Norwegian University of
Science and Technology

Security in a Service-Oriented Architecture

Magne Rodem

Master of Science in Computer Science
Submission date: June 2008
Supervisor: Torbjørn Skramstad, IDI
Co-supervisor: Armaz Mellati, UNINETT FAS

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

In recent years, there have been an increased adoption of service-oriented systems. Applications are made available as services, and this approach is commonly referred to as a service-oriented architecture (SOA). This approach to systems development is still evolving, and one of the challenges in adopting SOA is to ensure the security of the solution.

In this project, the student should look at SOA and the security issues that are relevant in such an architecture. The student should identify threats and countermeasures, and give a set of security guidelines for development of service-oriented systems. Using these guidelines, a simple and secure SOA-based system should be designed.

Assignment given: 15. January 2008
Supervisor: Torbjørn Skramstad, IDI

Abstract

In a service-oriented architecture (SOA), parts of software applications are made available as services. These services can be combined across multiple applications, technologies, and organizations. As a result, functionality can be more easily reused, and new business processes can be assembled at a low cost. However, as more functionality is exposed outside of the traditional boundaries of applications, new approaches to security are needed.

While SOA shares many of the security threats of traditional systems, the countermeasures to some of these threats may differ. Most notably, eavesdropping, data tampering, and replay attacks must be countered on the message level in a complex SOA environment. In addition, the open and distributed nature of SOA leads to new ways of handling authentication, authorization, logging, and monitoring. Web Services are the most popular way of realizing SOA in practice, and make use of a set of standards such as WS-Security, XML Encryption, XML Signature, and SAML for handling these new security approaches.

Guidelines exist for development of secure software systems, and provide recommendations for things to do or to avoid. In this thesis, I use my findings with regard to security challenges, threats, and countermeasures to create a set of security guidelines that should be applied during requirements engineering and design of a SOA. Practical use of these guidelines is demonstrated by applying them during development of a SOA-based system. This system imports personal data into multiple administrative systems managed by UNINETT FAS, and is designed using Web Services and XML-based security standards. Through this practical demonstration, I show that my guidelines can be used as a reference for making appropriate security decisions during development of a SOA.

Preface

This master's thesis documents the work performed by one student during the spring of 2008. The thesis was written at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The assignment was initiated by UNINETT FAS AS.

I would like to thank my supervisor, Torbjørn Skramstad (IDI/NTNU), and my co-supervisor, Armaz Mellati (UNINETT FAS), for their continuous support and valuable feedback during this project.

Magne Rodem

Trondheim, June 10, 2008

Contents

List of Abbreviations **xi**

1 Introduction **1**
1.1 Background and Motivation 1
1.2 Research Agenda 2
1.3 Limitations 3
1.4 Related Work 3
1.5 Report Outline 4

2 Service-Oriented Architecture **7**
2.1 Definitions and Concepts 7
2.2 Terminology 8
2.3 When to Adopt 9
2.4 Security Challenges 9
2.5 Summary 11

3 Web Services **13**
3.1 Definitions and Concepts 13
3.2 Fundamental Standards 14
3.3 Security Standards 19
3.4 Summary 29

4 Threats and Countermeasures **31**
4.1 SQL Injection 31
4.2 XPath Injection 32
4.3 Buffer Overflow Attacks 34
4.4 Entity Recursion Attacks 35
4.5 External Entity Attacks 36
4.6 Service Description Scanning Attacks 37
4.7 Distributed Denial of Service (DDoS) Attacks 37
4.8 Eavesdropping 38
4.9 Tampering Data 39
4.10 Replay Attacks 40
4.11 Repudiation 41
4.12 Summary 42

5 Security Guidelines **45**
5.1 General Security Guidelines 45
5.2 SOA Security Guidelines 47
5.3 Summary 49

6 Requirements Engineering **51**
6.1 System Introduction 51
6.2 Security Challenges 54
6.3 Requirements 56
6.4 Summary 61

7 Design **63**
7.1 Standards 63
7.2 Technologies 65
7.3 Service Design 69
7.4 Summary 75

8 Discussion **77**
8.1 The Benefit of Having Security Guidelines 77
8.2 Practical Use of My Guidelines 78
8.3 The Side Effects of Centralization 78
8.4 Security Standards and Interoperability 79
8.5 Experiences with Frameworks and Libraries 80

9 Conclusion **81**

10 Further Work **83**

Bibliography **85**

A Use Case and Misuse Case Descriptions **89**

B Source Code **95**
B.1 WSDL Documents 95
B.2 Input Validation 100
B.3 SAML Processing 102
B.4 Logging Appender 104
B.5 Service Configuration 106

List of Abbreviations

CERT	Computer Emergency Response Team
DDoS	Distributed Denial of Service
DoS	Denial of Service
DTD	Document Type Definition
ESB	Enterprise Service Bus
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
ISP	Internet Service Provider
OASIS	Organization for the Advancement of Structured Information Standards
SAML	Security Assertion Markup Language
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
SQL	Structured Query Language
TLS	Transport Layer Security
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WS-I	Web Services Interoperability Organization
WSDL	Web Service Definition Language
WSS4J	Web Services Security for Java
XML	eXtensible Markup Language
XSL	eXtensible Stylesheet Language

Chapter 1

Introduction

This introductory chapter describes the background and motivation for this thesis. The research agenda, limitations, and related work are presented, and an outline of the report is provided.

1.1 Background and Motivation

Modern organizations have come to rely on advanced IT systems for performing day-to-day tasks. Over time, business needs have grown, and different kinds of applications and infrastructure have evolved [20]. This has brought a large variety of applications into the the IT architecture of organizations. Different applications are used in relation to different tasks, such as sales, finances, and project management. By choosing different applications for each task, organizations achieve flexibility and rapid delivery of functionality [23].

While this diverse approach enables tailoring the IT infrastructure to suit the individual organization's needs, it also has some problems [20]. Applications that are developed independently of each other provide functionality that can only be used within the boundaries of each application. In addition, differences between platforms, programming languages, and protocols lead to technology boundaries that can be difficult to cross. These boundaries limit the organization's ability to integrate and reuse functionality of its applications.

In recent years, service-oriented architectures (SOA) have emerged as a viable option for integrating applications [22]. By supporting heterogeneity, decentralization, and fault tolerance [19], SOA excels at keeping systems scalable and flexible while growing. SOA sees IT systems as collections of units called services. These services are built in such a way that they can be readily consumed by other applications and services. This makes it possible to create new applications by reusing services provided by other applications [20].

As parts of applications are made available as services, it becomes easier to combine or integrate functionality of applications. However, as more function-

ality is made available outside the internal scope of the applications, security is affected. SOA lowers the boundaries of applications, technology, and organizations. Traditional security approaches assumed and took advantage of these boundaries, and thus new approaches need to be used when dealing with SOA [20].

As mentioned by Josuttis [19], building a SOA is hard, and a lot of effort is required to help improving peoples understanding of SOA. According to McGraw [28], getting things like state and authentication right in SOA is a real challenge. SOA introduces new approaches for dealing with security, and it is especially important that developers and IT architects have a clear understanding of new security approaches and issues in relation to SOA. Due to the significant growth in the development of SOA-based systems over the past decade [22], there is a need to raise awareness of potential security issues related to SOA, and to provide some guidelines on how to develop secure SOA-solutions.

1.2 Research Agenda

In this thesis, I will look at SOA and the security issues of such an architecture. I will compare security in SOA with security in traditional systems, describe relevant security standards used by SOA-based systems, and identify threats and countermeasures that can be related to SOA. Using this information, I will create a set of security guidelines that should be applied during requirements engineering and design of a SOA. Finally, I will demonstrate the use of these guidelines by designing a simple SOA-based system. This leads to the following research questions:

1. What is a service-oriented architecture?
2. How does security in a service-oriented architecture compare to security in traditional computing?
3. What technical security standards can be utilized in a service-oriented architecture?
4. What threats and countermeasures can be related to a service-oriented architecture?
5. What guidelines with regard to security should be followed during requirements engineering and design of a service-oriented architecture?
6. How can a secure service-oriented architecture be designed?

I will seek to answer these questions by studying available literature on this subject. I will answer question 6 by demonstrating how a secure SOA solution can be designed using security guidelines.

1.3 Limitations

SOA is a large subject comprising a lot of technologies and concepts. This project is limited both in time and resources, and because of this, I have had to add some constraints to my thesis.

The idea of a service can be implemented in multiple ways [20], and I will take this into account in this thesis. As a result, the security guidelines that I will present are on a high level, and do not assume any specific implementation. However, Web Services are the most popular way of realizing services in a SOA [19]. Because of this, I will focus on Web Services when describing the most relevant standards for SOA and security. This is necessary to be able to show threats and countermeasures, and to describe design of a SOA in practice.

Services in a SOA can be connected and managed using an IT infrastructure known as Enterprise Service Bus (ESB) [20]. An ESB may provide functionality such as routing and transformation of data that are sent between services. There are different opinions about the exact role and responsibilities of an ESB, and very different technical approaches are used for realizing such a facility [19]. Due to time constraints, and the many different opinions and approaches to how an ESB is implemented, I will not describe or use ESBs in this thesis.

Several companies such as IBM, Oracle, and SAP sell SOA-solutions [19]. I will not describe or use such commercial solutions in this thesis, as I do not have the resources to do so. Instead, I will concentrate on open source software.

1.4 Related Work

The book *SOA in Practice* [19] by Nicolai M. Josuttis shows how SOA can be created and maintained. Real-world problems of implementing and running a SOA in practice are described. The book also touches upon the subject of security in SOA and talks about aspects such as security requirements and how to deal with interoperability and heterogeneity while handling security. Web Services and their security standards are also briefly mentioned.

SOA Security [20] by Ramaro Kanneganti and Prasad Chodavarapu is a book about SOA and its security aspects. It was published about the same time as I started writing this thesis. Compared to Josuttis, Kanneganti and Chodavarapu focus more on Web Services and security standards. All standards, both fundamental and those related to security, are described in detail. In addition, the Web Services framework Apache Axis is used for showing examples of how the standards can be utilized in practice.

The book *Building Secure Software* [45] by John Viega and Gary McGraw [45] presents a set of guidelines for developing secure software systems. These guidelines are applicable regardless of programming language, platform, or system architecture.

In their master's thesis *Web Application Security* [12], Julie-Marie Foss and Nina Ingvaldsen created three security guidelines that should be applied in the context of web applications. The use of guidelines were demonstrated by developing a web application and applying their guidelines during the development process. I will perform a similar study as Foss and Ingvaldsen, but while they focused on web applications, I will focus on SOA.

1.5 Report Outline

This report is divided into ten chapters. In this section, I will give a brief description of each chapter.

- **Chapter 1 - Introduction**
  The first chapter introduces the background and motivation for this thesis. In addition, research goals, limitations, and related work are presented.
- **Chapter 2 - Service-Oriented Architecture**
  This chapter describes the basic concepts of service-oriented architectures. The term "SOA" is defined, and some advantages of using SOA is presented. At the end of this chapter, security in SOA is compared with security in traditional systems.
- **Chapter 3 - Web Services**
  In this chapter, Web Services are described. Web Services are the most popular way of realizing SOA in practice [19]. Knowing how Web Services work is essential for understanding potential threats and countermeasures in SOA-based systems. Fundamental standards are described and one section is devoted to security related standards.
- **Chapter 4 - Threats and Countermeasures**
  This chapter describes threats and countermeasures that may apply to SOA. While SOA shares many of the threats of traditional systems, the form of attacks may be a bit different, and a few different strategies for countering the threats are needed.
- **Chapter 5 - Security Guidelines**
  This chapter contains my main contribution in this thesis, which is security guidelines for SOA. Before presenting the SOA-specific guidelines, some general security guidelines are briefly mentioned. The guidelines that I have created are intended to be used in addition to these general guidelines.
- **Chapter 6 - Requirements Engineering**
  In this chapter, I show how some of the guidelines in chapter 5 can be applied when identifying requirements for a SOA-based system. A real-world case scenario provided by UNINETT FAS is described, and threats and countermeasures in relation to this case are identified.

- **Chapter 7 - Design**
  This chapter takes the findings from chapter 6 into account when designing the SOA. Some of the security guidelines from chapter 5 are applied, and appropriate standards and technologies for realizing the SOA are chosen. Finally, the services of the SOA are designed and described using figures and class diagrams.

- **Chapter 8 - Discussion**
  In this chapter, my SOA-specific security guidelines are discussed with regard to aspects such as their importance, ease of use, and side-effects. In addition, I comment on some of the experiences I have had with frameworks and libraries for Web Service development.

- **Chapter 9 - Conclusion**
  This chapter summarizes and concludes my work.

- **Chapter 10 - Further Work**
  This chapter discusses possible further work related to this project.

Chapter 2

Service-Oriented Architecture

This chapter aims to introduce the reader to the basic concepts of service-oriented architectures. A brief description of SOA and it advantages is provided, and the security challenges of SOA are introduced.

2.1 Definitions and Concepts

An exact definition of the term "service-oriented architecture", or SOA in short, is hard to find. This, however, does not mean that such a definition does not exist. In fact, many different definitions of SOA exist, often depending on the context in which the term is used. In this thesis, I will refer to the definition given in the reference model for service-oriented architecture [25] by OASIS[1]:

> "Service-Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains."

As can be seen from the definition, SOA is a paradigm. It is not a concrete architecture, but rather something that leads to a concrete architecture. In other words, SOA is an approach or a way of thinking, and thus it is not some concrete framework or tool that can be bought [19].

Even though SOA is just a paradigm, some technical concepts are often used for describing it. As the name implies, SOA is largely based on the concept of services. A service is an IT representation of some business functionality. The technical details of a service should not be externally visible, and their interfaces should be designed in such a way that business people are able to understand them [19]. This is not only an advantage when communicating with stakeholders, but also results in very abstract service descriptions. The

[1]The Organization for the Advancement of Structured Information Standards (OASIS) drives the development, convergence, and adoption of open standards for the global information society [32].

result is that platform-specific details no longer matter, and that heterogeneous systems can be more easily combined.

A common misconception regarding SOA is that Web Services (described in chapter 3) are a necessity in the realization of a SOA. It is important to recognize that even though services in a SOA are often implemented as Web Services, other technologies may be used as well [20].

2.2 Terminology

Before continuing, some words need to be said about the terminology used in this report. SOA represents a different way of thinking about software systems, and because of this, we need to define some new terms.

The terms ”client” and ”server” are often used in relation to traditional systems. However, their meanings in different contexts are a bit unclear. For instance, ”client” may be interpreted both as an application and a communicating party. In addition, what do we call a server that initiates communication with another server? Is it still a server or may it now be considered to be a client? This gets confusing in a service-oriented architecture where many services may be involved. Because of this, the terms ”provider” and ”consumer” are often used when talking about services [19]:

- A *provider* is a system that implements a service so that other systems can call it.
- A *consumer* is a system that calls a service (uses a provided service).

This terminology is illustrated by figure 2.1. The terms ”client” and ”server” will naturally still be used on some occasions, but only in cases where their meanings are expected to be clearly understood.

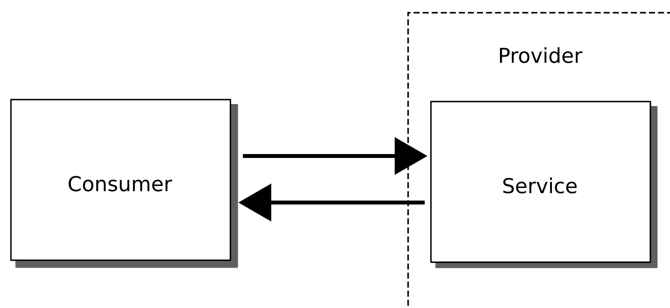

Figure 2.1: A consumer calls a service that is implemented by a provider.

2.3 When to Adopt

SOA is not the right solution to every integration problem. Seeing the current popularity of SOA [22], it can be tempting to use SOA even if it is not actually appropriate. Before adopting this paradigm, one should take a good look at the IT infrastructure and business processes of the organization and find out what can be gained by using SOA. Josuttis [19] presents three system characteristics that SOA is especially good at handling: Complex distributed systems, different owners and heterogeneity:

- In complex distributed systems, entities need to locate and use the distributed capabilities that are available. SOA allows this by enabling interactions between service providers and service consumers.
- When capabilities are controlled by different owners, different priorities come into play, and one cannot expect to control everything. The practices and processes of SOA take into account that distributed systems are not controlled by single owners.
- It is common for large systems to use different platforms and programming languages. This heterogeneity can be difficult to deal with. SOA, however accepts heterogeneity and supports this attribute.

Natis [31] is more specific, and mentions some major benefits of SOA. By dividing capabilities into services, SOA enables incremental development and deployment of business software. This also leads to the possibility of reusing business components, as one service can be used in multiple business processes. As services are often available for reuse, new business processes can be assembled at a low cost.

2.4 Security Challenges

SOA represents a new way of thinking with regard to business software. In SOA, functionality that previously were contained in one large application can be divided into multiple services, working together to achieve a business goal. How does this new way of thinking affect security? I will give an answer to this question in this section.

The traditional approach to security is well known. Let us look at an example. One server application provides various kinds of functionality to a client application. All this functionality can be represented as modules contained in the server application. To be able to access one or more of these modules, the client must to go through a security module. This module typically authenticates and authorizes the client, and takes care of other functional aspects of security such as protection against attacks. Figure 2.2, adopted from Kanneganti and Chodavarapu [20], illustrates the traditional approach to security.

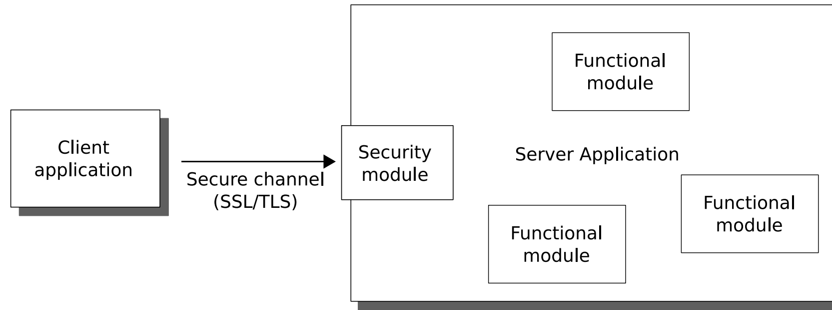

Figure 2.2: The traditional security approach, consisting of a single server application that offers independent functionality to the clients. This figure was adopted from Kanneganti and Chodavarapu [20].

The reason why this approach works is that all of the functionality is contained inside one single application. This means that one security module is sufficient for protecting all functional modules. However, in SOA the functional modules are not found inside one single application. The modules, or in this case services, are spread across multiple applications running on different servers. Figure 2.3, inspired by Kanneganti and Chodavarapu [20], illustrates a potential SOA environment.

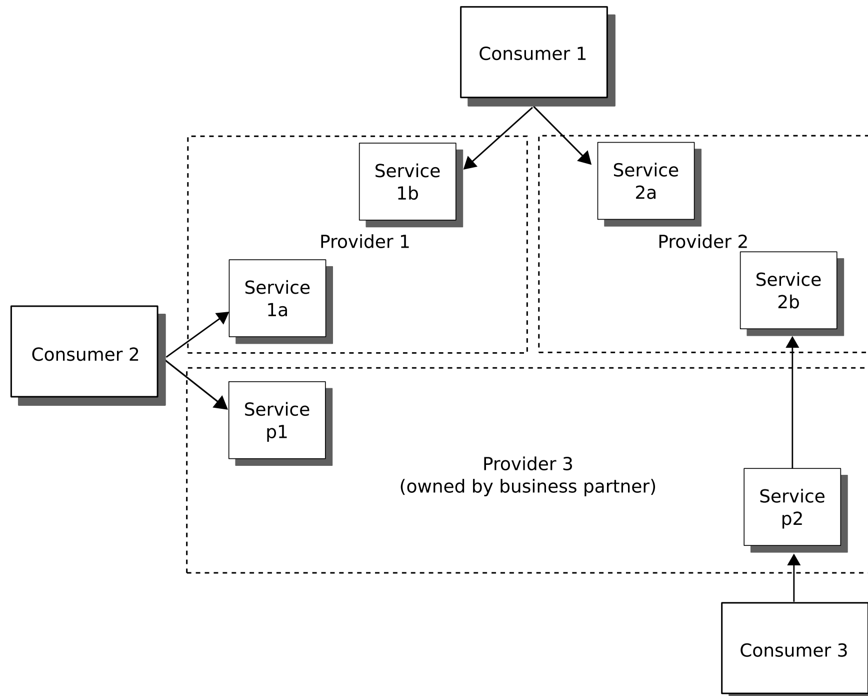

Figure 2.3: A potential SOA environment consisting of three providers with two services each, plus three consumers that make use of the services. The figure was inspired by Kanneganti and Chodavarapu [20].

As can be seen from figure 2.3, a SOA environment is a bit more complex than the server application in figure 2.2. SOA enables the possibility of recombin-

ing functionality from different applications, as illustrated by Consumer 1 in figure 2.3. Services in one enterprise can even be combined with services of partners (Consumer 2 in figure 2.3). In addition, services may be invoked directly or by partners, as shown in figure 2.3 by the call from Service p2 to Service 2b.

The many different ways of invoking services lead to a high degree of flexibility. However, this flexibility does not come without a cost; we need a new approach for dealing with security. Compared to the server application in figure 2.2, SOA is more open and unpredictable. Services may be invoked by different client applications in varying contexts. An application designer will find it difficult to foresee all possible situations in which a service may be invoked [20].

Traditionally, a secure channel between the client and server application have been enough to ensure data confidentiality and integrity between the two parties. However, in SOA additional parties may come into play. Take for example Consumer 3 in figure 2.3 which invokes Service p2. Parts of the message that the consumer sends to Service p2 might be sensitive and only intended for Service 2b. Service p2 is administrated by an external partner who should not be able to see the sensitive information. In this case, a secure channel between the consumer and Service p2 is not enough, as this only ensures confidentiality during transport between the services.

When functionality is made available as services, more entrances are available for accessing the applications. This introduces new challenges with respect to authentication and authorization, as access control needs to be enforced in multiple places. More entrances for users also lead to more potential places for attackers to exploit. As a result, keeping track of all events happening in a SOA is not as easy as in traditional single-entrance applications. Being able to find out who or what was the reason for a security attack or a security hole can bolster the security of a system [19]. Thus, a good strategy for handling logging and monitoring in a SOA should be in place.

When introducing approaches for handling security, it is important to retain the philosophy of SOA. More specifically, services should be kept as open and easy to use as possible, and the interoperability should not suffer because of security [20].

This has been a high-level introduction to the security challenges of SOA. In chapter 4 I will go more into specifics about the security concerns in SOA and how they can be addressed.

2.5 Summary

This chapter has introduced the main concepts of SOA. In addition, the main security challenges of SOA has been described.

A definition of SOA was provided in section 2.1. It was established that SOA is not a concrete architecture, but rather a paradigm. In other words, it is an

approach or a way of thinking. In section 2.3, the benefits of using SOA were presented. SOA is good at handling complex distributed systems, different owners and heterogeneity. In addition, SOA enables reuse of business components and assembly of new business processes at a low cost.

There are, however, a few security challenges related to SOA as described in section 2.4. SOA is more open and unpredictable than traditional applications, which makes it difficult to foresee all possible situations in which a service may be invoked. Combination of services may lead to new considerations with regard to confidentiality and integrity. In addition, services introduce more entrances to applications, which complicates authentication and authorization, as well as logging and monitoring.

The next chapter describes Web Services, which is a common way of realizing service-oriented architectures.

Chapter 3

Web Services

Web Services are the most popular way of realizing SOA in practice [20]. To be able to understand threats and countermeasures that are related to SOA, one needs to have a basic understanding of how Web Services work. This chapter defines what a Web Service is, and describes the most fundamental standards used by such services. In addition, one section is devoted to security standards.

