NTNU
Innovation and Creativity

Adding Security to Web Services

An Automatic, Verifiable, and Centralized Mechanism for Web Services Input Validation

Lars Arne Brekken
Rune Frøysa Åsprang

Master of Science in Communication Technology
Submission date: June 2006
Supervisor: Peter Herrmann, ITEM
Co-supervisor: Carl C. Christensen, Bekk Consulting

Norwegian University of Science and Technology
Department of Telematics

Problem Description

It is often desirable to be able to specify software security policies at the deployment stage rather than during development. Separation of code and policy offers flexibility and encourages software reuse as making changes becomes easier and code does not necessarily have to be recompiled when security requirements change. Moving input validation out of the application can also help performance, by using dedicated computing resources for validation.

Early web services standards did not address security, and a set of standards and specifications that covers security has subsequently been developed and released. One of these specifications is WS-SecurityPolicy, which can be used to specify message security requirements such as encryption and signing of messages in policy files. Lack of input validation is considered one of the greatest security threats to web applications, but is not addressed in the current web services standards. This project will investigate whether these standards can be extended to specify and automatically validate input parameter requirements for web services. In particular, the following tasks will be performed:

- An examination of web services and web services security
- Design of an extension of existing web services standards that enables validation of web service input parameters
- Development of a prototypical implementation of the design
- Elaboration of the implementation by means of an example application

Assignment given: 2006-01-16
Supervisor: Peter Herrmann, ITEM

Abstract

Accepting unvalidated input is considered today's greatest web security threat. This master's thesis addresses that threat by proposing an automatic and centralized mechanism for validating web services input. By building on existing web services standards, the proposed solution intercepts incoming web service requests and validates them against a security policy.

A major design goal for this work was to realize web services input validation without modifying existing functionality. That is, the input validation security mechanism should be added out of code. This is achieved by keeping the web services and the validation mechanism separate. Input validation configuration is accomplished by modifying a configuration file.

Even when the validation mechanism logic is correct, it may not function as intended. Such anomalies are in most cases caused by human-introduced errors in the configuration file, resulting in the need for a configuration file verification tool. This thesis proposes a verification tool that quantifies the level of security by analyzing the configuration file.

Preface

This thesis was written as part of our Master of Science degrees at the Norwegian University of Science and Technology (NTNU) in the spring semester of 2006.

We would like to use this opportunity to thank our academic supervisor at NTNU, Professor Peter Herrmann, and our supervisor at Bekk Consulting, Carl Christensen, for their valuable guidance and useful discussions during our work on this thesis.

Furthermore, we would like to thank Arne A. Baste, John T. Flåm, Marius R. Hanssen, and Linda D. Zgombic for their proofreading efforts.

Trondheim, 1st June 2006

Rune Frøysa Åsprang Lars Arne Brekken

Contents

List of Figures

List of Tables

List of Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
DES	Data Encryption Standard
DSS	Digital Signature Standard
GUI	Graphical User Interface
HMAC	Hash Message Authentication Code
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IDS	Intrusion Detection System
IP	Internet Protocol
OASIS	Organization for the Advancement of Structured Information Standards
OSI	Open Systems Interconnection
OWASP	Open Web Application Security Project
SHA	Secure Hash Algorithm
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security

UDDI	Universal Description, Discovery, and Integration
UDP	User Datagram Protocol
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WS	Web Service
WSDL	Web Services Discovery Language
WSE	Web Services Enhancements
XML	Extensible Markup Language
XPath	XML Path Language

Chapter 1

Introduction

The demand for web services is expected to proliferate in the coming years [1]. Web services provide a standard means of interoperation between heterogeneous software applications that run on a variety of platforms. Due to their growing popularity and increasingly widespread use, many web services standards have emerged. These standards address shortcomings in the early standards and extend web services in numerous ways.

None of the above-mentioned and newly evolved standards has explicitly addressed the field of web services input validation. Several experts on web security, among them the Open Web Application Security Project (OWASP), consider unvalidated input to be the greatest threat to web applications [2].

Mechanisms that secure web services against unwanted input are therefore of high importance. Preferably, such protection should be simple to add and its correctness and coverage should be easy to verify.

