht in XPath-Ausdrücke umwandeln.
[3] Beispielsweise `/wsse:Security/wsu:Timestamp`.

- Innerhalb jeder `sp:*SupportingTokens`-Forderung: für jede enthaltende Tokentyp-Forderung t und die Algorithmen-Forderung a (falls nicht vorhanden, wähle a aus der dazugehörigen `sp:*Binding`-Forderung): für jede Referenz r innerhalb eines `sp:EncryptedParts`- oder `sp:EncryptedElements`-Elements füge (t, r, a) zu Enc^P hinzu.

9.3 Überprüfung von XPath-Ausdrücken

Wie bereits dargestellt, können Forderungen für signierte und verschlüsselte Nachrichtenteile in WS-SecurityPolicy in XPath-Notation angegeben werden. Als Teil der Validierung einer Nachricht gegen eine Policy muss daher getestet werden, ob ein Element e, das signiert oder verschlüsselt ist, einen bestimmten XPath-Ausdruck r erfüllt. Bei baumbasierter Verarbeitung ist ein solcher Test sehr einfach durchzuführen: Man wendet den XPath-Ausdruck auf den XML-Baum an und testet, ob das Element e in der Menge der Ergebnisknoten enthalten ist. Das ist bei ereignisbasierter Verarbeitung so nicht möglich. Ebenfalls ist es nicht möglich, aus den XML-Ereignissen eines Elements *den* XPath-Ausdruck zu erzeugen, der dieses Element beschreibt.

Für die ereignisbasierte Policy-Überprüfung wird die folgende Methode verwendet. Der XPath Checker wird mit allen in der Sicherheits-Policy vorkommenden Referenzen konfiguriert. Dabei können XPath-Ausdrücke aus den `sp:EncryptedElements`- und `sp:SignedElements`-Elemente direkt übernommen werden. Die vordefinierten Ausdrücke für den SOAP-Body und die SOAP-Header-Blöcke, die in den `sp:EncryptedParts`- und `sp:SignedParts`-Elementen verwendet werden, werden in entsprechende XPath-Ausdrücke[4] umgewandelt.

```
<sp:Body/>
⇝ /env:Envelope/env:Body

<sp:Header Name="localname" Namespace="namespace"/>
⇝ /env:Envelope/env:Header/*[local-name()="localname"]
                            [namespace-uri()="namespace"]
```

Zur Laufzeit des Systems erhält der XPath Checker alle XML-Ereignisse als Eingabe. Diese werden gegen die vorgegebenen Ausdrücke getestet. Sobald ein XML-Ereignis (im Kontext der vorangehenden Ereignisse) einen XPath-Ausdruck erfüllt, wird dies vom Checker signalisiert. Details zur Realisierung der ereignisbasierten XPath-Überprüfung (insbesondere zur Behandlung von Rückwärts-Achsen) sind in [11] und [128] zu finden.

Dabei ist das Ereignis des Elements, auf das der XPath-Ausdruck passt, nicht immer identisch mit dem Ereignis, beim dem dieses entschieden werden kann. Als Beispiel sei das folgende XML-Dokument gegeben.

```
<a>
  <b></b>
</a>
```

Der XPath-Ausdruck `/a[./b]` wird vom Element `a` erfüllt, kann aber erst entschieden werden, wenn das Element `b` verarbeitet wurde. Die XPath-Komponente generiert daher zwei verschiedene Typen von Signalen:

[4] Die Benutzung des `env`-Präfixes in den XPath-Ausdrücken geschieht hier nur zur Erhöhung der Übersichtlichkeit. Korrekt wäre die Benutzung der Funktionen local-name() und namespace-uri().

- possibleMatch(e, r): Das Element[5] e erfüllt *möglicherweise* den XPath-Ausdruck r. Das Ereignis zu e ist auch das aktuelle Ereignis.

- definiteMatch(e, r): Das Element e erfüllt den XPath-Ausdruck r. Das Ereignis zu e ist nicht notwendigerweise das aktuelle Ereignis.

Aus Gründen der Vereinfachung sei angenommen, dass immer zuerst ein possibleMatch$(\cdot)$-Signal erzeugt wird, auch wenn das aktuelle Element bereits zu einem definiteMatch$(\cdot)$ führt. Diese Signale des XPath Checkers zusammen mit den Ereignissen aus der WS-Security-Komponente werden zur Generierung der Policy-Ereignisse verwendet.

9.4 Generierung von Policy-Ereignissen

Der Policy Handler generiert auf folgende Art und Weise Policy-Ereignisse für den Policy Validator. Dabei wird eine Tabelle $SigType \subseteq STRING \times \{REF, TOK, SIG\} \times STRING$ zur Behandlung signierter Elemente mitgeführt. Für jedes Element e (angezeigt durch start(e) oder consumedStart(e)), welches ein ID-Attribut mit Wert *ref* enthält und für das possibleMatch(e, r) oder definiteMatch(e, r) signalisiert wird, wird ein Eintrag $(ref, TYPE, r)$ zu $SigType$ hinzugefügt. Wenn $e =$ `ds:Signature`, dann ist $TYPE = SIG$. Wenn e ein Sicherheits-Token darstellt, dann ist $TYPE = TOK$. In allen anderen Fällen ist $TYPE = REF$.

Signaturen

Aus jeder Signatur (kommuniziert durch consumedStart(ds:Signature) bis consumedEnd(ds:Signature)) wird das Policy-Ereignis

signature(t, n)

generiert. Dabei ist t das Sicherheits-Token, das für diese Signatur verwendet wurde, und n die Anzahl der Referenzen, die diese Signatur signiert.

Verschlüsselte Blöcke

Wird ein consumedStart(xenc:EncryptedData) gelesen und wird für das erste darauf folgende nicht-konsumierte Ereignis e ein possibleMatch(e, r) ausgelöst, so wird bei einem nachfolgenden definiteMatch(e, r) das Policy-Ereignis

encryptedReference(r, t, a)

generiert, wobei t das Sicherheits-Token ist, das zu diesem verschlüsselten Block gehört, und a der verwendete Algorithmus (aus `xenc:EncryptionMethod`) ist.[6]

[5] XPath-Ausdrücke können im Allgemeinen nicht nur auf XML-Elemente verweisen, sondern beispielsweise auch auf XML-Attribute. Referenzen in WS-SecurityPolicy zeigen aber immer auf Elemente.

[6] Es ist möglich, dass auf ein Element mehrere (unterschiedliche) XPath-Ausdrücke passen. Wenn diese eintritt, d. h. *Match(e, r_1) und *Match(e, r_2) erzeugt werden, so wird statt encryptedReference(r, t, a) encryptedReference$(\{r_1, r_2\}, t, a)$ erzeugt. Da das keinen Einfluss auf die weitere Verarbeitung hat, aber die Notation verkomplizieren würde, wird das im Folgenden nicht weiter erwähnt.

Sicherheits-Token

Wird ein Sicherheits-Token eingelesen, z. B. consumedStart(wsse:BinarySecurityToken) so wird das Policy-Ereignis

securityToken(t)

generiert, wobei t das Sicherheits-Token darstellt.

Zeitstempel

Wird ein Zeitstempel eingelesen (also consumedStart(wsu:Timestamp) bis consumedEnd(wsu:Timestamp)), so wird das Policy-Ereignis

timestamp(c, e)

generiert, wobei c den Erzeugungszeitpunkt und e den Verfallszeitpunkt darstellt.

Signierte Elemente

Wird für ein $(ref, TYPE, r) \in SigType$ das Ereignis signedElement(ref, t, a) und das Signal definiteMatch(e, r) gelesen, so ist das Element mit der ID ref von dem Token t signiert und erfüllt den XPath-Ausdruck r. Zu beachten ist, dass dabei zuerst das signedElement($\cdot$)-Ereignis oder das definiteMatch($\cdot$)-Ereignis auftreten kann, dass aber beide in jedem Fall nach dem Erstellen des $SigType$-Eintrags auftreten. Es wird dann das Policy-Ereignis

signedElement(p, t, a)

generiert, wobei p abhängig von $TYPE$ ist. Wenn $TYPE = SIG$, dann ist p die signierte Signatur. Wenn $TYPE = TOK$, dann ist p das signierte Token. Wenn $TYPE = REF$, dann ist $p = r$ der XPath-Ausdruck, den das signierte Element erfüllt.[7]

9.5 Policy-Validierung

9.5.1 Strikte Policy-Interpretation

WS-SecurityPolicy definiert nur *Minimalanforderungen* an die Sicherheitselemente einer Nachricht. Nachrichten, die mehr als die geforderten Sicherheits-Token enthalten, werden explizit erlaubt. Nachrichten, die zusätzliche Elemente zum Integritäts- oder Vertraulichkeits-Schutz enthalten, werden nicht explizit erwähnt, aber auch nicht ausgeschlossen. Diese Eigenschaft kann für Ressourcenverbrauchs-Angriffe ausgenutzt werden, bei denen eine Nachricht viele (zusätzliche) Schutzelemente (d. h. Signaturen und verschlüsselte Blöcke) enthält und der Server mit kryptographischen Operationen belastet wird. Um derartige Angriffe zu verhindern, werden im Folgenden die Definitionen der WS-SecurityPolicy auch als *Maximalanforderungen* angesehen. Dies bedeutet, dass für den `wsse:Security`-Header, der (mittels des `env:actor`- bzw. `env12:role`-Attributs) dem aktuellen SOAP-Knoten zugeordnet ist, das Folgende erfüllt sein muss:

[7] Bezüglich nicht eindeutiger XPath-Ausdrücke gilt hier das gleiche wie oben unter „Verschlüsselte Blöcke“.

- Die Anzahl der Sicherheits-Token ist gleich der Anzahl der geforderten Token in der Policy.
- Die Nachricht enthält nicht mehr Signaturen und verschlüsselte Blöcke als in der Policy gefordert.

Es bleibt die Frage, ob diese Verschärfungen Auswirkungen auf die Benutzung oder die Sicherheit des Web Services haben. Die Sicherheitsanforderungen des Web-Service-Servers sind in der Sicherheits-Policy genau enthalten. Weitere Sicherheitselemente sind aus seiner Sicht nicht notwendig und können nur die Verarbeitung erschweren. Zur Erfüllung der Sicherheits-Policy muss ein Web-Service-Client diese bei der Erzeugung der Sicherheitselemente verarbeiten und hat damit genau die Policy erfüllt. Aus der Sicht der Nachrichtenverarbeitung gibt es keinen Grund, dann noch weitere Sicherheitselemente hinzu zu fügen.

Falls die Sicherheits-Policy die Sicherheitsanforderungen des Clients nicht erfüllt, so sollte der Web Service erst gar nicht aufgerufen werden.

9.5.2 Ereignisgesteuerte Validierung

Die vom Policy Handler erzeugten Policy-Ereignisse werden vom Policy Validator verwendet, um die Konformität einer Nachricht bezüglich der aktuellen WS-SecurityPolicy zu überprüfen. Einfache Policy-Anforderungen, wie das Layout des `wsse:Security`-Headers oder die Existenz eines Zeitstempels, sind dabei sehr einfach zu überprüfen und werden im Folgenden nicht weiter beschrieben.

Der nicht-triviale Teil der Policy-Validierung ist die Zuordnung der Sicherheits-Token und der von diesen geschützten Dokumentteilen zu den entsprechenden Anforderungen in der Policy. Dazu werden aus den Policy-Ereignissen zwei Mengen Sig^M und Enc^M (analog zu Sig^P und Enc^P) generiert, die die signierten und verschlüsselten Blöcke inklusive der verwendeten Sicherheits-Token darstellen. Diese werden dann mit Sig^P und Enc^P abgeglichen.

Formal sei für eine SOAP-Nachricht M, die gegen die Policy P überprüft werden soll, T^M die Menge der Sicherheits-Token in M und A^M die Menge aller in M verwendeten kryptographischen Algorithmen. Dann sei $Sig^M \subseteq T^M \times \left(R^P \cup T^M \cup \{Sig\}\right) \times A^M$ die Menge aller signierten Elemente in M. Analog sei $Enc^M \subseteq T^M \times R^P \times A^M$ die Menge der verschlüsselten Elemente in M.

Die Überprüfung, ob die Nachricht M die Policy P erfüllt, erfolgt mittels des PTMA-Algorithmus (*Policy Token Mapping Algorithm*, [66]).

PTMA-Algorithmus

- Überprüfe, ob $|T^M| = |T^P|$. Sei $n := |T^M|$, $T^M = \{t_1^M, \ldots, t_n^M\}$ und $T^P = \{t_1^P, \ldots, t_n^P\}$.
- Überprüfe jede mögliche Abbildung von T^M zu T^P: Für jede Permutation π der Menge $\{1, \ldots, n\}$ und für jedes $i \in \{1, \ldots, n\}$:
  1. Überprüfe, ob das Token t_i^M dem Typ $t_{\pi(i)}^P$ entspricht.
  2. Seien $S^M = \left\{(r^M, a^M) \mid (t_i^M, r^M, a^M) \in Sig^M\right\}$ und $S^P = \left\{(r^P, a^P) \mid (t_{\pi(i)}^P, r^P, a^P) \in Sig^P\right\}$ (die Referenz-Algorithmus-Paare der Signaturen des Tokens t_i^M bzw. der Tokenforderung $t_{\pi(i)}^P$).
  3. Überprüfe, ob $|S^M| = |S^P|$.

4. Überprüfe, ob für jedes $(r^M, a^M) \in S^M$ ein $(r^P, a^P) \in S^P$ existiert mit $r^M = r^P$ und $a^M \subseteq a^P$.
5. Seien $E^M = \{(r^M, a^M) \mid (t_i^M, r^M, a^M) \in Enc^M\}$ und $E^P = \{(r^P, a^P) \mid (t_{\pi(i)}^P, r^P, a^P) \in Enc^P\}$ (die Referenz-Algorithmus-Paare der verschlüsselten Blöcke des Tokens t_i^M bzw. der Tokenforderung $t_{\pi(i)}^P$).
6. Überprüfe, ob $|E^M| = |E^P|$.
7. Überprüfe, ob für jedes $(r^M, a^M) \in E^M$ ein $(r^P, a^P) \in E^P$ existiert mit $r^M = r^P$ und $a^M \subseteq a^P$.

- Falls keine Abbildung von T^M zu T^P passt, ist die Sicherheits-Policy verletzt.

Um während der ereignisgesteuerten Verarbeitung auch schon vor Ende der SOAP-Nachricht Überprüfungen vornehmen zu können, wird zusätzlich eine Variante PTMA* benötigt. Diese speichert zusätzlich für jedes Token t^M in $OS(t^M)$ die Anzahl der ausstehenden Signaturen dieses Tokens t^M und unterscheidet sich von PTMA in den folgenden Punkten:

3. Überprüfe, ob $|S^M| + OS(t^M) = |S^P|$.
4. *entfällt*
6. Überprüfe, ob $|E^M| \leq |E^P|$.
7. *entfällt*

Im Folgenden wird die Verwendung von PTMA und PTMA* zur ereignisgesteuerten Policy-Validierung innerhalb des Policy Validator beschrieben.

Ereignisgesteuerte Policy-Validierung

- Während der Verarbeitung des SOAP-Headers:

  securityToken(t^M): füge t^M zu T^M hinzu, überprüfe ob eine entsprechende Tokentyp-Forderung $t^P \in T^P$ existiert, überprüfe ob $T^M \leq T^P$.

  signature(t^M, n): setze $OS(t^M) := n$ die Anzahl der ausstehenden Signaturen des Tokens t^M.

  signedElement(p^M, t^M, a^M): füge (p^M, t^M, a^M) zu Sig^M hinzu und überprüfe ob $|Sig^M| \leq |Sig^P|$.

  encryptedReference(r^M, t^M, a^M): füge (r^M, t^M, a^M) zu Enc^M hinzu und überprüfe ob $|Enc^M| \leq |Enc^P|$.

- Am Ende des SOAP-Headers: Starte PTMA*.

- Während der Verarbeitung des SOAP-Bodys:

  signedElement(p^M, t^M, a^M): füge (p^M, t^M, a^M) zu Sig^M hinzu und starte PTMA*.

  encryptedReference(r^M, t^M, a^M): füge (r^M, t^M, a^M) zu Enc^M hinzu und starte PTMA*.

- Am Ende der SOAP-Nachricht: Starte PTMA.

Laufzeit und Speicherbedarf

Die Laufzeit des PTMA-Algorithmus ist $\mathcal{O}\big(|T^M| \cdot |T^P| \cdot (|Sig^P| + |Enc^P|) \cdot (|Sig^M| + |Enc^M|)\big)$. Weiterhin gilt nach Abschnitt 9.5.1:

$$|Sig^M| \leq |Sig^P| \text{ und } |Enc^M| \leq |Enc^P|$$

Außerdem gilt:

$$|T^M| \leq |Sig^M| + |Enc^M| \text{ und } |T^P| \leq |Sig^P| + |Enc^P|$$

Damit ist die Laufzeit des PTMA in $\mathcal{O}\big((|Sig^P| + |Enc^P|)^4\big)$. Die Laufzeit ist also unabhängig von der Nachricht und damit in $\mathcal{O}(1)$.

Der Speicherbedarf hängt nur von der Komplexität der Sicherheits-Policy ab und ist damit ebenfalls in $\mathcal{O}(1)$.

9.6 Identifikation der Hauptsignatur

Wie bereits erwähnt, übernimmt die in WS-SecurityPolicy definierte Hauptsignatur (und damit auch das Haupttoken) eine besondere Rolle, da sie den Nachrichtenerzeuger identifiziert und damit beispielsweise für die Zugriffskontrolle verwendet werden kann. Allerdings ist innerhalb einer SOAP-Nachricht weder das Haupttoken, noch die Hauptsignatur besonders ausgezeichnet. Die Unterscheidung zu anderen Token bzw. Signaturen kann nur über den in der Policy geforderten Tokentyp und die Benutzung (d. h. verwendeter Algorithmus und signierte Elemente) geschehen. So erfolgt es beispielsweise beim PTMA. Wenn er erfolgreich terminiert, so hat er u. a. ein $t^M \in T^M$ gefunden, das die Forderungen an das Haupttoken erfüllt. Diese Abbildung kann aber unter Umständen uneindeutig sein, falls eine Policy für ein Unterstützungs-Token die gleichen Forderungen wie für das Haupttoken enthält. Das kann in der Praxis sehr leicht passieren.

Aus Gründen der Vereinfachung wird meist nur ein einziger Tokentyp und eine einzige Algorithmen-Gruppe verwendet. Und selbst unterschiedliche Algorithmen-Gruppen unterscheiden sich nur geringfügig in den konkreten Algorithmen. So gibt es unter den 16 in WS-SecurityPolicy definierten Algorithmen-Gruppen insgesamt nur zwei mögliche Algorithmenvarianten für die digitale Signatur.

Um eine solche Situation zu vermeiden, müssen für die Hauptsignatur in der Policy eindeutige Forderungen (in Bezug auf die anderen geforderten Signaturen) aufgestellt werden. Die Eindeutigkeit der Forderungen kann dabei über die Algorithmen, den Tokentyp oder die signierten Elemente erfolgen. Während die ersten beiden Möglichkeiten zu starke Einschränkungen für die Unterstützungstoken darstellen, ist das Hinzufügen einer zusätzlichen, exklusiven Signierforderung problemlos möglich. Das zu signierende Element sollte dabei nicht innerhalb des SOAP-Bodys oder des `wsse:Security`-Headers liegen, um Überschneidungen mit typischerweise signierten Elementen zu vermeiden. Zur Gewährleistung der Interoperabilität sollte das Element weiterhin aus einem verbreiteten Standard stammen.

Ein guter Kandidat ist der WS-Addressing-Standard [73]. Zur Vermeidung von Angriffen sollten WS-Addressing-Elemente immer signiert werden [16]. Unter den von WS-Addressing definierten Elementen bietet sich das `wsa:Action`-Feld zur Erkennung der Hauptsignatur an, da es nicht optional ist und nur vom Nachrichtenerzeuger erstellt wird. Mit folgenden zusätzlichen Forderungen kann eine Policy die Signierung dieses Eintrags durch das Haupttoken und damit die eindeutige Identifizierung der Hauptsignatur ermöglichen.

```
<sp:SignedParts>
  <sp:Header Name="Action"
             Namespace="http://www.w3.org/2005/08/addressing"/>
</sp:SignedParts>

<sp:RequiredElements>
  <sp:xPath>
    /env:Envelope/env:Header/wsa:Action
  </sp:xPath>
</sp:RequiredElements>
```

Dieses Haupttoken t wird dann mittels des Ereignisses

$$\mathsf{messageSignerCandidate}(t)$$

an nachfolgende Komponenten weitergegeben. Aus Sicht der Abwehr von DoS-Angriffen kann diese zusätzliche Forderung auch dann sinnvoll sein, wenn das Haupttoken sich bereits ohne diese Erweiterung eindeutig bestimmen lässt. Der Grund liegt darin, dass die Identität, die das Haupttoken repräsentiert, für die Zugriffskontrolle verwendet wird. Zur Erfüllung der Grundregel aus Kapitel 2.3.2 sollte die Zugriffskontrolle möglichst früh erfolgen. Deshalb muss das Haupttoken möglichst früh während der Nachrichtenüberprüfung identifiziert werden. Durch das exklusiv signierte `wsa:Action`-Feld ist sichergestellt, dass das Haupttoken am Ende des SOAP-Headers feststeht.

9.7 Schutzwirkung gegen Angriffe

Die oben beschriebene Validierung gewährleistet eine strikte Konformität der Nachricht zu der Policy. Damit werden Angriffe, die auf der Abweichung von der Sicherheits-Policy basieren, erkannt und abgewehrt. Damit sind beispielsweise *Policy Violation*-Angriffe nicht mehr möglich. Durch strikte Interpretation der Policy werden weiterhin Nachrichten mit zu vielen Sicherheitselementen abgelehnt und damit *Oversized Cryptography*-Angriffe abgewehrt. Schließlich werden auch *XML-Rewriting*-Angriffe durch die Überprüfung gegen eine (gute) Sicherheits-Policy erschwert. Der Angriff in Kapitel 5.3.3 würde beispielsweise durch die Überprüfung gegen eine Policy mit der Signierforderung für das Element `/Envelope/Body` erkannt werden. Bei diesem Beispiel zeigt sich aber auch, dass die Qualität der Policy Einfluss auf die Angriffsmöglichkeiten hat. So würde beispielsweise eine Policy mit der Signieranforderung `//Body` den beschriebenen Angriff immer noch erlauben. Die Analyse von Sicherheits-Policys bezüglich *XML-Rewriting*-Angriffe ist also eine Voraussetzung für die Wirksamkeit der Policy-Überprüfung. Derartige Analysen werden exemplarisch in [16] präsentiert.