3.1 Definitions and Concepts

The World Wide Web Consortium (W3C[1]) uses the following definition of a Web Service [8]:

> "A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

As is evident from this definition, Web Services are realized using several specific standards which are used to enable interoperable interaction. XML plays a major part in Web Services. A consumer and a Web Service interacts by sending messages encoded in XML format to each other. This leads to high interoperability, as XML data is expressed using plain text and can be processed by almost any programming language [20]. The Simple Object Access Protocol (SOAP) is used for sending the XML-based messages. SOAP operates on a higher level than traditional transport protocols such as HTTP. A detailed description of a Web Service and its operations can be found in a Web Services

[1] The World Wide Web Consortium (W3C) develops web-related interoperable technologies (specifications, guidelines, software, and tools) [46].

Description Language (WSDL) document. Figure 3.1 illustrates interaction between a consumer and a Web Service.

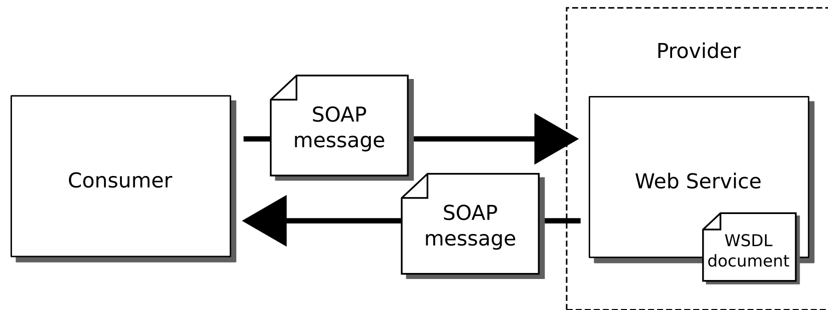

Figure 3.1: A consumer calls a Web Service that is implemented by a provider. The interface of the Web Service is described using WSDL, and the two parties interact using SOAP messages.

3.2 Fundamental Standards

W3C uses several technical concepts such as XML, WSDL, and SOAP for defining what a Web Service is. Thus, in order to understand how Web Services work, one should be familiar with these concepts. The responsibilities and relationships of XML, WSDL, SOAP, and HTTP are illustrated in figure 3.2.

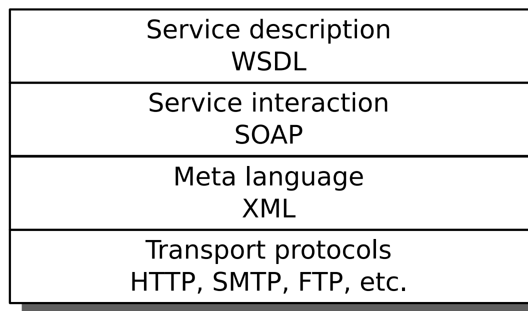

Figure 3.2: The fundamental concepts of Web Services.

At the bottom, a transport protocol is required for interacting with a Web Service. This protocol is typically HTTP, but other protocols may be used [19]. The messages that are sent using HTTP are expressed as XML documents. SOAP defines the structure of these documents, and WSDL describes how a consumer may compose and send SOAP messages to a Web Service. The following sections describe XML, WSDL, and SOAP in more detail.

3.2.1 eXtensible Markup Language (XML)

XML is the common denominator format for data exchange between Web Services. Some basic concepts of XML, such as elements, attributes, XML

Schemas, and namespaces are briefly introduced here. For a more detailed description of XML, I recommend Hunter et al. [16].

I will use a simple example to describe the basics of XML. Let us say that a library wants to store or communicate a list of books by using XML. Listing 3.1 shows how the library may express this list.

```
<lib:books xmlns:lib="http://library.com/">
  <lib:book id="1">
    <lib:isbn>1234-567-89-0</lib:isbn>
    <lib:title>Cooking for Dummies</lib:title>
    <lib:author>Donald Cook</lib:author>
    <lib:publisher>John Wiley and Sons</lib:publisher>
  </lib:book>
  <lib:book id="2">
    ...
  </lib:book>
  ...
</lib:book>
```

Listing 3.1: A list of books expressed using XML.

This example is quite self-explanatory, but I will go through the main concepts. Each book is described using `book` elements with four sub-elements each. The `book` elements also have an `id` attribute, specifying a unique book ID. Notice that each element has a prefix named `lib`. This prefix denotes the namespace of the elements [19], and is declared by the `xmlns` attribute of `books`. In this case, the namespace is *http://library.com*.

One might ask what we need namespaces for. A namespace is a unique identifier defined to help avoiding naming conflicts [20]. For instance, similar to the library, a bookstore may also communicate its list of books using XML. The bookstore uses a different set of sub-elements underneath the `book` element, such as pricing information and number of books in stock. If the library and the bookstore decides to merge their lists, there will be two different `book` structures in the same document. Thus, we have a naming conflict, and need a way to distinguish the two types of books from each other. In listing 3.1, the prefix `lib` is used to indicate that the elements belong to the namespace of the library. When adding books from the bookstore to the same document, a new namespace may be defined in a similar way to distinguish between the two types of books. As will be evident in the following sections, namespaces play an important role when multiple XML-based standards are combined in the same document.

In addition to namespaces, we need a way to define the structure of XML documents. This is done using XML Schemas, which enable us to specify the elements that may appear in the documents, and in what hierarchy they may be used [20]. For instance, a `book` element should only appear as a sub-element of `books`, and all `book` elements must have an `id` attribute. All XML-based standards that are described in this chapter have their own XML Schema(s). When a standard is being used, the Schema document(s) of this standard is

specified inside the XML document. XML parsers are then able to check that the XML document conforms to the XML Schema in question.

This has been a very brief introduction to XML. As mentioned, for those who have limited knowledge of XML, I recommend Hunter et al. [16] for a more in-depth description. In the rest of this report I will assume that the reader is familiar with the basic concepts of XML.

3.2.2 Web Services Description Language (WSDL)

Web Service Description Language (WSDL) provides a model for describing Web Services. A WSDL document uses abstractions such as messages, ports, and services, which support a generic way of specifying the protocols and data formats used by a Web Service [20]. WSDL documents have three layers of description [19]:

- The `portType` layer describes the interface of a service. This consists of the service operations, including input and output parameters. These parameters are defined in `message` sections.
- The `binding` layer defines the protocol and format used by the Web Service operations.
- The `service` layer defines the address where the Web Service is available.

Now, let us look at an example that shows what a WSDL document looks like. A university wants to offer a Web Service that returns a student's grade in a given course. A student should invoke the service by supplying a course ID. In return, the students grading for that specific course will be received. Listing 3.2 shows a WSDL document describing this Web Service. For simplicity, I have omitted XML Schemas, types, and binding details from this example.

```
<wsdl:definitions name="CourseRegisterService"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <wsdl:types>
      ...
  </wsdl:types>

  <wsdl:message name="getGradeInput">
    <wsdl:part name="courseId" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="getGradeOutput">
    <wsdl:part name="grade" type="xsd:string"/>
  </wsdl:message>

  <wsdl:portType name="CourseRegister">
    <wsdl:operation name="getGrade">
      <wsdl:input message="tns:getGradeInput" />
      <wsdl:output message="tns:getGradeOutput" />
    </wsdl:operation>
  </wsdl:portType>
```

```
  <wsdl:binding name="CourseRegisterSoapBinding" ...>
    ...
  </wsdl:binding>

  <wsdl:service name="CourseRegisterService">
    <wsdl:port name="CourseRegisterPort"
               binding="tns:CourseRegisterSoapBinding">
      <soap:address location="http://rodem.no:9090/courseregister"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Listing 3.2: A WSDL document describing a service with one operation.

The WSDL document in listing 3.2 contains one operation, `getGrade`, as seen from the `portType` section. When invoked, this operation receives a course ID, and responds with a grade associated with the course in question. The `service` section specifies the location of the service.

A WSDL document can be hand-coded, but often, it is more convenient to generate these documents using a tool [20]. This is especially useful if we have an existing application and want to make its interfaces available through a Web Service.

3.2.3 Simple Object Access Protocol (SOAP)

SOAP is the protocol that is used for exchanging messages with Web Services. While HTTP is typically the low-level protocol used for interaction, SOAP is the specific format for exchanging messages over HTTP [19]. A SOAP message is an XML document consisting of an envelope with a header and a body. Listing 3.3 shows the structure of SOAP messages.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    ...
  </soap:Header>
  <soap:Body>
    ...
  </soap:Body>
</soap:Envelope>
```

Listing 3.3: The structure of SOAP messages.

The `Body` element contains the actual data of the message. There are no constraints on the content of this element, except that it cannot carry free text by itself. In other words, the content should be XML elements only. The actual interpretation of the content of the `Body` element is up to the application receiving the message. [20]

The `Header` element may contain additional information about the message. This element can contain extensions to SOAP and/or additional application semantics. In addition to the `Body` and `Header` elements, a SOAP message

may carry a SOAP `Fault`. This element can be found within the `Body` of the message, and communicates error information back to the consumer. [20]

Let us look at a simple example of message exchange using SOAP. Recall the WSDL document shown in listing 3.2. A student would call this service, supplying a course ID, to get his/her grade for that specific course. Listing 3.4 shows such a call.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    ...
  </soap:Header>
  <soap:Body>
    <courseId>TDT4520</courseId>
  </soap:Body>
</soap:Envelope>
```

Listing 3.4: A SOAP request asking for a grade in course TDT4520.

As can be seen from listing 3.4, the `Body` of the SOAP message contains the input data to the operation, in this case a course ID. For simplicity I have excluded all `Header` elements in this example. Now, let us look at a SOAP response to this call.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    ...
  </soap:Header>
  <soap:Body>
    <grade>C</grade>
  </soap:Body>
</soap:Envelope>
```

Listing 3.5: A SOAP response returning a grade to the consumer.

The response in listing 3.5 is structured in the same way as the request. The `Body` contains the grade for the course in question. In this case the student apparently got a C.

This and the previous sections have introduced some of the fundamental standards used by Web Services. The simple examples in these sections show how services can be described, and how messages can be exchanged. However, I have still not mentioned anything about the security of these concepts. How can a consumer prove his or her identity, and how can messages be encrypted? I will answer these and more security related questions in the next section.

3.3 Security Standards

This section aims to introduce the reader to some of the most important security standards for XML and Web Services, and to show examples of their use. The number of security standards can be overwhelming, and because of this, I will begin by showing an overview of the standards and their area of application. The standards that I will present in this section cover five functional aspects of security:

- Authentication - Verifying the identity of users [20].
- Authorization - Deciding whether or not to permit an action on a resource [20].
- Confidentiality - Protecting secrecy of sensitive data [20].
- Integrity - Detecting data tampering [20].
- Non-repudiation - Preventing either the sender or the receiver from denying a transmitted message [40].

Figure 3.3 shows the main XML and Web Services security standards, and relates them to functional aspects of security.

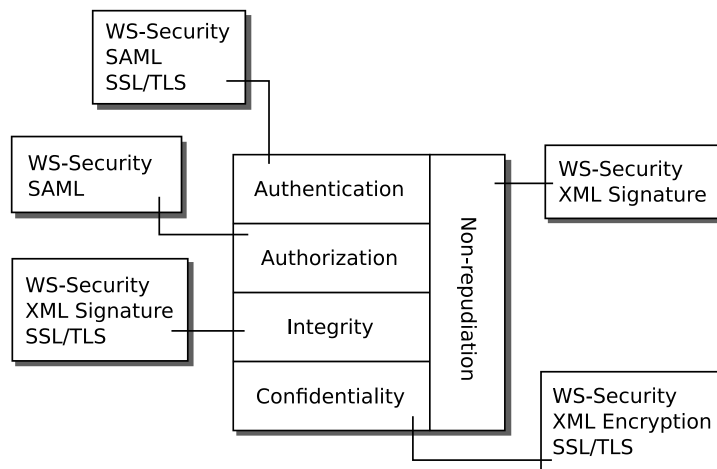

Figure 3.3: Security standards related to XML and Web Services.

There are a few things to note from this figure. For instance, WS-Security may be used in all cases. This does not mean that WS-Security is able to cover all of our needs. As we will see in the following sections, WS-Security provides the syntax and semantics for using other XML-based security standards in SOAP messages. In addition, we see that transport-level security (SSL/TLS) may be used for authentication, integrity, and confidentiality. As mentioned in section 2.4, however, transport-level security may not be appropriate in a complex SOA. In such cases, XML Encryption and XML Signature can be used as an alternative. Lastly, we see that a standard called SAML may be used

for authentication and authorization. WS-Security, XML Encryption, XML Signature, and SAML are described in the following sections.

Before diving into the standards, I will briefly mention the standardization organizations involved in the security of XML and Web Services. XML, WSDL, and SOAP have been developed by W3C. When it comes to Web Services security standards, however, W3C is joined by OASIS. W3C is the development base for XML Encryption and XML Signature, while OASIS is the development base for WS-Security and SAML. [27]

In relation to standards, WS-I should also be mentioned. WS-I (The Web Services Interoperability Organization) is an industry consortium that seeks to enhance interoperability among Web Services implementations [20]. WS-I publishes implementation guidelines known as profiles for popular Web Services standards to promote interoperability. WS-I has published a Basic Security Profile [29] that clarifies WS-Security and associated token profiles.

When choosing a vendor or open source solution for Web Services security, higher value should be placed on WS-I compliance than on simple conformance to standards [27]. Two implementations that both claim WS-Security compliance may not be interoperable, as they might implement different portions of WS-Security. On the other hand, two implementations that both claim WS-I compliance will be interoperable [27].

3.3.1 WS-Security

In section 3.2.3, examples of message exchange using SOAP were presented. However, nothing was said about the security of these messages. One might think that security information such as usernames and passwords are added to the body of messages just as any other information. While this is possible, it has a major drawback. The interpretation of the SOAP body is up to the service receiving the message [20]. The SOAP body is processed as part of the business operations of the service. This means that security would have to be handled as part of business operations, and that security information would be expressed in different ways by different services. We want to express security information in a standardized way that can be understood by all consumers and services. This is where the WS-Security standard comes into play.

WS-Security defines a SOAP header element, named `Security`, for making security claims and ensuring message-level security [20]. Security claims are statements that are made regarding security. These claims may state things like the users identity, who signed the message or what key was used for encryption of the message. Multiple security claims may be added to the same message. Listing 3.6 shows a simplified SOAP message that uses WS-Security for specifying a username and password.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <wsse:Security xmlns:wsse="../oasis-200401-wss-wssecurity-secext-1.0.xsd">
```

```
      <wsse:UsernameToken ...>
        <wsse:Username>magrode</wsse:Username>
        <wsse:Password>password</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soap:Header>

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

Listing 3.6: A SOAP message containing a WS-Security header with a username and a password.

In this example, the `Security` header element contains a `UsernameToken`. A `UsernameToken` carries the claim that a message is sent from a particular user [20]. In this case, the username and password are sent in clear text. `UsernameToken` is only one of many types of security tokens that can be added to the `Security` header. Examples include public key certificates[2] and SAML assertions [20].

While WS-Security supports security tokens that make security claims, it does not specify how security claims are verified, what keys and algorithms to use for encryption and signing, and how requests are authorized. WS-Security only provides the syntactic and semantic support needed to implement any security model in the context of Web Services [20].

3.3.2 XML Encryption

Currently, Secure Sockets Layer/Transport Layer Security (SSL/TLS) is commonly used for secure communication over the Internet [40]. While SSL/TLS is a very secure and reliable protocol for providing end-to-end security between two parties [36], it has some shortcomings with respect to complex service-oriented architectures. SSL/TLS encrypts the communication channel, but this may not be enough, as described in section 2.4. When a message travels through untrusted intermediary nodes on the path to its destination, we need to be able to encrypt parts of messages to prevent the intermediary nodes from reading sensitive data [20].

The XML Encryption standard allows parts of a SOAP message to be encrypted. XML Encryption is not intended to replace or supersede SSL/TLS, but rather provide mechanisms that are not covered by SSL/TLS [36]. With XML Encryption, both secure and non-secure data can be exchanged in the same document, and confidentiality can be ensured on the entire path between the sender and destination endpoints [20]. Another advantage of XML Encryption is that it provides persistent encryption, as encrypted documents can

[2]A public key certificate consists of a public key plus an identifier of the key owner, with the whole block signed by a trusted third party [40].

be stored. XML Encryption is tied to SOAP using WS-Security, and operates at the message level, which means that it is independent of the underlying transport [27].

To be able to understand the examples in this and the next section, the reader should know the basics of public key cryptography. With public key cryptography, a user has two keys. One key is private, while the other key is public. The keys are mathematically related, but it is not possible to derive one key from another [40]. These keys can be used for encryption. Figure 3.4 illustrates the use of public and private keys for encryption.

Alice
Bob's public key
Message to Bob
Encrypt
Bob
Bob's private key
Encrypted message
Decrypt

Figure 3.4: If Alice wants to send a message to Bob, she can encrypt the message using Bob's public key. When receiving the message, Bob can decrypt it with his private key. Nobody other than Bob is able to decrypt the message, as Bob is the only one who has the private key.

In the following example, I will show how XML Encryption is used together with WS-Security. Recall the SOAP message in listing 3.6. In this SOAP message, a username and password were transmitted in clear text. Let us say that we would like to encrypt the `UsernameToken` but keep the body of the message in clear text. Listing 3.7 shows a very simplified example of this.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <wsse:Security xmlns:wsse="../oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
        ...
      </xenc:EncryptedKey>
      <xenc:EncryptedData ...>
        ...
      </xenc:EnctyptedData>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    ...
  </soap:Body>
</soap:Envelope>
```

Listing 3.7: A SOAP message containing a WS-Security header with an encrypted username and password.

As can be seen from listing 3.7, the `UsernameToken` has been replaced by two

other elements, namely `EncryptedKey`[3] and `EncryptedData`:

- `EncryptedKey` contains information about the key that was used for encryption. This is provided to help the recipient in decryption of the message. This information includes the algorithm used for encrypting the key, the key used for encrypting the key, the actual encrypted key and a reference list that indicates which elements of the message that have been encrypted using the key. [20]
- `EncryptedData` contains the actual encrypted data, which in this case is a `UsernameToken`. In addition, this element contains a reference to the encrypted key that was used for encrypting the data, and the algorithm used for encryption. [20]

3.3.3 XML Signature

The integrity of data can be ensured using SSL/TLS [48]. However, as this operates on the transport-level, we cannot rely on SSL/TLS when messages travel through untrusted intermediary nodes. In such cases, we need to use digital signatures for protecting the integrity of messages. A digital signature is analogous to a handwritten signature, and verifies the author and the contents of a message [40]. Not only does this protect the integrity of messages, but as both the author and contents are verified, the author cannot repudiate that a message exchange has taken place.

Signing of messages is performed by computing a message digest and securely communicating this digest together with the message to the receiver. The receiver can then recompute the digest and compare this digest with the one provided by the sender [20]. Public key cryptography is commonly used for communicating digests in a secure manner. The sender encrypts the digest with his or her private key, and the receiver uses the sender's public key to decrypt the digest. Undetected modification of both the message and the encrypted message digest is not possible without knowing the private key that was used for encryption [20].

Let us say that Alice wants to send a message to Bob. Alice wants Bob to be able to verify that the message has not been tampered with. To do this, Alice needs to compute a digest of the message, and communicate this digest securely to Bob along with the actual message. Bob uses Alice's public key to decrypt the encrypted digest and recomputes the digest of the message. Then, Bob compares the two digests to verify that they are equal. Figure 3.5 illustrates the signing and verification process between Alice and Bob.

[3] Public key cryptography is computationally expensive [40]. Because of this, we do not want to encrypt large chunks of data using this scheme. Instead, a randomly chosen symmetric key is encrypted using public key cryptography and sent to the receiving party [20]. It is this symmetric key that is used for encryption and decryption of data. With symmetric encryption, the same key is used both for encryption and decryption, which is less computationally expensive [40].

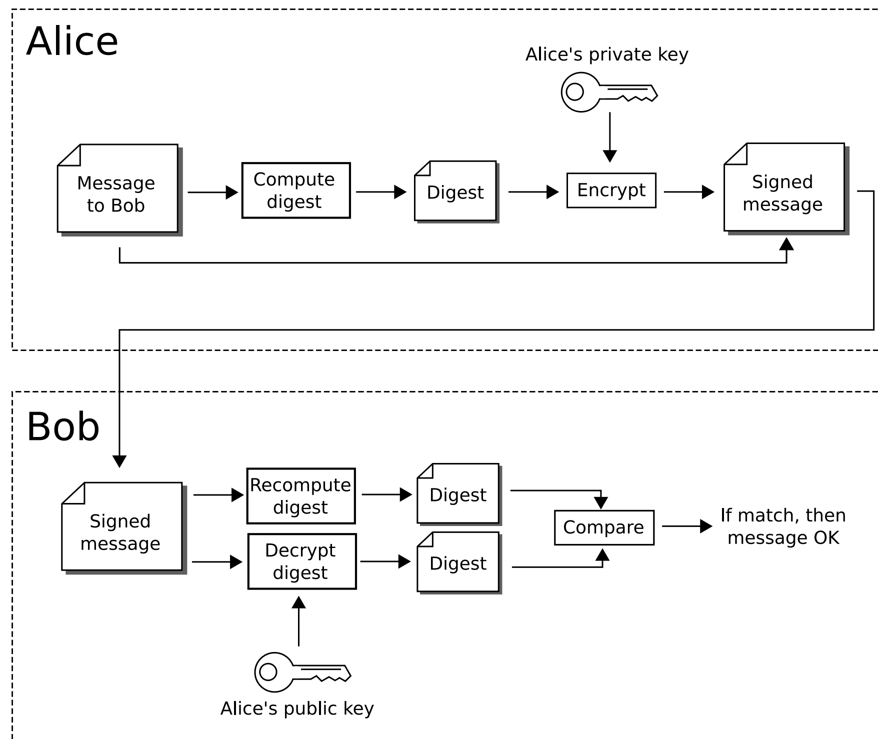

Figure 3.5: Alice computes a message digest of the message to Bob, and encrypts the digest using her private key. The encrypted digest is added to the message and sent to Bob. Bob recomputes the digest of the message, and uses Alice's public key to decrypt the digest that was contained in the message. If the two digests match, then the message has not been tampered with.

XML Signature is a standard that allows a digital signature to be expressed using XML. Similar to XML Encryption, XML Signature is bound to SOAP using WS-Security. The XML Signature structure is placed inside the `Security` element of the SOAP header. Listing 3.8 shows a simplified SOAP message where the body of the message has been signed.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <wsse:Security xmlns:wsse="../oasis-200401-wss-wssecurity-secext-1.0.xsd">

      <wsse:BinarySecurityToken wsu:Id="CertId-338156" ValueType="...#X509v3">
        ...
      </wsse:BinarySecurityToken>

      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">

        <ds:SignedInfo ...>
          <ds:Reference URI="#id-13033964">
            ...
            <ds:DigestValue ...>
              6429WQ5aLlydSX9COCHdChqgttY=
            </ds:DigestValue>
          </ds:Reference>
          ...
```

```
      </ds:SignedInfo>

      <ds:SignatureValue ...>
        ...
      </ds:SignatureValue>

      <ds:KeyInfo ...>
        <wsse:SecurityTokenReference ...>
          <wsse:Reference URI="#CertId-338156" />
        </wsse:SecurityTokenReference>
      </ds:KeyInfo>

    </ds:Signature>
  </wsse:Security>
 </soap:Header>
 <soap:Body wsu:Id="id-13033964">
   ...
 </soap:Body>
</soap:Envelope>
```

Listing 3.8: A SOAP message containing a WS-Security header with a signature.