Commonly, software developers are not security experts [3]. Software security is in general a difficult area to master; this also applies to web services security. Additionally, web services security standards have yet to mature, and keeping up-to-date with the different standards and specifications can be challenging.

There are many advantages to letting experts take care of the security aspects of an application, the foremost being that the quality of the security-related decisions can be expected to be higher than those made by ordinary developers.

In this thesis we propose a verifiable solution for input validation of web services that can be configured by those responsible for security at deployment time.

1.1 Motivating Scenario

The following scenario illustrates the need for the input validation mechanism discussed in this thesis.

A small IT company develops banking applications. In order to facilitate integration of their solutions with other suppliers' systems, web services are used.

Most of the developers in the company's development department have considerable development experience. However, only some of them have experience from implementing security.

An analysis of the company's existing web services has revealed that they were developed with varying levels of security. Input validation tends to be neglected or implemented poorly. Not surprisingly, it turns out that those developers familiar with security have taken the most security considerations during development. The rest of the developers have either introduced security vulnerabilities, or not addressed security issues at all. Security issues have not been under centralized control.

In order to meet the requirements from the banking industry, the IT company has been reorganized. The development department has been split into two separate divisions: one development division and one security division. The security division monitors the developers and adds security mechanisms when the developers have finished their work.

The security division is now looking for a security mechanism that will allow them to add input validation to new and existing web services at deployment time.

1.2 Problem Statement

From the scenario above we extract the following problem statement:

We want to design and implement an automatic, verifiable, and centralized mechanism for web services input validation that can be added to and configured for a web service at deployment time.

1.3 Related Work

Web application input validation is not a new field. In [4] and [5], emphasis is put on SQL injection, which is only a subset of all input validation problems.

Much research has been done in the field of monitoring and protecting web applications. Sirer and Wang [6] and Ardagna et. al. [7] both consider access control issues. Kruegel and Vigna [8] look at the monitoring of web services misuse by detecting usage anomalies, while Baresi et. al. [9] investigate policy-based monitoring of web service compositions.

There exist several attempts at developing formal semantics for web services [6, 10, 11]. In the two former papers, this formalization is used for specifying security policies. In [12], Bhargavan et. al. make use of the semantics developed in [10], and create a tool for verifying web services security policies.

1.4 Approach

In our introductory research, books, technical documentation, standards, and specifications will be used as our main sources for information. Also, reputable publishers such as IEEE, ACM, and Springer will be used as starting points when searching for additional background information.

Next, our problem statement will be broken down into a series of specific system requirements. The requirements will then be used as the basis for system analysis and design.

Last, a prototypical implementation of the design will be developed. The implementation will be elaborated by means of an example application.

1.5 Thesis Outline

The background of our thesis is found in chapters 2-4. Here, XML, web services, web services security, and input validation are introduced.

Next, we proceed to the system development of our solution. In chapter 5, "System Requirements", we state the requirements of the solution. In chapter 6, "System Analysis", we analyze the requirements and describe an implementation-independent architecture of the solution. In chapter 7, "System Design", implementational decisions are presented, and the result is a detailed design. In chapter 8, "Implementation", we describe our implementation of the design. In chapter 9, "An Example Application", we demonstrate the behavior of the system in an example application.

Conclusions are drawn in chapter 10. Here we also list this thesis' contributions and suggestions for future research.

Part I

Background

Chapter 2

XML and Web Services

This chapter introduces web services as well as XML, which is the corner stone in web services technology. Thus, understanding XML is important for understanding web services. Further, the XML schema standard, which will be introduced in this chapter, provides a standard means for validating XML data.

2.1 XML

The Extensible Markup Language (XML) is a general-purpose W3C-standard for document markup [13]. The standard defines a human-readable language that is flexible enough to let anyone define their own document structures using what is called *elements* and *attributes*. Both will be explained shortly. Although flexible in some regards, the standard is quite strict in other areas [14]. For instance the grammar, which defines element and attribute placement, legal names, and more, is very detailed. A document that adheres to this grammar is called *well-formed*.