Kapitel 10

Zugriffskontrolle

10.1 Einführung

Zugriffskontrolle ist ein wichtiger Mechanismus zur Wahrung aller drei elementaren Sicherheitsforderungen. Sie verhindert unberechtigte Zugriffe auf Daten und Dienste und hilft damit deren Integrität und Vertraulichkeit zu gewährleisten. Wie bereits in Kapitel 4.2.2 erwähnt, hilft Zugriffskontrolle aber auch bei der Abwehr von Denial-of-Service-Angriffen und damit bei der Sicherung der Verfügbarkeit. Wenn anstelle aller Benutzer des Internet nur eine kleine Gruppe von Benutzern auf einen Dienst zugreifen darf, wird die Wahrscheinlichkeit eines Angriffs entsprechend sinken. Zusätzlich kann man in den meisten Fällen davon ausgehen, dass bekannte Benutzer vertrauenswürdiger sind als der durchschnittliche Benutzer. Und selbst wenn diese Annahme falsch ist, kann man bei Verwendung von Zugriffskontrolle den Benutzern, die durch versuchte Angriffe aufgefallen sind, leicht den weiteren Zugriff verbieten.

Bei der Abwehr von DoS-Angriffen gibt es aber zwei entscheidende Probleme. Zum einen kann die Überprüfung der Authentizität – insbesondere bei der Verwendung von kryptographischen Methoden – selbst für DoS-Angriffe missbraucht werden. Zum anderen muss die Zugriffskontrolle auch der Art der Angriffe angepasst sein. Die Zugriffskontrolle ist wirkungslos gegen Angriffe, die ihre Wirkung entfalten, bevor die Zugriffskontrolle durchgeführt wurde. So ist beispielsweise die SSL-Client-Authentifizierung unwirksam gegen SYN-Flut-Angriffe, da der Angriff das TCP-Protokoll zu Ressourcenverbrauch zwingt, bevor das SSL-Protokoll startet.

Im Zusammenhang mit Web Services ist Zugriffskontrolle bislang nur wenig untersucht worden. WS-Security definiert zwar Sicherheitselemente wie Benutzer-Token, digitale Signatur, SAML-Zusicherungen [30] usw., die für Zugriffskontrolle verwendet werden können, macht aber keine Vorgaben, wie dies umgesetzt werden kann. In [40] und [5] werden Zugriffskontroll-Architekturen für Web Services vorgestellt. In [15] wird eine formale Semantik für die Authentifizierung von SOAP-Nachrichten entwickelt. Keine dieser Arbeiten berücksichtigt die Verarbeitung vom SOAP-Nachrichten und damit auch nicht deren Auswirkung auf die Verfügbarkeit eines Web-Services.

Im Folgenden wird ein Zugriffskontrollsystem mit drei Eigenschaften vorgestellt [67]:

1. Flexible Authentifizierung unter Berücksichtigung der Sicherheits-Policy
2. Autorisierung mit beliebigem Autorisierungssystem unabhängig von der Sicherheits-Policy
3. Ereignisgesteuerte, DoS-resistente Durchsetzung der Zugriffskontrolle

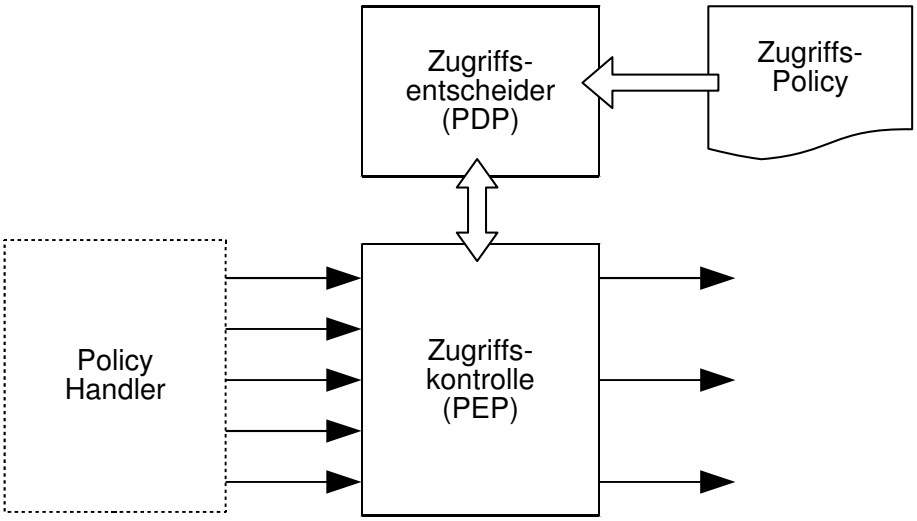


Abbildung 10.1: Architektur der Zugriffskontrollkomponente

Abbildung 10.1 zeigt die Architektur der Zugriffskontrollkomponente. Die eigentliche Zugriffskontrolle befindet sich in der Ereigniskette hinter der Policy-Komponente. Sie führt die Authentifizierung durch, befragt den Zugriffsentscheider nach einer Autorisierungsentscheidung und bricht dementsprechend die Verarbeitung ab oder oder nicht. Die Autorisierungsentscheidung wird auf Grundlage einer Zugriffs-Policy getroffen, in der die Zugriffsrechte der Benutzer festgelegt sind. Die beiden Zugriffskomponenten haben die Rolle eines *Policy-Durchsetzers* (engl. *Policy Enforcement Point (PEP)*) bzw. eines *Policy-Entscheiders* (engl. *Policy Decision Point (PDP)*) [24, 154].

10.2 Authentifizierung

Die Authentifizierung des Erzeugers einer Nachricht kann auf unterschiedlichen Protokollschichten erfolgen. Was schon für Web-Service-Sicherheit im Allgemeinen gesagt wurde, gilt für Authentifizierung im Speziellen: traditionelle Authentifizierungsmethoden, die auf der Transport- oder Vermittlungschicht realisiert werden (beispielsweise IPSec oder SSL), sind für Web Services ungeeignet. Allerdings gibt es im Web-Service-Umfeld keine Standardisierung für Authentifizierung. WS-Security definiert beispielsweise mehrere Sicherheits-Token zum Transport von Identitäten, schließt die Behandlung der Authentifizierung aber explizit aus. Ähnliches gilt für andere Web-Service-Sicherheitsstandards.

Ein erstes Problem für die Authentifizierung ergibt sich, falls eine Nachricht mehr als ein Sicherheits-Token enthält. Es gibt keinerlei Möglichkeit, eines dieser Token als Identität des Nachrichtenerzeugers nachzuweisen. Einen Hinweis für die Lösung dieses Problems gibt WS-SecurityPolicy. So ist in einer Sicherheits-Policy genau ein Token als Haupttoken ausgezeichnet, welches vom Nachrichtenerzeuger stammen muss und für die Authentifizierung verwendet werden kann. Zur Identifizierung dieses Tokens ist vorher eine Validierung der Nachricht gegen die Sicherheits-Policy notwendig.

Ein zweites Problem ist die Sicherung einer Nachricht gegen Modifikation. Nur falls dieses sichergestellt ist, kann eine Nachricht auch als authentisch angesehen werden. Lösungen für die-

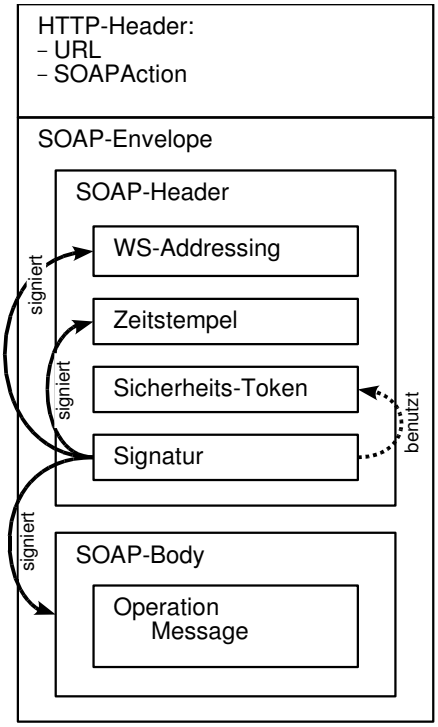


Abbildung 10.2: Zugriffskontrollelemente in einer SOAP-Nachricht (Beispiel)

ses Problem finden sich beispielsweise in [16]. Dort wird u. a. die Signierung von SOAP-Body, Zeitstempel und dem `wsa:Action`-Feld empfohlen. Die hier beschriebene Authentifizierungsmethode geht davon aus, dass die Sicherheits-Policy des Web Services diese Anforderungen enthält.

Abbildung 10.2 zeigt eine Beispielnachricht, in der das Sicherheits-Token die oben genannten Policy-Anforderungen erfüllt.

10.3 Autorisierung

Eine typische Autorisierungsfunktion [105] bildet

1. ein *Autorisierungs-Subjekt* s,
2. eine *Autorisierungs-Ressource* r und
3. eine *Autorisierungs-Operation* o

(die letzteren zwei auch teilweise als *Autorisierungs-Objekt* zusammengefasst) auf $\mathbb{B} := \{true, false\}$ ab[1]. Damit ergibt sich also die folgende Autorisierungsfunktion:

$$auth(s, r, o) \mapsto \mathbb{B}$$

Im Web-Service-Kontext ist das Haupttoken nach WS-SecurityPolicy (siehe Abschnitt 10.2) ein guter Kandidat für das Autorisierungs-Subjekt. Für die Autorisierungs-Ressource und die -Operation wird die Verwendung folgender WSDL-Elemente vorgeschlagen:

[1]Teilweise fließen noch weitere zusätzliche Bedingungen (beispielsweise die aktuelle Uhrzeit) in die Autorisierungsentscheidung mit ein.

- Autorisierungs-Ressource ↔ Web-Service-Endpunkt
- Autorisierungs-Operation ↔ Web-Service-Operation

Diese Abbildung hat wesentliche Vorteile gegenüber Lösungen, bei denen die gesamte SOAP-Nachricht in die Autorisierungsentscheidung mit einbezogen wird. So werden beispielsweise in [40] beliebige Elemente aus der SOAP-Nachricht (definiert durch XPath-Ausdrücke) als Autorisierungs-Objekte verwendet. Solche Ausdrücke können im Allgemeinen erst nach der Verarbeitung der gesamten SOAP-Nachricht ausgewertet werden. Damit kann die Zugriffskontrolle aber Denial-of-Service-Angriffe, bei denen der SOAP-Body „bösartige“ Nachrichtenteile enthält (beispielsweise ein übergroßer SOAP-Body), nicht abwehren.

Die oben vorgeschlagene Abbildung erlaubt es dagegen, bereits am Ende des SOAP-Headers eine Autorisierungsentscheidung zu treffen. Sollte eine fener granulierte Zugriffskontrolle notwendig sein, so kann diese später zusätzlich erfolgen – falls die Autorisierung über die Operation und den Endpunkt die Nachricht nicht bereits abgelehnt hat. Allerdings haben Erfahrungen gezeigt, dass Zugriffsentscheidungen, die Nachrichtenbestandteile innerhalb der Operation betreffen (beispielsweise: darf Benutzer x auf das Bankkonto y zugreifen), von der Web-Service-Applikation getroffen werden sollten und nicht von einem vorgeschaltetem Zugriffskontrollsystem.

Das hier beschriebene System ist vollständig unabhängig davon, wie die Autorisierungsfunktion implementiert wird. Das bedeutet, die Zugriffs-Entscheider-Komponente kann mit einem beliebigen PDP-System realisiert werden. So lassen sich natürlich auch WS-Federation [74] und WS-Trust [123] integrieren. Bei der Realisierung der Zugriffskontrolle für das im nächsten Kapitel beschriebene Firewall-System wurde XACML [122] als Zugriffskontroll-Policy verwendet. Im Anhang B findet sich eine Beispiel-Policy. Diese zeigt, wie bei XACML Autorisierungsregeln unter Benutzung von Subjekt, Ressource und Objekt formuliert werden können. In dem Beispiel erlaubt die Policy den folgenden Zugriff:

- Subjekt = `CN=Gruschka,OU=Unknown,O=CAU,L=Kiel,ST=Unknown,C=DE`
- Ressource = `http://server:8080/WebServices/MyService`
- Objekt = `urn:getLength`

10.4 Ereignisgesteuerte Zugriffskontrolle

Die folgenden ereignisgesteuerten Verarbeitungsschritte führen die Zugriffskontrolle durch. Zur Optimierung der Wirkung gegen DoS-Angriffe wird die Autorisierungsentscheidung möglichst früh im Laufe der Nachrichtenverarbeitung durchgeführt. Schlägt einer der im Weiteren beschriebenen Überprüfungen fehl, so wird die Nachrichtenverarbeitung abgebrochen.

httpURL(url): Die URL der HTTP-Anfrage ist eindeutig dem Web-Service-Endpunkt zugeordnet. Damit ist die Autorisierungs-Ressource r identifiziert:

$$r := url$$

httpSOAPAction(sa): Das HTTP-Header-Element `SOAPAction` ist ein Hinweis auf die tatsächliche Web-Service-Operation (siehe [9], R1127) und wird als vorläufige Autorisierungs-Operation verwendet:

$$o^* := sa$$

messageSignerCandidate(t): Das Token t ist das einzige Token in der SOAP-Nachricht, das die Anforderungen an das Haupttoken erfüllen kann. Die dazugehörige Identität wird daher zum (einzig möglichen) Kandidaten für das Autorisierungs-Subjekt:

$$s^* := t$$

Damit kann an dieser Stelle bereits eine *vorgezogene* Autorisierungsentscheidung getroffen werden:

$$auth(s^*, r, o^*)$$

startElement(o) *(erstes Kind von* `env:Body`*)*: Dieses Element definiert die tatsächliche Web-Service-Operation (siehe [9], R2710). Damit kann überprüft werden, ob die vorläufige Autorisierungs-Operation richtig angenommen war:[2]

$$o = o^*$$

Man erkennt, dass die Autorisierungsentscheidung mit vorläufigen Werten für das Autorisierungs-Subjekt und das Autorisierungs-Objekt durchgeführt wird. Das kann aber niemals dazu führen, dass eine Autorisierungsentscheidung fälschlicherweise negativ ausfällt (engl. *false negative*). Umgekehrt ist es möglich, dass eine Autorisierungsentscheidung fälschlicherweise positiv ausfällt (engl. *false positive*). In diesem Fall wird dies aber an anderer Stelle entdeckt und damit die Nachricht trotzdem (korrekt) abgelehnt, wie im Folgenden gezeigt wird.

Ist $auth(s^*, r, o^*) = false$ und wird dadurch die Nachricht abgelehnt, obwohl $auth(s^*, r, o) = true$ gewesen wäre, so ist das kein *false negative*, denn aufgrund des inkorrekten `SOAPAction`-Felds muss die Nachricht sowieso abgelehnt werden (siehe auch 5.3.1). Ist $auth(s^*, r, o^*) = true$ aber $auth(s^*, r, o) = false$ (*false positive*), so wird das durch den Test $o = o^*$ erkannt und die Nachricht abgelehnt.

Da es außer s^* keine andere Möglichkeit für das Autorisierungs-Subjekt gibt, kann es diesbezüglich kein *false negative* geben. Ein *false positive* ist möglich für den Fall, dass s^* nicht die Forderungen an das Haupttoken erfüllt (d. h. Signierung des Zeitstempels, Signierung des SOAP-Bodys usw.). Dies wird aber im Laufe der WS-Security-Verarbeitung und der Policy-Validierung (automatisch) überprüft. Sollte also beispielsweise s^* gefälscht sein, so wird die Nachricht von einer der beiden vorhergehenden Sicherheitskomponenten abgelehnt.

Laufzeit und Speicherbedarf

Die Autorisierungsfunktion wird pro Nachricht maximal einmal aufgerufen und ist bezüglich Laufzeit und Speicherbedarf offensichtlich unabhängig von der SOAP-Nachricht.

10.5 Schutzwirkung gegen Angriffe

Zugriffskontrolle kann keine konkreten Denial-of-Service-Angriffstypen (wie in Kapitel 5 beschrieben) verhindern, sondern reduziert lediglich durch Beschränkung auf einen geschlossenen Benutzerkreis die Möglichkeiten für Angriffe. Weiterhin ist Zugriffskontrolle natürlich eine wichtige Methode zur Vermeidung von Angriffen gegen Integrität oder Vertraulichkeit.

[2] Diese Überprüfung wird nicht hier sondern unabhängig von der Zugriffskontrolle für alle Nachrichten von der Basisüberprüfungskomponente durchgeführt.

Die Autorisierung verfolgt dabei die in [113] vorgestellte Methode des *monoton ansteigenden Vertrauens*. Der Aufruf der Autorisierungsfunktion führt zu einer schwachen Authentifizierung, da zu diesem Zeitpunkt noch nicht alle Autorisierungsanforderungen überprüft wurden. Die Authentifizierung wird durch weitere Überprüfungsschritte (Signatur-Verifizierung, Überprüfung der Operation) immer stärker. Auf diese Weise wird eine möglichst frühe und trotzdem korrekte Erkennung eines nicht-autorisierten Zugriffs erreicht.

Dabei können viele Angriffe bereits beim Aufruf der Autorisierungsfunktion erkannt werden. Zu diesem Zeitpunkt ist bereits die Signatur für den Zeitstempel überprüft worden. Damit ist das Kopieren einer Signatur und eines Zeitstempels aus der Nachricht eines autorisierten Benutzers die einzige Möglichkeit zur Erzeugung einer Angriffsnachricht, die die Autorisierungsfunktion erfüllt.[3] Eine solche gefälschte Nachricht kann aber nur innerhalb der Lebensdauer des Zeitstempels verwendet werden.

[3] Ausnahme: das `SOAPAction`-Feld stimmt nicht mit der Operation überein. Das wird allerdings gleich am Anfang des SOAP-Bodys erkannt.

Kapitel 11

Überprüfung der Nachrichtenreihenfolge

11.1 Einleitung

Ein BPEL-Prozess definiert implizit ein zustandsbehaftes Protokoll über Web-Service-Nachrichten. Ein möglicher Angriff gegen jedes zustandsbehafte Protokoll ist das Versenden von Nachrichten, die syntaktisch korrekt sind, aber nicht zum aktuellen Zustand des Protokolls passen, also von der korrekten Nachrichtenfolge abweichen. Ähnliche Angriffe sind auch von „klassischen“ zustandsbehafteten Protokollen (z. B. TCP) bekannt. Eine Gegenmaßnahme gegen solche Angriffe ist die Überprüfung aller Nachrichten auf Konformität bezüglich des aktuellen Zustands. So verwerfen die meisten Firewalls TCP-Pakete ohne SYN-Markierung, wenn diese zu keiner aktiven TCP-Verbindung gehören. Die Realisierung einer analogen Gegenmaßnahme für das Kommunikationsprotokoll eines BPEL-Prozesses wirft im Vergleich mit diesen klassischen Protokollen drei zusätzliche Probleme auf.

Zum Ersten ist das Kommunikationsprotokoll nicht *global* definiert, sondern abhängig von dem aktuellen BPEL-Prozess. Es ist also notwendig, aus dem BPEL-Dokument das Kommunikationsprotokoll zu extrahieren und einen Protokoll-Automaten für dieses zu generieren.

Zum Zweiten ist die Verarbeitung der Nachrichten und ihre Zuordnung zu BPEL-Prozessen, Prozessinstanzen und Web-Service-Operationen sehr aufwendig bezüglich des Ressourcenverbrauchs. Wie in Kapitel 5.2.6 gesehen, lässt sich dieses bei ungünstigen Implementierungen für Angriffe ausnutzen.

Soll die Protokollüberprüfung unabhängig von der BPEL-Ablaufumgebung stattfinden (beispielsweise in einem externen Gateway), so tritt schließlich das Problem auf, dass nicht in jedem Fall anhand der Kommunikation der aktuelle Zustand des BPEL-Prozesses (und damit die nächste erlaubte Nachricht) eindeutig festgestellt werden kann. Grundsätzlich ist der Ablauf des Prozesses durch das BPEL-Dokument genau festgelegt und selbst bedingte Ablaufentscheidungen (z. B. beim `switch`) enthalten Bedingungen, die im Normalfall nur Daten aus dem BPEL-Dokument und den ein- und ausgehenden SOAP-Nachrichten enthalten. In diesen Fällen ist der aktuelle Zustand auch für ein externes System eindeutig rekonstruierbar. Allerdings lassen sich Konditionen konstruieren, die nicht von außen nachvollzogen werden können, wenn beispielsweise die interne Uhrzeit der BPEL-Ablaufumgebung einbezogen wird.