This example might be a bit complex, but I will try to explain the different elements and how they relate to each other. The `Signature` element has three sub-elements:

- `SignedInfo` is the element that is actually signed [20]. This element contains the digest value of the SOAP body. We can see that `SignedInfo` refers to the `Id` of the SOAP body in the `URI` attribute.
- `SignatureValue` contains the value of the signature [20]. The signature is an encrypted digest of what is signed (`SignedInfo`).
- `KeyInfo` contains information about the key that can be used for verifying the signature [20]. In this case, we simply refer to a `BinarySecurityToken` by specifying its `Id` as the value of the `URI` attribute.

The `BinarySecurityToken` contains the public key certificate of the sender [20]. If the issuer of this certificate is trusted by the receiver, and the encrypted and recomputed digests match, the receiver can be certain that the message is intact and originates from the subject in question [40].

3.3.4 Security Assertion Markup Language (SAML)

I have already provided an example of authentication using usernames and passwords in WS-Security. By adding a `UsernameToken` to the `Security` header, usernames and passwords can be provided for authentication with a service. Now imagine that multiple services exist in a SOA, and that each service wants to authenticate and authorize its consumers. It is still possible to supply usernames and passwords when calling each service, but as more and more services are added to the architecture, authentication and authorization in each service might become difficult to manage. A subtle change in the user database(s)

or new requirements for methods of authentication might lead to changes in multiple services.

To address the challenge of deploying, managing, and evolving security enforcement mechanisms across a large number of services, Kanneganti and Chodavarapu [20] suggest using a dedicated security service for authentication and authorization. They argue that a security service can be centrally managed and quickly modified to meet rapidly changing business needs. Figure 3.6 illustrates this concept.

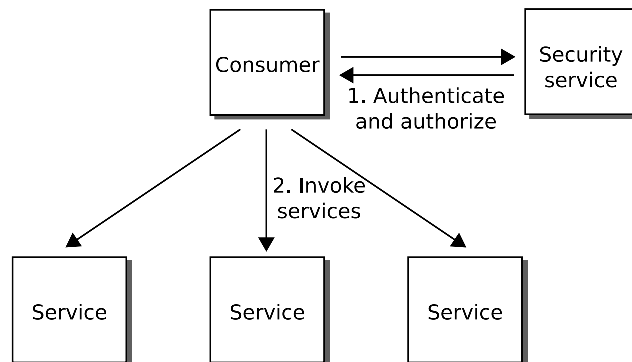

Figure 3.6: Before using regular services, the consumer sends security credentials to a security service that performs authentication and authorization. The security service returns some information back to the consumer. The consumer can use this information to prove his or her identity and authorizations when invoking other services.

As illustrated in figure 3.6, the security service returns some information that the consumer can use for proving his or her identity and authorizations when invoking other services. This information can be expressed using SAML (Security Assertion Markup Language) [20].

SAML is an XML-based language for management and exchange of security information between different systems [19]. Using SAML, one party can assert security information about a subject. This subject can for example be a user, system or company. In addition to the identity of the subject, other attributes such as permissions and email addresses can also be asserted [15]. Security information about a subject is communicated using SAML assertions. Listing 3.9 shows an example of such an assertion.

```
<saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
    AssertionID="_42da98fa7244b6ecba8c17f4cebd70c9"
    IssueInstant="2008-05-19T08:40:22.414Z"
    Issuer="http://rodem.no/securityservice/"
    MajorVersion="1" MinorVersion="1">
  <saml:AuthenticationStatement
    AuthenticationInstant="2008-05-19T08:40:22.363Z"
    AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:password">
    <saml:Subject>
      <saml:NameIdentifier NameQualifier="rodem.no">
        magrode
      </saml:NameIdentifier>
    </saml:Subject>
```

```
  ...
  </saml:AuthenticationStatement>
</saml:Assertion>
```

Listing 3.9: A SAML assertion containing an `AuthenticationStatement`.

The assertion in listing 3.9 contains an `AuthenticationStatement`, indicating that the user *magrode* has been authenticated. The attribute values of the `Assertion` element show the specific time when the assertion was issued, the URI of the assertion issuer and SAML version that is being used. In addition, the `AuthenticationMethod` attribute indicates that a password was used for authentication with the security service. The `Assertion` element may be inserted into the WS-Security header in the same way as a `UsernameToken`. Much more information can be added to SAML assertions. Refer to the OASIS SAML Token Profile [15] for a more extensive description of assertions.

SAML assertions may be generated and exchanged using a variety of protocols [26]. Examples are WS-Trust, which is briefly described in the next section, and the SAML protocol. Both these standards propose a request-response protocol for handling assertions. If only issuing, validating, renewing, and canceling SAML assertions is required, the SAML protocol is much simpler than WS-Trust [20].

The following simple example, inspired by Kanneganti and Chodavarapu [20], illustrates a SOAP message querying a security service to see if a subject is permitted to execute a resource.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <wsse:Security xmlns:wsse="../oasis-200401-wss-wssecurity-secext-1.0.xsd">
      ...
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <samlp:Request xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol">
      <samlp:AuthorizationDecisionQuery>
        <saml:Subject>
          <saml:NameIdentifier NameQualifier="rodem.no">
            magrode
          </saml:NameIdentifier>
        </saml:Subject>
      </samlp:AuthorizationDecisionQuery>
    </samlp:Request>
  </soap:Body>
</soap:Envelope>
```

Listing 3.10: Example of a request using the SAML protocol over SOAP. This example was inspired by Kanneganti and Chodavarapu [20].

The request in listing 3.10 contains an `AuthorizationDecisionQuery`, asking for authorization details for the user *magrode*. The consumer of the security service submits his credentials using the WS-Security header entry. The security service responds with the following SOAP message.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    ...
  </soap:Header>
  <soap:Body>
    <samlp:Response xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol">
      <samlp:Status>
        <samlp:StatusCode Value="samlp:Success"/>
      </samlp:Status>

      <saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion">
        <saml:AuthorizationDecisionStatement
          Resource="http://rodem.no/someservice/someOperation"
          Decision="Permit">

          <saml:Subject>
            <saml:NameIdentifier NameQualifier="rodem.no">
              magrode
            </saml:NameIdentifier>
          </saml:Subject>

          <saml:Action>
            Execute
          </saml:Action>
        </saml:AuthorizationDecisionQuery>
      </saml:Assertion>
    </samlp:Request>
  </soap:Body>
</soap:Envelope>
```

Listing 3.11: Example of a response from a security service that supports the SAML protocol. This example was inspired by Kanneganti and Chodavarapu [20].

Listing 3.11 shows that the security service responds with a `StatusCode` indicating whether the request succeeded or failed. The actual `Assertion` states that the user *magrode* is permitted to execute the specific resource. Refer to the SAML specification [26] for a more thorough description of the SAML protocol.

3.3.5 Other Security Standards

In addition to the standards already described in this chapter, some other important Web Services standards related to security deserves to be mentioned:

- **WS-Trust**
  Enables issuing, renewal, and validation of security tokens. WS-Trust describes a security token service (STS) that issues security tokens. The interface for the STS is designed to meet the requirements of a wide variety of situations, and supports multiple types of security tokens such as SAML assertions, Kerberos tickets or keys. [20]

- **WS-SecureConversation**
  Provides for the establishment and maintenance of a security context

across different messages in a conversation. The advantage of this is that keys used for encryption/decryption and signing/verification need to be exchanged only once for each conversation between a consumer and a service. [20]

- **WS-Policy and WS-SecurityPolicy**
  Provides a comprehensive framework for promoting interoperability among Web Services implementations. WS-Policy provides the basic syntax for expressing policies, which in this case are machine-readable expressions of what is required in a message exchange. WS-SecurityPolicy provides security-related policy assertions that can be used in WS-Policy documents. These assertions can for example state that all messages sent to the service must contain a username and password, or be encrypted using a specific algorithm [20].

- **WS-Addressing**
  Not exactly a security standard, but may be used for routing messages through intermediaries. This can be useful when a security service is used as an intermediary for performing authentication/authorization or validating data. WS-Addressing preserves addressing information of messages, which means that an intermediary security service is able to determine where a message should go next after having processed it. [20].

3.4 Summary

This chapter has defined what Web Services are, and described the standards used by such services. Web Services are the most popular way of implementing SOA, and makes extensive use of XML-based standards for communication.

Fundamental Web Service standards such as XML, WSDL, and SOAP were described in section 3.2. XML plays a major part in Web Services, as it is the common denominator format for data exchange. WSDL is used for describing the interface of a Web Service, and SOAP is used for exchanging messages over a lower-level transport such as HTTP.

The main security standards related to XML and Web Services were described in section 3.3. WS-Security defines a SOAP header for making security claims and ensuring message-level security. Other standards such as XML Encryption, XML Signature, and SAML make use of this header for adding security information to to SOAP messages. XML Encryption enables encryption of parts of SOAP messages, and XML Signature allows a digital signature to be expressed using XML. Finally, SAML enables management and exchange of security information between different systems. SAML may be used when having a central security service for authentication and authorization.

Now that the basics of SOA and Web Services have been presented, we can move on to describing security threats and countermeasures that are relevant for SOA.

Chapter 4

Threats and Countermeasures

This chapter describes threats and countermeasures that may apply to SOA. Before diving into this material, let us clearly define the meaning of these these concepts. According to McGraw [28] a *threat*, within information security, is the danger posed by a malicious agent for a variety of motivations. For a threat to be effective, it must act against a vulnerability in the system. A *vulnerability* is a defect or weakness in the system that can be used to break the systems security or violate security policy [28]. Following this, a countermeasure is a way of protecting the confidentiality, integrity, and availability of the system and its information [28]. Threats, vulnerabilities and countermeasures are closely related terms. A countermeasure can remove one or more threats, thus preventing the exploitation of one or more vulnerabilities.

For each threat presented in this chapter, I will describe countermeasures that can be applied. While this chapter describes the most common threats and countermeasures in relation to SOA, there is no hiding the fact that other issues may exist, and new issues will arise over time. Because of this, it is crucial that SOA developers stay up to date on the latest security issues. Good sources of information include mailing lists such as Bugtraq [34], CERTs[1], and of course articles and books.

4.1 SQL Injection

Services may use SQL (Structured Query Language) for communication with one or more databases. Some services might also build SQL queries based on input from external sources. In such cases, developers need to be aware of the dangers of SQL injection.

The principle of SQL injection is best described using an example. Consider a service that accepts an ISBN as input and returns a description of the corre-

[1] Computer Emergency Response Teams (CERT) assemble, process and provide information about vulnerabilities [33]. CERTs regularly publish advisories containing warnings about severe vulnerabilities, and also observe areas such as viruses and Trojan horses.

sponding book back to the consumer. For example, the service might receive the following call:

```
<bookInfo>
  <isbn>9788204119896</isbn>
</bookInfo>
```

The service uses the following SQL query to retrieve information about books:

```
SELECT * FROM book WHERE isbn = '" + isbn + "';
```

This query will cause no problems as long as the input actually is an ISBN. However, consider the following call:

```
<bookInfo>
  <isbn>
    0';
    DELETE FROM book;
  </isbn>
</bookInfo>
```

This results in the following SQL statements:

```
SELECT * FROM book WHERE isbn = '0';
DELETE FROM book;
```

The consumer was able to inject additional SQL code into the original query, in this case deleting all entries in the table. As illustrated by this example, an SQL injection vulnerability might leave the entire database exposed and writable to an attacker.

How can we avoid these types of attacks? The main countermeasure to SQL injection is to ensure that received data does not contain SQL metacharacters [17]. There are multiple ways of performing data validation. When filtering data, the recommended approach is to use whitelisting instead of blacklisting[2]. Besides filtering, another way of dealing with metacharacters is to use prepared statements, in which query parameters are passed separately from the SQL statement. Finally, an important countermeasure to these types of threats is to not reveal too much details regarding the implementation of the service in error messages. A potential attacker will most likely send various types of data to the service, trying to find out more about how the service is implemented. Thus, error messages or SOAP Faults should not be too verbose.

4.2 XPath Injection

Services that process XML documents using XPath might be vulnerable to XPath injection. XPath injection is based on the same principle as SQL injection, but in this case it is XPath metacharacters that might cause problems.

[2]Blacklisting means looking for bad data and blocking such data if found. When performing whitelisting however, we state the type of data that is considered harmless and block the rest. Whitelisting is preferred in a security context as it also blocks unknown types of data [17].

As almost all implementations of SOA involve a lot of XML processing [20], XPath injections are especially relevant for SOA.

Let us look at an example, inspired by Robi [35], that illustrates this type of threat. A service performs authentication by comparing credentials supplied by a consumer with user information stored in an XML document. This XML document contains user entries on the following form:

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <user>
    <firstname>Magne</firstname>
    <lastname>Rodem</lastname>
    <loginID>magrode</loginID>
    <password>password</password>
  </user>
  ...
</users>
```

When receiving a message from a consumer, the service extracts a username and password of the consumer. When using SOAP and WS-Security, the message header might look like this:

```
<soap:Header>
  <wsse:Security xmlns:wsse="...">
    <wsse:UsernameToken>
      <wsse:Username>magrode</wsse:Username>
      <wsse:Password>password</wsse:Password>
    </wsse:UsernameToken>
  </wsse:Security>
</soap:Header>
```

Before the service processes the body of this message, it uses the following XPath statement to locate the user in question:

```
//users/user[loginID/text()='" + username + "' and
             password/text()='" + password + "']
```

This statement goes through the entire XML document looking for user entries that have the desired `loginID` and `password`. If the service receives the username and password as shown in the header above, this will work without problems. The user information is found and access is granted. However, consider the following username and password:

```
<wsse:Username>' or 1=1</wsse:Username>
<wsse:Password>' or 1=1</wsse:Password>
```

This will result in an XPath statement that looks a little different than what was originally intended:

```
//users/user[loginID/text()='' or 1=1 and password/text()='' or 1=1]
```

Now, every user entry in the XML document will be returned as this statement will be true for all users (`1=1`). The user is then granted access as one or more corresponding user entries were found in the XML document.

Depending on the implementation, this type of attack yields a lot of possibilities for persons of malicious intent. If XPath is used in the retrieval of static data, the attacker might be able to read the entire XML document. Similarly, if an XSL transformation[3] is performed somewhere after reading data from external sources, an attacker could inject XPath statements to modify the data during transformation.

As with SQL injection, the main countermeasure to this type of attack is to ensure that received data does not contain metacharacters [35]. For data validation the same principles as for SQL injection (section 4.1) holds. XPath does not support the concept of parametrized queries, but similar functionality can be achieved by using other APIs such as XQuery. Refer to Robi [35] for an example of parametrized queries with XQuery. As always, error messages and SOAP Faults should be kept as brief as possible to not reveal too much implementation details of the service.

4.3 Buffer Overflow Attacks

The threat of buffer overflow attacks in SOA is very much the same as for other distributed software systems. In 2000, buffer overflows accounted for almost 45% of all security problems reported to CERT [47], and it is probably still the most commonly exploited vulnerability today [20].

In lower-level languages such as C and C++, memory handling is left to the developers. If a developer wants to use a part of the memory, he or she has to allocate a certain amount of bytes manually. When writing to memory, the size of the written data might be larger than the size of the allocated buffer. In such cases, data will be written outside the buffer, which might have serious consequences. For a more in-depth description of buffer overflow problems, refer to Wagner et al. [47].

If a service does not make sure that the size of received data fits in its allocated buffers, it may be vulnerable to buffer overflow attacks. For example, by sending more data than the service expects, an attacker might be able to execute arbitrary code. In addition, buffer overflow vulnerabilities can exist in underlying APIs or in the web/application server hosting the service.

While buffer overflow attacks are generally a significant threat, distributed systems such as SOA has an advantage over traditional applications. Similar to web applications [12], services benefit from the fact that an attacker is not able to analyze the application source code or binaries. It is much more difficult to perform a buffer overflow attack without knowledge of the service implementation.

The main countermeasure to buffer overflow attacks is data validation. Care should be taken to ensure that received data fits in its designated buffers. As

[3] eXtensible Stylesheet Language (XSL) is used for transforming XML documents into other XML documents [9].

buffer overflow problems can be exploited on the system level (i.e. starting new processes or accessing the file system), it is important not to give the service application more privileges than it really needs [20]. The user account running the service process should for instance not have administrative privileges. On UNIX, an additional way of limiting the damage is to run the service in a `chroot` jail[4].

The best way of countering the threat of buffer overflow attacks is however to build services using type-safe languages such as Java and C#. These languages do not suffer from this problem [28].

4.4 Entity Recursion Attacks

Almost all SOA implementations involve a lot of XML processing [20], and XML brings some new types of potential vulnerabilities. One type of threat that is specific to XML-based services is entity recursion attacks, also known as XML bombs.

XML supports declaration of entities within a Document Type Definition (DTD). Entities are much similar to constants in a program. An entity can be defined in the following way:

```
<!ENTITY entityName "entityValue">
```

This entity can then be referenced in the XML document as `&entityName;`. Consider the following DTD, adopted from Gallagher [13]:

```
<!ENTITY x0 "Foobar">
<!ENTITY x1 "&x0;&x0;">
<!ENTITY x2 "&x1;&x1;">
<!ENTITY x3 "&x2;&x2;">
<!ENTITY x4 "&x3;&x3;">
...
<!ENTITY x100 "&x99;&x99;">
```

If `&x100;` is used somewhere in the XML document, it will first be replaced by `&x99;&x99;`, which is further replaced by `&x98;&x98;&x98;&x98;` and so on. This replacement chain continues until the replacement string eventually becomes ”Foobar” repeated 2^{100} times. An attacker can easily generate the DTD listed above, but the service receiving and processing the XML document will eventually run out of memory. This type of attack may result in denial of service (DoS), making the service unavailable for legitimate users.

Web Services using SOAP are vulnerable to this type of attack if the SOAP implementation allows DTD within a SOAP message. The SOAP specification prohibits the presence of DTD in a SOAP message [20]. Regardless of this, many vendors (including IBM and Microsoft) have previously overlooked the

[4]The `chroot` system call is used to change the root of the file system as seen by the calling process [10]. A process running in a `chroot` jail will not be able to access files outside the new root directory.

SOAP specifications and processed DTD anyway [30]. This goes to show that one should not blindly trust SOAP vendors in these cases.

The most important countermeasure to entity recursion attacks is to make sure that the SOAP implementation is compliant with the SOAP specification, and thus do not accept DTD within a SOAP message.

4.5 External Entity Attacks

As described in the previous section, XML supports declaration of entities within a Document Type Definition (DTD). In that section I also described a type of attack that exploits this feature, namely entity recursion attacks. Another type of attack that is related to entities are external entity attacks.

Almost all XML parsers allow plugging in an external entity resolver that can resolve external entity references [20]. If an external entity resolver simply resolves any external reference, this leads to new vulnerabilities. The following shows an example, adopted from Gallagher [13], that contains an external entity resolving the system file */etc/passwd*.

```
<!DOCTYPE myTest [
  <!ELEMENT foobar ANY>
  <!ENTITY xxe SYSTEM "/etc/passwd">
]>
<foobar>&xxe;</foobar>
```

If this code is allowed somewhere in a SOAP message, the contents of */etc/-passwd* is effectively added to the message. This may have all sorts of consequences. If, for example the service echoes the message, an attacker will be able to read the system password file.

An attacker could also use this feature to launch a DDoS attack (see section 4.7). This could be done by simultaneously sending multiple SOAP messages to different services, resolving one specific URI. In addition, this could be exploited by repeatedly accessing a limited system resource such as */dev/random*[5] [20].

Similar to entity recursion attacks, external entity attacks can only be executed if the SOAP implementation allows DTD within a SOAP message. If entity resolving functionality is required by the service, the external entity resolver should be limited to only resolve a limited set of known locations.

The best way to counter external entity attacks however, is to make sure that the SOAP implementation is compliant with the SOAP specification, and thus do not accept DTD within a SOAP message.

[5] /dev/random is a random number generator on UNIX based systems that can degrade performance of the system if used very frequently. [20]

4.6 Service Description Scanning Attacks

Operations and invocation patterns of a service are specified in a service description. As seen in section 3.2.2, WSDL documents are used to describe Web Services. While such documents are helpful for consumers that need to invoke operations on the service, they also may contain debugging functions or expose operations that were never meant to be called externally [1]. An attacker may use this information to identify implementation details and possible weaknesses of the service.

One obvious countermeasure to this type of attack is to ensure that service descriptions do not contain more information than necessary. Web Services frameworks support automatic generation of WSDL documents from source code [20]. It is important to check that generated WSDL documents do not contain debugging functions or other unnecessary information.

Some attackers might try to search for service descriptions using Google or other search engines. By adding `filetype:wsdl` to a Google search query, an attacker can easily find a lot of WSDL documents [1]. Further, these documents can be scanned to find debugging functions and other interesting information. To avoid having the WSDL document show up in a Google `filetype` query, the WSDL document can be published in a more subtle way. For instance, services may make WSDL documents available on the form *http://example.com/ServiceName?WSDL*.

In some cases it can be necessary to only allow WSDL access to authenticated consumers. The exact implementation of this depends on how the WSDL document is published. For example, if the document is published as a file on a web server, HTTP authentication or SSL client certificates may be used.

4.7 Distributed Denial of Service (DDoS) Attacks

When performing a denial of service (DoS) attack, an attacker tries to reduce the availability of a service, thus attempting to stop legitimate users from operating the service. Some ways of doing this in a SOA are based on entity recursion attacks and external entity attacks as described in previous sections. However, there is another form of DoS attack that might be more difficult to counter, namely the distributed denial of service (DDoS) attack.

In a DDoS attack, services are hit with a flood of concurrent requests from multiple points in the network [20]. When hit by a powerful DDoS attack, a service would most likely crumble when trying to process all requests.

To cope with DDoS attacks, one needs to be able to quickly distinguish between legitimate and bad requests. This should be done as early as possible, and at multiple access points to the network [20]. Every access point to the network

should use a firewall to filter good traffic from bad. Figure 4.1, adopted from Kanneganti and Chodavarapu [20] illustrates this strategy.

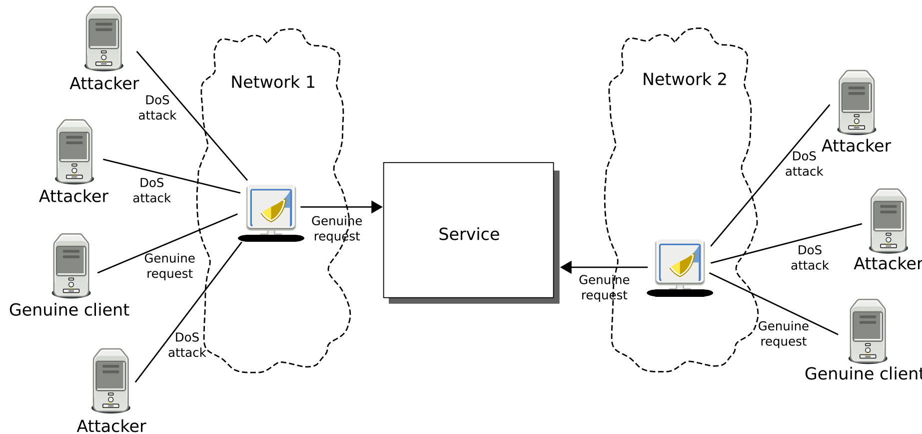

Figure 4.1: A strategy for coping with DDoS attacks. By filtering traffic generated by DoS attackers early and at multiple access points, DDoS attack on the service can be avoided. Adopted from Kanneganti and Chodavarapu [20].

In figure 4.1, multiple attackers send a flood of requests to the service. The service can be reached through two separate networks, which means that the access points of each network need to be equipped with firewalls so that every service request is filtered. This ensures that the service is able to process genuine requests without being affected by the many DoS attacks. Another strategy for handling DDoS attacks is to use load balancing for spreading the workload between multiple servers [38]. Load balancing can mitigate DDoS attacks, as multiple servers are able to handle more requests than just one single server.

Andrews and Whittaker [1] suggest using intrusion detection systems (IDS) or bandwidth management solutions for detecting DDoS attacks. When an unusual amount of activity are detected towards one host in the network, an IDS can trigger firewall updates to block this activity. However, this may be counterproductive. Attackers can launch attacks identifying themselves as originating from legitimate users (e.g. large ISP proxies), thus blocking these users from the service. Because of this, any countermeasure to DDoS attacks should be carefully designed and reviewed [1].

4.8 Eavesdropping

If an attacker is located somewhere between a service and a consumer, he or she might be able to eavesdrop on transmissions. In such cases, an attacker would want to obtain information that is being sent. Two attacks related to eavesdropping are traffic analysis and release of message contents [40].

Traffic analysis is performed by observing message patterns. This can be done regardless of any encryption that may be used. By observing message patterns,

an attacker can determine the location and identity of communicating hosts, as well as observing the frequency and length of messages being exchanged [40]. By gathering this information, the attacker can guess the nature of the communication that is taking place.

In SOA, as in most cases where two parties are communicating, release of message contents is countered using encryption. When a message is sent directly between two parties, one common way of implementing confidentiality is to use encryption on the transport layer, typically with SSL/TLS. In many cases, this is also a good way of providing confidentiality in a SOA. However, when intermediate services are involved in a transmission, one has to consider if these intermediaries can be trusted. If this is not the case, SSL/TLS should not be used - at least not as the only method for providing confidentiality - as intermediaries will be able to read contents of messages.

When intermediaries cannot be trusted, confidentiality should be handled on the message level rather than on the transport level. This means that the message itself should be encrypted instead of solely relying on a secure connection such as SSL/TLS [19]. How this is handled depends on the SOA implementation. Currently the most prominent example is the WS-Security standard used by Web Services. WS-Security and XML Encryption were described in section 3.3.

4.9 Tampering Data

It is a common misconception that message integrity is a side benefit that is automatically achieved when using encryption. However, in reality an encrypted message may be subtly changed without impacting its ability to be decrypted [27].

I have already described how SOA differs from traditional architectures when it comes to confidentiality. When intermediate services are used in a transmission, it is no longer sufficient to use a secure connection, such as SSL/TLS, between endpoints. Intermediaries will be able to read the message contents, which may not be desirable in all cases. This also affects how integrity is achieved in a SOA.

Detecting that data has been tampered with can be done on the transport or message level. Similar to confidentiality, integrity can be handled on the transport level, often by using SSL/TLS, if no intermediate services are involved in a transmission. However, when there are one or more untrusted intermediaries between the two endpoints, integrity must be handled on the message level. One way of countering data tampering on the message level is to use digital signatures. When a message has been signed using the consumer's private key, the service can verify the integrity of the message by using the consumer's public key.

How this is done depends on the SOA implementation. A SOA built using Web Services can benefit from the WS-Security standard and XML Signature. This

was described in section 3.3.

4.10 Replay Attacks

One might assume that a message that has been properly encrypted and signed cannot be used by an attacker. This is not true. Even though a message has been encrypted and signed, it can still be used in a replay attack [27]. Replay attacks are performed by capturing a valid message and resending it. Figure 4.2 shows how an attacker can perform a replay attack.

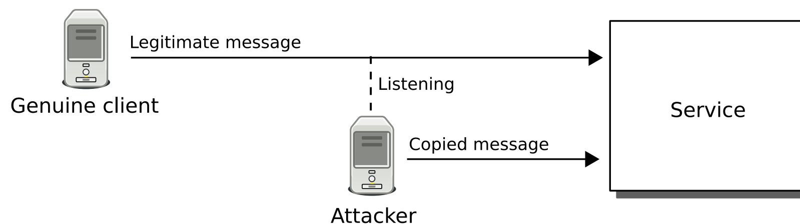

Figure 4.2: An attacker listening to the communication between a consumer and a service can capture a message and resend it.

Replay attacks are often performed to gain unauthorized access. If an attacker is able to capture a message that requests some kind of sensitive data from the service, he or she may be able to resend the message along with the original user's security credentials. As the message contains legitimate security credentials, the service will authorize the request and return the the sensitive data back to the attacker.

This attack is relevant for any type of service. One countermeasure is to sign some information that shows that the message actually was created by the consumer in question at that specific time. This information can be a timestamp or a nonce[6] When using timestamps, a replayed message will include the same timestamp as the original message [27]. The service will then be able to compare the timestamp in a received message with its local time, and discard the message if the timestamp has expired. It is important to carefully decide the timestamp trust interval. An attacker should not have time to capture and replay a message, yet the interval must be long enough to ensure that time differences between hosts do not result in legitimate messages being blocked [27].

A practical way of countering this threat exists for Web Services. As mentioned in section 3.3.3, SOAP messages can be signed using XML Signature. WS-Security includes support for timestamps. A timestamp can be specified in the SOAP header, below the WS-Security element:

```
<wsse:Security xmlns:wsse="..." xmlns:wsu="...">
  ...
  <wsu:Timestamp wsu:Id="as820d72sj3">
    <wsu:Created>2008-03-23T20:58:13Z"</wsu:Created>
```

[6] A nonce is an identifier or number that is used only once [40].

```
    <wsu:Expires>2008-03-23T21:00:13Z"</wsu:Created>
  </wsu:Timestamp>
  ...
</wsse:Security>
```

Further, this timestamp can be signed using XML Signature by referring to the timestamp Id:

```
<wsse:Security xmlns:wsse="..." xmlns:wsu="...">
  ...
  <dsig:Signature xmlns:dsig="...">
    <dsig:SignedInfo>
      ...
      <dsig:Reference URI="#as820d72sj3">...</dsig:Reference>
    </dsig:SignedInfo>
    ...
  </dsig:Signature>
</wsse:Security>
```

The above example is very simplified, but shows the principle of signing timestamps using XML Signature. Now, let us say that an attacker was able to capture a SOAP message containing the elements above. When the service receives the copied message from the attacker, it will either recognize that the message has already been received, or it will discover that the message has expired [27].

Another way of countering replay attacks is to secure the transport layer using SSL/TLS. SSL/TLS automatically protects against replay attacks by including an implicit sequence number [48].

4.11 Repudiation

In cases where messages are exchanged, situations may arise where one of the parties claim that a message has not been sent or received. If either the sender or receiver of the message cannot prove that the message exchange has taken place, they are vulnerable to repudiation attacks [40].

One common way of ensuring non-repudiation is to use public key cryptography to sign all messages, and keep records of all message exchanges [42]. In practice, this means that a sender signs outgoing messages with a private key, and then the receiver can verify the signature using the public key of the sender.

The strategy for ensuring non-repudiation in a SOA is very much similar to that of traditional system architectures [20]. Consumers and services may sign and verify messages using public key cryptography. Because of this, I will not go into detail on how to ensure non-repudiation. However, the actual implementation may vary. A SOA implemented using Web Services can make use of WS-Security and XML Signature (both described in section 3.3) for signing messages.

4.12 Summary

In this chapter, I have presented threats and countermeasures that are relevant for service-oriented architectures. Table 4.1 shows a summary of all threats and their corresponding countermeasures.

Threat	**Countermeasures**
SQL injection	Perform data validation. Use prepared statements.
XPath injection	Perform data validation. Use XQuery to build parametrized queries.
Buffer overflow attacks	Perform data validation. Limit system privileges. Use a type-safe language such as Java or C#.
Entity recursion attacks	Do not process DTD found within messages.
External entity attacks	Do not process DTD found within messages. Limit external entity resolving to known locations.
Service description scanning attacks	Ensure that service descriptions does not contain unnecessary information. Require authentication when accessing descriptions. Hide descriptions from search engines.
DDoS attacks	Use firewalls to filter traffic. Use intrusion detection systems to trigger firewall updates. Use load balancing.
Eavesdropping	Use a secure encrypted connection (SSL/TLS). Encrypt the actual messages if untrusted intermediaries are used.
Tampering data	Use a secure encrypted connection (SSL/TLS). Use digital signatures to sign messages if untrusted intermediaries are used.
Replay attacks	Use a secure encrypted connection (SSL/TLS). Sign timestamp or nonce.
Repudiation	Sign all messages and keep records of all message exchanges.

Table 4.1: Overview of threats and countermeasures.

As can be seen from this chapter, many of the threats that are present in traditional systems are also relevant in SOA. SQL injection and buffer overflow attacks are common in web applications [1]. In addition, eavesdropping, data tampering, DoS/DDoS, replay attack, and repudiation are well-known threats in network security [40]. Even though many of the threats are present in both traditional systems and SOA, some countermeasures differ. In specific, SSL/TLS may no longer be enough for countering eavesdropping, data tampering, and replay attacks. Message-level security need to be used to counter these threats when untrusted intermediaries are used.

Some threats are more relevant in SOA than in traditional systems. Most notably, service description scanning attacks are specific to SOA. XPath injection, entity recursion attacks, and external entity attacks exploit vulnerabilities related to XML processing. As SOA involves a lot of XML processing [20], these threats are especially relevant in SOA. Table 4.2 shows an overview of the threats that are specific to SOA, have different countermeasures, or are especially relevant in SOA compared to traditional systems.

Threat	**Specific**	**Diff. count.**	**Esp. rel.**
SQL injection	-	-	-
XPath injection	-	-	✓
Buffer overflow attacks	-	-	-
Entity recursion attacks	-	-	✓
External entity attacks	-	-	✓
Service description scanning attacks	✓	-	-
DDoS attacks	-	-	-
Eavesdropping	-	✓	-
Tampering data	-	✓	-
Replay attacks	-	✓	-
Repudiation	-	-	-

Table 4.2: Overview of the threats that are specific to SOA, have different countermeasures in SOA, or are especially relevant in SOA compared to traditional systems.

Now that the security challenges, standards, threats, and countermeasures in SOA have been presented, we are ready to identify some security guidelines for development of SOA.

Chapter 5

Security Guidelines

A guideline is a recommendation for things *to do* or *to avoid* [28]. In software development, guidelines exist for specific technical contexts, such as web applications, distributed systems, database connectivity, and so on. Guidelines are best enforced through human analysis, and should be kept in mind throughout the entire software development process [28].

In this chapter, I will present high-level security guidelines for software development. Some security guidelines should be followed regardless of the type of system that is being built. I will briefly describe these guidelines in the next section. In addition, I have provided a set of guidelines that are specific to security in service-oriented architectures. These SOA-specific guidelines should be applied during requirements analysis and design, and forms the basis of my contribution in this thesis.

I will make practical use of the guidelines presented in this chapter when identifying requirements and designing a secure service-oriented architecture.

5.1 General Security Guidelines

In software development there are some security guidelines that always should be kept in mind. Viega and McGraw [45] present 10 general design-level guidelines for developing secure software systems. I will quickly reiterate these guidelines here:

- **GG1: Secure the weakest link**
  A system is only as secure as its weakest link. An attacker will always try to attack the weakest part of a system, as this is what requires the least amount of effort. Because of this, extra work should be done to improve the security of the weakest part of a system.

- **GG2: Practice defense in depth**
  A system that is built with defense in depth is more difficult to attack.

The reason for this is that an attacker has to go through multiple defense layers to succeed. Systems should not have a single point of failure.

- **GG3: Fail securely**
  When a system fails, it should do so in a secure manner. In particular, bad handling of unexpected errors may lead to security breaches. For example, a system that prints too detailed error messages back to the user might reveal sensitive information about the system or its data.

- **GG4: Follow the principle of least privilege**
  Only the minimum access necessary to perform operations should be given. In addition, access should be granted for a minimum amount of time. This ensures that users are not able to do things other than what was intended. Also, if an attacker is able to compromise a user account, he or she will only be able to do minimal damage.

- **GG5: Compartmentalize**
  A system should be broken into as many isolated units as possible. Then, an attacker that has compromised one unit will only be able to take advantage of that single unit. This can also increase the simplicity of the system. An example of compartmentalization can be to use different machines for different tasks.

- **GG6: Keep it simple**
  By keeping a system as simple as possible, the code becomes easier to maintain. Complex code is harder to review, and tend to have more bugs [28]. Components should be reused when possible.

- **GG7: Promote privacy**
  The privacy of users should naturally be ensured, but the privacy of systems must also be handled with care. A system might give away details about themselves that can be used by an attacker. System information that should be kept out of the hands of attackers include operating system version, database structure, and file system details.

- **GG8: Remember that hiding secrets is hard**
  Security by obscurity should not be relied upon. Hard coding secrets in code or hiding passwords in clear text somewhere is bad practice. One should always assume that attackers are able to see the source code or reverse engineer binaries.

- **GG9: Be reluctant to trust**
  Developers should assume that the environment where their system resides is insecure. Trust can be given to external systems, code, people, etc. In any case, trust should always be closely held and never loosely given [7]. User input should not be trusted, and must be validated before being used. Also, the security of a system can be increased by minimizing the trust in other systems.

- **GG10: Use community resources**
  It is good practice to use community resources instead of designing a

home-brew security solution. Community resources are often tried and tested, and developed by security experts. By using community resources that are known to be secure, the likelihood of avoiding known mistakes can be reduced.

5.2 SOA Security Guidelines

Service-oriented architectures are a bit different from traditional software systems. As described in previous chapters, this results in new approaches to security. In this section, I will summarize the main security aspects of SOA, and use this knowledge to create a set of guidelines that should be applied during requirements engineering and design of service-oriented architectures. These guidelines should be considered in addition to the 10 general guidelines described in the previous section.

First, let us recap on the main security aspects of SOA that have been presented in this report:

- **Aspect 1: The interoperability concern**
  As mentioned in section 2.4, services should be kept as open and easy to use as possible when choosing approaches for handling security. In section 3.3, several security standards for SOA, or more specifically Web Services, were presented. These standards are open and widely accepted, and lead to standardized handling of security.

- **Aspect 2: Different levels of security**
  In section 2.4, the problem with transport-level security and the use of intermediaries were presented. Three threats in chapter 4, namely eavesdropping, data tampering, and replay attacks, also mentioned this, and message-level security was suggested as a countermeasure. However, transport-level security has better performance, and is often more widely supported [19].

- **Aspect 3: Security may complicate business operations**
  Adding security information to the body of messages is bad practice, as described in section 3.3.1. Then, security information would need to be handled as part of the business logic, as the message body is processed by business operations. Not only does this result in different ways of expressing security information, but it also leads to increased complexity of business operations.

- **Aspect 4: Multiple entrances to applications**
  SOA leads to multiple entrances to applications, as functionality is made available as services. As mentioned in section 2.4, this introduces new challenges with respect to security functionality such as authentication and authorization. SAML (described in section 3.3.4) helps in this respect by enabling centralized authentication and authorization. A central

security facility eases maintenance and allows for uniform handling of security.

- **Aspect 5: Keeping track of events can be difficult**
  As described in section 2.4, due to multiple entrances to applications, logging and monitoring of a SOA is not as easy as in traditional systems. When many services are involved it can be difficult to keep track of what is going on, and what has happened. If we were able to log messages to one central location, we would be better equipped to track issues in our SOA [43].

When looking at this list of aspects, it becomes clear that certain recommendations can be made for things to do, and things to avoid during requirements engineering and design of a SOA. I will use this list of aspects as a basis for my SOA security guidelines.

One could argue that I could have included more aspects in this list. For instance, I have not mentioned the threats of service description scanning, XPath injection, entity recursion, and external entity attacks, which were identified as specific or especially relevant in chapter 4. However, service description scanning is covered by the general guideline GG7, *promote privacy*. Thus, there is no need to create a new guideline for this. The same goes for XPath injection, which is covered by guideline GG10, *be reluctant to trust*. Entity recursion and external entity attacks are mainly related to bad SOAP implementations. As Web Services and SOAP are not the only way of implementing a SOA, I do not want my guidelines to assume any specific implementation. My guidelines should be on a high level, and entity recursion and external entity attacks are a bit too specific to Web Services. Thus, I have not included these threats in the above list of aspects.

As a result of my main findings in this report, which are summarized in the above list of aspects, I propose the following five security guidelines for SOA:

- **SG1: Retain interoperability when handling security**
  When handling security in a SOA, it is important to retain the philosophy of keeping services as open and easy to use as possible. Interoperability should not suffer because of security, and this means that open and widely accepted security standards should be chosen when realizing a service-oriented architecture. Examples of such standards are those defined by OASIS and W3C. When building a SOA using Web Services, one should especially focus on compliance with the WS-I Basic Security Profile [29], which promotes interoperability between different implementations of standards.

- **SG2: Assess the need for message-level security**
  Messages that are exchanged in a SOA might travel through untrusted intermediary services. In such cases, transport-level security does not suffice, as the untrusted intermediaries can eavesdrop on, or tamper with, messages. To ensure confidentiality and integrity in these cases, encryption and signing of data should be handled on the message level [21].

However, in more simple cases, where messages only travel between two nodes or through trusted intermediaries, transport-level security might be sufficient [20]. Transport-level security has better performance, and is often more widely supported than message-level security [19]. When developing a SOA, one should consider relevant requirements and threats in detail, and decide whether or not message-level security is a necessity.

- **SG3: Handle security outside of business operations**
  Separation of concerns is a core principle in SOA. This principle should not only be applied at the architectural level, but also on the implementation level [24]. This means that concerns such as security should be handled outside of the business logic of the service. This reduces the complexity of business operations, increases modifiability, and enables the possibility of independent implementation of security by experts. An example of this can be seen in Web Services using WS-Security, where security credentials can be placed in the header of SOAP messages and checked before the business operation is invoked.

- **SG4: Centralize security**
  Multiple services often have similar security policies and mechanisms. Security functionality such as authentication and authorization are commonly present in services. When dealing with a large number of services, it can be hard to keep the security enforcements of all services consistent and up to date. In such cases, security is best handled at a central location. This can be done in several ways. Examples include using the same user database for authentication in all services, adding an intermediary facility for security processing between the consumers and the services, or using a dedicated security service.

- **SG5: Centralize logging**
  If an attacker is somehow able to cause harm to one or more services in a SOA, an administrator should be able to detect and grasp the details of the malicious event as quickly as possible. The distributed nature of SOA leads to more entrances for an attacker as opposed to a large single-entrance application [20]. If logs need to be manually gathered from different services, this makes it more difficult to react in the case of an attack. In addition, an attacker that has gained system privileges might be able to tamper with logs that reside locally on the service host. Because of this, logs should preferably be stored, not only locally, but also centrally for easier monitoring and attack detection/recovery.

5.3 Summary

In this chapter I have presented a total of 15 guidelines for development of secure service-oriented architectures. The ten guidelines that are identified by Viega and McGraw [45] are applicable to all software systems. In addition to these, I have composed a set of five guidelines specific to SOA.

In the remaining chapters, I will describe development of a SOA and show how the security threats described in chapter 4 can be countered using existing standards and Java-based frameworks. During the development process I will make use of the security guidelines presented in this chapter.

Chapter 6

Requirements Engineering

Previous chapters have described the theory behind SOA and security. In the remaining of this report, I will look at a real-world case and demonstrate the development of a SOA-based system. UNINETT FAS is one of many organizations that are interested in SOA. They have provided me with a real-world case scenario that will not only allow me to illustrate the security challenges in SOA, but also to show how my guidelines can be applied in practice.

This chapter details requirements for a SOA-based system that imports personal data into multiple administrative systems. The chapter begins with a brief introduction to the system, which includes background information and a system overview. After the system has been introduced, some of the security guidelines from chapter 5 are applied. Using these guidelines, I am able to identify the services and functionality that are needed to fulfill the security requirements of the system. Following this, the requirements of the services are specified using use cases and misuse cases. The chapter ends with a summary of the guidelines that have been used during requirements engineering.

6.1 System Introduction

This section describes UNINETT FAS and their requirements for a SOA-based import mechanism.

6.1.1 Background

UNINETT FAS is responsible for buying and managing administrative systems for over 30 universities and colleges in Norway. These systems perform operations related to management of finances, documents, and projects. The systems run on different platforms and are located on servers with different physical locations and operators. As the systems are bought from different vendors and use different databases, there is an obvious need for integration.

Lately, UNINETT FAS has taken an interest in SOA and its advantages in integration. In section 2.3, some reasons for using SOA were presented. Three system characteristics that SOA is especially good at handling were described [19]: Complex distributed systems, different owners, and heterogeneity. Deciding whether or not SOA is a good approach for UNINETT FAS is out of the scope of this report. However, SOA seems to be able to handle many of the characteristics that are evident in the IT architecture at UNINETT FAS.

UNINETT FAS is, however, as many organizations, concerned about the security aspects of SOA. In the previous chapters, I have tried to address this concern by explaining the differences between SOA and traditional systems with regard to security, describing countermeasures to common threats and providing guidelines for development of secure service-oriented architectures. Now it is time to demonstrate this in practice.

UNINETT FAS has provided me with a case scenario, which involves import of personal information into multiple administrative systems. The same personal information, such as names, addresses, phone numbers etc. need to be present in multiple systems, and this should not have to be entered manually in each system. UNINETT FAS wants to populate the systems with as little human intervention as possible. The institutions (colleges and universities) should be able to send information about persons to UNINETT FAS, which automatically populates each system with the given information. UNINETT FAS wants to see how a secure infrastructure for handling this process can be designed using the SOA paradigm.

6.1.2 System Overview

UNINETT FAS would like to offer a service for importing information about persons into multiple systems. Institutions should be able to import persons by using a simple client application that communicates with the import service. The client application should read an XML document containing a list of persons, and invoke a specified service that handles the import process. In return, the client application should receive a summary describing what was done during the import process.

It is assumed that the administrative systems managed by UNINETT FAS have dedicated services that offer functionality related to person management. Quite a few vendors are beginning to expose some of their systems' capabilities as services out of the box [20]. In cases where these systems do not provide dedicated services for this purpose, such services need to be made specifically. The import service should make use of these services (henceforth called system services) when importing persons. Figure 6.1 shows how UNINETT FAS envisions the communication between the client application and the services.

The main tasks of the import service will be comparing received and currently stored persons, and transforming personal data to formats that the communicating parties are able to understand. All services send and receive data

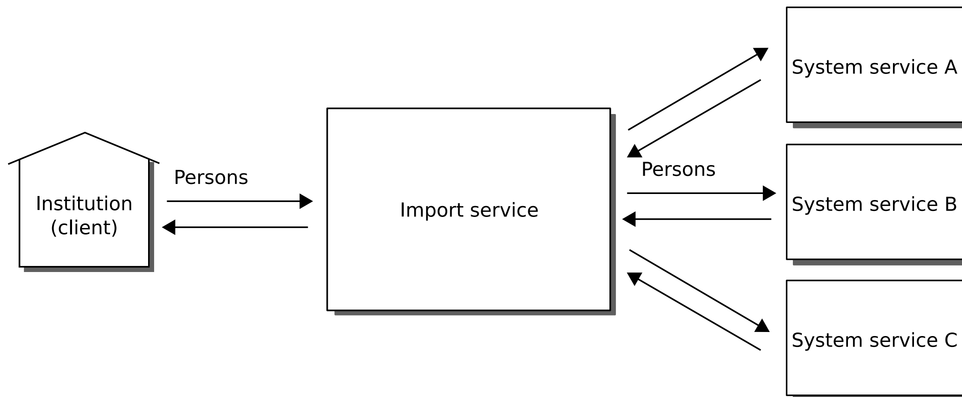

Figure 6.1: An overview of the main services involved in import of information about persons.

using XML-based technologies, however the format (XML Schema) used by the system services may differ.

Institutions must authenticate when using the functionality of the import service. This process should be made as easy as possible for the institutions, while minimizing the amount of administration required. In addition, UNINETT FAS does not rule out the possibility of offering additional services to the institutions in the future. Thus, it should be easy to add new services to the architecture. Also, UNINETT FAS wants to be able to keep track of events that occur on each service.

As the messages that are sent between institutions and the import service contain personal information, UNINETT FAS wants to keep messages confidential. Also, care should be taken to ensure that it is not possible to modify the personal information or any other data while it is transferred between institutions and the import service.

6.2 Security Challenges

Now that we have a basic overview of the system, we can start thinking about how to solve the security challenges of the SOA. In any form of software development it is important to have security in mind during the entire development process [28]. Thus, security guidelines should not be neglected in the requirements engineering phase. In this section, I will use my current knowledge of the system that is to be built, and see what security guidelines can be applied at this time.