2.1.1 Elements

The main building-block of an XML document is the element. An element is a container that can have a value and zero or more attributes. Also, an element may contain other elements. An element consists of two corresponding start and stop definitions. An example is displayed below. The start and stop elements are on lines 1 and 3, respectively.

```
1 <exampleElement>
2   element value
3 </exampleElement>
```

2.1.2 Attributes

In addition to using elements, one can also use attributes to contain information in an XML document. The main difference from an element is that an attribute can only have a value and cannot contain other attributes or elements. An attribute is always contained inside an element. Extending the previous example, we get the following:

```
<exampleElement anAttribute="attribute value">
   element value
</exampleElement>
```

A simple example of an XML document with both elements and attributes is shown below:

```
1 <?xml version="1.0"?>
2 <person sex="male">
3    <firstName>Peter</firstName>
4    <lastName>Pan</lastName>
5    <birthYear>1970</birthYear>
6 </person>
```

The first line defines that the document is XML version 1.0. Next, a person element is defined beginning on line 2 and ending on line 6. The person element has an attribute called “sex” with the value “male” Inside the element we also find three other elements ("firstName", "lastName", "birthYear"), each with individual values.

2.1.3 Namespaces

According to Harold & Means, namespaces have two purposes in XML [14]:

- To be able to distinguish elements or attributes from different vocabularies that happen to have identical names.
- To group related elements and attributes.

Elements and attributes from different namespaces can appear in the same document. This may happen if a document from one namespace is nested inside a document from another namespace. A namespace is simply a prefix prepended to element and attribute names, with a colon separating the two. A namespace is often formed as a URI[1], and as it can be rather long, it is possible to define a shorthand version of it inside the XML document. If the namespace for the previous example was "http://www.example.org", and "ex" is used as the shorthand version, the document would be:

```
<?xml version="1.0"?>
<ex:person ex:sex="male" xmlns:ex="http://www.example.org">
   <ex:firstName>Peter</ex:firstName>
   <ex:lastName>Pan</ex:lastName>
   <ex:birthYear>1970</ex:birthYear>
</ex:person>
```

2.1.4 XML Schema

In order to restrict what information an XML document can contain, it is possible to create an *XML Schema*[16] for it. A schema can be used to specify what elements and attributes a document can contain, the relationships between them, and what values they can take.

[1]A URI is a string that identifies a web resource. URIs are defined as a superset of URLs [15].

A schema for the previous XML document can be:

```
<?xml version="1.0"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.org">
   <xs:element name="person">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="firstName" type="xs:string" />
            <xs:element name="lastName" type="xs:string" />
            <xs:element name="birthYear" type="xs:integer" />
         </xs:sequence>
         <xs:attribute name="sex" type="xs:string" />
      </xs:complexType>
   </xs:element>
</xs:schema>
```

In the definition above, a complex type is defined inside the person element. When an element has attributes and nested elements, these must be specified within a complex type. The nested elements are defined as a sequence, which means that the elements in an XML document that adheres to the schema must appear in the specified order. "birthYear" is defined as an integer, while "firstName", "lastName", and "sex" are defined as strings. The namespace that the schema defines is specified in the "targetNamespace" attribute.

The schema could be made more sophisticated, e.g. by adding a restriction on the values for sex ("male" or "female"), or specifying that a year of birth must be 1900 or later. The main use of XML schemas is to validate XML documents. A document that adheres to the XML grammar as well its associated XML schema is said to be both *well-formed* and *valid.*

2.1.5 XPath

In addition to validating XML documents, it is also important to be able to query XML documents in order to retrieve information that fulfills certain criteria. The XML Path Language (XPath) [17] is a language developed for this purpose. Using what is called XPath expressions one can select a particular subset of the elements and attributes of an XML document [18, p122].

2.2 Web Services

A great part of our practical work with this thesis has been associated with web services. Web services are designed to support system interoperability by allowing server-to-server communication over a network. Web services are often associated with protocols such as SOAP, WSDL, and UDDI. These protocols will be further introduced and a general definition of web services will be provided in this section.

2.2.1 Web Services Definition

Cerami [19, p3] defines web services to be any service that is available over the Internet, uses a standardized XML messaging system, and is not tied to any one operating system or programming language. Additionally, even though it is not required, web services should be discoverable and self-describing. When these criteria are fulfilled, the web service can add value to other developers, because they become able to discover the service and invoke it from their applications.