Abbildung 11.1 zeigt die Architektur der Komponente zur Zustandsverfolgung innerhalb eines BPEL-Prozesses, die diese Eigenschaften berücksichtigt. Die Zustände des Kommunikationsprotokolls eines Prozesses werden durch einen speziellen Automatentyp, dem

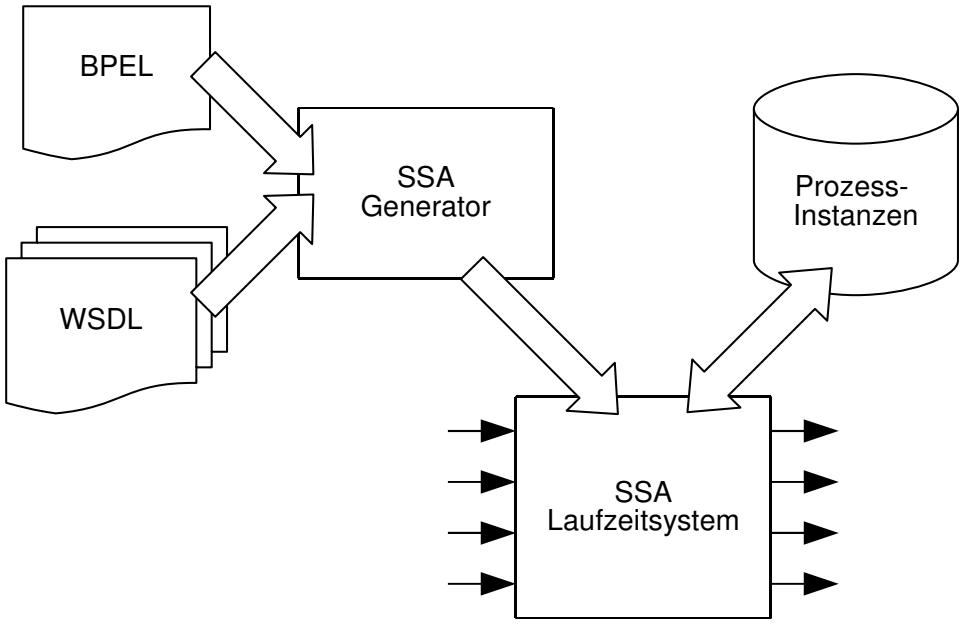


Abbildung 11.1: Architektur der Komponente zur Zustandsverfolgung

Nachfolgermengen-Automat (engl. *Successor Set Automaton*), kurz SSA, abgebildet. Der SSA-Generator erzeugt aus einem BPEL-Dokument und den dazugehörigen Web-Service-Beschreibungen einen SSA [69]. Dieser Automat wird vom SSA-Laufzeitsystem verwendet, um die aktuelle Nachricht auf Konformität zum aktuellen Zustand des Kommunikations-Protokolls zu überprüfen. Zustandsinformationen der Prozessinstanzen werden in einem Zwischenspeicher abgelegt. Im einfachsten Fall werden hier lediglich die Werte der Prozessvariablen gespeichert. Es gibt auch bessere (weil weniger speicherintensive) Varianten, siehe dazu [91]. Zum Schutz vor Angriffen sorgt die ereignisbasierte Verarbeitung im Laufzeitsystem für geringen Ressourcenverbrauch. Desweiteren ist die Verarbeitung so ausgelegt, dass ungültige Nachrichten möglichst früh (bezogen auf die Anzahl der Elemente, die bereits verarbeitet wurden) erkannt und abgelehnt werden können. Und schließlich ist der SSA so ausgelegt, dass er auch nicht eindeutig entscheidbare Verzweigungen im Prozessablauf behandeln kann und dann mehr als einen Zustand als Nachfolger des aktuellen Zustands zulässt.

Der folgende Abschnitt beschreibt den SSA und seine Generierung aus einem BPEL-Dokument im Detail.

11.2 Zustandsverfolgung bei BPEL

11.2.1 Der Nachfolgermengen-Automat

Zur Modellierung und teilweise auch zur Realisierung von zustandsbehafteten Systemen werden oft Transitionssysteme verwendet (siehe beispielsweise [51]). Dabei sind eine Menge von Zuständen durch Transitionen verbunden, die mit Wörtern des Eingabealphabets dekoriert sind. Tritt ein externes Ereignis (beispielsweise eine eingehende Nachricht) ein, wird ein Zustandswechsel vom aktuellen Zustand über die Transition durchgeführt, die mit diesem Ereignis dekoriert ist. Da bei einem BPEL-Prozess der nächste Protokollzustand alleine anhand der aus-

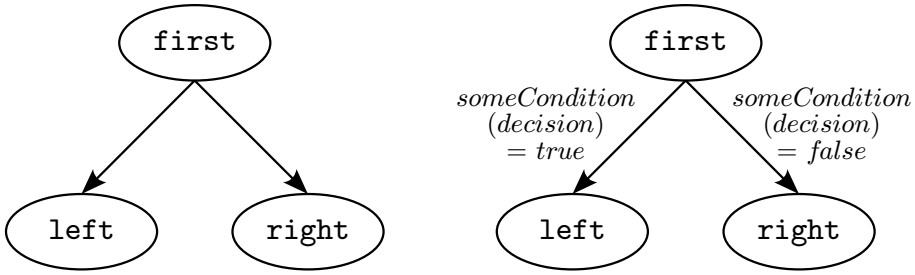


Abbildung 11.2: Beispiel für einen SSA (links) und einen cSSA (rechts)

getauschten Nachricht nicht eindeutig bestimmt werden kann, ist diese Modellierung hier nicht verwendbar. Zur genaueren Diskussion dieses Problems siehe auch [91].

Zur Modellierung des Kommunikationsprotokolls für BPEL-Prozesse zur externen Protokollüberwachung wurde daher das Modell des Nachfolgermengen-Automats entwickelt. Dabei werden die ausgetauschten Nachrichten mit den Zuständen (und nicht mit den Transitionen) identifiziert. Der aktuelle Zustand des Automaten ist dann die zuletzt ausgetauschte Nachricht. Eine Transition von x nach y beschreibt die Relation „Nachricht y ist (ein) Nachfolger der Nachricht x“ im Kommunikationsprotokoll des BPEL-Prozesses. Eine Nachricht (= Zustand) ist also mit der **Menge seiner Nachfolger** verbunden. Eine Nachricht n wird genau dann akzeptiert, wenn eine Transition vom aktuellen Zustand zu dem Zustand existiert, der mit n assoziiert ist.

Zur Illustration sei das folgende Beispiel-BPEL-Dokument gegeben.

```
<sequence>
  <receive operation="first" variable="decision" />
  <switch>
    <case condition="someCondition(decision)">
      <receive operation="left" />
    </case>
    <otherwise>
      <receive operation="right" />
    </otherwise>
  </switch>
</sequence>
```

Der daraus resultierende SSA ist in Abbildung 11.2 links zu sehen. Man erkennt, dass dieser Automat nach der Nachricht `first` sowohl die Nachricht `left` als auch die Nachricht `right` akzeptiert. Diese Zustandsverfolgung ist natürlich nur *unscharf*, da die Auswertung der Bedingung der `case`-Aktivität nicht berücksichtigt wird und damit teilweise Nachrichten akzeptiert werden, die nicht dem aktuellen Zustand des BPEL-Prozesses entsprechen.

Dieses Problem eliminiert der *bedingungsgesteuerte* (engl. *condition driven*) Nachfolgermengen-Automat (cSSA). Bei diesem werden die Transitionen mit den Bedingungen dekoriert, die für diesen Übergang erfüllt sein müssen. Abbildung 11.2 rechts zeigt den cSSA für das obige Beispiel. Der cSSA kann dadurch die Zustandsverfolgung schärfer durchführen als der SSA, hat aber aufgrund der Bedingungsauswertung höhere Laufzeit und Speicherbedarf.

Beim praktischen Einsatz von SSAs bzw. cSSAs muss daher die Präzision der Erkennung gegen den Ressourcenbedarf abgewägt werden.

Definition eines SSA

Sei P ein BPEL-Prozess. Dann ist der *Nachfolgermengen-Automat* SSA von P durch das folgende Tupel beschrieben:

$$SSA = (Q, Q_0, \Delta)$$

Dabei ist:

- Q die (endliche) Menge der Operationen aller Kommunikationsaktivitäten des Prozesses P
- $Q_0 = \{op \in Q \mid op \text{ hat Attribut } \texttt{createInstance="yes"}\}$ die Menge der Instantiierungsoperationen von P
- $\Delta \subseteq Q \times Q$ die bedingungsbehaftete Nachfolgerelation der Kommunikationsaktivitäten von P

Jeder SSA besitzt einen *aktuellen Zustand* $q \in Q$. Ein $q' \in Q$ wird von einer Instanz *akzeptiert*, falls $(q, q') \in \Delta$. In diesem Fall wird q' der neue aktuelle Zustand.

Definition eines cSSA

Sei P ein BPEL-Prozess. Sei weiterhin für eine Menge von Variablen V: $\mathcal{B}(V) = \{b : V \dashrightarrow \mathbb{B}\}$ die Menge der (partiellen) booleschen *Belegungen* von V. Sei schließlich $\mathfrak{A} = \{a : \mathcal{B}(V) \to \mathcal{B}(V)\}$ die Menge der *Variablenzuweisungen*. Dann ist der *bedingungsgesteuerte Nachfolgermengen-Automat* $cSSA$ von P durch das folgende Tupel beschrieben:

$$cSSA = (Q, Q_0, V, C, \Delta)$$

Dabei ist:

- Q die (endliche) Menge der Operationen aller Kommunikationsaktivitäten des Prozesses P
- $Q_0 = \{op \in Q \mid op \text{ hat Attribut } \texttt{createInstance="yes"}\}$ die Menge der Instantiierungsoperationen von P
- V eine Menge von Variablen von P
- $C = \{c : \mathcal{B}(V) \to \mathbb{B}\}$ eine Menge von booleschen Ausdrücken über V
- $\Delta \subseteq Q \times C \times \mathfrak{A} \times Q$ die bedingungsbehaftete Nachfolgerrelation der Kommunikationsaktivitäten von P

Eine *Instanz* von $cSSA$ besteht aus einem *aktuellen Zustand* $q \in Q$ und einer Variablenbelegung $b \in \mathcal{B}(V)$. Ein $q' \in Q$ wird von einer Instanz *akzeptiert*, falls ein $c \in C$ existiert mit $(q, c, a, q') \in \Delta$ und $c(b) = true$ (für ein beliebiges $a \in \mathfrak{A}$). In diesem Fall wird q' der aktuelle Zustand und $a(b)$ die neue Variablenbelegung.

Im folgenden Abschnitt wird die Konstruktion eines SSA bzw. cSSA aus einem BPEL-Dokument beschrieben.

11.2.2 Der Nachfolgermengen-Algorithmus

Zur praktischen Verwendung des SSA-Konzepts zur Zustandsüberprüfung von Nachrichten eines BPEL-Prozesses ist es notwendig, den Nachfolgermengen-Automaten aus einem BPEL-Dokument zu generieren. Die Mengen Q, Q_0, V und C können leicht aus der Prozessdefinition herausgelesen werden. Einzig die Konstruktion der Nachfolgerrelation Δ ist nicht-trivial. Dies wird im Folgenden für den cSSA beschrieben. Für die Konstruktion eines (einfachen) SSA fallen einfach die Bedingungen und die Variablenzuweisungen weg.

Vorher- und Nachherzustände

Zur Erzeugung des Nachfolgermengen-Automaten wird zunächst eine vorläufige Form $cSSA^* = (Q \cup Q^*, Q_0, V, C, \Delta^*)$ erzeugt. Diese enthält neben den Kommunikationsoperationen weitere temporäre Zustände Q^*, die in einem zweiten Schritt eliminiert werden. Zur Erzeugung des cSSA* wird der Baum der Aktivitäten von P rekursiv durchlaufen, für jede Aktivität wird ein Teilautomat generiert und diese dann zum Gesamtautomaten kombiniert. Zur Kombination der Teilautomaten werden auch Zwischenzustände erzeugt. Sei m eine BPEL-Aktivität mit den Kindaktivitäten $m_1, \ldots, m_n$. Dann gelten folgende Bezeichnungen (mit $i \in \{1, \ldots, n\}$):

- $\alpha(m)$ bezeichne den Zustand vor Beginn der Ausführung von m. Das wird *Vorherzustand* genannt.
- $\omega(m)$ bezeichne den Zustand nach Ende der Ausführung von m. Das wird *Nachherzustand* genannt.
- $\alpha'_i(m)$ bezeichne den Zustand im Teilautomaten vom m, der mit dem Vorherzustand der i-ten Kindaktivität (also mit $\alpha(m_i)$) übereinstimmt.
- $\omega'_i(m)$ bezeichne den Zustand im Teilautomaten vom m, der mit dem Nachherzustand der i-ten Kindaktivität (also mit $\omega(m_i)$) übereinstimmt.

Falls m im jeweiligen Zusammenhang eindeutig ist, wird als Bezeichner auch vereinfacht α, ω, α'_i und ω'_i verwendet.

Generierung von Teilautomaten

Im Folgenden wird für die einzelnen BPEL-Aktivitäten der Teilautomat beschrieben, der dieser Aktivität bezüglich der Zustandsfolge entspricht. Zur Vereinfachung wird die Relation Δ graphisch dargestellt. Ein Übergang $x \xrightarrow{c} y$ bedeutet dabei, dass $(x, c, a, y) \in \Delta^*$ gilt. Dabei gilt standardmäßig $a = id_{\mathfrak{A}}$ (die Identität auf $\mathfrak{A}$). Die einzige Ausnahme gilt für die `assign`-Aktivität (siehe unten).

Die `sequence`-Aktivität (siehe Abbildung 11.3) fordert die sequentielle Ausführung der Kindaktivitäten in Dokumentreihenfolge. Im Nachfolgermengen-Automaten wird also für jede Kindaktivität der Nachherzustand ω'_i mit dem Vorherzustand α'_{i+1} der nächsten Kindaktivität verbunden. Die erste und letzte Kindaktivität wird entsprechend mit dem Vorher- und Nachherzustand der `sequence`-Aktivität verbunden.

Die `switch`-Aktivität (siehe Abbildung 11.4) erlaubt die Ausführung genau einer Kindaktivität. Es werden also Vorherzustände der Kindaktivitäten mit dem Vorherzustand der `switch`-Aktivität verbunden (analog für die Nachherzustände). Die Bedingungen für die Transitionen

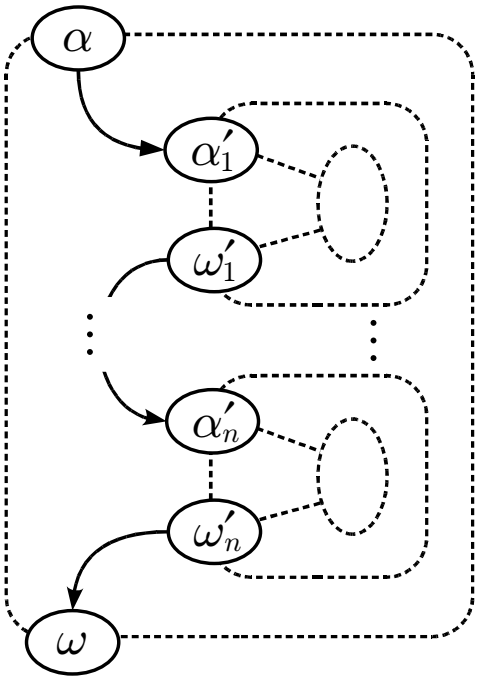


Abbildung 11.3: Teilautomat für `sequence`-Aktivität

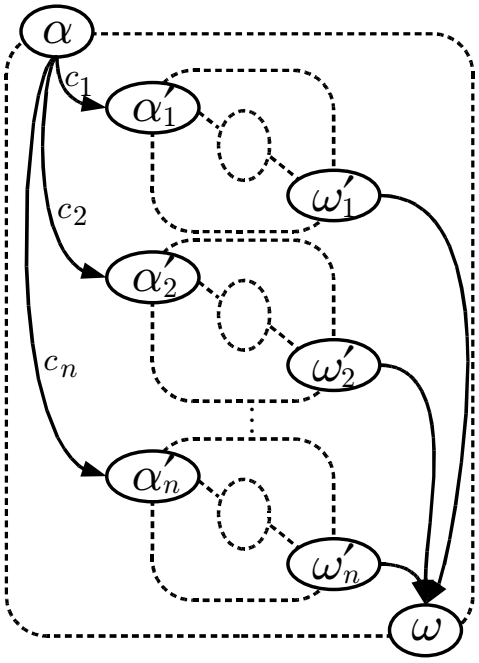


Abbildung 11.4: Teilautomat für `switch`-Aktivität

ergeben sich aus den `condition`-Attributen c_i der `case`-Fälle der `switch`-Aktivität. Eine Besonderheit gilt für den `otherwise`-Fall. Dieser hat kein `condition`-Attribut und wird genau dann ausgeführt, wenn alle anderen Bedingungen nicht erfüllt sind. Ist also der n-te Fall ein `otherwise`, so gilt: $c_n = \neg c_1 \wedge \ldots \wedge \neg c_{n-1}$. Falls die Bedingungen c_i nicht disjunkt sind, müssen sie wie folgt umgeformt werden: $c_i^* = c_i \wedge \neg c_1 \wedge \ldots \wedge \neg c_{i-1}$.

Die `while`-Aktivität (siehe Abbildung 11.5) enthält eine Bedingung c. Ist diese Bedingung beim Erreichen der `while`-Aktivität erfüllt, so wird die Kindaktivität ausgeführt. Ist nach dem Beenden der Kindaktivität die Bedingung immer noch erfüllt, wird die Kindaktivität nochmals ausgeführt. Ist zu irgendeinem Zeitpunkt c nicht erfüllt, so wird die `while`-Aktivität beendet (Verbindung zu ω).

Innerhalb der `pick`-Aktivität (siehe Abbildung 11.5) gibt es zwei Möglichkeiten. Ist die Zeitbedingung des `onAlarm`-Konstrukts erfüllt (hier durch *timeout* abstrahiert), so muss die Kindaktivität des `onAlarm`-Elements als nächstes ausgeführt werden (in der Zeichnung α'_n bis ω'_n). Ist die Zeitbedingung *nicht* erfüllt ($\neg$*timeout*), so kann als nächstes (genau) eine der `onMessage`-Aktivitäten, die eine Kommunikations-Aktivität repräsentieren, ausgeführt werden und danach deren Kindaktivität (in der Zeichnung α'_i bis ω'_i, wobei $i \in \{1, \ldots, n-1\}$).

Enthält die `pick`-Aktivität mehrere `onAlarm`-Konstrukte (nicht in der Zeichnung dargestellt), so können die Zeitbedingungen entsprechend ihrer zeitlichen Abfolge geordnet und daraus mehrere *timeout*-Bedingungen konstruiert werden. Ein Beispiel: die Startzeit der `pick`-Aktivität ist 12 Uhr, die `onAlarm`-Zeitbedingungen sind „um 14 Uhr“ und „nach 1 Stunde“. Dann ergibt sich daraus $timeout_1$ = „um 14 Uhr“ und $timeout_2$ = „zwischen 13 und 14 Uhr“. Statt $\neg$*timeout* wäre die Bedingung für die `onMessage`-Aktivität dann „vor 13 Uhr“.

Das obige Beispiel illustriert auch das Problem von Zeitangaben für eine externe Zustandsüberwachung. Eine Instanz außerhalb der BPEL-Ablaufumgebung kann nicht wissen, wann die `pick`-Aktivität gestartet wird und kann damit relative Zeitangaben nicht genau

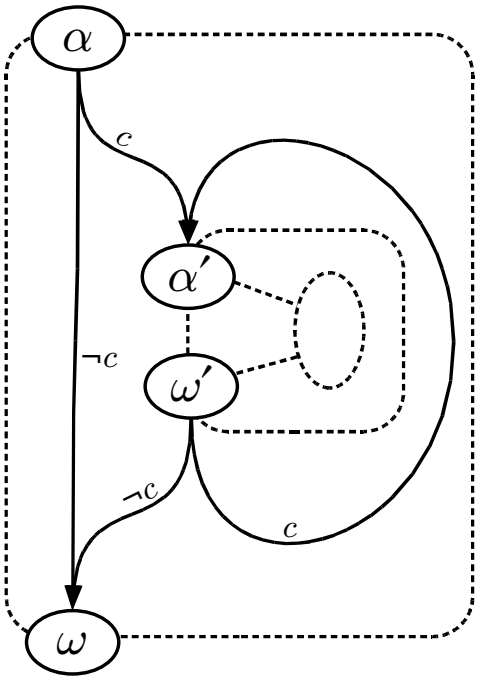


Abbildung 11.5: Teilautomat für `while`-Aktivität

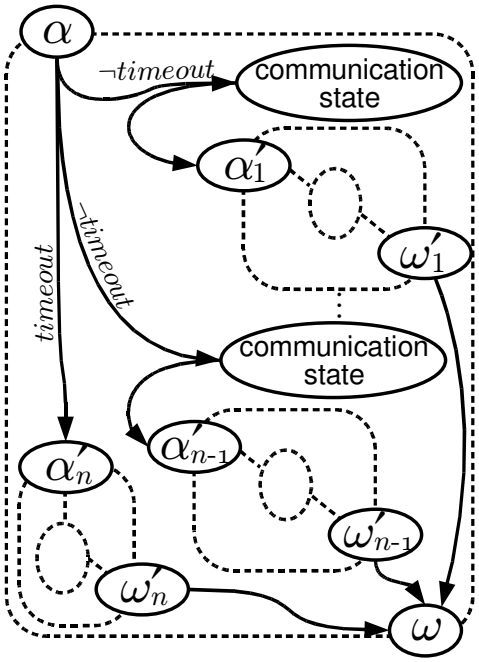


Abbildung 11.6: Teilautomat für `pick`-Aktivität

überwachen. Der Start einer `pick`-Aktivität muss dann durch die letzte vorherige Kommunikationsaktivität abgeschätzt werden.

Für Kommunikationsaktivitäten (siehe Abbildung 11.7) wird ein Zustand aus Q erzeugt, der mit Vorher- und Nachherzustand verbunden wird.

Alle anderen Aktivitäten spielen für die Nachrichtenabfolge keine Rolle und werden daher durch einen leeren Teilautomaten modelliert (siehe Abbildung 11.8). Das gilt auch die `assign`-Aktivität. Die Transition von α nach ω erhält aber zusätzlich eine (nicht-triviale) Variablenzuweisung a. Das ergibt sich aus den `copy`-Anweisungen, die die `assign`-Aktivität enthält.

Eliminierung von Zwischenzuständen

Der durch die oben beschriebene Methode konstruierte cSSA* kann durch Eliminierung der Vorher- und Nachherzustände in einen cSSA überführt werden. Dazu wird für jeden temporären Zustand jeder Quellzustand (bezüglich Δ^*) mit jedem Zielzustand verbunden. Danach kann der temporäre Zustand entfernt werden. Formal bedeutet das für einen gegebenen $cSAA^* = (Q \cup Q^*, Q_0, V, C, \Delta^*)$:

Für jedes $e_t \in Q^*$ wähle $S = \left\{e \in Q \cup Q^* \;\middle|\; (e, c, a, e_t) \in \Delta^*\right\}$ und $Z = \left\{e_z \in Q \cup Q^* \;\middle|\; (e_t, c, a, e) \in \Delta^*\right\}$. Für jedes $(e, c, a, e_t) \in S$ und für jedes $(e_t, c', a', e') \in Z$: füge (e, cc', aa', e') zu Δ^* hinzu. Schließlich wird e_t aus Q^* entfernt und S und Z aus Δ^* entfernt.