Institutions need to authenticate when using the import service. Currently, only one service is going to be made available to the institutions. However, according to the system description, new services might be added in the future, and the process of adding such services should be as easy as possible. According to guideline SG4, *centralize security*, common security functionality such as authentication should be centralized when many services are involved. I propose using a dedicated security service for centralizing authentication in this system.

The security service should receive security credentials from a consumer, check these credentials against a list of user entries and respond with a token that serves as proof of the consumer's identity. When the consumer calls a service, it adds the token to the message, and the service is able to verify the identity of the consumer based on the token. Figure 6.2 illustrates how the security service should operate.

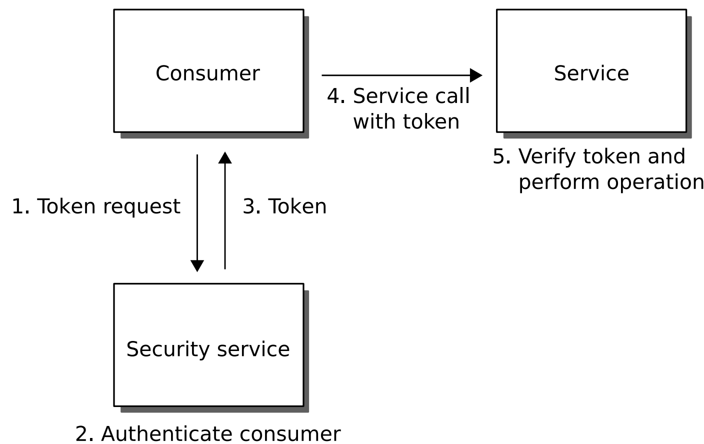

Figure 6.2: A consumer invokes the security service, requesting an identity token. The security service authenticates and authorizes the consumer and returns a token. The token can then be used when invoking other services.

According to the system description, events that occur on each service should be easily monitored. This means that all events should be logged by the services. Looking at the guidelines in section 5.2, we see that logging should be centralized according to guideline SG5, *centralize logging*. This is needed to get the full picture of what is going on in all services. A centralized logging facility is likely to be reused, and thus it should be possible to add log entries by invoking a

service. When adding the security service and a logging service to the service-oriented architecture, we end up with a set of services as shown in figure 6.3.

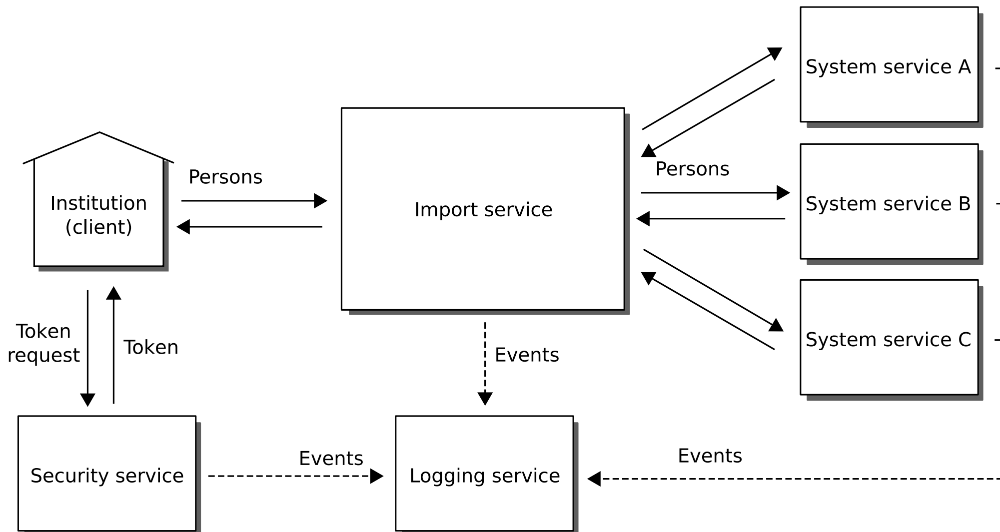

Figure 6.3: The client application authenticates with the security service, and requests an identity token. The client then invokes the import service with a list of persons, and adds the identity token to the message as proof of its identity. The import service verifies the identity token and authorizes the client. It may then invoke the system services to perform the actual import. All events are logged to the logging service.

According to SG2 in section 5.2, we should assess the need for message-level security. As the guideline says, one of the main causes for using message-level security is to protect data against untrusted intermediary services. In our case scenario, there are no untrusted intermediary services. The import service acts as an intermediary between the institutions and the system services, but we trust the import service not to eavesdrop on or tamper with messages. Because of this, transport-level security is sufficient for protecting the data flow between institutions and the system services. Transport-level security has better performance than message-level security, and is often better supported [19]. Thus, transport-level security is the best choice in our scenario.

Message-level security is however needed for identity tokens that are issued by the security service. If identity tokens are not signed, consumers can tamper with tokens before they are sent to a service. Consumers may do this to grant themselves access to operations other than what they have been authorized to perform. Because of this, the security service needs to sign all tokens that are issued, and services need to verify the signature when receiving tokens from consumers.

6.3 Requirements

This section details the requirements of the SOA-based system that is to be built. I will be using use cases, or more specifically misuse cases, for documenting the requirements of the SOA solution. Misuse cases will allow me to relate threats and countermeasures to functionality in a way that is easily comprehensible. Use cases and misuse cases are described only briefly here. Refer to Vidgen [44], and Sindre and Opdahl [37] for a more in-depth description of use cases and misuse cases.

Use cases are a common technique for specifying, documenting, and communicating requirements. A use case typically represents some functionality as perceived by the user [44]. Use cases are often documented using both a textual description and an UML diagram[1]. The use case notation comprises actors, use cases, and associations. Figure 6.4 illustrates the use case diagramming notation.

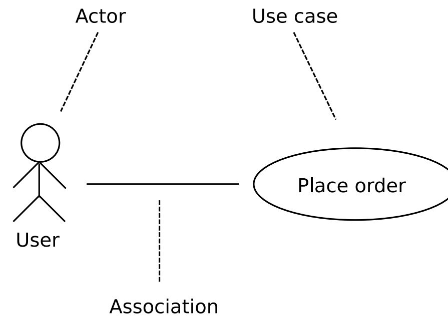

Figure 6.4: The use case diagramming notation.

While use cases are suitable for most functional requirements, they may lead to neglect of extra-functional requirements, such as security requirements [37]. For a closer integration between functional and extra-functional requirements, Sindre and Opdahl [37] describe another approach called misuse cases. Misuse cases are a systematic approach to eliciting security requirements based on use cases. In misuse cases, traditional use cases are extended to also cover misuse.

Misuse cases model both functionality and threats of a system. New concepts such as *misusers* and *misuse cases* are added to the existing UML definitions of actors and use cases. A misuse case is a sequence of actions performed by a misuser, with the intention of causing harm to a system. Countermeasures to threats posed by a misuser are modeled as use cases. Misuse cases threaten use cases, and use cases mitigate misuse cases. Figure 6.5 illustrates the misuse case diagramming notation. Note that I have omitted the threaten relation in the misuse case diagrams in this report to avoid cluttering.

In addition to misuse case diagrams I have provided lightweight textual misuse case descriptions for each service. This lightweight approach embeds the

[1] The Unified Modeling Language (UML) is a modeling notation that has gained widespread acceptance for the analysis and design of software systems [44].

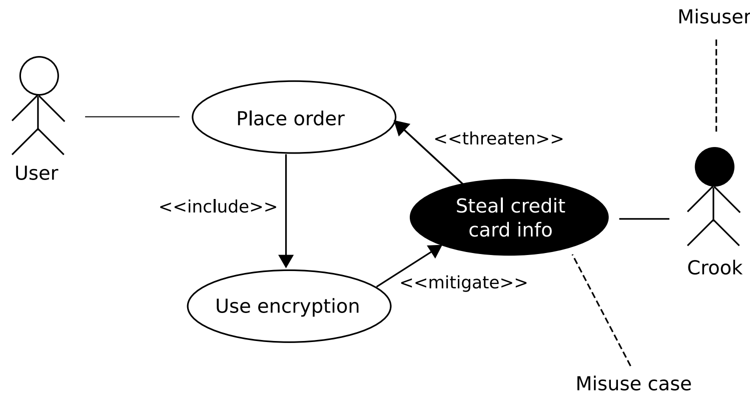

Figure 6.5: The misuse case diagramming notation.

description of misuse cases within a regular use case template [37]. These descriptions can be found in appendix A.

The following use cases are described in this section:

- **UC1: Import persons**
  A consumer invokes the import service, submitting a list of persons for import. The import service invokes one or more system services to retrieve the current lists of persons. The import service then compares the new and current list of persons, and invokes the system services to update the person registries.
- **UC2: Authenticate consumer**
  A consumer invokes the security service, requesting an identity token. The security service reads the request, authenticates the consumer, and responds with an identity token that can be used when invoking other services.
- **UC3: Add log entry**
  A consumer invokes the logging service, requesting that a log entry should be stored. The logging service logs the event and replies with a status code indicating whether the event was logged or not.

6.3.1 UC1: Import Persons

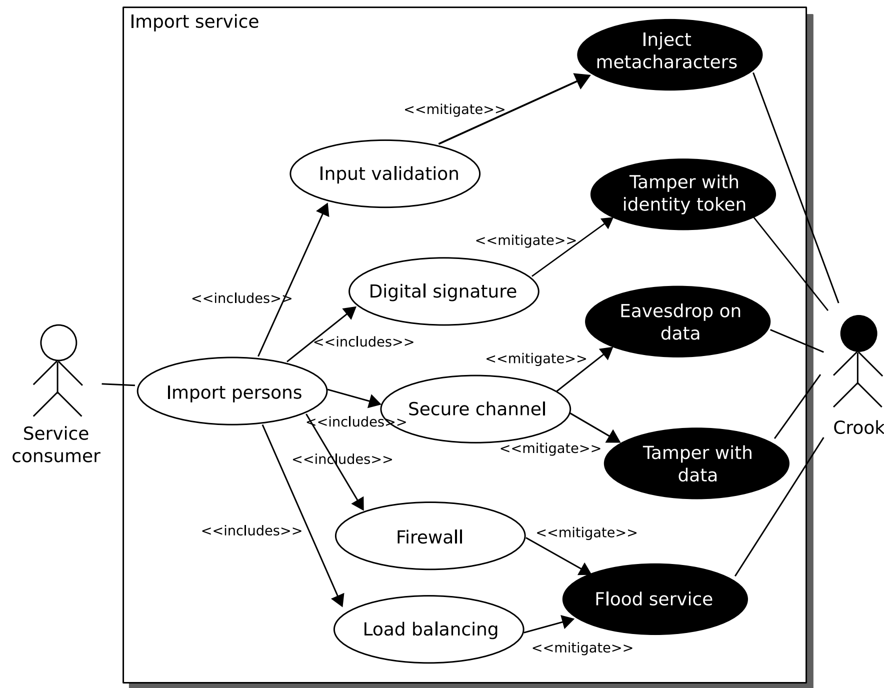

Figure 6.6: Misuse case diagram for UC1: Import persons

The misuse case shown in figure 6.6 illustrates the threats and countermeasures that are related to the process of importing persons:

- A malicious user may perform metacharacter injection, such as SQL and XPath injection. To mitigate this attack, it is clear from the figure that input to the service should be validated.
- The consumer might have tampered with the identity token that have been issued by the security service. To counter this, the digital signature of the identity token is verified by the import service.
- Eavesdropping of data is also a threat, in which the attacker could be able to both look at personal information and steal identity tokens that are transmitted. A secure channel is used to counter eavesdropping.
- An attacker could tamper with data that is being sent between the consumer and the import service. Tampering of data is countered using a secure channel.
- An attacker could flood the service[2], which is a threat that is countered using a firewall to filter requests, or using load balancing.

[2]Note that while I recognize that the threat of service flooding (DDoS) exists in this system, I will not describe how to counter it in later chapters. This threat is countered using mechanisms that are applied either by the host running the service or in the network infrastructure, and thus it does not greatly affect how services are designed.

6.3.2 UC2: Authenticate Consumer

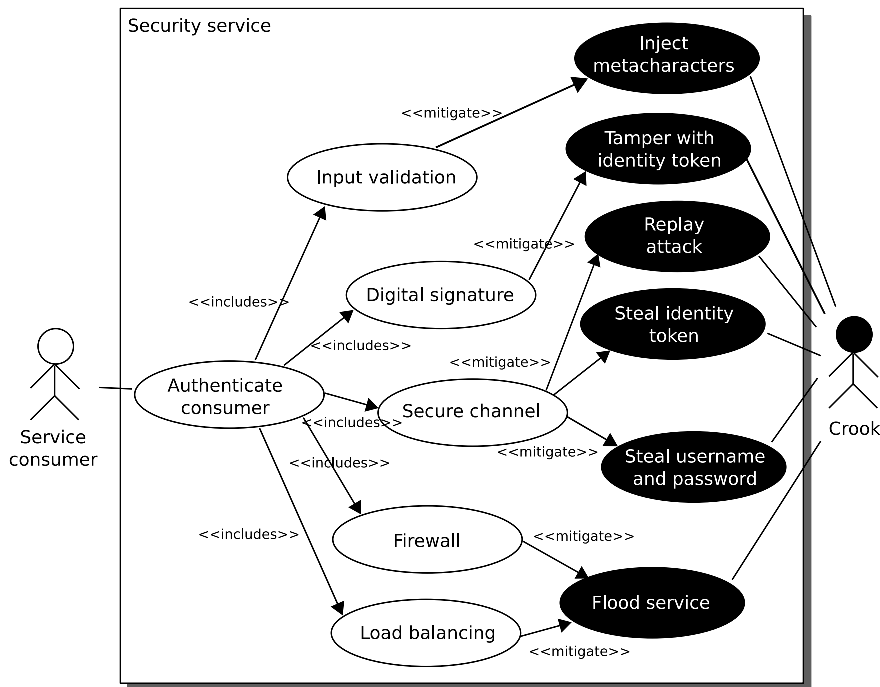

Figure 6.7: Misuse case diagram for UC2: Authenticate consumer

Figure 6.7 shows the threats and countermeasures that can be related to authentication of consumers:

- As was also the case for import of persons, a malicious user can try to inject metacharacters together with data that are sent to the service. In this case, such injections could be used as a means of gaining unauthorized access. The input to the service need to be validated to mitigate this attack.
- The consumer of the security service may try to tamper with an identity token to get unauthorized access to services. This is countered by using digital signatures.
- Replay attacks, described in chapter 4, can also be performed for gaining unauthorized access. Replay attacks are mitigated by using a secure channel.
- While usernames and passwords are transmitted between the consumer and the security service, these credentials can be stolen. A secure channel is used for countering this.
- While identity tokens are transmitted between the consumer and the security service, these tokens can be stolen by an outside attacker. A secure channel is used for countering this.
- Flooding of the service is countered using firewalls and load balancing.

6.3.3 UC3: Add Log Entry

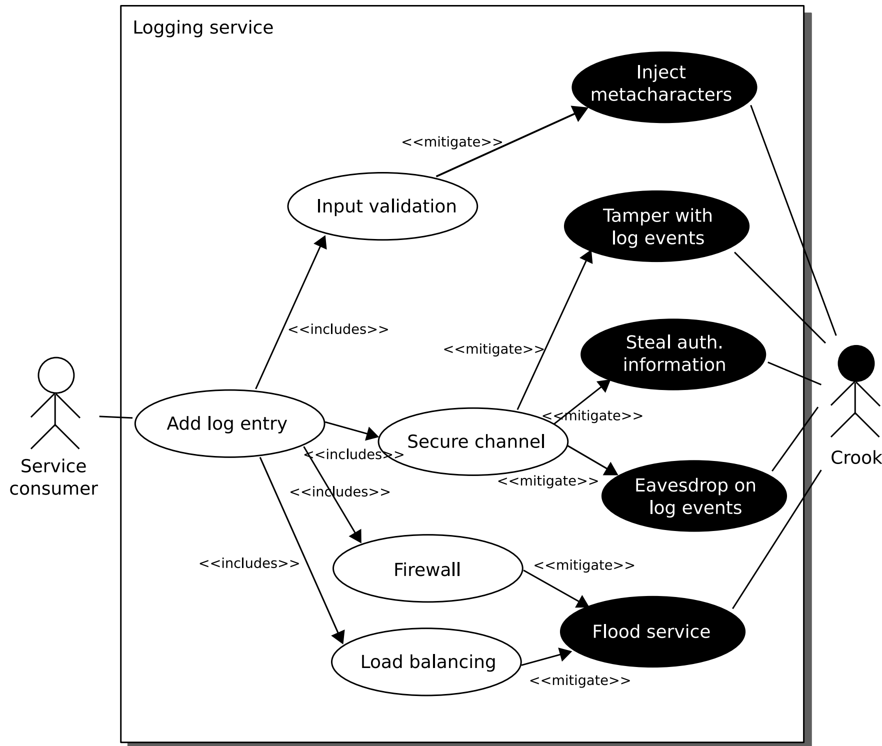

Figure 6.8: Misuse case diagram for UC3: Add log entry

The misuse case diagram in figure 6.8 shows threats and countermeasures related to logging of events:

- A consumer can try to inject metacharacters together with log events that are sent to the service. The input to the service needs to be validated to mitigate this attack.

- An outside attacker may try to tamper with log events that are sent from the consumer to the service. This may be performed to hide attacks on other services. A secure channel is used for mitigating this attack.

- An outside attacker may eavesdrop on log events that are sent from the consumer to the service. By doing this, the attacker can get information regarding the implementation of the SOA, or read personal information that might be logged. This is countered using a secure channel.

- An outside attacker may try to steal authentication information as it is transmitted between the consumer and the service. A secure channel is used for countering this.

- Flooding of the service is countered using firewalls and load balancing.

6.4 Summary

In this chapter, the requirements of the service-oriented architecture have been identified.

After applying some of the SOA security guidelines in section 5.2, we found that two additional services were needed in our architecture: A security service that helps centralizing authentication, and a logging service that enables centralization of logging. In addition, we assessed the need for message-level security, and found that transport-level security is sufficient in this scenario. However, message-level security is needed for signing of identity tokens that are issued by the security service. Table 6.1 summarizes the use of security guidelines in this chapter.

ID	**Name**	**Practical use**
SG4	Centralize security	Using a security service for authenticating service consumers.
SG5	Centralize logging	Sending log events to a centralized logging service.
SG2	Assess the need for message-level security	Transport-level security is sufficient, except when exchanging identity tokens.

Table 6.1: Security guidelines applied during requirements engineering.

In section 6.3, security requirements were identified using misuse cases. One misuse case diagram was presented for each service to illustrate the threats and countermeasures that are related to our SOA.

Now that the requirements have been identified, we can move on to the design of the service-oriented architecture.

Chapter 7

Design

In the previous chapter, the requirements and threats of our SOA were identified. I will now describe design of the SOA by taking these requirements and threats into account.

The chapter begins by identifying the standards that will be used for realizing the SOA. After the standards have been identified, we are able to decide which existing technologies, such as platforms and frameworks, that best suite our needs. When both the standards and the technologies are known, we can move on to describing the actual design of our services. Throughout the whole chapter, special attention will be given to the adherence to security guidelines. The chapter ends with a summary of the guidelines that have been used during the design phase.

7.1 Standards

Guideline SG1, *retain interoperability when handling security*, in section 5.2, states that open and widely accepted security standards should be used when realizing a SOA. In this section, I will list the standards that will be used in this system, and explain why they have been chosen. When selecting appropriate XML-based standards, I have focused on standards that are defined by W3C and OASIS. These standards are open and expected to be supported in many situations.

7.1.1 Web Services

Web Services, described in chapter 3, are the most popular way of implementing SOA [20]. It is therefore the most natural technology to use when demonstrating SOA security in practice. Web Services are realized using a set of specific standards which are used to enable interoperable interaction.

The requirements specification in chapter 6 detailed three services: An import service, a security service, and a logging service. All of these services will be implemented as Web Services, using the standards described in chapter 3.

7.1.2 WS-Security

WS-Security, described in section 3.3.1, defines a SOAP header element for making security claims [20]. Instead of adding security information to the body of SOAP messages, we use WS-Security as it expresses security information in a standardized way that is widely understood. In addition, by keeping security information out of the body of messages, security is not processed as part of business operations. This is in line with guideline SG3 in section 5.2, *handle security outside of business operations*.

As seen from the requirements specification in chapter 6, we will need to make several security claims in our SOA:

- Institutions must claim their identity when authenticating with the security service. This will be done using usernames and passwords.
- Institutions must claim their identity when using the import service, and also prove that they are authorized to use it.
- The security service must prove that identity tokens have not been tampered with by signing the tokens and adding information required for verification of the signature.

WS-Security allows all of these claims to be added to the SOAP header.

7.1.3 XML Signature

XML Signature, described in section 3.3.3, allows digital signatures to be expressed using XML [20]. Such signatures can be added to the header of SOAP messages using WS-Security, and can be used to detect if some part of a message has been tampered with.

In the requirements specification in chapter 6, I found that transport-level security will suffice in this SOA. This means that SSL/TLS can be used instead of digital signatures. However, there was one exception: The identity tokens that are issued by the security service. These tokens need to be digitally signed to detect potential tampering. We will be using XML Signature to sign identity tokens.

7.1.4 SAML

SAML, described in section 3.3.4, is a language that can be used to assert security information about a subject [20]. This information is communicated using

SAML assertions. These assertions may contain the identity of the subject, permissions, email addresses, and other useful information.

The requirements specification in chapter 6 stated that a central security service should be used for authentication and authorization of institutions. This security service should issue identity tokens that institutions could use to prove their identity when accessing the import service. These identity tokens can be expressed as SAML assertions.

The security service is responsible for issuing SAML assertions, and should support authentication and authorization requests through a standardized interface. The SAML protocol, introduced in section 3.3.4, can be used for issuing, validation, renewal, and cancellation of SAML assertions. The SAML protocol provides exactly the interface we need for communicating authentication requests and assertions between consumers and the security service.

7.1.5 SSL/TLS

We will be relying on transport-level security for countering eavesdropping and tampering of data. Transport-level security has better performance and is often better supported than message-level security [19]. Thus, it is wise to use transport-level security when message-level security is not required. SSL/TLS is commonly used for security on the transport-level, and is widely used for secure communication on the Internet. This means that it is likely to be supported by most web servers and clients. SSL/TLS also protects against replay attacks [48], which was identified as a threat in chapter 6.

A few steps need to be taken before SSL/TLS can be utilized. Each service should have its own SSL certificate. A self-signed certificate can be generated using the `keytool` application that is bundled with Java [20]. In addition, the consumers that are going to invoke the services must be configured to trust the certificates.

7.2 Technologies

As stated by guideline GG10 in section 5.1, it is good practice to use existing community resources instead of designing home-brew solutions. This section presents the main technologies that we will use for developing the service-oriented architecture.

When working with service-oriented architectures, there are a lot of platforms and libraries that can be used. Many commercial vendors offer products related to SOA. However, in this thesis I will focus on open source software. The platforms and libraries used in this project have been chosen primarily based on security requirements and the author's prior knowledge and experience.

7.2.1 The Java Platform

In theory, the choice of a programming platform is not as critical in SOA as in traditional application development. No matter what platform we choose, services and consumers should be expected to interoperate well with those written on other platforms [20]. However, there are some aspects that need to be considered when choosing a programming platform. Mainstream programming platforms for SOA are Microsoft's .NET platform and Java Enterprise Edition, but relevant tools are also written for C/C++ [2] [3]. As mentioned in section 4.3, one of the countermeasures to buffer overflow attacks is to use type-safe languages. Because of this, C/C++ should be avoided if possible. I will focus on open source frameworks and libraries in this project, which means that the Java platform is a good choice.

7.2.2 Apache CXF

The SOA will be designed using Web Services, which means that we need a framework that can help us build such services. At the moment, there are many open source Web Service frameworks to choose from. The Apache Software Foundation maintains an updated list [6] of frameworks and the features they support. Three of the most prominent frameworks at the time of writing are Apache Axis 1.x, Apache Axis2, and Apache CXF. I will briefly describe these frameworks here.

Apache Axis 1.x [2] is one of the older frameworks and supports the most basic features and standards. While Axis 1.x is stable and described in some books, like SOA Security [20] by Kanneganti and Chodavarapu, it it is no longer under active development. Apache Axis2 [3] is a redesign of Axis 1.x and supports many of the latest standards [6]. Axis2 has some advantages over Axis 1.x, such as high performance and hot deployment.

Apache CXF [4] is one of the most recent efforts in the field of open source Web Service frameworks. It is the offspring of two older Web Service projects, namely XFire and Celtix [41]. Some of the advantages of Apache CXF are ease of use, good support for Web Service standards and high performance [4]. I have decided to use Apache CXF for Web Service development, mainly because of its ease of use and Spring-based configuration[1].

Before continuing, a few words need to be said about the interceptor chain of Apache CXF. Interceptors are fundamental processing units inside Apache CXF. When a service is invoked, an interceptor chain is created and invoked [4]. Each interceptor on the chain gets a chance to do what it wants with a message, which includes operations such as reading, transformation, and header

[1]Spring is a Java/JEE application framework that claims to increase development productivity and runtime performance while improving test coverage and application quality [39]. Spring is often referred to as a dependency injection framework because of its ability to link objects together using XML files.

processing. I will use interceptors to add security functionality to services. This is in line with SOA security guideline SG3, *handle security outside of business operations*, as security functionality is placed outside of the service implementation class. Figure 7.1 illustrates how interceptors are connected to the service class.

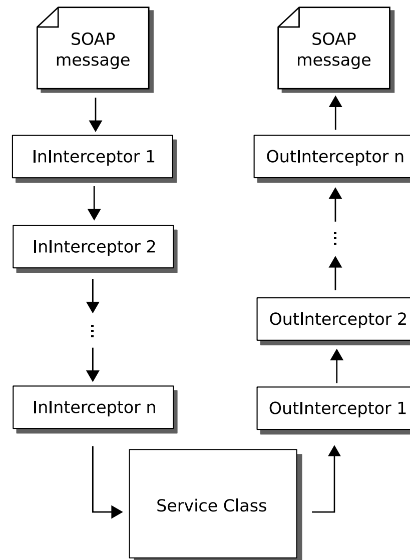

Figure 7.1: In Apache CXF, SOAP messages that arrive to the service are processed by a chain of interceptors. After the service implementation class has been invoked, a new set of interceptors process SOAP messages on their way out.

Interceptors can be configured for both services and client applications. A large set of interceptors are included with the Apache CXF distribution. Among these are two interceptors for handling inbound and outbound WS-Security processing respectively. These are described in section 7.2.3.

7.2.3 WSS4J

Apache CXF makes use of many underlying libraries for creating services. One of these is WSS4J (Web Services Security for Java) which is used for generation, signing, and verification of WS-Security information within SOAP messages [5]. WSS4J is also being used by Axis 1.x, Axis2 and XFire, which means that the WS-Security support and configuration in these frameworks are very similar. According to guideline SG1, *retain interoperability when handling security*, interoperability should not suffer because of security. Thus, we should find out if WSS4J is interoperable with other Web Services frameworks or libraries. Version 1.5 of WSS4J can be configured to be compliant with the WS-I Basic Security Profile [5]. This means that it is interoperable with other software that is compliant with the Basic Security Profile.

Configuration of WSS4J is done through the interceptors that is included in the Apache CXF distribution. Two interceptors, `WSS4JInInterceptor` and

`WSS4JOutInterceptor`, process SOAP messages on their way in and out. Settings such as location of keys for encryption and signing may be specified in configuration files [4].

7.2.4 Log4J

As can be seen from the requirements specification in chapter 6, the SOA should use a dedicated logging service for centralizing logging. This service will receive log events from multiple services and store them centrally for easier monitoring and attack detection/recovery.

To ease development and configuration, an existing logging framework named Log4J will be used. This framework can be configured to use different types of appenders for logging to the console, files or databases. Uria [43] describes an appender that sends log events to a Web Service. This appender can be used on all hosts where services are deployed. It can be configured to log events that correspond to specific levels (e.g. info, warning, and error) or events that originate from specific Java classes. By using a strict configuration, only specific events can be logged to avoid overloading the logging service.

7.2.5 Input Validation Framework

Guideline GG9, *be reluctant to trust* states that user input should not be trusted, and that it should be validated before being used. In doing this, we face the challenge of having multiple services and thus multiple entrances to applications. This means that handling input validation in one place, as would be normal in traditional architectures, becomes difficult.