2.2.2 Web Services Architecture

The web services architecture can be considered from two different points of view. Both alternatives are described here.

Alternative 1: Description by Role Model

The first alternative for describing the web services architecture is describing different roles in the architecture. Web services include three communicating parts, and each of these parts has its own role. The three roles are:

- **The Client.** The client is the machine which invokes and consumes the web service.
- **The Registry.** In order to use a web service, the client does a lookup in the registry to find the desired service. The registry holds references to web services implemented by different service providers and returns web service addresses to the clients.
- **The Service.** The service is the web service itself. The web service responds to client requests and contains the web service logic.

Figure 2.1 shows a schematic depiction of these roles and how they are related.

Alternative 2: Description by Protocol Stack

When the three web services roles are to be implemented, three systems have to communicate using the same protocol stack. The second alternative for describing the web services architecture is outlining the protocol stack. The stack is depicted in figure 2.2.

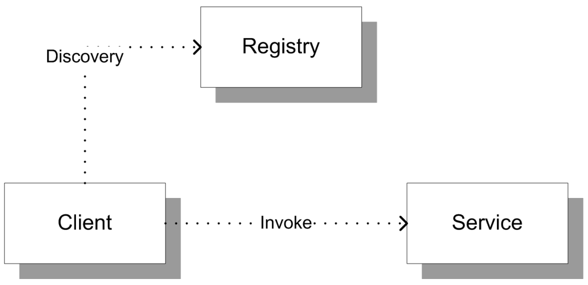


Figure 2.1: Web Service Roles

Different references describe different protocol stacks with different level of detail. The stack presented in figure 2.2 is extracted from [20] and [21]. This protocol stack provides a sufficient level of detail for illustrating the web services architecture in this section.

Web services run on the OSI application layer. Therefore, all layers of the web services protocol stack are situated on the OSI application layer. The layers are:

- **The Service Transport Layer.** Situated at the bottom of the web services protocol stack is the service transport layer. This layer's main task is transporting messages between applications. Traditional protocols such as HTTP over TCP or UDP are common choices on this layer, but, in principle, a number of other protocols can be used (e.g. FTP and SMTP). HTTP, TCP, and UDP will not be further described. For more details on these protocols, Tanenbaum [22] is an excellent source.

- **The Messaging Layer.** This layer handles XML parsing and encoding. The XML formats play a major role when web services are used for achieving interoperability between heterogeneous end-systems. SOAP is the most common protocol choice on this layer, and will be further examined in section 2.3.

- **The Discovery and Description Layer.** This layer exists side by side with the Security layer. The discovery part of this layer implements the lookup functionality of the registry role described above. The UDDI protocol is the natural choice for this functionality. The description protocol defines the interface to the web service, and the WSDL protocol is typically used for this purpose. Both the UDDI and WSDL protocols will be further described in coming sections.

- **The Security Layer.** The security layer exists side by side with the discovery and description layer. This layer adds security services such as authentication and confidentiality to the messages.

- **The Application Layer.** The top layer of the web services stack consists of the applications that use web services.

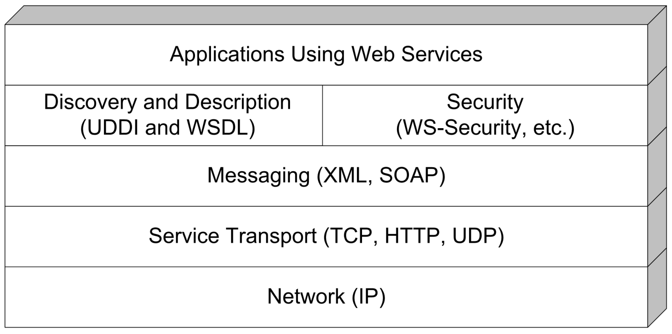

Applications Using Web Services
Discovery and Description
(UDDI and WSDL)
Security
(WS-Security, etc.)
Messaging (XML, SOAP)
Service Transport (TCP, HTTP, UDP)
Network (IP)

Figure 2.2: Web Service Protocol Stack

Throughout this chapter, the different protocols used in our work are further described.

2.3 XML Messaging Using SOAP

The SOAP protocol is developed for passing XML messages between computers. The web services architecture strongly depends on XML messaging. SOAP and web services are thus tightly interconnected.