Das entstandene Δ^* enthält damit keine Relation über Elemente aus Q^* mehr. Mit $\Delta := \Delta^*$ ergibt sich dann $cSAA = (Q, Q_0, V, C, \Delta)$.

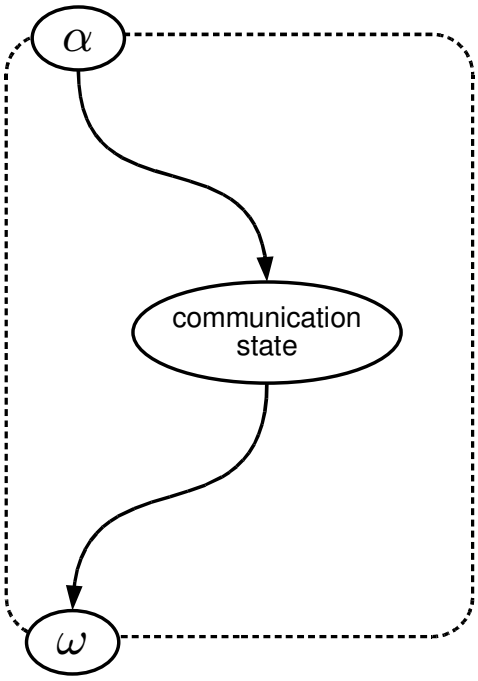


Abbildung 11.7: Teilautomat für Kommunikations-Aktivitäten

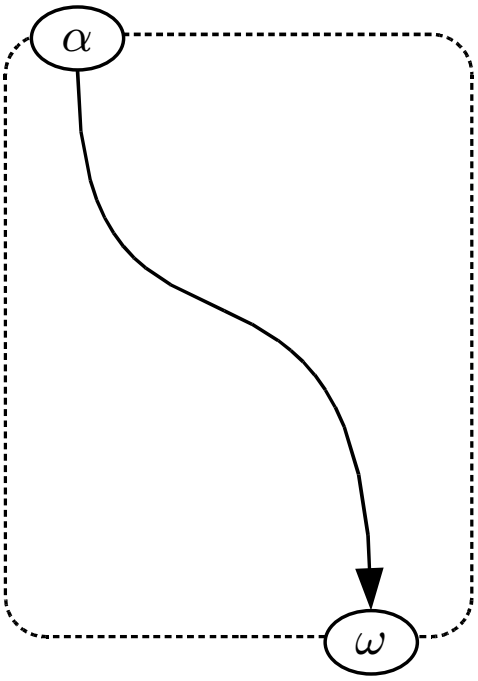


Abbildung 11.8: Teilautomat für alle anderen Aktivitäten

11.3 Ereignisgesteuerte Protokollvalidierung

Der so erzeugte Protokollautomat kann für die effiziente Überprüfung der Konformität eines Web-Service-Aufrufs bezüglich der BPEL-Definition verwendet werden. Um eine eingehende SOAP-Nachricht auf den SSA anwenden zu können, ist es zunächst notwendig, die Nachricht einem BPEL-Prozess und dann auch einer BPEL-Prozessinstanz zuzuordnen.

11.3.1 Prozess- und Prozessinstanzzuordung

Der BPEL-Prozess wird mit einem WSDL-Endpunkt identifiziert, welcher wiederum durch die HTTP-URL adressiert wird. Damit kann der BPEL-Prozess P durch das Ereignis $\mathsf{httpRequest}(URL)$ ausgewählt werden. Durch das `SOAPAction`-Feld im HTTP-Request wird ebenfalls die BPEL-Operation *op* adressiert, die für die Ausführung des SSAs benötigt wird.

Für die Auswahl der Instanz innerhalb des BPEL-Prozesses gilt es zwei Fälle zu unterscheiden. Falls *op* zu einer BPEL-Operation mit dem Attribut `createInstance="yes"` gehört, so erzeugt dieser Aufruf eine neue Prozessinstanz in der BPEL-Ablaufumgebung und damit auch in dieser Komponente. Es wird also für P eine neue Instanz eines entsprechenden SSAs erzeugt. Der aktuelle Zustand dieses SSAs ist *op*.

Im anderen Fall müssen die Werte der zu *op* gehörenden Korrelationsgruppe mit denen der bereits laufenden Instanzen verglichen werden. Das bedeutet konkret: Für jede Prozessinstanz wird im Prozessinstanz-Speicher eine Tabelle für alle Werte aller im Prozess definierten Korrelationseigenschaften C angelegt. Für jede Operation o ist eine Teilmenge $C_o \subseteq C$ mit Korrelationseigenschaften für diese Operation festgelegt. Diese Eigenschaften müssen in der dazugehörigen eingehenden Nachricht enthalten sein und dienen der Identifikation der Prozessinstanz. Die Auswahl der richtigen Instanz kann mittels einer *Kandidatenliste* ereignisbasiert

erfolgen. Am Anfang der Nachricht sind alle Instanzen des adressierten Prozesses in der Kandidatenliste. Für jede Ereignisfolge

startElement(c)
character(v)
endElement(c)

mit $c \in C_{op}$, wird der aktuelle Wert v mit den gespeicherten Werten für c der Instanzen in der Kandidatenliste verglichen. Die Instanzen, für die diese Werte ungleich sind, werden aus der Kandidatenliste entfernt. Wird die Kandidatenliste im Verlauf der Verarbeitung der Nachricht leer, so passt die Nachricht zu keiner laufenden Instanz und wird abgelehnt. Ist nach Verarbeitung aller Werte aus C_{op} mehr als eine Instanz in der Kandidatenliste, so kann die Nachricht nicht eindeutig zugeordnet werden. Dieser Fall darf laut Standard nicht auftreten und so wird die Nachricht auch in diesem Fall abgelehnt.

Falls die Kandidatenliste einelementig ist, kann die Instanz eindeutig bestimmt werden. Das SSA-Laufzeitsystem aktualisiert die gespeicherten Korrelationseigenschaften mit denen aus der Nachricht und wählt den zugehörigen SSA aus. Dieser kann dann für die Überprüfung der korrekten Nachrichtenfolge verwendet werden.

11.3.2 Anwendung des Nachfolgermengen-Automaten

Wie bereits oben beschrieben, wird die Operation *op* von dem SSA akzeptiert, falls im SSA ein Übergang vom aktuellen Zustand q nach *op* existiert. Das bedeutet:

bei einem (einfachen) SSA: (q, op) existiert

bei einem cSSA: (q, c, a, op) existiert und zusätzlich ist die Kondition an diesem Übergang erfüllt, d. h. $c(b) = true$.

Ist dieses nicht erfüllt, so wird die Nachricht abgelehnt. Ansonsten wird *op* der neue aktuelle Zustand. Beim cSSA wird zusätzlich die Variablenbelegung aktualisiert, d. h. $a(b)$ wird die neue Variablenbelegung.

11.3.3 Laufzeit und Speicherbedarf

Die Laufzeit der Auswahl der korrekten Prozessinstanz ist linear abhängig von der Größe der SOAP-Nachricht (und von der Anzahl der laufenden Instanzen). Alle anderen Operationen (Auswahl des Prozesses, Durchlauf des SSAs usw.) sind sogar nur vom BPEL-Dokument abhängig. Damit ist die Forderung $R \in \mathcal{O}(n)$ erfüllt. Durch die Längenbeschränkung der SOAP-Nachricht durch die Schema-Validierung ist dann auch $R \in \mathcal{O}(1)$ erfüllt.

Der Speicherbedarf pro Instanz ergibt sich aus der Komplexität des SSAs, der Anzahl der Korrelationseigenschaften und deren (längenbeschränkte) Schemadefinitionen. Damit ist der Speicherbedarf unabhängig von der Nachrichtengröße ($M \in \mathcal{O}(1)$). Es ist aber leicht einzusehen, dass der Gesamtspeicherbedarf über alle Instanzen unbeschränkt ist. Damit ist dieses System (ebenso wie eine BPEL-Laufzeitumgebung) angreifbar, beispielsweise durch eine große Anzahl Zugriffe, durch die jeweils neue Instanzen erzeugt werden (*Instantiation Flooding* [92]). Derartige Angriffe sind ein offenes Problem, das in zukünftigen Arbeiten behandelt werden soll.

11.4 Schutzwirkung gegen Angriffe

Die Überprüfung von eingehenden Nachrichten durch die Zustandsverfolgungskomponente detektiert alle Nachrichten, die nicht zu einer laufenden Instanz eines BPEL-Prozesses gehören, und solche, die zwar zu einer laufenden Instanz gehören, aber im aktuellen Zustand dieser Instanz nicht erlaubt sind. Damit können Angriffe, bei denen (syntaktisch korrekte) Nachrichten verwendet werden, die von der definierten Nachrichtenreihenfolge abweichen, erkannt und verhindert werden. So ist beispielsweise der *BPEL State Deviation*-Angriff nicht mehr möglich.

Teil III

Realisierung und Abschluss

Kapitel 12

Implementierung

12.1 Netzwerkarchitektur

Wie bereits erwähnt, können die in den vorangegangenen Kapiteln vorgestellten Schutzkonzepte auf unterschiedliche Weise in die Web-Service-Verarbeitung integriert werden. Als *Proof-of-Concept*-Implementierung wurden die Konzepte als Web-Service-Firewall realisiert. Das entstandene System trägt den Namen CheckWay.

Abbildung 12.1 zeigt die Netzwerkarchitektur und die Einbindung in die Web-Service-Verarbeitung von CheckWay. Firewalls [31] werden typischerweise an der Grenze eines Sicherheitsbereichs (engl. *Security Domain*), also beispielsweise am Zugangspunkt eines Unternehmensnetzwerks, eingesetzt. Das gilt natürlich auch für eine Web-Service-Firewall wie CheckWay. Eine solche Firewall wird dann aber nicht als erste Verteidigungslinie, sondern hinter einem Paketfilter o. ä. eingesetzt werden.

Die Web-Service-Kommunikation läuft in diesem Szenario grob wie folgt ab. Die Firewall erhält vom Server die Web-Service-Beschreibung und die Sicherheits-Policy. Diese Metadaten werden modifiziert und an den Client weitergegeben. In der Web-Service-Beschreibung werden dabei im Wesentlichen zwei Änderungen durchgeführt. Zum einen werden die Schemata, wie in Kapitel 7 beschrieben, modifiziert. Zum anderen wird der Web-Service-Endpunkt abgeändert, so dass die angegebene URL nicht mehr auf den Web-Service-Server, sondern auf die Web-Service-Firewall zeigt.

Die Sicherheits-Policy wird in der Art modifiziert, dass die Sicherheits-Anforderungen von CheckWay hinzugefügt werden. Dies sind genau die Anforderungen, die die Firewall verarbeitet. Im einfachsten Fall ist die Sicherheits-Policy des Servers leer bzw. existiert nicht, beispielsweise auch dann, wenn der Web-Service-Server gar keine Sicherheitsmaßnahmen unterstützt. Von dieser Situation soll im Folgenden ausgegangen werden.

Der eigentliche Aufruf des Web Services erfolgt in diesem Fall derart, dass der Client eine SOAP-Nachricht erstellt, diese entsprechend der Sicherheits-Policy mit WS-Security-Elementen versieht und sie an die Firewall verschickt. Dort wird die Nachricht entsprechend der Vorschriften aus der Web-Service-Beschreibung, der Sicherheits-Policy und der Zugriffs-Policy überprüft. In Verlauf dieser Überprüfungen werden auch die Sicherheits-Elemente der Firewall-eigenen Sicherheits-Policy aus der Nachricht entfernt. Falls diese Überprüfung fehlerfrei durchläuft, wird die dadurch entstehende SOAP-Nachricht an den Web-Service-Server weitergeleitet. Im anderen Fall erhält der Client eine SOAP-Fault-Nachricht [21].

Die SOAP-Response des Web-Service-Servers durchläuft auch die Firewall und kann dabei ebenfalls bestimmten Überprüfungen, wie beispielsweise der Schema-Validierung, unterzogen

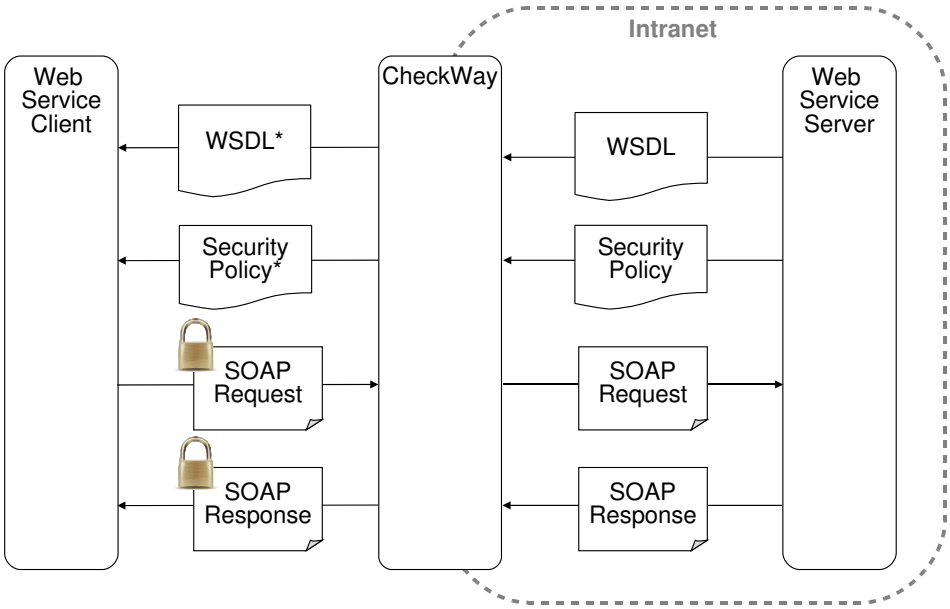


Abbildung 12.1: Vereinfachte Netzwerkarchitektur für Web-Service-Firewall

werden. Dies dient nicht dem Schutz der Verfügbarkeit des Servers, kann aber Angriffe gegen den Client verhindern oder das „Herausschmuggeln“ von Informationen aus dem Sicherheits-Bereich erschweren. Weiterhin wird die Nachricht mit WS-Security-Elementen angereichert, wie sie in der Sicherheits-Policy für die SOAP-Response spezifiziert sind. Diese Policy-konforme Anreicherung der SOAP-Nachricht mit Sicherheits-Elementen geschieht ebenso wie die Verarbeitung der eingehenden Nachricht ereignisbasiert. Hier stehen allerdings nicht die Robustheit gegen Angriffe sondern Effizienzgründe im Vordergrund. Dieser Teil von CheckWay wird im Folgenden nicht weiter betrachtet; Details dazu finden sich in [68].

In der Kommunikation zwischen Web-Service-Client und Web-Service-Server agiert CheckWay also als Proxy, der Server ist für den Client nicht sichtbar. Diese Art der Realisierung des Web-Schutzes hat mehrere Vorteile:

- Gleichzeitiger Schutz von mehreren Web-Service-Servern auch unterschiedlicher Web-Service-Frameworks. Das ist insbesondere bei einer heterogenen IT-Infrastruktur von Interesse.

- Unterschiedliche Sicherheitsniveaus innerhalb und außerhalb des Sicherheitsbereichs. Für Nachrichten innerhalb eines lokalen Netzes herrschen typischerweise geringere Sicherheitsanforderungen als für Nachrichten von außerhalb. Dies betrifft sowohl den Schutz vor Angriffen als auch die Anforderungen bzgl. Vertraulichkeit und Integrität. Mit der oben vorgestellten Architektur lassen sich diese unterschiedlichen Anforderungen feingranular realisieren.

- Verstecken des Web-Service-Servers. Die Modifikation des Web-Service-Endpunkts erlaubt es, nicht nur den Endpunkt-Host sondern auch den Endpunkt-Pfad zu verändern und so Details über die interne Web-Service-Struktur vor Außenstehenden zu verstecken.

- Verstecken von Web-Service-Operationen. CheckWay erlaubt es, den Zugriff auf einzelne Web-Service-Operationen eines Web Services zu verhindern. Paketfilter können dieses

nicht leisten und auch HTTP-Firewalls können beispielsweise von *SOAPAction-Spoofing*-Angriffen überlistet werden.

Ein gewichtiger Nachteil dieser Architektur ist, dass die Web-Service-Firewall einen *Single Point of Failure* darstellt. Allerdings lässt sich dieser dann auch wiederum redundant auslegen und auf mehrere Rechner verteilen. Eine solche Hardware-Skalierung ist bei einer Einbindung des Schutzsystems in ein Web-Service-Framework nicht möglich.

Konfiguration

Zur Realisierung als Web-Service-Firewall muss CheckWay im Einzelnen mit folgenden Informationen konfiguriert werden (hier für den Schutz eines einzelnen Servers angegeben):

- Metadaten des Web-Service-Servers:
  - BPEL-Beschreibung (nur für BPEL-Server)
  - Web-Service-Beschreibung
  - Sicherheits-Policy (falls vorhanden)
- Konfigurationsdaten für die Firewall:
  - Endpunkt, unter dem der Web Service von außen erreichbar sein soll
  - Operationen, die versteckt werden sollen
  - Konfigurationsdaten für Schema-Beschränkungen
  - Sicherheits-Policy der Firewall
  - Zugriffskontroll-Policy
  - Zertifikate und krytopgraphische Schlüssel

12.2 Aufbau der Firewall

12.2.1 Architektur

Abbildung 12.2 zeigt die interne Struktur von CheckWay. Die Firewall besteht aus zwei Hauptkomponenten: dem Compiler zum Vorverarbeiten der Metadaten und der Laufzeitkomponente, dem Gateway. Der Compiler hat im Einzelnen folgende Aufgaben:

Verarbeitung der Web-Service-Beschreibung: Dabei werden zunächst aus der Web-Service-Beschreibung XML Schemata für die Menge der gültigen SOAP-Nachrichten des Web Services generiert. Auf diese Schemata werden dann die Schema-Beschränkungen angewendet. Aus den so entstandenen Schemata und dem konfigurierten „äußeren“ Endpunkt wird dann eine Web-Service-Beschreibung generiert, die Web-Service-Clients für die Benutzung des Web Services (über die Firewall) zur Verfügung gestellt werden kann. Die modifizierten Schemata werden außerdem der Gateway-Komponente übergeben.

Verarbeitung der Sicherheits-Policy: Die Sicherheits-Policy des Web-Services wird mit der Sicherheits-Policy vereinigt. Die so entstandene Policy wird Web-Service-Clients zur Verfügung gestellt. Weiterhin werden aus dieser Policy die Sicherheits-Forderungen extrahiert und an die Gateway-Komponente übergeben.

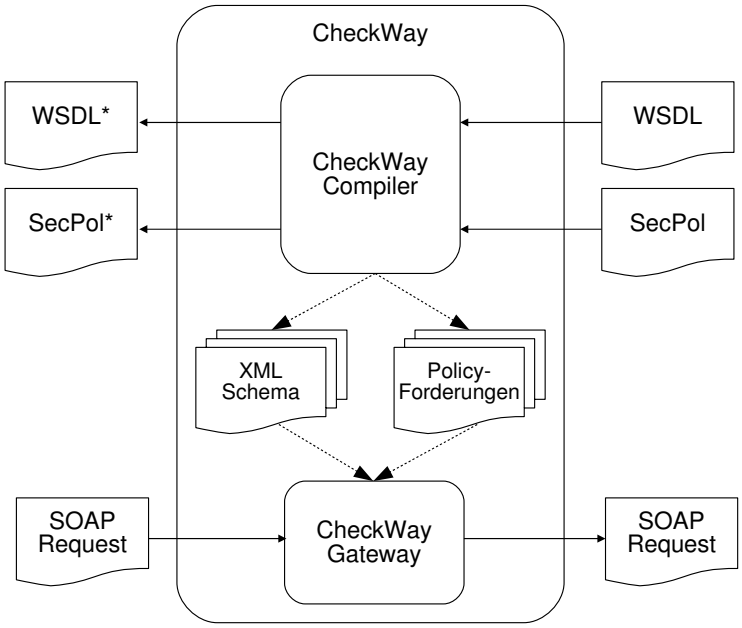


Abbildung 12.2: Architektur von CheckWay

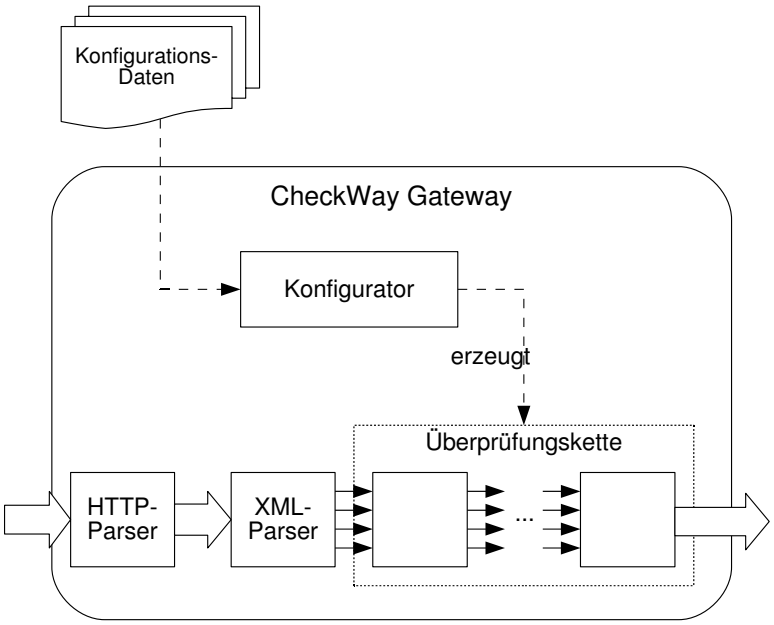


Abbildung 12.3: Architektur der CheckWay-Gateway-Komponente

Abbildung 12.4: Kette der Überprüfungskomponenten

Der Aufbau des Gateways ist in Abbildung 12.3 dargestellt. Die Konfigurationsdateien (die auch die Schemata und die Policy-Forderungen umfassen) werden von einer Konfigurationskomponente eingelesen und daraus werden die Kette der Überprüfungskomponenten erstellt. Diese Kette ist so aufgebaut, wie sie im Kapitel 6 entwickelt wurde (siehe Abbildung 12.4).