According to guideline SG4, *centralize security*, we should try to centralize security functionality when possible. In his master's thesis, Jensen [18] has developed an input validation framework for Web Services. He hints at the possibility of making a central validation service based on this framework. However, there are some potential problems of such a solution. Every message would have to be routed through the validation service, and performance or the possibility of denial of service attacks might be an issue.

The services in our SOA have different operations and XML Schemas, which means that a different validation configuration for each service needs to be made in any case. One might also assume that the validation configurations do not need to be frequently modified after deployment. Because of this, instead of using a central validation service, we will validate input in each service by using Jensen's input validation framework [18].

7.3 Service Design

Now that we have identified which standards and technologies to use in the development of our SOA, we can move on to the design of the individual services. The services have individual security mechanisms, and this section describes how these mechanisms are realized in each service. In addition, the business logic of the services is briefly described.

7.3.1 Import Service

When invoked, the import service receives a SOAP message from the consumer. The body of this message contains the list of persons that should be imported, while the header of the message contains a signed SAML assertion. The consumer must provide this assertion to prove his or her identity to the import service. To avoid eavesdropping and tampering of data during transmission, a secure channel (SSL/TLS) is used for exchanging messages between the two parties. Figure 7.2 illustrates the communication between a consumer and the import service.

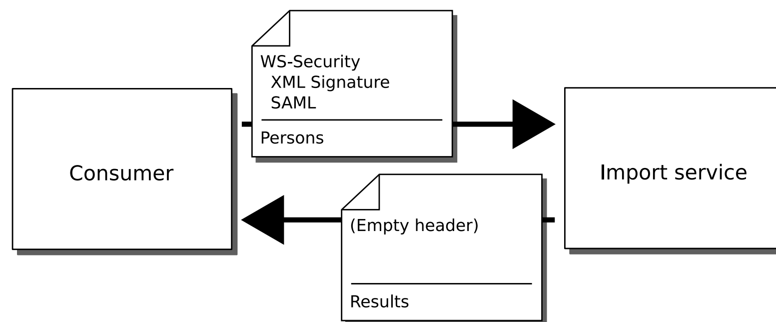

Figure 7.2: A consumer sends a SOAP message to the import service. The message body contains a list of persons, and the header contains a signed SAML assertion. The import service responds with a list of results.

Upon receiving the message from the consumer, the import service performs a set of security checks. As described in section 7.2.2, we are using interceptors to add security functionality to our services. Figure 7.3 illustrates the three interceptors that are used by the import service.

First of all, the message must be validated to counter metacharacter injection attacks. This is done as early as possible to minimize unvalidated processing of data. The import service uses a `ValidationInterceptor` to validate the contents of messages. This interceptor is configured to use the input validation framework provided by Jensen [18]. Following validation, the signature of the SAML assertion must be verified to ensure that the assertion has not been tampered with. The `WSS4JInInterceptor` that is included in Apache CXF is configured to do this. However, this interceptor is not able to process the contents of SAML assertions, and thus an additional interceptor, `SAMLInInterceptor`,

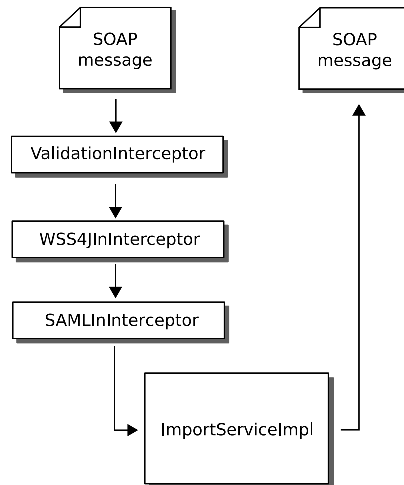

Figure 7.3: An incoming SOAP message to the import service is processed by three interceptors to perform validation of data, verification of the digital signature, and processing of the SAML assertion.

is used to authorize the consumer.

When the service implementation class is invoked, all security processing has been performed, and the implementation class can concentrate on the business logic. We have successfully been able to handle security outside of the business logic as recommended by guideline SG3 in section 5.2.

As described in chapter 6, the import service invokes other services for doing the actual import. These services, which are referred to as *system services* in this report, are assumed to be provided by system vendors. If we made the system services ourselves, we could propagate the SAML assertion from the import service request to the system service requests. However, as the system services are made by different vendors, they use different security mechanisms. Thus, we cannot assume that the system services support SAML. Instead, the import service needs to be configured to invoke the system services using the security mechanisms that they require.

I will briefly describe the classes that perform the business logic inside the import service. Figure 7.4 shows a class diagram, illustrating the main classes and their relations.

Four classes and one interface are depicted in the class diagram:

- `ImportServiceImpl` is the main service class, and implements the methods that are available to service consumers. The method `importPersons()` calls the `PersonImporter` class to start the import process.

- `PersonImporter` is responsible for carrying out the import process. It contains a set of `SystemClient` instances, and uses these instances to read and modify data about persons. In addition, it makes use of a `PersonComparator` that compares current and new data.

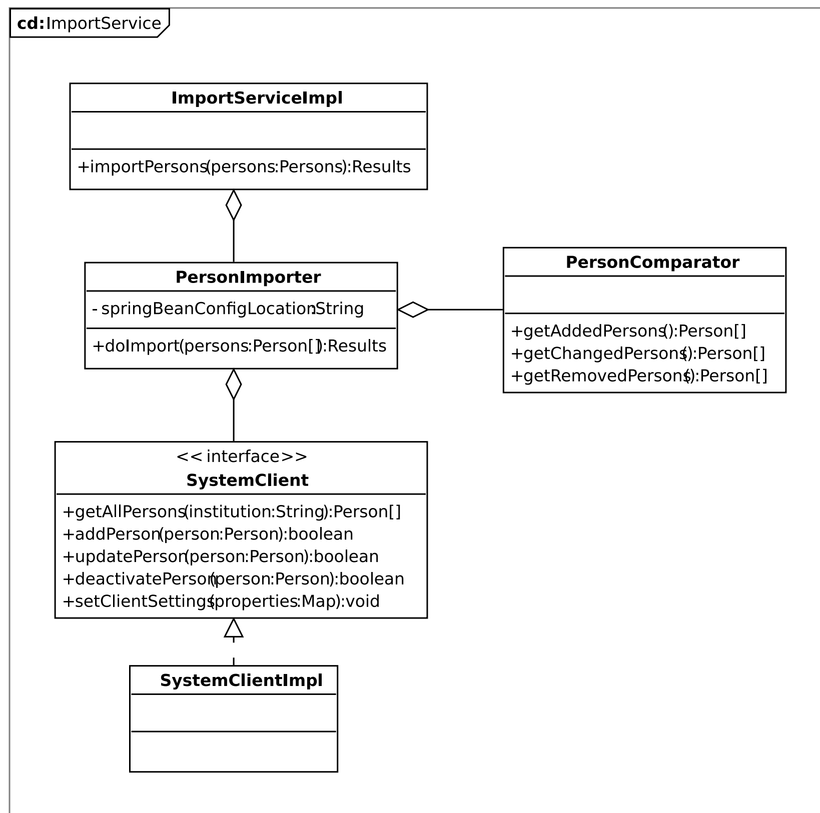

Figure 7.4: The classes that perform the business logic of the import service.

- `PersonComparator` compares two lists of person instances, and identifies new, removed, and updated persons.
- `SystemClient` is an interface that contains methods for retrieval and modification of personal information.
- `SystemClientImpl` is a client implementation class that invokes a system service to read and modify personal information. Each system service has a corresponding `SystemClient` implementation class such as this.

Each client implementation class needs to be configured to be able to authenticate with their respective system services. The Spring framework allows us to specify this in an XML configuration file [39].

This has been a quick description of the design of the import service. Now, let us move on to the security service.

7.3.2 Security Service

The security service utilizes the SAML protocol for communicating identity tokens. It receives a SOAP message from a consumer, where the body contains

a SAML request, and the header contains the username and password of the consumer. The security service responds with a signed SAML assertion describing the consumer in question. The consumer can then use the SAML assertion for authentication and authorization with other services. Figure 7.5 shows the communication between a consumer and the security service. A secure channel (SSL/TLS) is used for exchanging messages between the two parties.

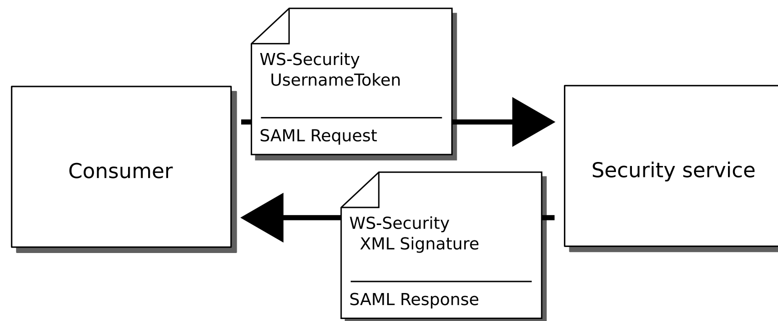

Figure 7.5: A consumer sends a SOAP message to the security service. The body of this message contains a SAML request, and the header contains the username and password of the consumer. The security service returns a signed SAML assertion back to the consumer.

After receiving the SOAP message from the consumer, the security service performs a set of security checks. Similar to the import service, the security service performs security processing in interceptors. The interceptors involved in security processing are illustrated in figure 7.6.

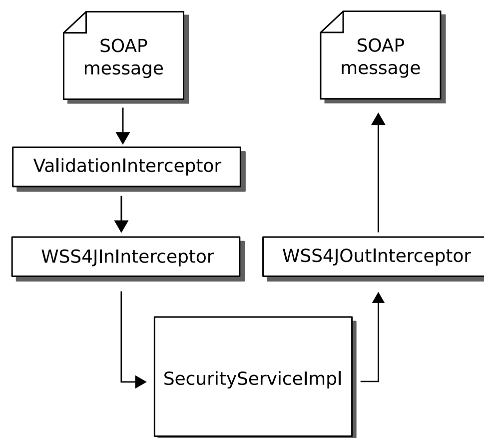

Figure 7.6: An incoming SOAP message to the security service is processed by two interceptors to perform validation of data and authentication of the consumer. The `WSS4JInInterceptor` is configured to check the username and password against the user database.

To counter the threat of metacharacter injection, all input from the consumer is validated. This is performed in a `ValidationInterceptor`. In addition, the username and password of the consumer is checked against a user database. This is done by configuring the `WSS4JInInterceptor` to communicate with the database. After the service implementation class is done processing the

message, an additional interceptor is needed for signing the SAML assertion that is sent back to the consumer. `WSS4JOutInterceptor` is configured to do this.

When the service implementation class is invoked, the SAML issuing process begins. The interceptors have verified that the consumer is authorized to invoke the security service, but no SAML assertion has been made. The business logic of the security service takes care of this. One might argue that issuing of SAML assertions is a security operation, and that according to guideline SG3, *handle security outside of business operations*, we should not issue assertions as part of the business logic. However, in this case, security *is* the business operation of the service. Thus, in the case of the security service, we can justify issuing SAML assertions as part of the business logic.

I will briefly describe the classes involved in issuing of SAML assertions. The class diagram in figure 7.7 shows an overview of the most important classes.

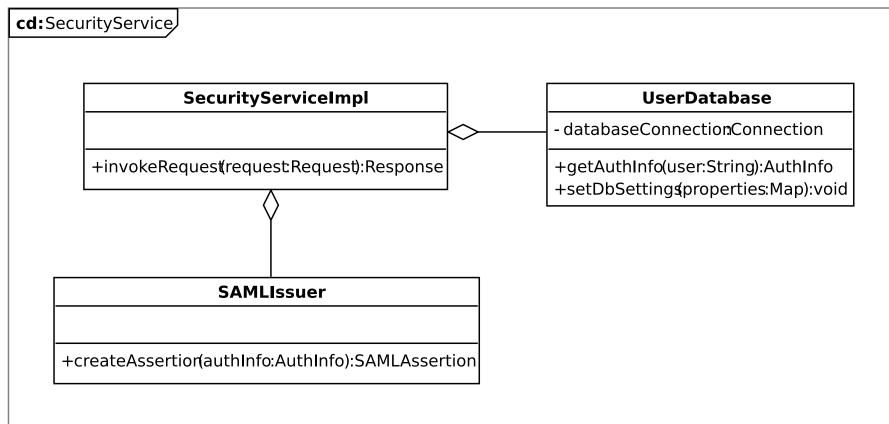

Figure 7.7: The classes that are involved in issuing of SAML assertions.

Three classes are depicted in the class diagram:

- `SecurityServiceImpl` is the main service class, and implements the methods that are available to service consumers. The method `invokeRequest()` receives a request, connects to the `UserDatabase` and uses `SAMLIssuer` to issue SAML assertions based on the information found in the database.
- `UserDatabase` is responsible for communication with the database, and returns authentication and authorization information that can be used when issuing assertions.
- `SAMLIssuer` generates the actual SAML assertions. It receives authentication and authorization information regarding a user, and generates assertion based on this information.

The `UserDatabase` needs to be configured to connect to the database. The Spring framework allows us to configure this in an XML configuration file.

When using configuration files that contain security information such as a password for connecting to the database, it is important to consider guideline GG4, *follow the principle of least privilege*. The configuration file must be readable only to the security service.

7.3.3 Logging Service

The logging service receives a SOAP message from the consumer, containing a log event in the message body. In return, the consumer receives a result indicating whether the event was successfully logged or not. The logging service differs from other services in that it is invoked directly by systems. No external consumers or individuals will be invoking the logging service, and thus there is no need to rely on the security service and SAML for authentication. Instead, consumers must present a trusted public key certificate (SSL client certificate) when invoking the logging service. Authentication based on client certificates is a technically superior to most other authentication schemes [20]. Similar to the other services, a secure channel (SSL/TLS) is used for exchanging messages between a consumer and the logging service. Figure 7.8 illustrates the communication between the two parties.

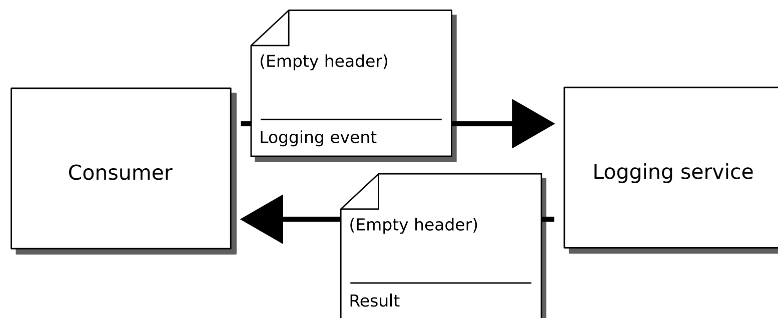

Figure 7.8: A consumer sends a SOAP message to the logging service. The body of the message contains an event that should be logged. In this case there are no header elements, as authentication is performed on the transport level using SSL client certificates. The logging service responds with a result, indicating whether or not the event was logged.

Yet again interceptors are used for security processing of SOAP messages. However, the logging service authenticates consumers on the transport level using SSL client certificates, which means that interceptors for authentication and authorization are not required. The logging service only has one interceptor, `ValidationInterceptor`. Figure 7.9 illustrates this.

One might argue that validation of log events is not necessary, as these messages originate from trusted systems. However, imagine a situation where a system logs attack attempts and forwards these logs to the logging service. If some of these attack attempts contain metacharacters, this may affect both the logging service and a potential user interface that parses the logs. Because of this,

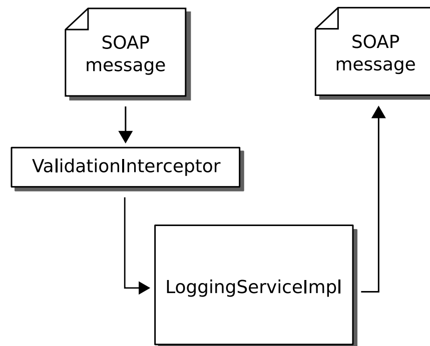

Figure 7.9: An incoming message to the logging service is processed by one interceptor, `ValidationInterceptor`, that validates the data of the message.

incoming messages to the logging service need to be validated. This is in line with guideline GG9, *be reluctant to trust*, in section 5.1.

I have not provided a class diagram for the business logic of the logging service, as the business logic consists of only one class, namely `LoggingServiceImpl`. This class reads the log event from the incoming message and simply logs it using the Log4J framework. Log4J can then be configured to log the event to different locations using different formats.

7.4 Summary

In this chapter, I have described the design of a simple service-oriented architecture.

Before designing the services, I identified the standards and technologies that were best suited for this specific SOA. Open and widely adopted standards were chosen to minimize potential interoperability problems. The Apache CXF framework was used for Web Services development, and allows us to handle security outside of business operations by using interceptors. Log4J is a logging framework that can be configured to log to multiple places, including remote Web Services. In addition, an input validation framework were used for validating input in each service.

I have not implemented the entire SOA-solution that has been described in this chapter. However, I have provided a few implementation examples in appendix B. These examples include WSDL documents describing all services, a validation interceptor, a SAML interceptor, a Web Services logging appender, and an example of service configuration.

Even though this SOA is relatively small, we have seen that many security considerations come into play. Table 7.1 summarizes the security guidelines that have been applied when designing the SOA.

ID	**Name**	**Practical use**
SG1	Retain interoperability when handling security	Security standards defined by W3C and OASIS are used. The WSS4J library can be configured to be compliant with the WS-I Basic Security Profile for interoperability with other platforms and frameworks.
SG3	Handle security outside of business operations	Security claims are inserted to the header of SOAP messages. Interceptors are used to do security processing before and after the service implementation class is invoked.
GG4	Follow the principle of least privilege	Configuration files containing passwords are readable only to the service in question.
GG9	Be reluctant to trust	All input to the services is validated, regardless of who the consumer is.
GG10	Use community resources	Existing community resources are chosen instead of designing home-brew solutions.

Table 7.1: Security guidelines applied during design.

Chapter 8

Discussion

In this chapter, I will discuss the security guidelines that I have proposed, and mention how they may affect other aspects of a SOA. In addition, I will briefly comment on the experiences that I have had with frameworks and libraries during this project.

8.1 The Benefit of Having Security Guidelines

In chapter 4, threats and countermeasures related to SOA were presented. It turns out that many of the threats that are present in traditional systems are also relevant in SOA. In fact, I have found only one threat that is specific to SOA, which is service description scanning attacks. XPath injection, entity recursion, and external entity attacks are especially relevant in SOA as they exploit vulnerabilities related to XML-processing, but other than that, the threats in chapter 4 are well-known in traditional systems [1] [40]. However, I found that different countermeasures are needed for countering some of the threats. In addition, the nature of SOA leads to new approaches for handling security functionality such as authentication and authorization.

What I gather from this is that people who are new to SOA will have few problems understanding the threats that are evident in such architectures. If a person has dealt with security in other kinds of systems, most of the threats will be familiar. However, as countermeasures may differ from those of traditional systems, persons who are new to SOA will need to gain some knowledge on how to apply countermeasures to threats in SOA. Similarly, as new approaches are introduced for handling security functionality, newcomers will need to familiarize themselves with these approaches.

This shows that people who are new to SOA will benefit from having a set of security guidelines when developing SOA-based systems. In my thesis, I have made such guidelines and demonstrated how they can be applied in a real-world case scenario.

8.2 Practical Use of My Guidelines

Faced with the challenge of developing a SOA, how difficult is it to use my security guidelines during the development process? To answer this question, we need to look at how the guidelines are written. Each guideline has a descriptive title and is written in a short and concise way, describing the reason why the guideline should be applied, and of course how it may be applied. As a result, anyone involved in the development of a SOA can quickly glance through the guidelines at any time during development. This reduces the possibility of forgetting important security aspects, and makes sure that security is appropriately handled.

The security guidelines do not refer to any other chapters or sections of my thesis. This is done on purpose to avoid dependencies between the guidelines and the rest of the report. As a result, the guidelines can easily be taken out of the context of this report and be used independently as a brief overview of how to solve security challenges in SOA.

As mentioned, the guidelines are short and concise. While this is good in many cases, it can also be seen as a disadvantage. Detailed information on how to apply the guidelines are omitted. This means that using guidelines as the only means of gaining SOA-related security knowledge is not enough. For instance, one should also have knowledge of security standards and how to use them. In this respect, section 3.3 is a good starting point for anyone new to SOA, or more specifically Web Services.

In chapters 6 and 7, I have demonstrated how to use my guidelines. As can be seen from these chapters, there is no streamlined process for applying guidelines. The guidelines are simply used when appropriate. For instance, when the issue of handling authentication in multiple services comes up, the list of guidelines is consulted, and it becomes clear that authentication is best handled centrally. Remember that guidelines are just recommendations for what to do and what not to do in a certain context [28]. Thus, it is natural to simply use them when appropriate.

8.3 The Side Effects of Centralization

Two of my security guidelines, SG4 and SG5 in section 5.2, recommend centralizing security and logging. While this makes the security enforcements easier to maintain, and allows for better monitoring, it also has a few side effects.

Performance might be an issue, depending on how the central security and logging facilities are realized. In chapters 6 and 7, I have used a dedicated Web Service for centralization. Web Services need to parse XML data when receiving requests, resulting in a more resource intensive processing as opposed to for example a simple HTTP request. This may have a negative impact on performance if a large number of consumers use the security and logging

services simultaneously. Especially, the logging service might be affected, as each event occurring in each service is logged. An attacker might also exploit this by sending a large number of requests to these services, effectively launching a (distributed) denial of service attack. This threat, including measures for countering it, was described in section 4.7.

In the case scenario described in this report, performance is not expected to be a problem, as the number of services in the SOA are limited. However, as more services are added to the SOA, one might have to consider other strategies for centralization. Mallery [27] suggests using a network gateway that processes incoming and outgoing messages to the SOA. Such advanced gateways are able to process XML data more effectively, and can be configured to process SOAP headers and issue identity tokens. However, such solutions need to be acquired from a vendor. As mentioned in section 1.3, I have decided not to look at commercial solutions in this report.

8.4 Security Standards and Interoperability

Guideline SG1 recommends retaining interoperability while handling security by using open and widely adopted standards. In addition, it states that one should focus on compliance with the WS-I Basic Security Profile for promoting interoperability between different implementations of standards. While these recommendations help making services and consumers more interoperable, they cannot guarantee full interoperability. The reason for this is simple: Different services may use different security mechanisms.

A good example of this can be seen in the SOA that have been described in this report. I mentioned this briefly in section 7.3.1. While the import service require that a SAML assertion is used for proving a consumer's identity and authorizations, the system services (provided by vendors) cannot be expected to support such assertions. This means that the import service cannot propagate the SAML assertions received from the institutions when interacting with the system services. The import service has no other choice than to use the authentication mechanisms required by the system services. As a result, the security credentials required by the system services need to be known by the import service.

Instead of specifying the security credentials in the configuration of intermediary services, such as the import service, Dushin and Newcomer [11] suggest using a data structure for handling multiple credential types and formats in the same request. By using this approach, the import service could receive both a SAML assertion and any other credentials required by the system services. However, this would potentially result in added complexity for consumers of the import service, as many credentials need to be added to each message.

This shows that the interoperability concern can involve more considerations than just choosing the right standards and ensuring compliance of implemen-

tations.

8.5 Experiences with Frameworks and Libraries

In this project, I have used a few frameworks and libraries when demonstrating how to design a SOA. While I have not implemented the entire system, I have made some implementation examples, as shown in appendix B. When making these examples, I had to make practical use of the frameworks and libraries described in chapter 7. This has given me the opportunity to share some of the experiences that I have had with these tools.

Getting started with the Apache CXF Web Services framework is not hard. People who are familiar with the basic standards of Web Services will have few problems developing simple Web Services, as the Apache CXF web page [4] provides good tutorials that guide you through the process of service development. However, once you start developing more complex services, things tend to get difficult. The documentation provided on the Apache CXF web page only serves as an introduction, and at the time of writing, some sections are still under construction. Because of this, I had to download the source code of Apache CXF and look through it on some occasions.

The Web Services security library, WSS4J, suffered from the same problems as Apache CXF. Simple scenarios such as sending and verifying a username and password using WS-Security is documented, but once things like SAML assertions are being used, you might have to figure things out by looking though the WSS4J source code.

In my experience, the documentation of Apache CXF and WSS4J has not yet caught up with the functionality of these tools. What impact does these issues have on security? As developers have difficulties finding information on how to implement certain functionality, they might resort to trial and error for making things work. This can result in implementations that are vulnerable, while at the same time seemingly work as intended. Thus, thorough testing is important for identifying vulnerabilities in such implementations.

Chapter 9

Conclusion

Service-oriented architecture is an interesting paradigm for combining and reusing functionality across applications, technologies, and organizations. However, if organizations decide to adopt SOA, they need to be aware of the security challenges that come with this paradigm. SOA differ from traditional systems in ways that affect how security is handled. In a large or complex SOA, new approaches are needed for ensuring confidentiality and integrity of data, and handling authentication, authorization, and logging. If adopters of SOA do not have a clear understanding of these new approaches, they risk ending up with a set of services that are either vulnerable to attacks or difficult to manage.