SOAP is mainly developed for remote procedure calls over HTTP. The SOAP functionality may look like the functionality of middleware frameworks such as CORBA. Unlike similar frameworks, SOAP uses plain XML for all messages and is therefore language and operating system independent. This independence makes SOAP excellent for system integration processes. SOAP is a natural choice for XML messaging when e.g. a java platform system is to be integrated with a system developed on the .NET platform.

A sketch of the SOAP message format is shown in figure 2.3. The SOAP message must always include a SOAP envelope, inside which all data is placed. Inside the envelope, a body element is required. Additionally, header and fault elements can be optionally included in the envelope.

For further details and practical examples on the SOAP message format, the reader is referred to appendix A.

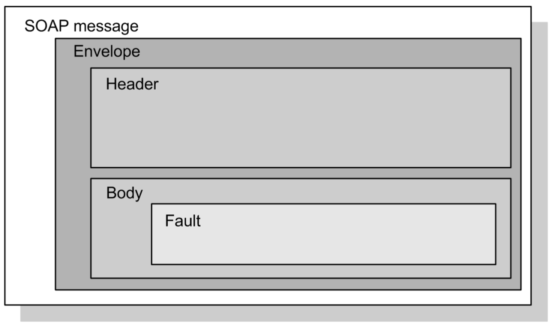


Figure 2.3: SOAP Message Format

2.4 Describing Web Services with WSDL

The Web Services Description Language (WSDL) is an XML standard that describes web services. WSDL describes the operations and messages used for a web service. A WSDL document thus represents the interface for a given web service and leaves the implementation of the web service to the developer. WSDL is language and platform independent and is commonly used for describing SOAP services.

The WSDL standard includes definitions of several elements, including the definitions element, the types element, the message element, the port-type element, the binding element, and the service element.

Our work does not deal with details of the WSDL document. Readers can get more details about the WSDL document in appendix A.

2.5 Describing, Discovering and Integrating Web Services

The Universal Description, Discovery and Integration (UDDI) technical specification provides a standard method for giving publicity to and discovering of web services. However, UDDI is not included in our work, and is just mentioned here because of its tight connections with web services. Some details of UDDI are found in appendix A

2.6 Summary

This chapter introduced the basic building blocks for the work presented by this thesis. The protocols and standards presented here will be referred to in the following chapters.

XML is a standard that defines a human-readable language that lets anyone define their own document structure using elements and attributes. XML documents can be validated against an XML Schema, which is used to specify what elements and attributes a document can contain. XPath is a language for querying an XML document.

Web services provide a standard means for server-to-server communication over a network. Protocols such as SOAP, WSDL and UDDI are often associated with web services. All of these protocols were briefly presented in this chapter.

The next chapter introduces several security aspects related to web services, and is the second of three background chapters.

Chapter 3

Securing Web Services

When developing a security solution, it is important to keep in mind that a chain is only as strong as the weakest link. Although our goal is to add input validation to web services, we want to examine the existing security mechanisms for web services, for instance those providing confidentiality and authenticity. An important reason for this is that we want to understand the whole web services security picture. This understanding may prove beneficial when we start developing our system in part II.

Web services may expose business critical systems and information, and it is crucial that a proper level of protection is applied. Security was not addressed in early web services standards, but has been gradually added later on, through a plethora of standards.

This chapter consists of three parts. In the first, transport-level and messaging-level security is discussed. Then, the core XML and web services security standards are examined. In the last part, web services security policy standards and specifications are examined.

3.1 The Case for Messaging Layer Security

Web services security can be applied both on the transport layer and on the messaging layer. Both methods have benefits and drawbacks. In general, transport layer security is faster and less flexible, whereas messaging layer security is slower and more flexible. The two methods are sometimes used in conjunction for added security when particularly sensitive data is processed and transferred.

3.1.1 Transport Layer Security Insufficiencies

A very common way of securing application data is by using the Transport Layer Security (TLS) protocol [23]. TLS can provide data confidentiality and integrity by means of encryption and hash algorithms. Additionally, by using both client and server authentication (e.g. using digital certificates), the peers can authenticate each other. TLS provides point-to-point protection between two hosts, as illustrated in figure 3.1. Communication is secured between Host A and Host B, and between Host B and Host C.