Der Überprüfungskette vorangestellt ist ein HTTP-Parser zur Verarbeitung des HTTP-Headers und ein XML-Parser, der die SOAP-Nachricht in SAX-Ereignisse zerlegt.

Dabei werden für jeden Web-Service-Endpunkt eigene Verarbeitungsketten erzeugt. Nach der Verarbeitung des HTTP-Headers ist der Web-Service-Endpunkt (durch die HTTP-URL) identifiziert und der Eingabestrom wird an die entsprechende Verarbeitungskette übergeben. Zur Behandlung paralleler Aufrufe des gleichen Endpunktes steht für jeden Endpunkt sogar eine Menge von Verarbeitungsketten zur Verfügung. Beim Aufruf des Endpunkts wird eine freie Verarbeitungskette aus diesem Reservoir entnommen. Ist dort keine freie vorhanden, wird eine neue Kette erzeugt. Nach der Bearbeitung der Nachricht wird die Verarbeitungskette wieder als „frei“ gekennzeichnet. Diese Verfahren wird auch als *Objekt-Pool* [64] bezeichnet.

12.2.2 Design

Zur Realisierung einer Kette von ereignisverarbeitenden Komponenten existieren verschiedene Design-Lösungen. Eine der bekanntesten ist das Entwurfsmuster *Zuständigkeitskette* (engl. *Chain of Resposibility*) der sogenannten „Gang of Four“ [60]. Darin werden zwei Möglichkeiten für die Schnittstelle zwischen zwei Komponenten vorgeschlagen. Die erste verwendet nur eine einzige Methode für die Übergabe von Ereignissen. Der Typ des Ereignisses wird nur durch das dabei übergebene Objekt definiert. Diese Realisierung hat zwei Nachteile. Zum einen ist dadurch keine Typsicherheit gegeben. Zum anderen widerspricht sie der üblichen Vorgehensweise bei der SAX-basierten XML-Verarbeitung. Die zweite Möglichkeit ist eine feste Schnittstelle mit unterschiedlichen Methoden für die einzelnen Ereignisse. Das entspricht genau der typischen SAX-Realisierung, führt aber zu Problemen bei der Erweiterbarkeit eines solchen Systems.

Wie in den vorangegangenen Kapiteln gezeigt, werden die Ereignisse nicht nur zur Weitergabe von Elementen des XML-Dokuments verwendet, sondern auch zum Transport von Metainformationen, wie z. B. aus der Nachricht entfernte Sicherheitselemente, Ergebnisse der Policy-Validierung oder auch Informationen aus dem HTTP-Header. Wird im Verlauf der Entwicklung der Komponenten ein neues Ereignis erzeugt und damit eine neue Methode der Schnittstelle hinzugefügt, so muss diese Methode allen Komponenten der Ereigniskette hinzugefügt werden. Um dieses Problem zu umgehen, wurde für das Design eine modifizierte Version des Entwurfsmusters entwickelt. Diese wird im nächsten Abschnitt vorgestellt.

Entwurfsmuster für Ereignisketten

Abbildung 12.5 zeigt das Klassendiagramm für das Entwurfsmuster *Ereigniskette*, das eine Weiterentwicklung der oben genannten Zuständigkeitskette darstellt. Die zentrale Rolle spielt dabei die abstrakte Klasse AbstractEventHandler, welche alle Schnittstellen (HandlerInterface*)

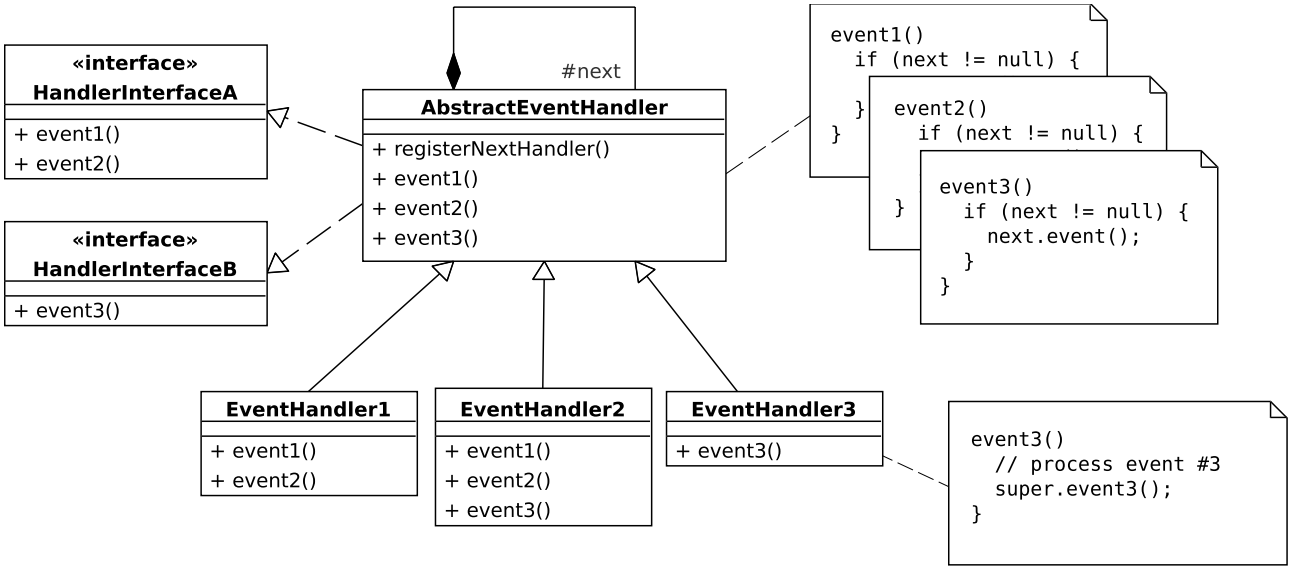


Abbildung 12.5: Klassendiagramm des Entwurfsmusters *Ereigniskette*

der Ereigniskette implementiert. Sie enthält ein Attribut vom Typ AbstractEventHandler, dass als Referenz auf die nächste Komponente in der Ereigniskette dient. Die Ereignismethoden enthalten als Default-Implementierung den Aufruf der gleichnamigen Methode bei dieser nächsten Komponente. Die Komponenten der Ereigniskette (EventHandler*) erweitern die Klasse AbstractEventHandler und brauchen lediglich die Methoden zu überschreiben, deren Ereignis sie bearbeiten wollen. Alle anderen Ereignisse werden durch die Vorgabeimplementierung an die nachfolgende Komponente weitergegeben.

Dies bedeutet ebenfalls, dass bei Erweiterung der Schnittstelle für die Ereigniskette nur die Klassen modifiziert werden müssen, die die neue Methode explizit benutzen. So würde in dem Beispiel in Abbildung 12.5 das nachträgliche Hinzufügen der Schnittstelle HandlerInterfaceB keinerlei Änderungen der Klasse EventHandler1 erfordern.

Dieses Entwurfsmuster erlaubt damit Realisierungen einer Ereigniskette mit folgenden Eigenschaften:

- Typsicherheit der Ereignismethoden
- Konformität zur SAX-Methodik
- Operation auf Teilmengen der Gesamtschnittstelle
- leichte Erweiterbarkeit der Schnittstelle der Ereigniskette

Design von CheckWay

Abbildung 12.6 zeigt die Realisierung der Kette der Überprüfungskomponenten in CheckWay unter Benutzung des oben vorgestellten Entwurfsmusters. Die abstrakte Komponentenklasse trägt hier den Namen EventHandler. Sie implementiert alle Schnittstellen der Überprüfungskette und ist die Superklasse aller Komponenten dieser Kette. Zu erkennen sind dabei auch die Methoden der original SAX-Spezifikation, die in der Schnittstelle ExtendedContentHandler enthalten sind. Die folgende Tabelle zeigt den Zusammenhang zwischen den Überprüfungskomponenten und der realisierenden Klasse im CheckWay-System.

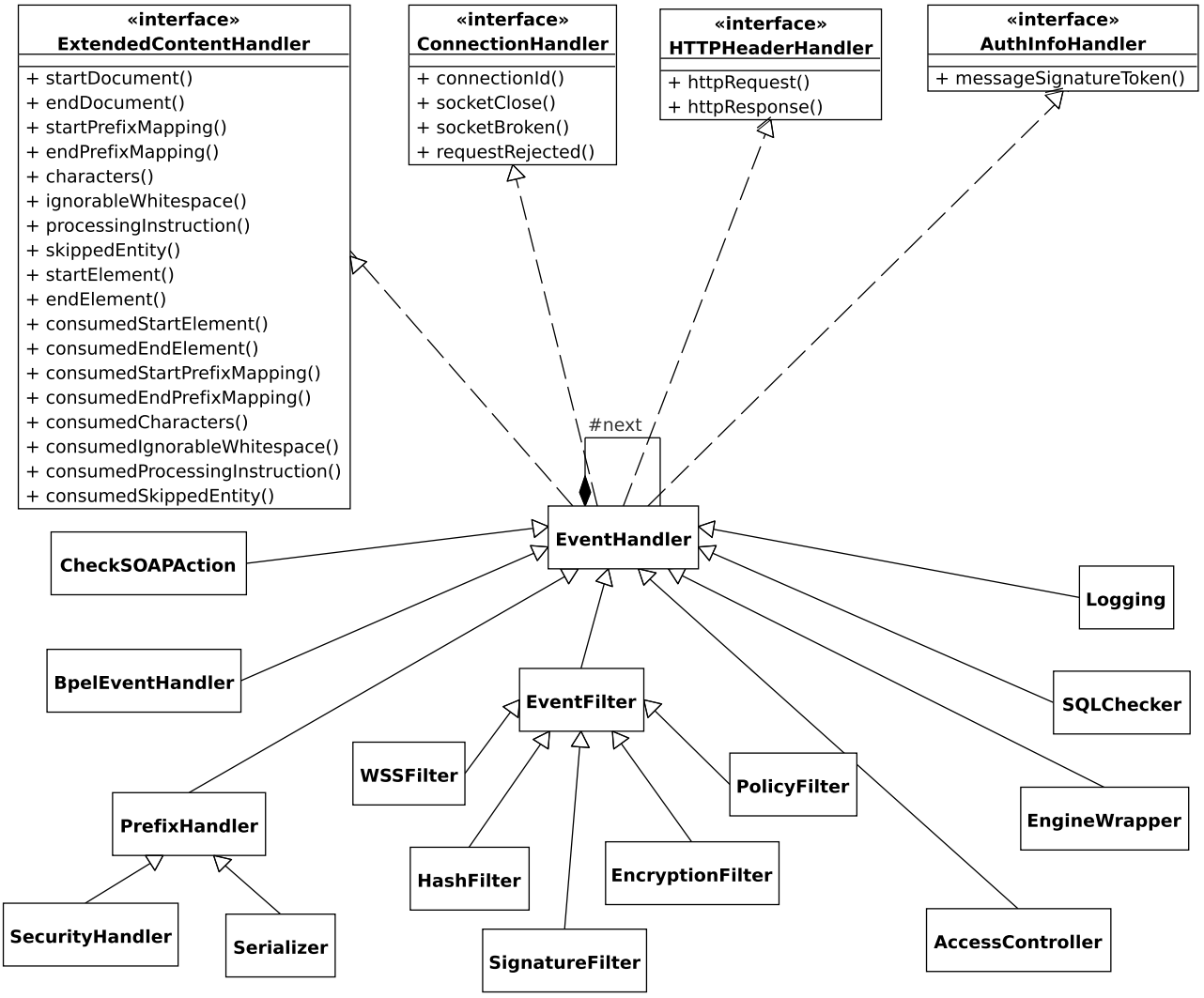


Abbildung 12.6: Klassendiagramm der Ereignisklassen von CheckWay

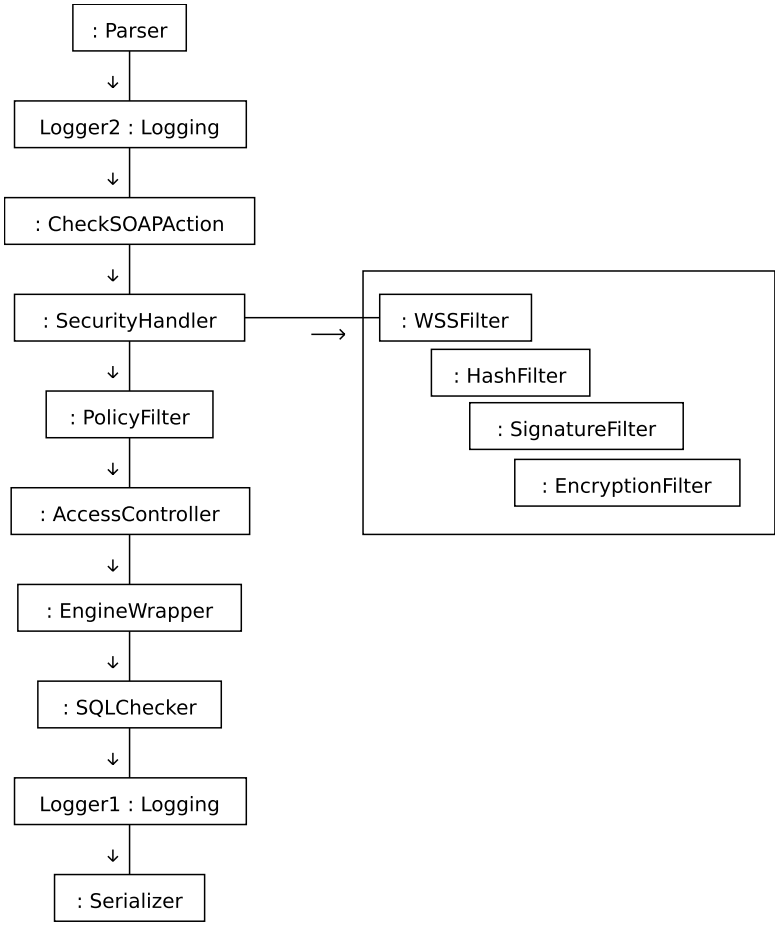


Abbildung 12.7: Ereigniskette zur Laufzeit

Komponente	**Klasse**
Basisüberprüfung	CheckSOAPAction
Zustandsverfolgung	BpelEventhandler
Security-Verarbeitung	SecurityHandler
Policy-Verarbeitung	PolicyFilter
Zugriffskontrolle	AccessController
Schema-Validator	EngineWrapper
Serialisierer	Serializer

Abbildung 12.7 zeigt eine typische Überprüfungskette, wie sie zur Laufzeit des Systems aus den oben genannten Klassen instantiiert wird.

Die Klasse Logging schreibt (beispielsweise zur Fehlersuche) alle durchlaufenden Ereignisse in eine Log-Datei und kann an beliebiger Stelle der Überprüfungskette eingefügt werden. Die Klassen WSSFilter, HashFilter, SignatureFilter und EncryptionFilter realisieren die Verarbeitung des `wsse:Security`-Headers, das Hashing von signierten Blöcken, die Überprüfung von Signaturen und das Entschlüsseln von verschlüsselten Blöcken. Instanzen dieser Klasse werden, wie in Kapitel 8 beschrieben, in die Verarbeitungskette dynamisch zur Laufzeit eingefügt.

Kapitel 13

Evaluation der Implementierung

Die im vorherigen Kapitel vorgestellte Implementierung wurde in verschiedenen Testreihen bezüglich Speicherverbrauch und Laufzeit überprüft. Ziel dieser Evaluation war zum einen der praktische Nachweis, dass die in Kapitel 6.2.1 aufgestellten komplexitätstheoretischen Forderungen an den Ressourcenverbrauch erfüllt sind. Zum anderen sollte beim Vergleich mit einem weit verbreiteten Web-Service-Framework gezeigt werden, dass die Performance von CheckWay mit handelsüblichen Implementierungen konkurrieren kann und es damit auch für praktische Einsätze brauchbar ist.

13.1 Testsystem und Messmethoden

Alle Tests wurden auf einem Athlon 64 3000+ mit 1 GByte Hauptspeicher durchgeführt. Als Betriebssystem kam Debian GNU/Linux 4.0 mit einem Kernel Version 2.6.18 zum Einsatz. Als Java-VM wurde Sun Microsystems Java Version 1.6.0 verwendet. Als Vergleichs-Framework wurde Apache Axis2 1.3 mit dem WS-Security-Modul Apache Rampart 1.3 gewählt. Da die Schema-Validierung in CheckWay nicht direkt mit der Verarbeitung in Axis2 verglichen werden kann, wurde dieser Test mit Apache Xerces 2.9.0 durchgeführt.

In allen Testreihen wurde die Laufzeit und der Speicherverbrauch für die Verarbeitung der jeweiligen Nachricht gemessen. Während die Messung der Laufzeit einfach ist, stellt die Messung des Speicherverbrauchs bei Java-basierten Anwendungen immer ein Problem dar. Zunächst einmal ist der vom Betriebssystem angeforderte Speicher (welcher leicht messbar ist) kein brauchbarer Indikator für den tatsächlichen Speicherverbrauch eines Programms. Der tatsächliche Speicherverbrauch lässt sich seit Java Version 1.5 allerdings mit der *Java Monitoring & Management Console* `JConsole` messen [35]. Sie erlaubt für jede laufende Java-VM das Auslesen einer Vielzahl von Parametern, u. a. auch des aktuellen Heap-Speicherverbrauchs.

Das zweite Problem bei Speicherverbrauchsmessungen ist die automatische Speicherbereinigung (engl. *Garbage Collection*) der Java-VM. Diese sorgt dafür, dass Objekte wieder aus dem Speicher entfernt werden, wenn keine Referenz mehr auf sie existiert. Allerdings wird diese Bereinigung aus Effizienzgründen nicht sofort durchgeführt, sondern kann beliebig verzögert werden. Das führt bei langen Laufzeiten einer Anwendung zu wenig aussagekräftigen Werten für Speicherverbräuche. Daher wurde bei den Messungen für diese Evaluation immer nur eine Nachricht an das jeweilige System (CheckWay bzw. Axis2) gesendet und danach das System neu gestartet. Im Einzelnen wurden zur Evaluation der Verarbeitung einer Nachricht vom Typ T mit n Elementen auf System S folgende Testschritte durchgeführt.

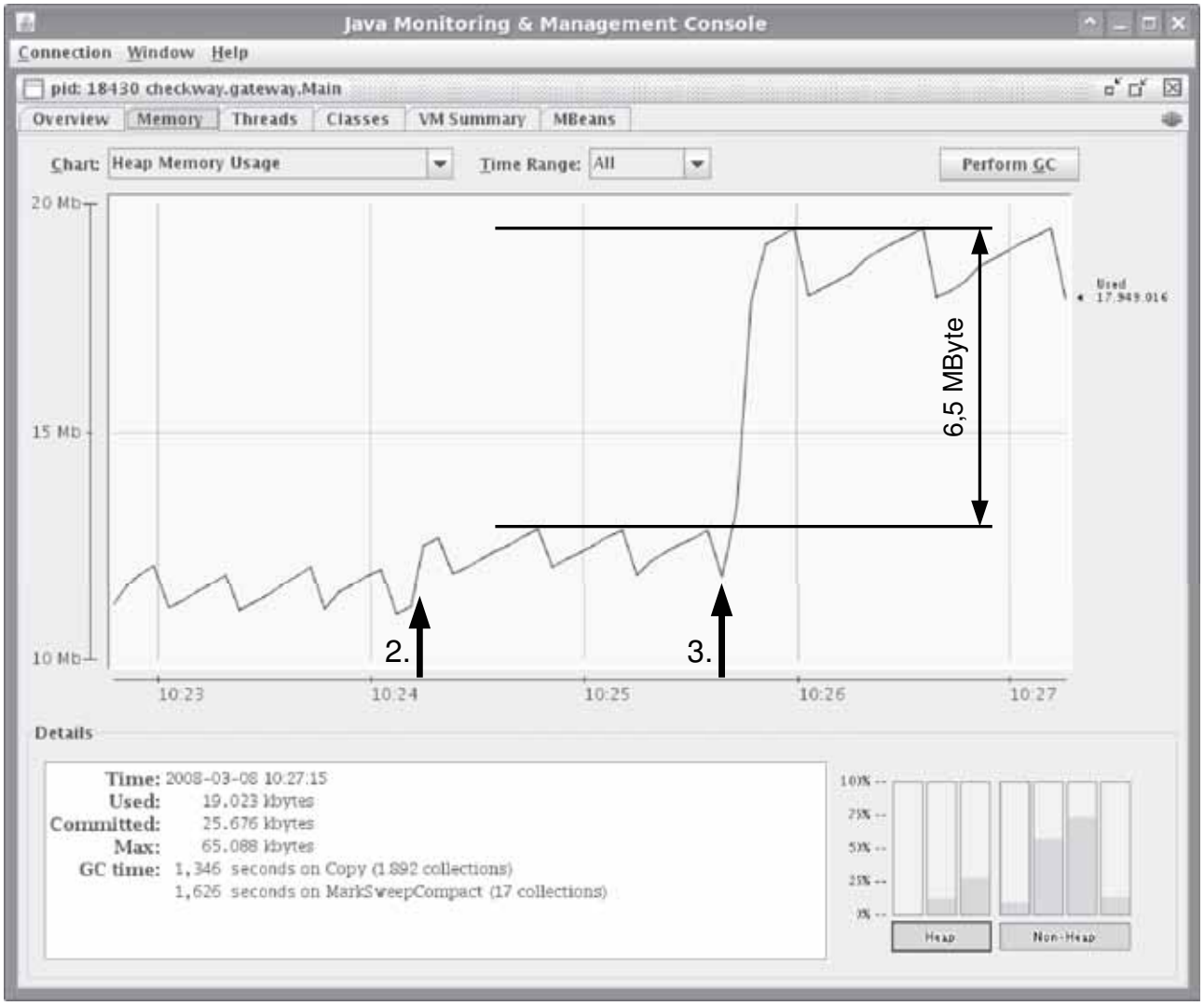


Abbildung 13.1: Anzeige des Speicherverbrauchs in `JConsole`

1. Starte S.
2. Sende Nachricht vom Typ T mit einem Element an S.
3. Sende Nachricht vom Typ T mit n Elementen an S.
4. Stoppe S.