In this thesis, I have identified security challenges, standards, threats, and countermeasures that are related to SOA, especially focusing on the differences between SOA and traditional systems. Based on this knowledge, I have created five guidelines that provide recommendations for how to handle security in a SOA. These guidelines should be applied during requirements engineering and design. Web Services are the most popular way of realizing SOA in practice, but other technologies can also be used. Because of this, I have made sure that my SOA security guidelines do not assume any specific implementation. As a result, the guidelines can be applied regardless of the technology that is used for SOA development. The guidelines are written in a short and concise way, and can be taken out of the context of this report and used without modifications.

In specific, the guidelines recommend retaining interoperability, assessing the need for message-level security, handling security outside of business operations, centralizing security, and centralizing logging. By demonstrating the use of these guidelines in a real-world case scenario, I have shown that the guidelines are helpful during development of a SOA. Using these guidelines as a reference during requirements engineering and design, the right decisions with regard to security in service-oriented architectures are made. As a result, vulnerabilities and bad security design choices can be avoided.

Chapter 10

Further Work

I have identified some possible further work that can be performed in relation to this project:

1. **Finalize implementation**
   In this report, I have described development of a service-oriented architecture, and implemented some of its functionality. The next step with regard to this SOA is obviously to finalize its implementation. Appendix B contain WSDL documents for all services. Skeletons for implementation classes of services and consumers can be generated automatically by using these documents together with a tool. In addition, interceptors for handling security functionality has been implemented. Thus, the work that remains is primarily implementation of the business logic of services and consumers.

2. **Evaluate the security in Web Services frameworks**
   As described in section 7.2.2, there are a lot of open source Web Services frameworks to choose from. Future studies could look at how security is handled in one or more of these frameworks, find out what standards and features are supported, and assess aspects such as ease of use, default security settings, and quality of documentation.

3. **Survey of SOA adoption and security approaches**
   Kontogiannis et al. [22] mentions that there has been a significant growth in the development of SOA-based systems over the past decade. However, I have yet to see any specific numbers indicating the degree of SOA adoption. A survey could be performed to examine the current adoption of SOA in Norwegian organizations. In addition to the degree of adoption, it would be interesting to see what approaches organizations choose for realizing SOA in practice - especially with regard to security.

4. **Approaches to Logging and Monitoring in SOA**
   One of my security guidelines recommend centralizing logging in SOA. In this report, I have realized this in practice by using a dedicated logging service. However, this is not necessarily the best way to centralize

logging in SOA. If a large number of services are present, there might be performance issues, as discussed in chapter 8. Future studies could look at approaches to logging and monitoring in SOA, and identify advantages and disadvantages of each approach with regard to aspects such as performance, security, and interoperability.

Bibliography

[1] Andrews, Mike and Whittaker, James. *How to Break Web Software*. Addison-Wesley, Boston, 2006. ISBN 0321369440.

[2] Apache. Web Page of the Apache Axis 1.x Web Service Framework, 2008 (accessed 2008-05-05). URL http://ws.apache.org/axis/.

[3] Apache. Web Page of the Apache Axis2 Web Service Framework, 2008 (accessed 2008-05-05). URL http://ws.apache.org/axis2/.

[4] Apache. Web Page of the Apache CXF Web Service Framework, 2008 (accessed 2008-05-05). URL http://cxf.apache.org/.

[5] Apache. Web Page of the WSS4J Web Services Security Library, 2008 (accessed 2008-05-05). URL http://ws.apache.org/wss4j/.

[6] Apache. Web Service Frameworks Stack Comparison, 2008 (accessed 2008-05-05). URL http://wiki.apache.org/ws/StackComparison.

[7] Barnum, Sean and Gegick, Michael. Reluctance to Trust, 2005 (accessed 2008-04-07). URL https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/principles/355.html.

[8] Booth, David, Ferris, Christopher, Champion, Mike, Orchard, David, McCabe, Francis, and Haas, Hugo. Web Services Architecture by the W3C Working Group, 2004 (accessed 2008-02-21). URL http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[9] Clark, James. W3C Recommendation: XSL Transformations Version 1.0, 1999 (accessed 2008-06-02). URL http://www.w3.org/TR/xslt.

[10] Dreyfus, Emmanuel. Securing Systems with chroot, 2003 (accessed 2008-03-19). URL http://www.onlamp.com/pub/a/bsd/2003/01/23/chroot.html.

[11] Dushin, Fred and Newcomer, Eric. Handling Multiple Credentials in a Heterogeneous SOA Environment. *IEEE Security and Privacy, vol. 5, no. 5*, 2007.

[12] Foss, Julie-Marie and Ingvaldsen, Nina. *Web Application Security*. Master's thesis, Norwegian University of Science and Technology, NTNU, 2005.

[13] Gallagher, Tom. *Hunting Security Bugs*. Microsoft Press, Redmond, 2006. ISBN 9780735621879.

[14] Haas, Hugo. Web service use case: Travel reservation, 2002 (accessed 2008-04-15). URL `http://www.w3.org/2002/06/ws-example`.

[15] Hallam-Baker, Philip, Kaler, Chris, Monzillo, Ronald, and Nadalin, Anthony. Web Services Security: SAML Token Profile, 2004 (accessed 2008-05-19). URL `http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0.pdf`.

[16] Hunter, David, Rafter, Jeff, Fawcett, Joe, van der Vlist, Eric, Ayers, Danny, Duckett, Jon, Watt, Andrew, and McKinnon, Linda. *Beginning XML, 4th Edition*. Wrox, Indianapolis, 2007. ISBN 978-0470114872.

[17] Huseby, Sverre H. *Innocent Code: A Security Wake-Up Call for Web Programmers*. John Wiley and Sons, 2004. ISBN 0-470-85744-7.

[18] Jensen, Henning. *Input Validation Framework for Web Services*. Master's thesis, Norwegian University of Science and Technology, NTNU, 2007.

[19] Josuttis, Nicloai M. *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, City, 2007. ISBN 0-596-52955-4.

[20] Kanneganti, Ramarao and Chodavarapu, Prasad. *SOA Security*. Manning Publications Co., Greenwich, 2008. ISBN 1-932394-68-0.

[21] Kearney, Paul. Message level security for web services. Technical report, Security Research Centre, BT, UK, 2005 (accessed 2008-05-02). URL `http://linkinghub.elsevier.com/retrieve/pii/S1363412704000044`.

[22] Kontogiannis, Kostas, Lewis, Grace A., Smith, Dennis B., Litoiu, Marin, Muller, Hausi, Schuster, Stefan, and Stroulia, Eleni. The Landscape of Service-Oriented Systems: A Research Perspective. In *Proceedings of the International Workshop on Systems Development in SOA Environments*. 2007.

[23] Light, Ben, Holland, Christopher P., Kelly, Sue, and Wills, Karl. Best Of Breed IT Strategy: An Alternative To Enterprise Resource Planning Systems. In *Proceedings of the 8th European Conference on Information Systems*. 2000.

[24] Luu, Tieu. Separation of Concerns in Web Service Implementations, 2006 (accessed 2008-04-11). URL `http://www.onjava.com/pub/a/onjava/2006/09/06/separation-of-concerns-in-web-services.html`.

[25] MacKenzie, C. Matthew, Laskey, Ken, McCabe, Francis, Brown, Peter, Metz, Rebekah, and Hamilton, Booz Allen. Reference Model for Service Oriented Architecture, Committee Draft 1.0, 7 February 2006, 2006 (accessed 2008-02-14). URL `http://www.oasis-open.org/committees/download.php/16587/wd-soa-rm-cd1ED.pdf`.

[26] Maler, Eve, Mishra, Prateek, and Philpott, Rob. Assertions and Protocol for the OASIS Security Assertion Markup Language V1.1, 2003 (accessed 2008-05-19). URL `http://www.oasis-open.org/committees/download.php/3406/oasis-sstc-saml-core-1.1.pdf`.

[27] Mallery, John. *Hardening Network Security*. McGraw-Hill/Osborne, New York, 2005. ISBN 9780072257038.

[28] McGraw, Gary. *Software Security*. Addison-Wesley Professional, Reading, 2006. ISBN 0321356705.

[29] McIntosh, Michael, Gudgin, Martin, Morrison, K. Scott, and Barbir, Abbie. WS-I Basic Security Profile Version 1.0, 2007 (accessed 2008-05-22). URL `http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html`.

[30] Naraine, Ryan. DoS Flaw in SOAP DTD Parameter, 2003 (accessed 2008-03-21). URL `http://boston.internet.com/news/article.php/3289191`.

[31] Natis, Yefim V. Gartner Research Note AV-19-6751, 2003 (accessed 2008-02-14). URL `http://www.gartner.com/resources/114300/114358/114358.pdf`.

[32] OASIS. Web Page of the Organization for the Advancement of Structured Information Standards (OASIS), 2008 (accessed 2008-05-16). URL `http://www.oasis-open.org/`.

[33] Schumacher, Markus. *Security Engineering with Patterns: Origins, Theoretical Models, and New Applications*. Springer, 2007. ISBN 3540407316.

[34] SecurityFocus. Bugtraq Archive, 2008 (accessed 2008-03-17). URL `http://www.securityfocus.com/archive/1`.

[35] Sen, Robi. Avoid the dangers of XPath injection, 2007 (accessed 2008-03-18). URL `http://www.ibm.com/developerworks/library/x-xpathinjection.html`.

[36] Siddiqui, Bilal. Exploring XML Encryption, Part 1, 2002 (accessed 2008-05-18). URL `http://www.ibm.com/developerworks/xml/library/x-encrypt/`.

[37] Sindre, Guttorm and Opdahl, Andreas L. Eliciting security requirements with misuse cases. *Requirements Engineering, vol. 10, no. 1*, 2004.

[38] Specht, Stephen M. and Lee, Ruby B. Distributed Denial of Service: Taxonomies of Attacks, Tools and Countermeasures. In *Proceedings of the 17th International Conference on Parallel and Distributed Computing and Systems*. 2004.

[39] SpringSource. Web Page of the Spring Framework, 2008 (accessed 2008-05-05). URL `http://springframework.org/`.

[40] Stallings, William. *Cryptography and Network Security*. Prentice Hall, Englewood Cliffs, 2005. ISBN 9780131873162.

[41] Townsend, Bjorn. Axis, Axis2 and CXF: Surveying the WS Landscape, 2007 (accessed 2008-05-05). URL `http://www.theserverside.com/tt/articles/article.tss?l=AxisAxis2andCXF`.

[42] Tsai, Chii-Ren. Non-Repudiation In Practice, 2003 (accessed 2008-03-25). URL `http://dsns.csie.nctu.edu.tw/iwap/proceedings/proceedings/sessionD/6.pdf`.

[43] Uria, Carmelo. Manage message logging with the Web Service Appender for Log4j, 2005 (accessed 2008-05-19). URL `http://www.ibm.com/developerworks/webservices/library/ws-log4j.html`.

[44] Vidgen, Richard. Requirements Analysis and UML. *Computing and Control Engineering, April 2003*, 2003.

[45] Viega, John and McGraw, Gary. *Building Secure Software*. Addison-Wesley, Boston, 2002. ISBN 9780201721522.

[46] W3C. Web Page of the World Wide Web Consortium (W3C), 2008 (accessed 2008-04-15). URL `http://www.w3.org/`.

[47] Wagner, David, Foster, Jeffrey S., Brewer, Eric A., and Aiken, Alexander. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Year 2000 Network and Distributed System Security Symposium (NDSS)*. 2000.

[48] Wagner, David and Schneier, Bruce. Analysis of the SSL 3.0 Protocol. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*. 1996.

Appendix A

Use Case and Misuse Case Descriptions

This chapter contains textual use case and misuse case descriptions for the SOA-based system described in this report. Use case descriptions are normally seen from the perspective of the user of the system [44]. I have described the activities involved in the use cases in a more detailed fashion to show exactly what happens as a result of consumer-service interaction. W3C provides Web Service use case examples that are written in a similar level of detail [14].

Services in a SOA can be used in multiple contexts. For instance, the security service may be used for authentication in a process involving storage or retrieval of documents, and the logging service may be used to log events occurring in normal web applications. These services might one day be involved in other operations than import of personal data, and because of this, use case descriptions of these services should be as general as possible. Today, the import service will only be invoked by the client application. In the future, however, this service might be invoked by other consumers, and thus the import service needs to have its own use case description.

I have added two new columns to extend the use case descriptions to misuse case descriptions, namely *threats* and *countermeasures*. To get the full picture of how the services operate, I have also included the following columns in the use case descriptions:

- **Brief description**
  Briefly describes the intention behind the use case and what happens during the use case.
- **Triggers**
  The event that triggers the use case.
- **Pre-conditions**
  Conditions that need to be satisfied before the use case is started.

- **Post-conditions**
  Conditions that need to be satisfied after the use case has completed.
- **Activities**
  The sequence of activities that occur during the use case.
- **Alternative flows**
  Flows that deviate from the normal sequence of activities.
- **Exception flows**
  Flows that occur as a result of an exception.

Use Case #1 **Title:** Import persons **Actor:** Consumer of import service
Brief Description: This use case is performed when a consumer invokes the import service to update the person registries of one or more systems. When updating a person registry, the import service compares the current list of persons with the new list submitted by the institution. New persons are added to the registry, persons that have been changed are updated and persons that are not present in the submitted list are deactivated. Finally, a summary of the changes made by the import operation is returned to the consumer.
Triggers: A consumer invokes the import service, submitting a list of persons for import.
Pre-Conditions: **PRC1:** Authentication information for verification of consumers must have been added to the service.
Post-Conditions: **POC1:** A summary of the changes made by the import operation, or alternatively an error message is returned to the consumer.

Continued on next page

Table A.1: UC1: Import persons (continued)

Activities: **A1:** The consumer invokes the import service with a list of persons. **A2:** The import service verifies the identity token and authorizes the consumer. **A3:** The import service reads all persons contained in the message received from the consumer. **A4:** The import service groups all persons based on which system they are destined for. **A5:** The import service invokes a given system service, requesting all persons from the system. **A6:** The import service compares the persons received from the institution with the persons fetched from the system. **A7:** The import service adds new persons to the given system by invoking the system service and supplying a list of persons. **A8:** The import service updates changed persons in the given system by invoking the system service and supplying a list of persons. **A9:** The import service deactivates persons that were not present in the list received from the institution by invoking the system service and supplying a list of persons. **A10:** The import service returns a summary of the changes back to the consumer. *A5-A9 are repeated for each system that needs to be updated.*
Alternative flows: **AF1:** At A6, the import service finds that no changes need to be made for a given system. The import service skips A7-A9.
Exception flows: **EF1:** At A2, the import service is unable to verify the identity token received from the consumer. The import service returns an error message back to the consumer explaining the problem. **EF2:** At A5, A7, A8 or A9, the import service is unable to communicate with the system service. The import service returns an error message back to the consumer explaining the problem.

Continued on next page

Table A.1: UC1: Import persons (continued)

Threats:
T1: The consumer injects metacharacters along with data. Possible outcomes:
T1-1: Implementation details of the service are revealed to the consumer.
T1-2: Personal data are revealed to the consumer.
T2: The consumer tampers with an identity token before transferring it to the service. Possible outcomes:
T2-1: The attacker is able to gain unauthorized access to the services.
T3: An outside attacker is eavesdropping on data that is being transferred from the consumer to the service. Possible outcomes:
T3-1: Personal information is revealed to unauthorized entities.
T4: An outside attacker tampers with data while it is being transferred from the consumer to the service. Possible outcomes:
T4-1: Unauthorized persons are added to the system registries.
T4-2: Authorized persons are modified or removed from the system registries.
T5: An attacker floods the service with requests. Possible outcomes:
T5-1: The service is overloaded, resulting in denial of service.

Countermeasures:
C1: To counter T1, input from the consumer is validated.
C2: To counter T2, the signature of identity tokens is verified
C3: To counter T3 and T4, a secure channel is used for transferring data.
C4: To counter T5, a firewall is used for limiting access.
C5: To counter T5, load balancing is used to handle a large number of requests.

Use Case #2
Title: Authenticate consumer
Actor: Consumer of security service

Brief Description: This use case is performed when a service consumer wants to obtain a proof of its identity. The consumer authenticates with the security service by providing its security credentials, and receives a signed identity token in return. This token can be used to prove the consumers identity when accessing a service. This centralizes authentication and authorization, and allows the consumer to use the same security credentials for accessing multiple services.

Triggers: A consumer invokes the security service, requesting an identity token.

Pre-Conditions:
PRC: An underlying user database or directory service must be configured to communicate with the security service.

Post-Conditions:
POC: An identity token or alternatively an error message is returned to the consumer.

Continued on next page

Table A.2: UC2: Authenticate consumer (continued)

Activities: **A1:** The consumer invokes the security service, supplying his/her security credentials. **A2:** The security service reads the security credentials of the consumer. **A3:** The security service validates the security credentials against the underlying user database or directory service. **A4:** The security service reads the token request from the consumer. **A5:** The security service creates an identity token based on the consumer request and the information that was found. **A6:** The security service signs the token as proof that the token was in fact issued by the security service. **A7:** The security service sends token back to the consumer.
Alternative flows: -
Exception flows: **EF1:** At A3, the security service cannot find a user that matches the given username and password. The security service returns an error message back to the consumer explaining the problem.
Threats: **T1:** The consumer injects metacharacters along with data. Possible outcomes: T1-1: The consumer is able to circumvent authentication. T1-2: Implementation details of the service are revealed to the consumer. **T2:** The consumer tampers with the identity token that is returned from the service. Possible outcomes: T2-1: The consumer is able to gain unauthorized access to other services. **T3:** An outside attacker captures a message that was sent between the consumer and the security service and replays it. Possible outcomes: T3-1: The attacker is able to perform operations on services that he or she is not authorized to access. **T4:** An outside attacker steals an identity token while it is being sent from the service to the consumer. Possible outcomes: T4-1: The attacker is able to perform operations on services that he or she is not authorized to access. **T5:** An outside attacker steals a username and password that is being transferred from the consumer to the service. Possible outcomes: T5-1: The attacker is able to request an identity token using a stolen username and password. **T6:** An attacker floods the service with requests. Possible outcomes: T6-1: The service is overloaded, resulting in denial of service.
Countermeasures: **C1:** To counter T1, input from the consumer is validated. **C2:** To counter T2, data is signed using a digital signature. **C3:** To counter T3, T4 and T5, a secure channel is used for transferring data. **C4:** To counter T6, a firewall is used for limiting access. **C5:** To counter T6, load balancing is used to handle a large number of requests.

Use Case #3
Title: Add log entry
Actor: Consumer of logging service

Brief Description: This use case is performed when a service consumer wants to add a log entry to the central log repository. A centralized log repository makes it easier for an administrator to monitor all services and detect and investigate potential errors and security problems.

Triggers: A consumer invokes the logging service, requesting that a log entry should be stored.

Pre-Conditions:
PRC1: An underlying mechanism for logging of events must be configured.

Post-Conditions:
POC1: A status message indicating whether the log event was successfully stored or not should be returned to the consumer.

Activities:
A1: The consumer invokes the logging service, supplying a log event.
A2: The logging service reads the log event from the message.
A3: The logging service stores the log event using the underlying logging mechanism.
A4: The logging service sends a status message back to the consumer.

Alternative flows: -

Exception flows:
EF1: At A2, the logging service finds a problem when parsing the log message. The logging service returns an error message back to the consumer explaining the problem.

Threats:
T1: The consumer injects metacharacters along with data. Possible outcomes:
T1-1: The consumer is able to add logs that is processed in a potentially dangerous manner by subsystems.
T1-2: Implementation details of the service are revealed to the consumer.
T2: An outside attacker tampers with the log events that are sent from the consumer to the service. Possible outcomes:
T2-1: The attacker is able to change logs to hide break-ins or attempts thereof.
T3: An outside attacker steals authentication information while it is being sent from the consumer to the service. Possible outcomes:
T3-1: The attacker is able to add log entries on behalf of the consumer in question.
T4: An outside attacker eavesdrops on log messages. Possible outcomes:
T4-1: Implementation details of the SOA are revealed to the attacker.
T4-2: Sensitive data about persons are revealed to the attacker.
T5: An attacker floods the service with requests. Possible outcomes:
T5-1: The service is overloaded, resulting in denial of service.

Countermeasures:
C1: To counter T1, input from the consumer is validated.
C2: To counter T2, T3 and T4, data is encrypted during transfer.
C3: To counter T5, a firewall is used for limiting access.
C4: To counter T5, load balancing is used to handle a large number of requests.

Appendix B

Source Code

Due to time constraints, I have not implemented the entire SOA that is designed in this report. However, I have implemented some examples that show how my design can be implemented in practice.

The following source code is provided:

- WSDL documents describing all Web Services.
- An input validation interceptor for Apache CXF.
- A SAML interceptor for Apache CXF.
- A Web Services logging appender for Log4J.
- A Spring [39] configuration example for the import service.

B.1 WSDL Documents

As described in section 3.2.2, WSDL documents describe operations, protocols, and data formats used by a Web Service. In this section, I have provided such WSDL documents for the import service, security service, and logging service. Implementation classes of services and consumers can be generated automatically based on these documents by using the `wsdl2java` tool distributed with the Apache CXF Web Services framework [4]. Thus, these WSDL documents are a good starting point for implementing the services and their business logic.

B.1.1 Import Service

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions name="ImportService"
    targetNamespace="http://uninett.no/fas/ws/importservice"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://uninett.no/fas/ws/importservice"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