While ideal for certain web service security scenarios, the protocol is insufficient in more complex environments [24]. We now look at two cases where TLS is insufficient for protecting Web Services.

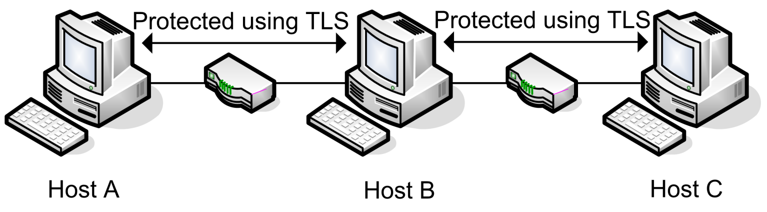


Figure 3.1: Security Applied Using TLS

Case 1 - Intermediary Web Services and Application Firewalls

A common web service scenario involves intermediaries that have to be able to access parts of a message. Application firewalls may need access to the message to decide whether to allow it to pass or not, and intermediary web services may need to process parts of the message. In this case, TLS does not provide enough flexibility as it only provides end-to-end and all-or-nothing protection when TLS is used between the endpoints.

Case 2 - Post-Transit Protection

Also, with transport-level security protocols such as TLS, communication is secured when in transit only. Often it is desirable to protect data in subsequent storage, for instance when persisted in a database. An illustration is shown in figure 3.2, where the data sent from Host A to Host B is only protected between the two hosts, and not if Host B decides to save it in a database.

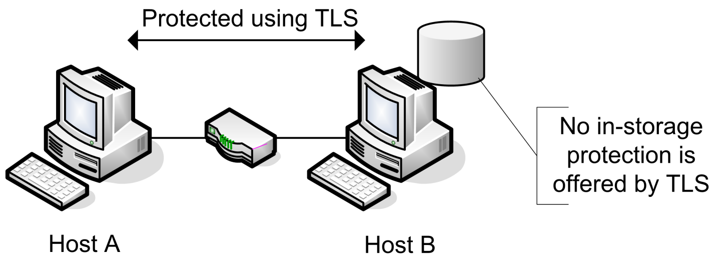


Figure 3.2: TLS Only Offers Data Protection During Transit

3.1.2 Messaging Layer Security

It will now be illustrated how protection can be applied to the two cases by applying security on the messaging level.

Case 1 - Intermediary Web Services and Application Firewalls

One of the main benefits of applying security on the messaging level is the flexibility this gives. For instance, security can be applied to selected parts of messages, making it possible for intermediate web services or application firewalls to access parts of it. Please see figure 3.3 for an illustration.

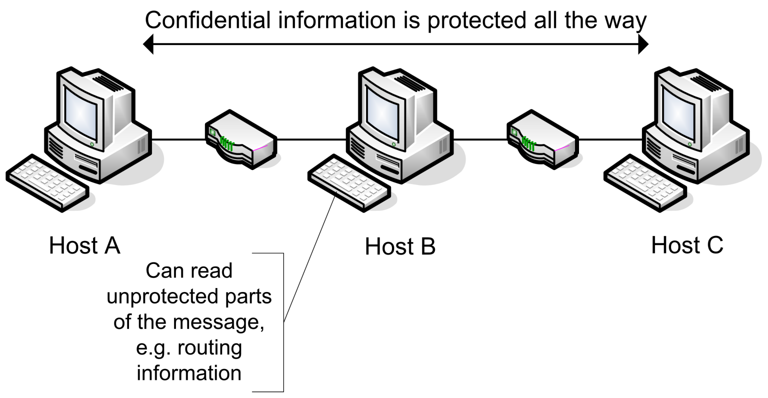

Confidential information is protected all the way
Host A
Host B
Host C
Can read unprotected parts of the message, e.g. routing information

Figure 3.3: Messaging Layer Security Allows Flexibility of Message Protection

Case 2 - Post-Transit Protection

Additionally, when security is applied to messages this means that the messages easily can be stored in a protected state. This principle is shown in figure 3.4.