Dabei dient der Punkt 2 der Initialisierung des Systems. Dadurch wird beispielsweise die Erzeugung global verwendeter Objekte angestoßen. Da sowohl deren Speicherverbrauch als auch die Laufzeit ihrer Erzeugung für die Evaluation der Nachrichtenverarbeitung irrelevant sind, wird vor der eigentlichen Testnachricht eine „kleine“ Nachricht an das zu testende System geschickt.

Abbildung 13.1 zeigt das Muster, das sich bei dieser Vorgehensweise für den Verlauf des Speicherverbrauchs ergibt. In jeder Phase zeigt sich das typische Sägezahnmuster, das sich aus der periodischen Speicherbereinigung ergibt. Der Speicherverbrauch, der in die Auswertung eingeht, wird aus der Differenz zwischen dem Maximum nach dem Senden der Initialisierungsnachricht und dem Maximum nach dem Senden der Testnachricht gebildet.

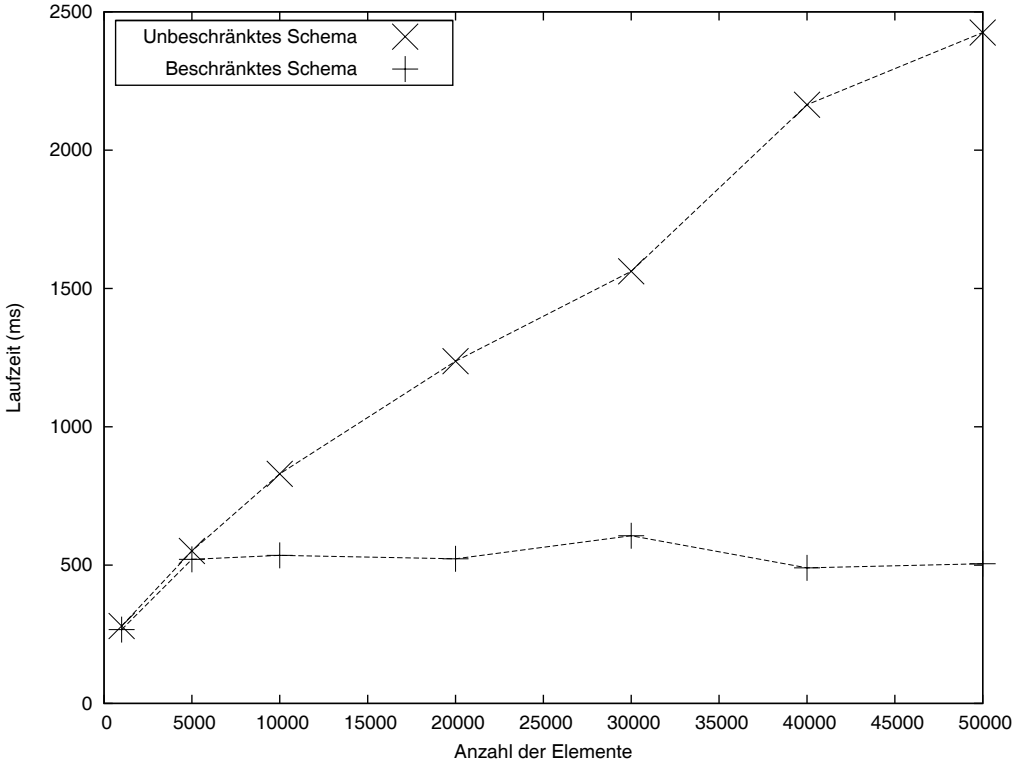


Abbildung 13.2: Laufzeit der Schema-Validierung bei CheckWay

13.2 Evaluation

Es wurde die Testszenarien ausgewählt, bei denen erfahrungsgemäß der größte Ressourcenverbrauch entsteht: Schema-Validierung und Entschlüsselung verschlüsselter Blöcke. Weiterhin wurden Nachrichten mit verschiedenen Arten von Abweichungen von den geforderten Definitionen verwendet, beispielsweise Verstoß gegen die Zugriffskontroll-Policy.

13.2.1 Schema-Validierung

Für die Evaluation der Schema-Validierung wurde eine Operation gewählt, die eine (unbeschränkte) Liste von Zeichenketten enthält. Die Testnachrichten wurden durch Variation der Anzahl der Listenelemente gebildet und enthielten keinerlei WS-Security-Elemente.

Abbildung 13.2 zeigt die Laufzeit der Schema-Validierung in CheckWay. Die obere Kurve zeigt dabei das Verhalten ohne Beschränkung des Schemas (`maxOccurs="unbounded"`), die untere Kurve mit aktivierter Beschränkung auf 5000 Elemente. Alle Nachrichten mit mehr als 5000 Elementen wurden hier also abgelehnt. In Abbildung 13.3 ist analog der Speicherverbrauch zu sehen. Dabei dienen die Werte ohne Schema-Beschränkung nur der deutlicheren Illustration des Zusammenhangs zwischen Dokumentgröße und Ressourcenverbrauch. Zur Erfüllung der Forderungen aus Kapitel 6.2.1 wird nur die zweite Kurve betrachtet.

Für die Wirkung der Schema-Beschränkung ist die Validierung gegen Schemata mit Konstrukten der Art `maxOccurs="`l`"` von großer Bedeutung. Gerade dieses ist für viele Schema-Validatoren aber problematisch. Abbildung 13.4 zeigt den Speicherverbrauch der Schema-Validierung bei CheckWay und Apache Xerces. Dabei wurden die Testnachrichten gegen

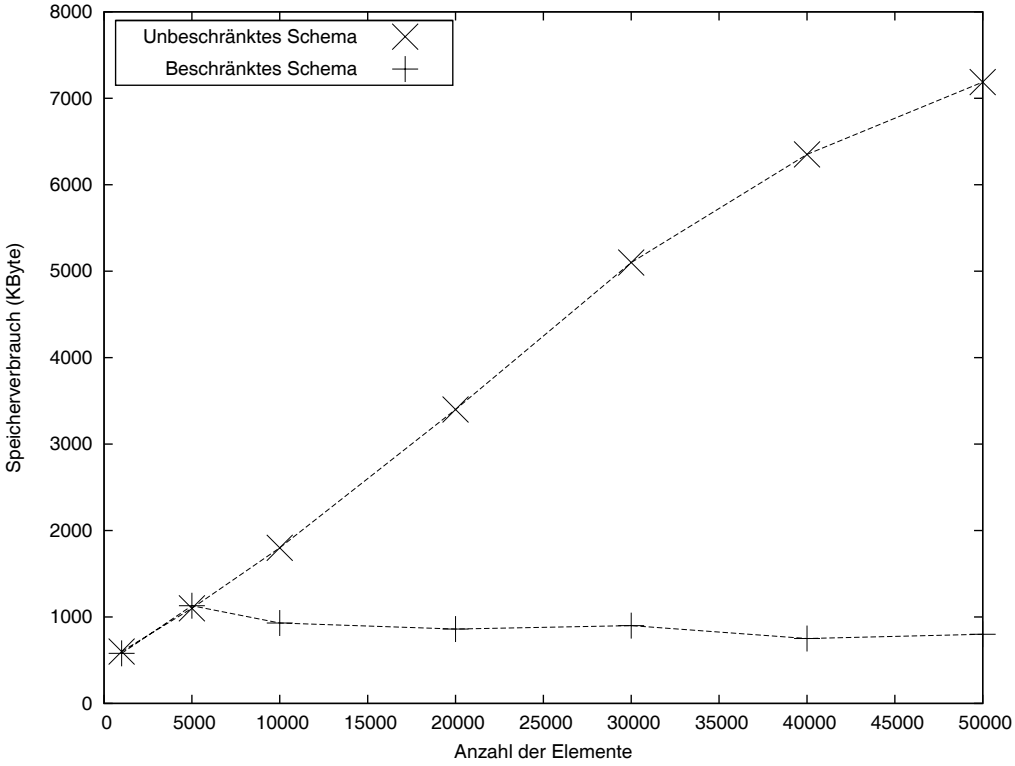


Abbildung 13.3: Speicherbedarf der Schema-Validierung bei CheckWay

Schemata mit ansteigendem l validiert.

13.2.2 Entschlüsselung

Für diese Testreihe wurden ähnliche Nachrichten wie im Abschnitt zuvor verwendet. Im Unterschied dazu wurde allerdings der Inhalt des SOAP-Bodys gemäß WS-Security verschlüsselt.

Die Abbildungen 13.5 und 13.6 zeigen Laufzeiten und Speicherverbrauch von CheckWay bei der Verarbeitung (d. h. primär Entschlüsselung und Schema-Validierung) dieser Nachrichten. Wiederum wurde danach unterschieden, ob die Schema-Beschränkung aktiviert ist oder nicht.

Zum Vergleich der Verschlüsselung-Verarbeitung zwischen CheckWay und Axis2, wurde bei beiden Systemen für jede Nachricht der Ressourcenverbrauch für die unverschlüsselte Nachricht und dieselbe Nachricht mit Verschlüsselung gemessen. Die Differenz zwischen beiden Werte ergibt dann den zusätzlichen Ressourcenverbrauch für die Entschlüsselung der Nachricht. Das Ergebnis dieses Tests ist in den Abbildungen 13.7 und 13.8 zu sehen.

13.2.3 Ungültige Nachrichten

Als Beispiel für ungültige Nachrichten (die nicht die Schema-Beschränkung verletzen) wurden Nachrichten mit einer Verletzung der Zugriffskontroll-Policy gewählt. Die Nachrichten waren also von einem nicht zugelassenen Tokeninhaber signiert. Abbildung 13.9 zeigt den Speicherverbrauch und die Laufzeit, die bei der Verarbeitung (und Ablehnung) einer solchen Nachricht auftreten. Ein ähnliches Bild ergibt sich für alle Nachrichten, bei denen Abweichung von den

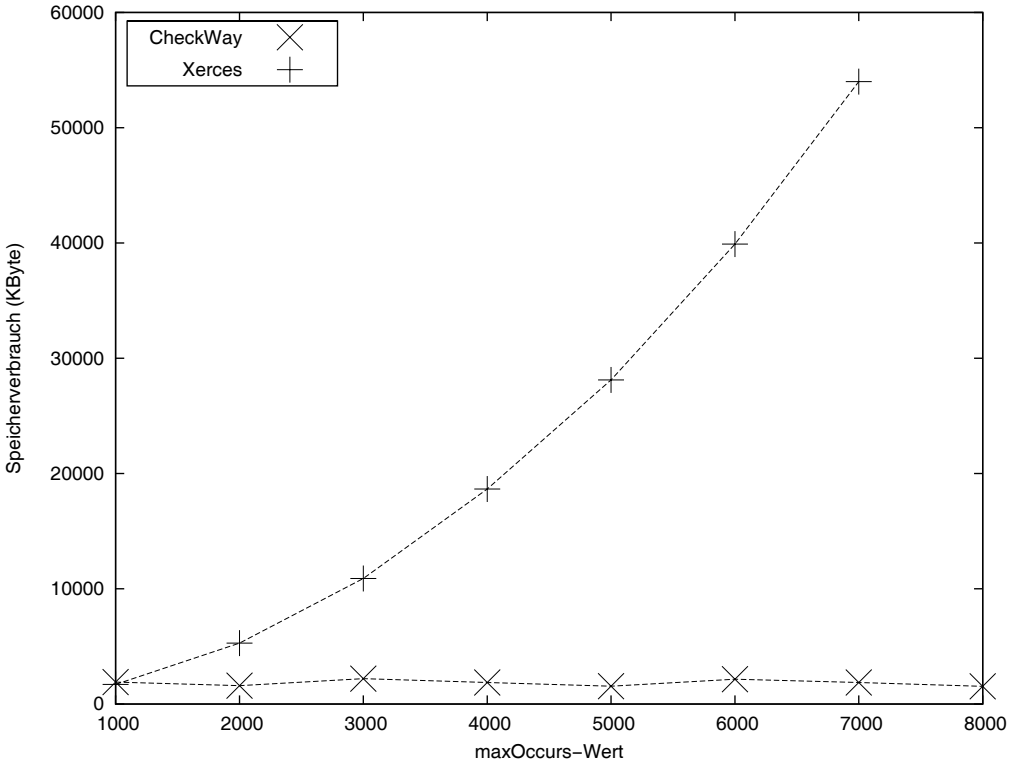


Abbildung 13.4: Speicherbedarf der Schema-Validierung bei CheckWay und Xerces

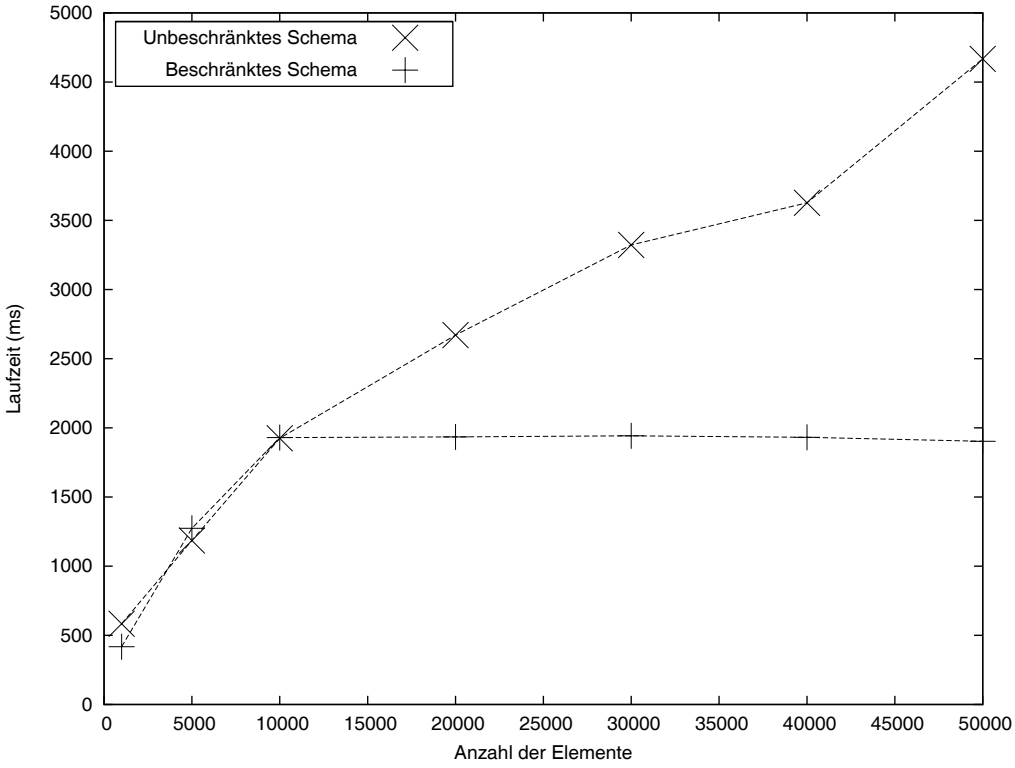


Abbildung 13.5: Laufzeit der Verarbeitung von verschlüsselten Blöcken bei CheckWay

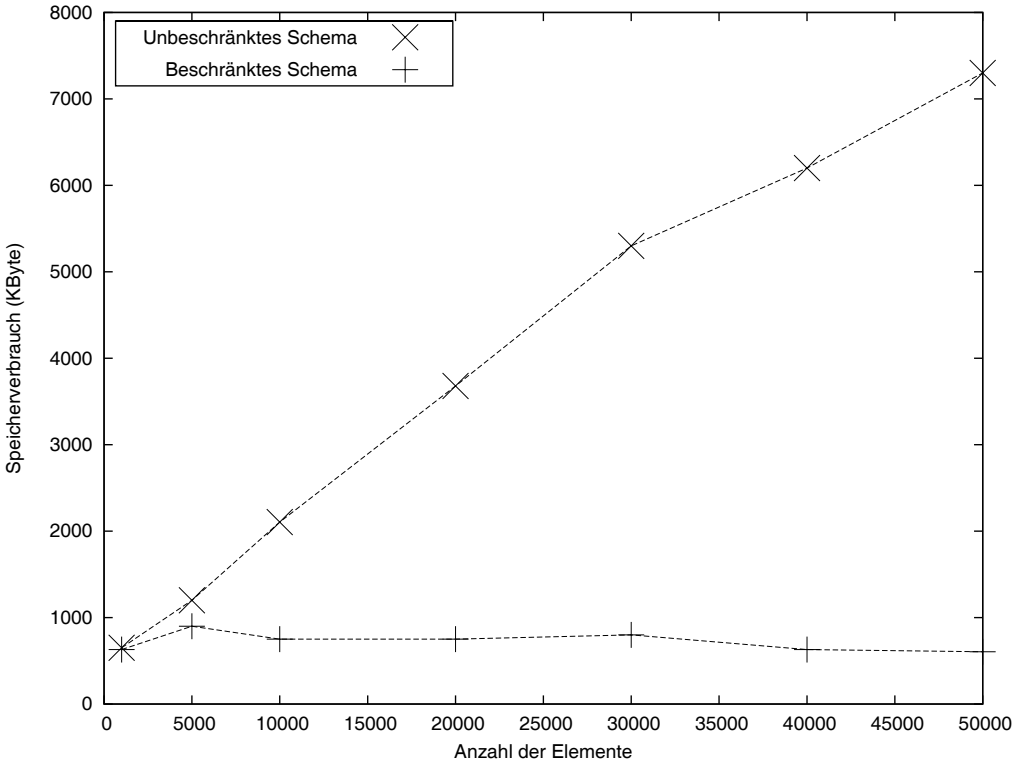


Abbildung 13.6: Speicherbedarf der Verarbeitung von verschlüsselten Blöcken bei CheckWay

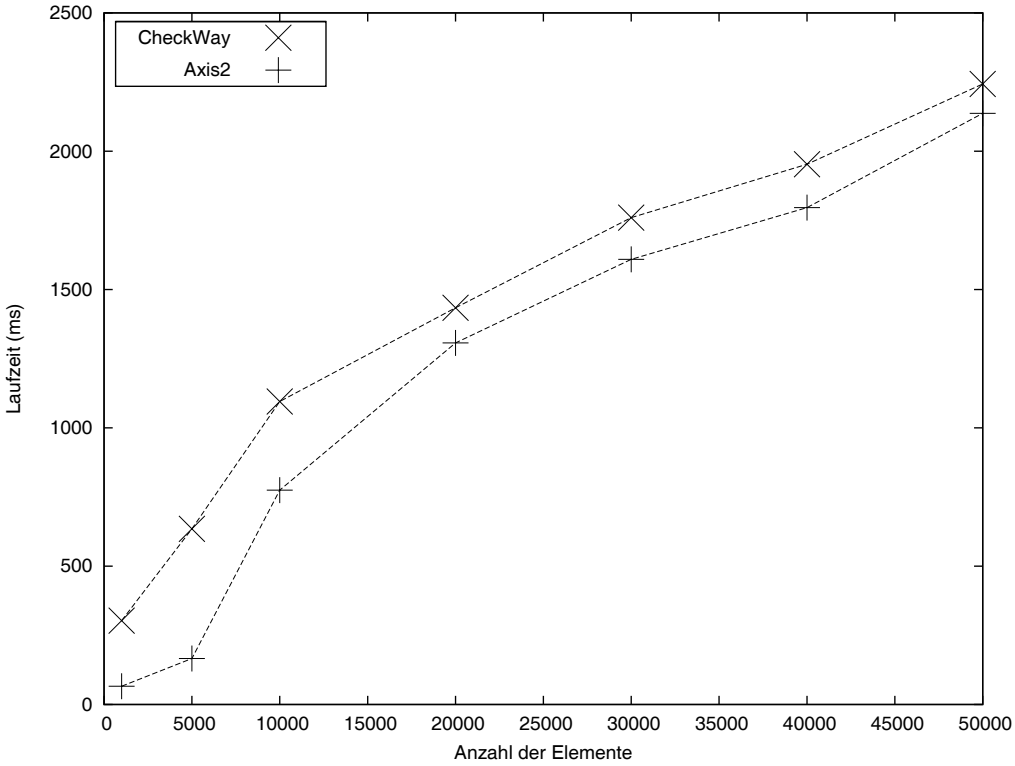


Abbildung 13.7: Vergleich der Laufzeit der Verarbeitung von verschlüsselten Blöcken

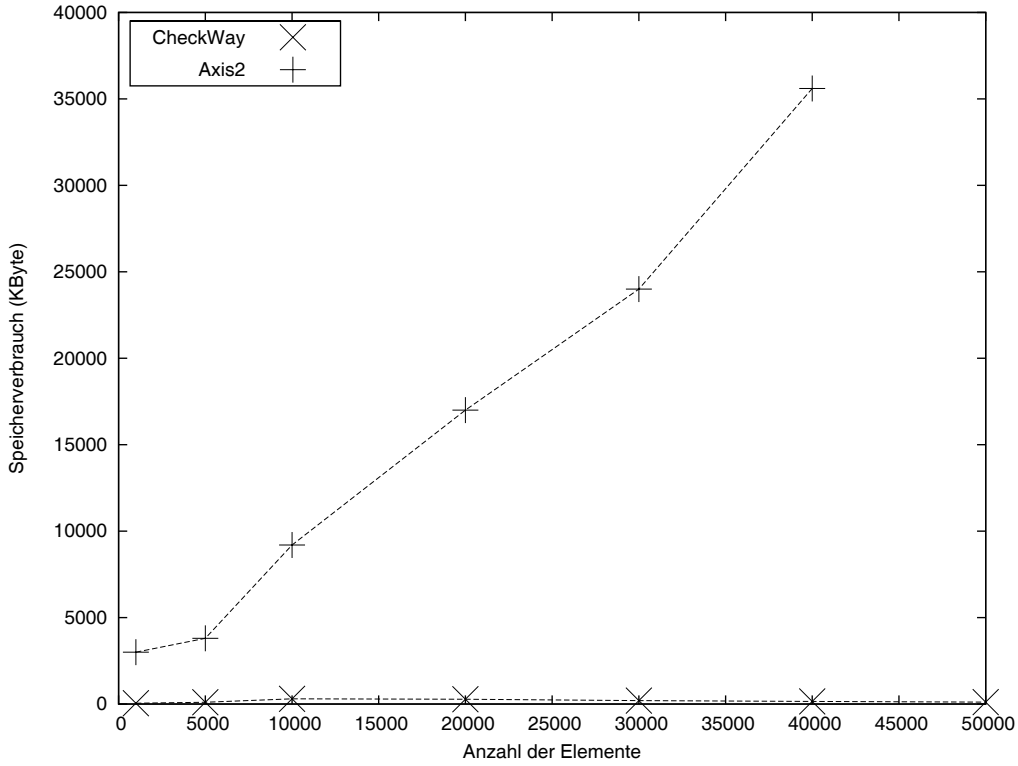


Abbildung 13.8: Vergleich des Speicherbedarfs der Verarbeitung von verschlüsselten Blöcken