```
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:types>
  <xsd:schema elementFormDefault="unqualified"
      attributeFormDefault="unqualified"
      targetNamespace="http://uninett.no/fas/ws/importservice"
      xmlns:tns="http://uninett.no/fas/ws/importservice"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <!-- The XML Schema that defines the format of personal data -->
  <xsd:include schemaLocation="http://www.uninett.no/trofast/ \
    Integrasjon/Importformat.xsd" />

  <!-- The results that are sent back to the consumer -->
  <xsd:element name="results" type="tns:results" />
    <xsd:complexType name="results">
      <xsd:sequence>
        <xsd:element ref="tns:result" minOccurs="0"
            maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>

  <xsd:element name="result" type="tns:result" />
    <xsd:complexType name="result">
      <xsd:sequence>
        <xsd:element name="personid" type="xsd:string"
          minOccurs="1" maxOccurs="1" />
        <xsd:element name="statusCode" type="xsd:string"
          minOccurs="1" maxOccurs="1" />
      </xsd:sequence>
    </xsd:complexType>

  </xsd:schema>
</wsdl:types>


<wsdl:message name="persons">
  <wsdl:part name="persons" element="tns:persons" />
</wsdl:message>
<wsdl:message name="results">
  <wsdl:part name="results" element="tns:results" />
</wsdl:message>


<wsdl:portType name="ImportServicePortType">
  <wsdl:operation name="importPersons">
    <wsdl:input name="persons" message="tns:persons" />
    <wsdl:output name="results" message="tns:results" />
  </wsdl:operation>
</wsdl:portType>


<wsdl:binding name="ImportServiceSoapBinding"
      type="tns:ImportServicePortType">
  <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="importPersons">
```

```
      <soap:operation soapAction="" style="document"/>
        <wsdl:input name="persons">
          <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="results">
          <soap:body use="literal"/>
        </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>


  <wsdl:service name="ImportService">
    <wsdl:port name="ImportServicePort"
        binding="tns:ImportServiceSoapBinding">
      <soap:address location="https://localhost:9091/ImportService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Listing B.1: WSDL document describing the import service (importservice.wsdl)

B.1.2 Security Service

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions name="SecurityService"
    targetNamespace="http://uninett.no/fas/ws/securityservice/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://uninett.no/fas/ws/securityservice/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol">

  <wsdl:types>
    <xsd:schema elementFormDefault="unqualified"
      attributeFormDefault="unqualified"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="urn:oasis:names:tc:SAML:1.0:protocol"
      xmlns:tns="urn:oasis:names:tc:SAML:1.0:protocol">

      <!-- The XML Schema that defines the SAML protocol -->
      <xsd:import namespace="urn:oasis:names:tc:SAML:1.0:protocol"
          schemaLocation="http://www.oasis-open.org/committees/download.php/ \
              3408/oasis-sstc-saml-schema-protocol-1.1.xsd"/>
    </xsd:schema>
  </wsdl:types>


  <wsdl:message name="request">
    <wsdl:part name="body" element="samlp:Request" />
  </wsdl:message>
  <wsdl:message name="response">
    <wsdl:part name="body" element="samlp:Response" />
  </wsdl:message>


  <wsdl:portType name="SecurityServicePortType">
```

```
    <wsdl:operation name="invokeRequest">
      <wsdl:input name="request" message="tns:request" />
      <wsdl:output name="response" message="tns:response" />
    </wsdl:operation>
  </wsdl:portType>


  <wsdl:binding name="SecurityServiceSoapBinding"
      type="tns:SecurityServicePortType">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="invokeRequest">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input name="request">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="response">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>


  <wsdl:service name="SecurityService">
    <wsdl:port name="SecurityServicePort"
        binding="tns:SecurityServiceSoapBinding">
      <soap:address location="http://localhost:9092/SecurityService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Listing B.2: WSDL document describing the security service (securityservice.wsdl)

B.1.3 Logging Service

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions name="LoggingService"
    targetNamespace="http://uninett.no/fas/ws/loggingservice"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://uninett.no/fas/ws/loggingservice"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <wsdl:types>
    <xsd:schema elementFormDefault="unqualified"
      attributeFormDefault="unqualified"
      targetNamespace="http://uninett.no/fas/ws/loggingservice"
      xmlns:tns="http://uninett.no/fas/ws/loggingservice"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">

      <!-- The elements used to describe logging events -->
      <xsd:complexType name="throwables">
        <xsd:sequence>
          <xsd:element name="throwableInfo" type="xsd:string"
              minOccurs="1" maxOccurs="unbounded"/>
        </xsd:sequence>
```

```
    </xsd:complexType>

    <xsd:element name="event" type="tns:event" />
    <xsd:complexType name="event">
      <xsd:sequence>
        <xsd:element name="level" type="xsd:int"
            minOccurs="1" maxOccurs="1" />
        <xsd:element name="localTime" type="xsd:long"
            minOccurs="1" maxOccurs="1" />
        <xsd:element name="hostname" type="xsd:string"
            minOccurs="1" maxOccurs="1" />
        <xsd:element name="logger" type="xsd:string"
            minOccurs="1" maxOccurs="1" />
        <xsd:element name="eventMessage" type="xsd:string"
            minOccurs="1" maxOccurs="1" />
        <xsd:element name="threadName" type="xsd:string"
            minOccurs="1" maxOccurs="1" />
        <xsd:element name="throwables" type="tns:throwables"
            minOccurs="1" maxOccurs="1" />
      </xsd:sequence>
    </xsd:complexType>

    <!-- The status code that is returned to the consumer -->
    <xsd:element name="statusCode" type="xsd:string" />

  </xsd:schema>
</wsdl:types>


<wsdl:message name="event">
  <wsdl:part name="event" element="tns:event" />
</wsdl:message>
<wsdl:message name="statusCode">
  <wsdl:part name="statusCode" element="tns:statusCode" />
</wsdl:message>


<wsdl:portType name="LoggingServicePortType">
  <wsdl:operation name="logEvent">
    <wsdl:input name="event" message="tns:event" />
    <wsdl:output name="statusCode" message="tns:statusCode" />
  </wsdl:operation>
</wsdl:portType>


<wsdl:binding name="LoggingServiceSoapBinding"
    type="tns:LoggingServicePortType">
  <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="logEvent">
    <soap:operation soapAction="" style="document"/>
    <wsdl:input name="event">
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="statusCode">
      <soap:body use="literal"/>
    </wsdl:output>
```

```
    </wsdl:operation>
  </wsdl:binding>


  <wsdl:service name="LoggingService">
    <wsdl:port name="LoggingServicePort"
        binding="tns:LoggingServiceSoapBinding">
      <soap:address location="http://localhost:9095/LoggingService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Listing B.3: WSDL document describing the logging service (loggingservice.wsdl)

B.2 Input Validation

The data that is received by services should be validated to mitigate metacharacter injection. When describing the design of services in the SOA, I have used a validation interceptor for validating data. This interceptor, named `ValidationInterceptor`, makes use of a validation framework by Jensen [18], and is provided in listing B.4. The `ValidationInterceptor` may be utilized for validation by anyone implementing Web Services using the Apache CXF Web Services framework.

```
package no.uninett.fas.ws.security;

import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Logger;

import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPMessage;

import no.fjas.wspv.ConfigParser;
import no.fjas.wspv.ServiceConfig;
import no.fjas.wspv.ValidationException;
import no.fjas.wspv.ValidatorSupport;
import no.fjas.wspv.util.ClassLoaderUtil;

import org.apache.cxf.binding.soap.SoapFault;
import org.apache.cxf.binding.soap.SoapMessage;
import org.apache.cxf.common.i18n.Message;
import org.apache.cxf.common.logging.LogUtils;
import org.apache.cxf.endpoint.Endpoint;
import org.apache.cxf.interceptor.Fault;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;
import org.w3c.dom.Document;

/**
 * An input validation interceptor for the Apache CXF Web Services framework.
 * The interceptor extracts the SOAP body of an incoming message and validates
```

```
 * it using the Web Services Payload Validator (no.fjas.wspv) by Henning Jensen.
 *
 * @author Magne Rodem
 */
public class ValidationInterceptor extends AbstractPhaseInterceptor<SoapMessage>
{
  public static final String PROP_CONFIG_FILE = "configFile";

  private static final Logger LOG =
    LogUtils.getL7dLogger(ValidationInterceptor.class);
  private Map<String, Object> properties = new HashMap<String, Object>();
  private Map<String, ServiceConfig> globalServiceConfig =
    new HashMap<String, ServiceConfig>();

  /**
   * Reads a set of properties and loads the configuration of the validation
   * framework.
   *
   * @param properties properties that are read by the interceptor
   */
  public ValidationInterceptor(Map<String, Object> properties) {
    super(Phase.PRE_INVOKE);
    this.properties = properties;
    loadConfig();
  }

  private void loadConfig() {
    String configFile = (String)properties.get(PROP_CONFIG_FILE);

    InputStream is = null;
    if (configFile != null) {
        is = ClassLoaderUtil.getResourceAsStream(
            configFile, ValidationInterceptor.class);
    } else {
      LOG.warning("Config file not specified.");
    }

    if (is != null) {
      globalServiceConfig = ConfigParser.parseValidatorConfigs(is, configFile);
    } else {
      LOG.warning("Validation configuration not loaded, " +
          "config file not found.");
    }
  }

  /**
   * Retrieves the configuration of the service, extracts the body of the
   * message and validates it.
   *
   * @param message the SOAP message that should be validated
   */
  public void handleMessage(SoapMessage message) throws Fault {
    Endpoint endpoint = message.getExchange().get(Endpoint.class);
    String serviceName = endpoint.getService().getName().getLocalPart();
    ServiceConfig config = globalServiceConfig.get(serviceName);

    SOAPMessage msg = message.getContent(SOAPMessage.class);
```

```
    Document doc = null;
    try {
      doc = msg.getSOAPBody().extractContentAsDocument();
    } catch (SOAPException e) {
      LOG.warning("Unable to extract SOAP body");
    }

    try {
        ValidatorSupport.validateEnvelope(config, doc);
    } catch (ValidationException e) {
      throw new SoapFault(new Message("VALIDATION_FAILED", LOG), e,
          message.getVersion().getSender());
    }
  }
}
```

Listing B.4: An input validation interceptor for the Apache CXF Web Services framework.

B.3 SAML Processing

As mentioned in section 7.3.1, the `WSS4JInInterceptor` that is included in Apache CXF does not process the contents of SAML assertions. To do this, we need an additional interceptor. Listing B.5 contains an interceptor that extracts authorization information from SAML assertions, and adds this information to the message context. As a result, the service implementation class is able to determine whether or not the consumer has been authorized. This interceptor may be utilized for authorization by anyone implementing Web Services using the Apache CXF Web Services framework.

```
package no.uninett.fas.ws.security.saml;

import java.util.Iterator;
import java.util.List;
import java.util.Vector;
import java.util.logging.Logger;

import no.uninett.fas.ws.security.SecurityConstants;

import org.apache.cxf.binding.soap.SoapFault;
import org.apache.cxf.binding.soap.SoapMessage;
import org.apache.cxf.common.i18n.Message;
import org.apache.cxf.common.logging.LogUtils;
import org.apache.cxf.interceptor.Fault;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;
import org.apache.ws.security.WSSecurityEngineResult;
import org.apache.ws.security.handler.WSHandlerConstants;
import org.apache.ws.security.handler.WSHandlerResult;
import org.opensaml.SAMLAction;
import org.opensaml.SAMLAssertion;
import org.opensaml.SAMLAuthorizationDecisionStatement;
import org.opensaml.SAMLStatement;
```

```java
import org.opensaml.SAMLSubject;

/**
 * The SAMLInInterceptor parses the content of an incoming SOAP message to
 * find a SAML assertion. If an assertion was found, the name, resource, and
 * permissions are added to the message context so that the service
 * implementation class is able to see that the consumer has been authorized.
 *
 * @author Magne Rodem
 */
public class SAMLInInterceptor extends AbstractPhaseInterceptor<SoapMessage> {
  private static final Logger LOG = LogUtils.getL7dLogger(
      SAMLInInterceptor.class);

  /**
   * Initializes the SAMLInInterceptor.
   */
  public SAMLInInterceptor() {
    super(Phase.PRE_INVOKE);
  }

  /**
   * Extracts the SAML assertion from the message and adds the authorization
   * details of the assertion to the message context.
   *
   * @param message a SOAP message containing an assertion
   */
  public void handleMessage(SoapMessage message) throws Fault {
    SAMLAssertion assertion = getAssertion(message);
    if (assertion == null) {
      throw new SoapFault(new Message("NO_SAML_ASSERTION", LOG),
          message.getVersion().getReceiver());
    }

    setAuthorization(assertion, message);
  }

  /*
   * Searches the Security header and extracts a SAMLAssertion instance.
   */
  private SAMLAssertion getAssertion(SoapMessage message) {
    List<Object> results = (Vector<Object>)message.get(
        WSHandlerConstants.RECV_RESULTS);
    SAMLAssertion assertion = null;

    for (Object result : results) {
      if (result instanceof WSHandlerResult) {
        List<Object> wsHndlResults = ((WSHandlerResult)result).getResults();
        for (Object wsHndlResult : wsHndlResults) {
          if (wsHndlResult instanceof WSSecurityEngineResult) {
            WSSecurityEngineResult wsEngineResult =
                ((WSSecurityEngineResult)wsHndlResult);
            assertion = (SAMLAssertion)wsEngineResult.get(
                WSSecurityEngineResult.TAG_SAML_ASSERTION);
          }
        }
      }
```

```
    }
    return assertion;
  }

  /*
   * Finds the authorization details of the assertion and adds it to
   * the message context.
   */
  private void setAuthorization(SAMLAssertion assertion, SoapMessage message) {
    SAMLAuthorizationDecisionStatement authzStm = null;
    Iterator it = assertion.getStatements();
    while (it.hasNext()) {
      SAMLStatement st = (SAMLStatement)it.next();
      if (st instanceof SAMLAuthorizationDecisionStatement) {
        authzStm = (SAMLAuthorizationDecisionStatement)st;
        break;
      }
    }

    SAMLSubject subject = authzStm.getSubject();
    String nameId = subject.getNameIdentifier().getName();
    String resource = authzStm.getResource();

    String actions = (String)message.get(SecurityConstants.PERMISSIONS);
    Iterator acIt = authzStm.getActions();
    while (acIt.hasNext()) {
      SAMLAction ac = (SAMLAction)acIt.next();
      if (actions == null || actions.equals("")) {
        actions = ac.getData();
      } else {
        actions += " " + ac.getData();
      }
    }

    message.put(SecurityConstants.USERID, nameId);
    message.put(SecurityConstants.RESOURCE, resource);
    message.put(SecurityConstants.PERMISSIONS, actions);
    LOG.info("SAML authorization: " + nameId + " has been given " +
        "permissions '" + actions + "' on " + resource);
  }
}
```

Listing B.5: An interceptor for the Apache CXF Web Services framework that extracts authorization information from SAML assertions.

B.4 Logging Appender

The SOA that has been designed in this report uses a logging service for centralized handling of logs. As described in section 7.2.4, I have used the Log4J logging framework to realize this functionality. By configuring Log4J to use specific appenders, logging events may be logged in different ways. To be able to send log messages to a Web Service, a custom appender need to be implemented and configured. Listing B.6 contains an appender that sends log events

to the central logging service.

```
package no.uninett.fas.ws;

import java.net.InetAddress;

import no.uninett.fas.ws.loggingservice.Event;
import no.uninett.fas.ws.loggingservice.Throwables;

import org.apache.log4j.AppenderSkeleton;
import org.apache.log4j.spi.LoggingEvent;

/**
 * The WebServiceLogAppender is a Log4J appender that invokes a Web Service
 * to log an event. The URL of the logging service is specified in the
 * log4j.xml configuration file as follows:
 *
 * <pre>
 * <appender name="webservice" class="no.uninett.fas.ws.WebServiceLogAppender">
 *   <param name="Url" value="https://localhost:9095/LoggingService" />
 * </appender>
 * </pre>
 *
 * @author Magne Rodem
 */
public class WebServiceLogAppender extends AppenderSkeleton {
  private LoggingClient client = null;
  private String url = null;

  /**
   * Indicates whether or not this appender must have a layout associated
   * with it.
   *
   * @return boolean whether or not this appender needs a layout
   */
  public boolean requiresLayout() {
    return false;
  }

  /**
   * Sets the URL of the logging service. This method is normally called by
   * the Log4J framework depending on what is configured in log4j.xml.
   *
   * @param url the URL of the logging service
   */
  public void setUrl(String url) {
    this.url = url;
  }

  /**
   * Invokes a logging service to append a new log message.
   *
   * @param event the logging event that is going to be logged
   */
  public synchronized void append(LoggingEvent event) {
    Event ev = new Event();
    ev.setLevel(event.getLevel().toInt());
    ev.setLocalTime(System.currentTimeMillis());
```

```java
    ev.setThreadName(event.getThreadName());
    ev.setLogger(event.getLoggerName());
    ev.setEventMessage(event.getRenderedMessage());

    String[] thrInfo = event.getThrowableStrRep();
    if (thrInfo != null) {
      Throwables tables = new Throwables();
      for (String info : thrInfo) {
        tables.getThrowableInfo().add(info);
      }
      ev.setThrowables(tables);
    }

    if (client == null) {
      client = new LoggingClient(url);
    }

    Object[] response = null;
    try {
      ev.setHostname(InetAddress.getLocalHost().getCanonicalHostName());
      response = client.invoke("logEvent", ev);
      if (response == null || !response.equals("success")) {
        throw new Exception("Bad response from logging service!");
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  /**
   * Closes the Web Service client.
   */
  public void close() {
    client.getClient().destroy();
  }
}
```

Listing B.6: A Log4J appender that invokes a Web Service to log an event.

B.5 Service Configuration

A lot has been said about interceptors in this report, and how they allow us to handle security outside of business operations. How are these interceptors configured in practice? The configuration of the import service is shown in listing B.7. The Spring framework [39] allows us to configure interceptors in an XML configuration file. This results in loose coupling between the service implementation class and interceptors.

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://cxf.apache.org/configuration/security"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
```

```
    http://cxf.apache.org/configuration/security
    http://cxf.apache.org/schemas/configuration/security.xsd
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

  <!--
    Configures the import service by specifying the implementation class,
    location of the WSDL document, and the address where the service should be
    made available.
  -->
  <jaxws:endpoint
      id="ImportService"
      implementor="no.uninett.fas.ws.importservice.ImportServiceImpl"
      wsdlLocation="no/uninett/fas/ws/importservice/importservice.wsdl"
      address="/ImportService">

    <!--
      Specifies inbound interceptors for this service.
    -->
    <jaxws:inInterceptors>

      <!--
        Interceptor for input validation. Reads the validation configuration
        to find out what values are allowed for individual elements, and
        validates incoming data.
      -->
      <bean class="no.uninett.fas.ws.security.ValidationInterceptor">
        <constructor-arg>
          <map>
            <entry key="configFile"
                value="no/uninett/fas/ws/importservice/validation.xml" />
          </map>
        </constructor-arg>
      </bean>

      <!--
        Interceptor for WS-Security processing. WSS4JInInterceptor depends
        on SAAJInInterceptor to be able to process security info. The WSS4J
        interceptor is configured to validate the signature of the message.
        The keystore containing trusted keys are specified in sign.properties.
        In addition, SAMLTokenUnsigned indicates that the service should
        check that a SAML assertion is present.
      -->
      <bean class="org.apache.cxf.binding.soap.saaj.SAAJInInterceptor"/>
      <bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
        <constructor-arg>
          <map>
            <entry key="action" value="Signature SAMLTokenUnsigned"/>
            <entry key="signaturePropFile"
                value="no/uninett/fas/ws/importservice/sign.properties" />
          </map>
        </constructor-arg>
      </bean>

      <!--
        Interceptor used for processing SAML assertions. Authorization
        information contained in the assertion is extracted and added to
        the message context. This way, the service implementation class
```

```
        is able to see that the consumer has been authorized.
      -->
      <bean class="no.uninett.fas.ws.security.saml.SAMLInInterceptor"/>

    </jaxws:inInterceptors>
  </jaxws:endpoint>
</beans>
```

Listing B.7: A Spring [39] configuration file that configures the import service.