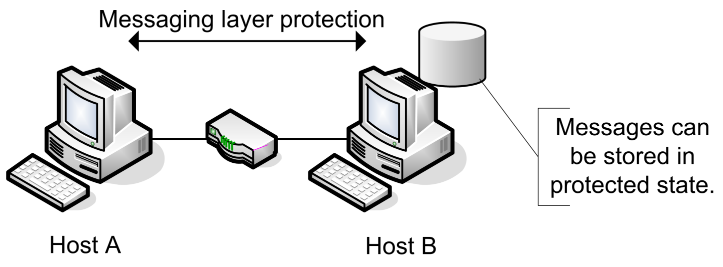

Messaging layer protection
Host A
Host B
Messages can be stored in protected state.

Figure 3.4: Messaging Layer Security Offers Data Protection Also After Transit

3.2 Core Security Standards

Whereas TLS was used to secure the transport layer, a set of security standards is used to apply security on the messaging layer. Next, the foundations of web services security will be presented. First, XML Encryption and XML Signature are discussed. Then, WS-Security, which extends the two XML standards, is examined.

3.2.1 XML Security Standards

The World Wide Web Consortium (W3C) defined two standards for XML security which are employed by web service security standards; XML encryption and XML signature. As the names imply, these standards define methods for encryption and signing of XML documents. The standards can be combined and applied with a high degree of flexibility, and one can, for instance, sign or encrypt only certain parts of an XML document.

XML Encryption

The XML Encryption standard [25] specifies how to encrypt arbitrary parts of XML documents using well-known encryption algorithms such as triple DES [26] and AES [27]. The standard also defines methods for key agreement and key exchange. The operation of XML Encryption is shown in Appendix B.1.

XML Signature

Complementing the encryption standard, the XML Signature standard [28] specifies how to ensure integrity and authenticity of XML messages using standards such as SHA-1 [29] and HMAC [30]. Sender authenticity is ensured using digital signatures (RSA [31] or DSS [32]). The operation of XML Signature is shown in Appendix B.2.

3.2.2 WS-Security

A central standard for web services security is WS-Security. It was originally developed by Microsoft, Verisign, and IBM, and was published in April 2002 [33]. It was later submitted to the Organization for the Advancement of Structured Information Standards (OASIS), which continues to work on the standard. OASIS released version 1.0 in April 2004, and version 1.1 was approved in February 2006.

Microsoft, IBM and other cooperating organizations have later developed a set of security specifications building on WS-Security. Among these are WS-Policy and WS-SecurityPolicy, which will be covered later in the chapter. Many of these specifications have been submitted to OASIS for standardization.

The standard specifies XML document details, and leaves most information related to implementation (such as algorithms used) up to the implementors. As Bhargavan et. al. point out, it "focuses on interoperability details rather than security" [34].

Basically, WS-Security is an extension of SOAP, adding support for authentication, integrity and confidentiality to SOAP messages. Confidentiality and integrity are achieved using the aforementioned XML Encryption and XML signatures, respectively. WS-Security also specifies how to convey authentication information in an XML message using what is called a security token.

According to the WS-Security standard [35], a security token is a collection of claims, which are declarations made by an entity regarding name, identity, etc. A signed security token is a security token that has been cryptographically signed by an authority. A security token may contain authentication information, but may also be used for authorization details, for instance.

When it comes to authentication, there exist specifications of several kinds of security tokens, some which can be found in additional profiles. There is support for user name/password, X.509 certificates, and Kerberos authentication, but the model is generic and extensible and allows for other authentication mechanisms as well.

3.3 Security Policy

The security policy is a very important part of the security of any information system. The SANS Institute defines a security policy as [36]:

> *A set of rules and practices that specify or regulate how a system or organization provides security services to protect sensitive and critical system resources.*

In a web services context, a security policy may define who can access the web service and how information is protected to and from the web service. Building on WS-Security, several web services security policy specifications and standards have been developed. These allow web service developers to formalize and communicate the web service policy. The mechanisms also let clients discover web service policy requirements and to make decisions on which services to use and how to use them, based on their associated security policies. A generic model can be seen in figure 3.5 (adapted from [37, p131]).