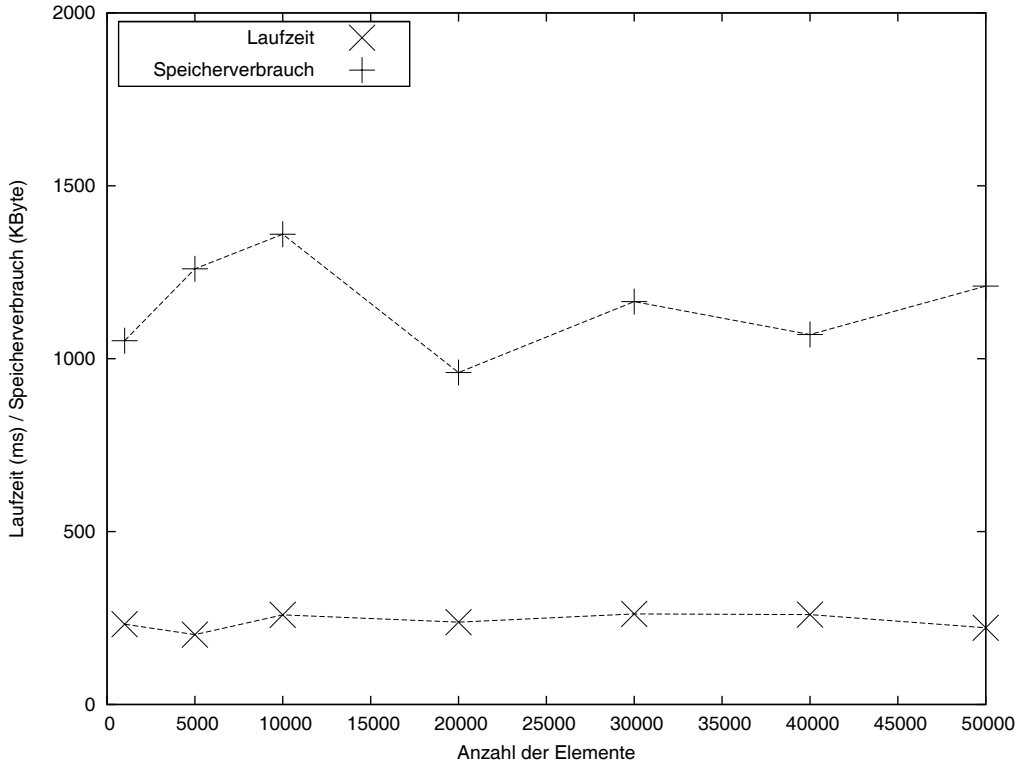


Abbildung 13.9: Verletzung der Zugriffskontroll-Policy

Definitionen im SOAP-Header festgestellt werden kann. Dazu gehören u. a. ungültige Zeitstempel, nicht Policy-konforme Sicherheits-Token und nicht erfüllte Sicherheits-Policy-Forderungen für den Header (beispielsweise signierte Signaturen).

Andere Verletzungen von Forderungen, wie ungültige Signaturen oder Verletzung von Schema-Typen im SOAP-Body, führen zu höheren Ressourcenverbräuchen, diese sind allerdings in jedem Fall durch die Werte, die sich die Schema-Beschränkung ergeben, limitiert.

13.3 Bewertung

Die oben präsentierten Evaluationsergebnisse illustrieren, dass die theoretisch gezeigten Forderungen an Speicherverbrauch und Laufzeiten erfüllt sind. Zum einen sind beide Ressourcenverbräuche in jedem Fall beschränkt, zum anderen ist für gültige Nachrichten ein linearer Zusammenhang zwischen Dokumentgröße und Ressourcenverbrauch vorhanden.

Weiterhin zeigen die Tests, dass CheckWay mit handelsüblichen Systemen mithalten kann bzw. diese sogar übertrifft. Insbesondere der in den Abbildungen 13.4 und 13.8 aufgetragene Speicherverbrauchsvergleich zeigt die Vorteile des hier gewählten Verarbeitungsmodells.

Kapitel 14

Abschluss

14.1 Zusammenfassung

In dieser Arbeit wurde untersucht, wie die Sicherheit von Web Services durch eine erweiterte und effiziente Validierung von eingehenden Web-Service-Nachrichten erhöht werden kann.

Erstellung eines Klassifizierungssystems für DoS-Angriffe. Es wurde ein multidimensionales Klassifizierungssystem für Denial-of-Service-Angriffe auf Netzwerkdienste entwickelt, das verschiedene Aspekte eines Angriffs berücksichtigt. Dieses System lässt sich mit kleineren Einschränkungen auch auf andere Arten von Angriffen, d. h. Angriffe gegen Vertraulichkeit und Integrität, übertragen.

Analyse von Angriffen gegen Web Services. Hier wurden eine Vielzahl von Angriffen gegen Web Service präsentiert. Diese Angriffsmöglichkeiten wurden durch theoretische Untersuchungen von Schwachstellen und durch praktische Experimente mit weitverbreiteten Web-Service-Frameworks gefunden. Diese Angriffe wurden analysiert und unter Benutzung des vorher entwickelten Klassifizierungssystems kategorisiert.

Modellierung der XML-Verarbeitung. Als Grundlage für die Verarbeitung von Web-Service-Nachrichten wurde ein einfaches Modell für die Darstellung und Verarbeitung von XML-Dokumenten erstellt. In diesem Modell wurden die beiden wichtigsten Methoden der XML-Verarbeitung, die baumbasierte und die ereignisbasierte Verarbeitung, analysiert und miteinander verglichen. Als Ergebnis wurden Aussagen über den (theoretischen) Speicherbedarf und die Laufzeit bei der Verwendung dieser Verarbeitungtechnologien getroffen.

Erweiterte Validierung als Schutzmaßnahme für Web Services. Web Services sind durch eine Vielzahl von formalen Metadaten beschrieben. Es wurde gezeigt, wie die Validierung gegen diese Beschreibungen als Schutzmaßnahme für Web-Service-Systeme verwendet werden kann. Es wurde weiterhin gezeigt, dass diese Methoden durch weitere Maßnahmen wie der Entschlüsselung von verschlüsselten Nachrichtenteilen oder der Einschränkung der protokolldefinierenden Metadaten ergänzt und verbessert werden können.

Ereignisbasierte SOAP-Verarbeitung. Die Art der Verarbeitung von SOAP-Nachrichten bei der Durchführung dieser Verfahren hat großen Einfluss auf die Effektivität der Maßnahmen. Auf Grundlage des Modells für die XML-Verarbeitung wurde gezeigt, dass eine

baumbasierte Realisierung der Schutzmaßnahmen ungeeignet ist. Ein derartiges Schutzsystem wäre selbst durch viele der Angriffe verwundbar, die es erkennen und verhindern soll.

Die Lösung ist eine vollständige ereignisbasierte Durchführung der oben genannten Schutzmaßnahmen. Eine solche ereignisbasierte Verarbeitung von SOAP-Nachrichten, insbesondere solcher mit WS-Security-Sicherheitselementen, lässt sich mit den Standardmechanismen allerdings nicht durchführen. Daher mussten neue Algorithmen und Verfahren zur ereignisbasierten SOAP-Verarbeitung und -Überprüfung entwickelt werden. Im Einzelnen wurden folgende Mechanismen realisiert:

- Schema-Validierung
- WS-Security-Verarbeitung
- Sicherheits-Policy-Überprüfung
- Zugriffskontrolle
- Überwachung der Nachrichtenreihenfolge

Implementierung eines Schutzsystems. Diese ereignisbasierten Verfahren zur Realisierung der Schutzkonzepte wurden prototypisch aber vollständig als Firewall-ähnliches Schutzsystem implementiert. Die Evaluation der Implementierung hat gezeigt, dass die Laufzeit und der Speicherverbrauch der Lösung nicht nur den theoretischen Forderungen genügt, sondern sie auch für praktische Einsätze geeignet machen.

Diese Arbeit präsentiert also die vollständige Entwicklung eines Schutzsystems für Web Services, angefangen bei der Analyse der Angriffe über die Entwicklung von abstrakten Schutzkonzepten und die konkrete Anwendung dieser Konzepte auf die Web-Service-Nachrichtenverarbeitung bis hin zur Erstellung einer lauffähigen Implementierung der entstandenen Lösung.

14.2 Offene Fragen und Ausblick

Im Verlaufe dieser Arbeit konnten nicht alle Probleme im Kontext von Angriffen auf Web Services behandelt werden.

So sind die dargestellten Lösungen ungeeignet für viele Angriffe vom Typ „Abweichung von der Protokoll-Semantik“ wie beispielsweise *Metadata Spoofing* oder *WS-Addressing-Spoofing* [92]. Ein weiteres grundlegendes Problem stellen Angriffe dar, die eine große Anzahl von Nachrichten verwenden, die sogenannten *Flut-Angriffe* (engl. *Flooding Attacks*) [93]. Beide Arten von Angriffen erfordern weitergehende Forschungsaktivitäten.

Ein weitere offene Frage ist die Auswirkungen der entwickelten Schutzmaßnahmen auf Angriffe gegen Integrität und Vertraulichkeit. Während für Ressourcenverbrauchsangriffe gezeigt wurde, dass diese durch die Schutzmaßnahmen vollständig verhindert werden können, wird das für einige andere Angriffe zwar vermutet, ist aber (noch) nicht beweisbar. So wird angenommen, dass *XML-Rewriting*-Angriffe gegen SOAP-Signaturen durch folgende Maßnahmen vollständig erkannt werden können:

- strenges Nachrichtenschema
- Schema-Validierung

- korrekte – beispielsweise im Sinne von [16] – Sicherheits-Policy
- Policy-Überprüfung

Diese Vermutung soll in zukünftigen Arbeit formal bewiesen werden.

Wie bereits im Verlauf der Arbeit angedeutet, können die ereignisbasierten Validierungmaßnahmen, die hier im Kontext eines Web-Service-Schutzsystems beschrieben wurden, auch als Teil eines Web-Service-Frameworks verwendet werden. Eine interessante Weiterführung dieser Arbeit wäre daher auch die Erstellung eines komplett ereignisbasierten Web-Service-Frameworks inklusive DoS-Schutz und WS-Security- und Sicherheits-Policy-Unterstützung.

Teil IV

Anhänge

Anhang A

Verwendete Namensraum-Präfixe

A.1 Standardisierte Namensräume

Präfix	Namensraum	Standard
bpel	http://schemas.xmlsoap.org/ws/2003/03/business-process/	[3]
ds	http://www.w3.org/2000/09/xmldsig#	[10]
env	http://schemas.xmlsoap.org/soap/envelope/	[21]
env12	http://www.w3.org/2003/05/soap-envelope	[72]
soap	http://schemas.xmlsoap.org/wsdl/soap12/	[56]
sp	http://schemas.xmlsoap.org/ws/2005/07/securitypolicy	[95]
wsa	http://www.w3.org/2005/08/addressing	[73]
wsdl	http://schemas.xmlsoap.org/wsdl/	[34]
wsse	http://docs.oasis-open.org/wss/2004/01/ oasis-200401-wss-wssecurity-secext-1.0.xsd	[124]
wsp	http://schemas.xmlsoap.org/ws/2004/09/policy	[49]
wsu	http://docs.oasis-open.org/wss/2004/01/ oasis-200401-wss-wssecurity-utility-1.0.xsd	[124]
xenc	http://www.w3.org/2001/04/xmlenc#	[86]
xsd	http://www.w3.org/2001/XMLSchema	[150, 17]

A.2 Namensräume aus Beispielen

Präfix	Namensraum
ns1	http://www.example.org/ns1

Anhang B

XML-Dokumente aus Beispielen

B.1 Zugriffskontrolle

```
<Policy PolicyId="ExamplePolicy"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:
                            rule-combining-algorithm:deny-overrides">
 <Target/>
 <Rule RuleId="urn:oasis:names:tc:xacml:2.0:example:SimpleRule1"
       Effect="Permit">
  <Target>
   <Subjects>
    <Subject>
     <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:
                            string-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
       CN=Gruschka,OU=Unknown,O=CAU,L=Kiel,ST=Unknown,C=DE
      </AttributeValue>
      <SubjectAttributeDesignator
       AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
       DataType="http://www.w3.org/2001/XMLSchema#string"/>
     </SubjectMatch>
    </Subject>
   </Subjects>
   <Resources>
    <Resource>
     <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:
                             anyURI-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI">
       http://server:8080/WebServices/MyService
      </AttributeValue>
      <ResourceAttributeDesignator
         AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
         DataType="http://www.w3.org/2001/XMLSchema#anyURI"/>
     </ResourceMatch>
    </Resource>
```

```
   </Resources>
   <Actions>
    <Action>
     <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:
                           string-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
       urn:getLength
      </AttributeValue>
      <ActionAttributeDesignator
         DataType="http://www.w3.org/2001/XMLSchema#string"
       AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
     </ActionMatch>
    </Action>
   </Actions>
  </Target>
 </Rule>
</Policy>
```

Anhang C

Glossar

Anfragen-Durchsatz: Anzahl der Dienstanfragen, die in einem bestimmten Zeitraum beantwortet werden, beispielsweise in $\frac{1}{s}$.

***Cookie Poisoning*:** *Cookie Poisoning* bezeichnet einen Angriff, bei dem in einem HTTP-Cookie [100] gespeicherte Informationen vom Client so verändert werden, dass sie den Server zu ungültigen Operationen veranlassen. Ein Beispiel ist ein Cookie einer Bestellapplikation, der explizit den Preis der Waren enthält. Der Client kann dann den Cookie mit einem geringerem Preis zurückschicken. Wird das serverseitig nicht nochmals überprüft, so war das *Cookie Poisoning* erfolgreich.

Base64: Die Base64-Kodierung erlaubt die Kodierung von 8-Bit-Binärdaten in Zeichen aus der Menge `[A-Z,a-z,+,/]`. Das ermöglicht die Darstellung von Binärdaten in Systemen, die nur die Darstellung von 7-Bit-ASCII-Zeichen erlauben, wie beispielsweise MIME [103]. Bei der Base64-Kodierung werden jeweils 3 Byte (= 24 Bit) der Originaldaten in 4 mal 6 Bit zerlegt und diese 6-Bit-Zahlen durch die o. g. Zeichen dargestellt.

Parser, Parsen: Der Vorgang des Parsens bezeichnet im Allgemeinen das Zerlegen und Umwandeln einer Eingabezeichenfolge in ein abstrakteres Format, auf dem die Weiterverarbeitung durchgeführt werden kann. Dabei werden in einem ersten Schritt von einem sogenannten *Scanner* die Eingabedaten in „Wörter“ der Sprache zerlegt. Danach analysiert der eigentliche Parser diese Wörter gemäß einer Grammatik und gibt das Ergebnis weiter.

Bei einem XML-Parser werden beispielsweise im ersten Schritt die spitzen Klammern ausgewertet und die XML-Kennungen (engl. *XML Tags*) extrahiert. Im zweiten Schritt wird dann die korrekte Schachtelung der Kennungen, die Namensraumdeklarationen usw. überprüft. Die Ausgabe des Parsers ist dann typischerweise ein DOM-Baum oder SAX-Ereignisse.

Reaktions-Latenz: Die Verzögerung zwischen Ende der Dienstanfrage und dem Beginn der Antwort auf diese Anfrage.

Wiederherstellungs-Geschwindigkeit: Zeitdauer zwischen einem erfolgreichen Denial-of-Service-Angriff und dem Zeitpunkt, zu dem der Dienst wieder vollständig verfügbar ist.

ZIP-Bombe: Eine *ZIP-Bombe* oder auch *Archivbombe* ist eine relativ kleine gepackte Datei (beispielsweise ein komprimiertes Archiv oder Bild), die aber eine extrem große Roh-Datei

enthält. Die gepackte Datei verbraucht beim Opfer beim Entpacken oder Betrachten des Inhalts große Menge an Ressourcen und kann so als DoS-Angriff verwendet werden.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Nayef Abu-Ghazaleh und Michael J. Lewis. Differential Deserialization for Optimized SOAP Performance. In: *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, S. 21, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Gustavo Alonso, Fabio Casati, Harumi Konu und Vijay Machiraju. *Web Services*. Springer, 2004.

[3] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic und Sanjiva Weerawarana. Business Process Execution Language for Web Services Version 1.1. May 2003.

[4] Chris Anley. Advanced SQL injection in SQL server applications. Technical report, NGSSoftware Insight Security Research, 2002.

[5] Claudio Agostino Ardagna, Ernesto Damiani, Sabrina De Capitani di Vimercati und Pierangela Samarati. A Web Service Architecture for Enforcing Access Control Policies. *Proc. of the First International Workshop on Views On Designing Complex Architectures (VODCA 2004)*, September 2004.

[6] Rajith Attapattu. Introduction to Apache Axis2. *Red Hat Magazine*, 21, 2006.

[7] Tuomas Aura und Pekka Nikander. Stateless Connections. In: *Proceedings of Information and Communication Security, 1st International Conference (ICICS '97)*, Band 1334, S. 87–97, Bejing, China, November 1997. Springer.

[8] Tuomas Aura, Pekka Nikander und Jussipekka Leiwo. DOS-Resistant Authentication with Client Puzzles. *Lecture Notes in Computer Science*, 2133:170+, 2001.

[9] Keith Ballinger, David Ehnebuske, Christopher Ferris, Martin Gudgin, Canyang Kevin Liu, Mark Nottingham und Prasad Yendluri. Basic Profile Version 1.1. *WS-I Organisation*, 2004.

[10] Mark Bartel, John Boyer, Barb Fox, Brian LaMacchia und Ed Simon. XML-Signature Syntax and Processing. *W3C Recommendation*, 2002.

[11] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, Marcus Fontoura und Vanja Josifovski. Streaming XPath processing with forward and backward axes. In: *Proceedings of the 19th International Conference on Data Engineering*, 2003.

[12] Tim Bass, Alfredo Freyre, David Gruber und Glenn Watt. E-Mail Bombs and Countermeasures: Cyber Attacks on Availability and Brand Integrity. *IEEE Network*, 12:10–17, 1998.

[13] Tim Berners-Lee, Roy Fielding und Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. *IETF request for comments*, RFC 3986, 2005.

[14] Karthikeyan Bhargavan, Cedric Fournet und Andrew D. Gordon. Verifying Policy-based Security for Web Services. In: *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, S. 268–277, New York, NY, USA, 2004. ACM Press.

[15] Karthikeyan Bhargavan, Cedric Fournet und Andrew D. Gordon. A Semantics for Web Services Authentication. *Theoretical Computer Science*, 340(1):102–153, 2005.

[16] Karthikeyan Bhargavan, Cedric Fournet, Andrew D. Gordon und Greg O'Shea. An Advisor for Web Services Security Policies. In: *SWS '05: Proceedings of the 2005 workshop on Secure Web Services*, S. 1–9, New York, NY, USA, 2005. ACM Press.

[17] Paul V. Biron und Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. *W3C Recommendation*, 2004.

[18] Matt Bishop. *Computer Security*. Addison Wesley, 2003.

[19] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris und David Orchard. Web Services Architecture. *W3C Recommendation*, 2004.

[20] Christian Borkowski. Automatische Schemagenerierung aus Web Service Beschreibungen in WSDL 2.0. *Bachelor thesis*, 2006.

[21] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte und Dave Winer. Simple Object Access Protocol (SOAP) 1.1. *W3C Note*, 2000.

[22] John Boyer, Donald E. Eastlake und Joseph Reagle. Exclusive XML Canonicalization Version 1.0. *W3C Recommendation*, 2002.

[23] John Boyer, Merlin Hughes und Joseph Reagle. XML-Signature XPath Filter 2.0. *W3C Proposed Recommendation*, 2002.

[24] J. Boyle, R. Cohen, S. Herzog, R. Rajan und A. Sastry. The COPS (Common Open Policy Service) Protocol. *IETF request for comments*, RFC 2748, 2000.

[25] Tim Bray, Dave Hollander, Andrew Layman und Richard Tobin. Namespaces in XML 1.1. *W3C Recommendation*, 2004.

[26] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Male und François Yergeau. Extensible Markup Language (XML) 1.0 (Fourth Edition). *W3C Recommendation*, 2006.

[27] Gerald Brose. A Gateway to Web Services Security - Securing SOAP with Proxies. In: *Web Services - ICWS-Europe 2003*, 2003.

[28] Aaron Brown und David A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. In: *Proceedings of 2000 USENIX Annual Technical Conference*, 2000.

[29] David Brownell. *SAX2*. O'Reilly, 2002.

[30] Scott Cantor, John Kemp, Rob Philpott und Eve Maler. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. *OASIS Standard*, 2005.

[31] D. Brent Chapman und Elizabeth D. Zwicky. *Building Internet Firewalls*. O'Reilly, 1995.

[32] Hang Chau. Network Security – Defense Against DoS/DDoS Attacks. `http://www.securitydocs.com/library/2576`, 2004.

[33] Yan Chen, Adam Bargteil, David Bindel, Randy H. Katz und John Kubiatowicz. Quantifying Network Denial of Service: A Location Service Case Study. In: *ICICS '01: Proceedings of the Third International Conference on Information and Communications Security*, S. 340–351, London, UK, 2001. Springer-Verlag.

[34] Erik Christensen, Francisco Curbera, Greg Meredith und Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. *W3C Note*, 2001.

[35] Mandy Chung. Using JConsole to Monitor Applications. *SUN Developer Network*, 2004.

[36] James Clark und Steve DeRose. XML Path Language (XPath) Version 1.0. *W3C Recommendation*, 1999.

[37] Frank Cohen. Discover SOAP encoding's impact on Web service performance. *IBM developerWorks*, 2003.

[38] John Cowan und Richard Tobin. XML Information Set (Second Edition). *W3C Recommendation*, 2004.

[39] Francisco Curbera und Rania Khalaf. Implementing BPEL4WS: The Architecture of a BPEL4WS Implementation. *Concurrency and Computation: Practice and Experience*, 18:1219–1228, 2005.

[40] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi und P. Samarati. Securing SOAP E-Services. *International Journal of Information Security*, 2002.

[41] Doug Davis, Anish Karmarkar, Gilbert Pilz und Ümit Yalcinalp. Web Services Reliable-Messaging Policy Assertion(WS-RM Policy). *Oasis Committee Draft 03*, 2006.

[42] Drew Dean und Adam Stubblefield. Using Client Puzzles to Protect TLS. In: *Proceedings of the 7th Network and Distributed System Security Symposium*, 2001.

[43] Dorothy E. Denning. An Intrusion-Detection Model. *sp*, 00:118, 1986.

[44] T. Dierks und E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. *IETF request for comments*, RFC 4346, 2006.

[45] Danny Dolev und Andrew Chi-Chih Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[46] Naganand Doraswamy und Dan Harkins. *IPSec, The New Security Standard for the Internet, Intranets, and Virtual Private Networks*. Prentice Hall, 2nd edition, 2003.

[47] Cynthia Dwork und Moni Naor. Pricing via Processing or Combatting Junk Mail. In: *CRYPTO '92: Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, S. 139–147, London, UK, 1993. Springer-Verlag.

[48] Chris Sharp (Editor). Web Services Policy 1.2 - Attachment (WS-PolicyAttachment). *W3C Member Submission*, 2006.

[49] Jeffrey Schlimmer (Editor). Web Services Policy 1.2 - Framework (WS-Policy). *W3C Member Submission*, 2006.

[50] Jon Postel (Editor). Internet Protcol. *IETF request for comments*, RFC 791, 1981.

[51] Jon Postel (Editor). Transmission Control Protocol. *IETF request for comments*, RFC 793, 1981.

[52] Thomas Erl. *Service-Oriented Architecture – Concepts, Technology, and Design*. Prentice Hall, 2005.

[53] ICANN Factsheet. Root server attack on 6 February 2007. Technical report, Internet Corporation for Assigned Names and Numbers, 2007.

[54] David C. Fallside und Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. *W3C Recommendation*, 2004.

[55] Ruchith Fernando. Secure Web Services with Apache Rampart. Technical report, WSO2 Oxygen Tank, 2006.

[56] Christopher Ferris und Jonathan Marsh (Editors). WSDL 1.1 Binding Extension for SOAP 1.2. 2006.

[57] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach und T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. *IETF request for comments*, RFC 2616, 1999.

[58] N. Freed und N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. *IETF request for comments*, RFC 2045, 1996.

[59] Alan O. Freier, Philip Karlton und Paul C. Kocher. The SSL Protocol Version 3.0 Internet Draft. `http://wp.netscape.com/eng/ssl3/`, 1996.

[60] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, 1995.

[61] Nalneesh Gaur. Assessing the Security of Your Web Applications. *Linux Journal*, (72), 2000.

[62] Virgil D. Gligor. A Note on the Denial-of-Service Problem. In: *Proceedings of the 1983 Symposium on Security and Privacy*, S. 139–149, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press.

[63] Virgil D. Gligor. On Denial-of-Service in Computer Networks. In: *Proceedings of the Second International Conference on Data Engineering*, S. 608–617, Washington, DC, USA, 1986. IEEE Computer Society.

[64] Mark Grand. *Patterns in Java, Volume 1.* Wiley Computer Books, 1998.

[65] Paul Grosso, Eve Male, Jonathan Marsh und Norman Walsh. XPointer Framework. *W3C Recommendation*, 2003.

[66] Nils Gruschka, Ralph Herkenhöner und Norbert Luttenberger. WS-SecurityPolicy Decision and Enforcement for Web Service Firewalls. In: *Proceedings of the IEEE/IST Workshop on Monitoring, Attack Detection and Mitigation*, S. 19?25, 2006.

[67] Nils Gruschka, Ralph Herkenhöner und Norbert Luttenberger. Access Control Enforcement for Web Services by Event-Based Security Token Processing. In: T. Braun, G. Carle und B. Stiller, editors, *15. ITG/Gi Fachtagung Kommunikation in Verteilten Systemen (KiVS 2007)*, S. 371–382, 2007.

[68] Nils Gruschka, Meiko Jensen und Torben Dziuk. Event-Based Application of WS-SecurityPolicy on SOAP Messages. In: *Proceedings of the 2007 ACM Workshop on Secure Web Services (SWS'07)*, S. 1–8, Fairfax, Virginia, USA, November 2007. Association for Computing Machinery.

[69] Nils Gruschka, Meiko Jensen und Norbert Luttenberger. A Stateful Web Service Firewall for BPEL. In: *Proceedings of the 2007 IEEE International Conference on Web Services (ICWS 2007)*, S. 142–149, 2007.

[70] Nils Gruschka und Norbert Luttenberger. Protecting Web Services from DoS Attacks by SOAP Message Validation. In: *Proceedings of the IFIP TC-11 21. International Information Security Conference (SEC 2006)*, S. 171–182, 2006.

[71] Nils Gruschka, Norbert Luttenberger und Ralph Herkenhöner. Event-based SOAP Message Validation for WS-SecurityPolicy-Enriched Web Services. In: *Proceedings of the 2006 International Conference on Semantic Web & Web Services*, S. 80–86, 2006.

[72] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau und Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. *W3C Recommendation*, 2003.

[73] Martin Gudgin, Marc Hadley und Tony Rogers. Web Services Addressing 1.0 - SOAP Binding. *W3C Recommendation*, May 2006.

[74] Hal Lockhart et al. Web Services Federation Language (WS-Federation). 2006.

[75] Vivek Haldar, Deepak Chandra und Michael Franz. Dynamic Taint Propagation for Java. In: *Proceedings of the 21st Annual Computer Security Applications Conference (2005 ACSAC)*, 2005.

[76] Bret Hartman, Donald J. Flinn, Konstantin Beznosov und Shirley Kawamoto. *Mastering Web Service Security.* Wiley Publishing, 2003.

[77] L. Todd Heberlein und Matt Bishop. Attack class: address spoofing. In: *Proceedings of the 19th National Information*, S. 147–157, New York, NY, USA, 1996. ACM Press/Addison-Wesley Publishing Co.

[78] Ralph Herkenhöner. Eventbased and Policy-driven Web Service Firewall. *Diploma thesis*, 2005.

[79] Shouichi Hirose und Kanta Matsuura. Enhancing the Resistance of a Provably Secure Key Agreement Protocol to a Denial-of-Service Attacks. In: *ICICS'99, LNCS 1729*, 1999.

[80] Maryann Hondo, Nataraj Nagarathnam und Anthony Nadalin. Securing Web Services. *IBM Systems Journal*, 41:228–241, 2002.

[81] John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 2007.

[82] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion und Steve Byrne. Document Object Model (DOM) Level 3 Core Specification. *W3C Recommendation*, 2004.

[83] Russ Housley, Tim Polk, Warwick Ford und David Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. *IETF request for comments*, RFC 3280, 2002.

[84] Chia-Hsin Huang, Tyng-Ruey Chuang, James J. Lu und Hahn-Ming Lee. XML Evolution: a two-phase XML processing model using XML prefiltering techniques. In: *VLDB'2006: Proceedings of the 32nd international conference on Very large data bases*, S. 1215–1218. VLDB Endowment, 2006.

[85] Takeshi Imamura, Andy Clark und Hiroshi Maruyama. A stream-based implementation of XML encryption. In: *XMLSEC '02: Proceedings of the 2002 ACM workshop on XML security*, S. 11–17, New York, NY, USA, 2002. ACM Press.

[86] Takeshi Imamura, Blair Dillaway und Ed Simon. XML Encryption Syntax and Processing. *W3C Recommendation*, 2002.

[87] Takeshi Imamura und Michiaki Tatsubori. Patterns for Securing Web Services Messaging. In: *OOPSLA 2003 Workshop on Web Services and Service Oriented Architecture*, 2003.

[88] Insecure.org. Ping of Death. `http://www.insecure.org/sploits/ping-o-death.html`.

[89] Tomasz Janczuk. Protect Your Web Services Through The Extensible Policy Framework In WSE 3.0. *MSDN Magazine*, 2006.

[90] Phil Janson. Web Services Interface Definition for Intrusion Defense. *IBM alphaworks*, 2005.

[91] Meiko Jensen. Konzeption und Implementierung einer BPEL-Firewall. *Diploma Thesis*, 2007.

[92] Meiko Jensen, Nils Gruschka, Ralph Herkenhöner und Norbert Luttenberger. SOA and Web Services: New Technologies, New Standards – New Attacks. In: *Proceeding of the 5th IEEE European Conference on Web Services (ECOWS '07)*, S. 35–44, 2007.

[93] Meiko Jensen, Nils Gruschka und Norbert Luttenberger. The Impact of Flooding Attacks on Network-based Services. In: *Proceedings of the Third International Conference on Availability, Reliability and Security (ARES 2008)*, S. 509–513, 2008.

[94] Laurent Joncheray. A Simple Attack Against TCP. In: *Proceedings of the 5th USENIX UNIX Security Symposium*, 1995.

[95] Chris Kaler und Anthony Nadalin (editors). Web Services Security Policy Language (WS-SecurityPolicy) 1.1. 2005.

[96] B. Kaliski und J. Staddon. PKCS #1: RSA Cryptography Specifications Version 2.0. *IETF request for comments*, RFC 2437, 1998.

[97] A. Keller und H. Ludwig. Policy-basiertes Management: State-of-the-Art und zukünftige Fragestellungen. *PIK*, 27:93–102, 2004.

[98] Hristo Koshutanski und Fabio Massacci. An access control framework for business processes for web services. In: *XMLSEC '03: Proceedings of the 2003 ACM workshop on XML security*, S. 15–24, New York, NY, USA, 2003. ACM Press.

[99] Margaret G. Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets und Martha Mercaldi. XML screamer: an integrated approach to high performance XML parsing, validation and deserialization. In: *WWW '06: Proceedings of the 15th international conference on World Wide Web*, S. 93–102, New York, NY, USA, 2006. ACM Press.

[100] D. Kristol und L. Montulli. HTTP State Management Mechanism. *IETF request for comments*, RFC 2965, 2000.

[101] Jussipekka Leiwo, Pekka Nikander und Tuomas Aura. Towards network denial of service resistant protocols. In: *Proc. of the 15th International Information Security Conference (IFIP/SEC)*, 2000.

[102] Jussipekka Leiwo und Yuliang Zheng. Layered Protection of Availability. In: *Proceedings of the Pacific Asia Conference on Information Systems*, 1997.

[103] E. Levinson. The MIME Multipart/Related Content-type. *IETF request for comments*, RFC 2387, 1998.

[104] Pete Lindstrom. Attacking and Defending Web Service. Technical report, Spire Security, 2004.

[105] Carsten Link. *Integration applikationsspezifischer Zugriffskontrolle in Serverprogramme durch Erweiterung der Programmiersprache Java um Konstrukte zur Einbettung von Referenzmonitoren.* Dissertation, Christian-Albrechts-Universität, Kiel, 2006.

[106] Hongbin Liu, Shrideep Pallickara und Geoffrey Fox. Performance of Web Services Security. In: *Proceedings of the 13th Annual Mardi Gras Conference*, February 2005.

[107] Wei Lu, Kenneth Chiu, Aleksander Slominski und Dennis Gannon. A Streaming Validation Model for SOAP Digital Signature. *In The 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, 2005.

[108] Satoshi Makino, Michiaki Tatsubori, Kent Tamura und Yuichi Nakamura. Improving WS-Security Performance with a Template-Based Approach. In: *Proceedings of the International Conference on Web Services (ICWS 2005)*, S. 581–588, 2005.

[109] Hiroshi Maruyama et al. *XML and Java – Developing Web Applications*. Addison-Wesley, 2002.

[110] Michael McIntosh und Paula Austel. XML signature element wrapping attacks and countermeasures. In: *SWS '05: Proceedings of the 2005 workshop on Secure web services*, S. 20–27, New York, NY, USA, 2005. ACM Press.

[111] Michael McIntosh, Martin Gudgin, K. Scott Morrison und Abbie Barbir. Basic Security Profile Version 1.0. *WS-I Organisation*, 2007.

[112] Brett McLaughlin. *Java and XML Data Binding*. O'Reilly, 2002.

[113] Catherine Meadows. A Formal Framework and Evaluation Method for Network Denial of Service. In: *PCSFW: Proceedings ot the 12th Computer Security Foundation Workshop*, 1999.

[114] Catherine Meadows. Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trends. *IEEE Journal on Selected Areas in Communications*, 21(1):44–54, January 2003.

[115] Jonathan Millen. Denial of Service: A perspective. *Dependable Computing for Critical Applications*, 4:93–108, 1995.

[116] Jonathan K. Millen. A Resource Allocation Model for Denial of Service. In: *Proceedings of the IEEE Symposium on Security and Privacy*, S. 137–147, 1992.

[117] Stefan Mintert, editor. *XML & Co. Die W3C-Spezifikationen für Dokumenten- und Datenarchitektur*. Addison-Wesley, 2002.

[118] Jelena Mirkovic und Peter Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, 2004.

[119] R. Moats. URN Syntax. *IETF request for comments*, RFC 2141, 1997.

[120] David Moore und Colleen Shannon. The spread of the Witty worm. *IEEE Security & Privacy Magazine*, 2:46–50, 2004.

[121] David Moore, Colleen Shannon, Douglas J. Brown, Geoffrey M. Voelker und Stefan Savage. Inferring Internet denial-of-service activity. *ACM Transactions on Computer Systems*, 24(2):115–139, 2006.

[122] Tim Moses. eXtensible Access Control Markup Language (XACML) Version 2.0. *OASIS Standard*, 2005.

[123] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir und Hans Granqvist. WS-Trust 1.3. *OASIS Standard*, 2007.

[124] Anthony Nadalin, Chris Kaler, Ronald Monzillo und Phillip Hallam-Baker. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). 2006.

[125] Roger M. Needham. Denial of service. In: *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, S. 151–153, New York, NY, USA, 1993. ACM Press.

[126] Roger M. Needham. Denial of service: an example. *Commununications of the ACM*, 37(11):42–46, 1994.

[127] Markus L. Noga, Steffen Schott und Welf Löwe. Lazy XML processing. In: *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, S. 88–94, New York, NY, USA, 2002. ACM Press.

[128] Dan Olteanu, Tim Furche und Francois Bry. An efficient single-pass query evaluator for XML data streams. In: *SAC'04: Proceedings of the 2004 ACM symposium on Applied computing,*, S. 627–631, 2004.

[129] Mark O'Neill. *Web Services Security*. McGraw-Hill, 2003.

[130] Kouichi Ono, Yuichi Nakamura, Fumiko Satoh und Takaaki Tateishi. Verifying the Consistency of Security Policies by Abstracting in Security Types. In: *Proceedings of 2007 IEEE International Conference on Web Services*, S. 497–504, 2007.

[131] The Apache XML Project. Xerces2 Java Parser. `http://xerces.apache.org/xerces2-j/`.

[132] K. Raeburn, S. Hartman und C. Neuman. The Kerberos Network Authentication Service (V5). *IETF request for comments*, RFC 4120, 2005.

[133] Mohammad Ashiqur Rahaman, Andreas Schaad und Maarten Rits. Towards secure SOAP message exchange in a SOA. In: *SWS '06: Proceedings of the 3rd ACM workshop on Secure web services*, S. 77–84, New York, NY, USA, 2006. ACM Press.

[134] Valentin Razmov. Denial of Service Attacks and How to Defend Against Them. *Graduate Networking Course, Survey Project Paper*, 2000.

[135] Florian Reuter und Norbert Luttenberger. Cardinality Constraint Automata: A Core Technology for Efficient XML Schema-aware Parsers. Technical report, SWARMS Project, 2003.

[136] Jothy Rosenberg und David Remy. *Security Web Services with WS-Security*. Sams Publishing, 2004.

[137] Karen Scarfone und Peter Mell. Guide to Intrusion Detection and Prevention Systems (IDPS). *Recommendations of the National Institute of Standards and Technology*, 2007.

[138] Günter Schäfer. Sabotageangriffe auf Kommunikationsstrukturen: Angriffstechniken und Abwehrmaßnahmen. *PIK 28*, S. 130–139, 2005.

[139] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram und Diego Zamboni. Analysis of a Denial of Service Attack on TCP. In: *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, S. 208–223, Washington, DC, USA, May 1997. IEEE Computer Society.

[140] Luc Segoufin und Victor Vianu. Validating Streaming XML Documents. In: *Symposium on Principles of Database Systems (PODS)*, S. 53–64, 2002.

[141] Clay Shields. What do we mean by Network Denial of Service? In: *Proceedings of the 2002 IEEE Workshop on Information Assurance and Security*, June 2002.

[142] R. Shirey. Internet Security Glossary. *IETF request for comments*, RFC 2828, 2000.

[143] Anoop Singhal, Theodore Winograd und Karen Scarfone. Guide to Secure Web Services. *Recommendations of the National Institute of Standards and Technology*, Special Publication 800-95, 2007.

[144] Adam Smith. Estonia: Under Siege on the Web. *Time Magazine,* `http://www.time.com/time/magazine/article/0,9171,1626744,00.html`, 2007.

[145] William Stallings. *Cryptography and Network Security.* Prentice Hall, 2003.

[146] Michiaki Tatsubori, Takeshi Imamura und Yuhichi Nakamura. Best-Practice Patterns and Tool Support for Configuring Secure Web Services Messaging. In: *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, S. 244, Washington, DC, USA, 2004. IEEE Computer Society.

[147] Java Web Services Performance Team. Streaming APIs for XML Parsers. Technical report, Sun Microsystems, 2005.

[148] The Apache Software Foundation. Performance FAQs. *The Apache XML Project* `http://xerces.apache.org/xerces2-j/faq-performance.html`.

[149] The SAX Project. Simple API for XML – SAX 2.0.1. `http://www.saxproject.org/`, 2002.

[150] Henry S. Thompson, David Beech, Murray Maloney und Noah Mendelsohn. XML Schema Part 1: Structures Second Edition. *W3C Recommendation*, 2004.

[151] Mahesh V. Tripunitara und Partha Dutta. A Middleware Approach to Asynchronous and Backward Compatible Detection and Prevention of ARP Cache Poisoning. In: *ACSAC '99: Proceedings of the 15th Annual Computer Security Applications Conference*, S. 303, Washington, DC, USA, 1999. IEEE Computer Society.

[152] Wil M. P. van der Aalst, Marlon Dumas und Arthur H. M. ter Hofstede. Web Service Composition Languages: Old Wine in New Bottles? In: *Proceedings of the 29th Euromicro Conference*, S. 298–305, 2003.

[153] Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, N. Russell, H. M. W. Verbeek und P. Wohed. Life After BPEL? In: *WS-FM 2005, volume 3670 of Lecture Notes in Computer Science*, S. 35–50, 2005.

[154] R. Yavatkar, D. Pendarakis und R. Guerin. A Framework for Policy-based Admission Control. *IETF request for comments*, RFC 2753, 2000.

[155] Xinfeng Ye und Santokh Singh. A SOA approach to counter DDoS attacks. In: *Proceeding of 2007 IEEE International Conference on Web Services (ICWS)*, S. 567–574, 2007.

[156] Che-Fn Yu und Virgil D. Gligor. A specification and verification method for preventing denial of service. In: *IEEE Transactions on Software Engineering*, S. 581–592, 1990.

[157] Jesper Zedlitz. Spezifikation und Implementierung eines Application Level Gateways für Web Service. *Diploma thesis*, 2004.

[158] Hubert Zimmerman. OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions of Communications*, 28:425–432, 1980.

[159] Olaf Zimmermann, Mark R. Tomlinson und Stefan Peuser. *Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects*. Springer, 2003.

Danksagung

An dieser Stelle möchte ich mich sehr herzlich bei all denen bedanken, die mich bei der Erstellung der Arbeit fachlich und moralisch unterstützt haben und ohne deren Hilfe diese Arbeit nicht zustande gekommen wäre.

Mein besonderer Dank gilt Prof. Dr.-Ing. Norbert Luttenberger für die Betreuung dieser Dissertation. Seine fachlichen und redaktionellen Anmerkungen haben wesentlich zur Qualität der Arbeit beigetragen. Außerdem danke ich Prof. Dr. Stefan Fischer, der die Erstellung des Zweitgutachtens auf sich genommen hat, sowie Prof. Dr. Thomas Wilke, Prof. Dr. Bernd Stellmacher und Prof. Dr. Reinhard von Hanxleden für die Teilnahme an der Prüfungskommission.

Weiterhin möchte ich mich bei meinen Kollegen und Diplomanden Florian Reuter, Carsten Link, Jochen Koberstein, Hagen Peters, Jesper Zedlitz, Ralph Herkenhöner, Meiko Jensen und Torben Dziuk für die wertvollen fachlichen Anregungen bedanken. Für die Hilfe bei der Suche nach Rechtschreib- und Grammatikfehlern danke ich vielmals Caren Kollek, Hagen Peters und Timo Hebebrand.

Ich bedanke mich auch vielmals bei meinen Eltern für ihre Hilfe und Aufmunterung. Mein ganz besonderer Dank gilt meiner Frau June, die stets an mich geglaubt hat und mich in all den Jahren immer unterstützt und ermuntert hat.