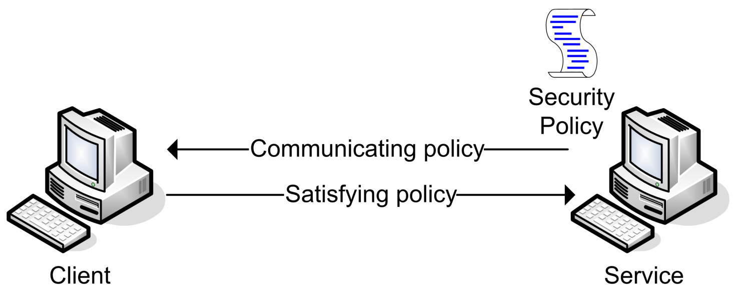


Figure 3.5: A Generic Security Policy Model

3.3.1 WS-Policy

WS-Policy was developed by a group of companies including Microsoft, IBM, and VeriSign, and specifies a generic model for describing web services security policies [38]. It defines a policy as a set of alternatives, and a client has to satisfy one of these alternatives. Each alternative in turn consists of a set of assertions that specify requirements that the client has to meet. An illustration of this model is shown in figure 3.6.

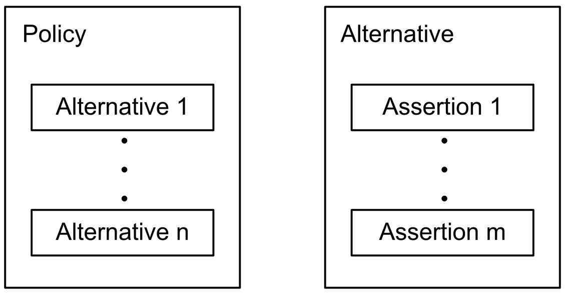


Figure 3.6: Policy as Defined by WS-Policy

3.3.2 WS-SecurityPolicy

Work on WS-SecurityPolicy was initiated by VeriSign, IBM, Microsoft, and RSA Security [39]. It was later handed over to OASIS, and as of February 2006 it has status as a working draft[1].

WS-Policy specifies a policy structure. However, it does not define any particular types of assertions, and leaves this for other specifications. WS-SecurityPolicy is such a specification, and extends WS-Policy by defining a set of security-related assertions. Hence, WS-Policy and WS-SecurityPolicy jointly constitute a framework for creating web services security policies. The assertions specified by WS-SecurityPolicy can be grouped into two groups:

- Protection Assertions (signing, encryption, required elements)
- Token Assertions (username, X.509, Kerberos, etc.)

The protection assertions state requirements for which parts of the SOAP messages that must be signed, encrypted, and present. Token assertions are used to specify authentication information.

[1]As the OASIS version of WS-SecurityPolicy is still on the draft stage, this paper is based on the v1.1 specification by IBM, Microsoft, RSA Security, and VeriSign.

In addition to specifying security assertions, WS-SecurityPolicy also develops WS-Policy further and adds some flexibility regarding nesting of policies and assertions.

3.4 Summary

This chapter, the second of three background chapters, introduced web services security aspects. First, it was discussed why it is sometimes desirable to apply security on the messaging layer, instead or in addition to security on the transport layer. Then, three fundamental XML and web services security standards - XML Encryption, XML Signature, and WS-Security - were introduced. Last, the thesis discussed security policies in a web services context, and presented the WS-Policy and WS-SecurityPolicy standards.

In the following chapter, we focus on a central topic for this thesis, namely input validation. Different approaches and concepts are introduced, and the chapter concludes the background part.

Chapter 4

Input Validation

Accepting unvalidated input from the user is considered the greatest threat of a web application by several influential organizations and scientists [2, 40]. By validating input data, one wants to prevent unexpected input data from harming the web service application execution.

Although web services differ from regular web applications in many ways, input validation is similar for both. While traditional web application input is input from users who download a form and post it back to the server, web services receive input from applications. In other words, web applications receive input from users, while web services receive input from machines.

In practice, this difference does not change the need for input validation, as there is no difference between input from machines and input from users. Malicious applications exist as well as malicious users. Thus, one cannot trust web service input more than web application input.

Before defining the requirements for the input validator solution in the next chapter, a brief introduction to key input validation concepts is provided in this chapter.

4.1 Web Service Input

Input to web services can be divided into two categories. The first category is what is usually meant by input, namely input parameters received through the web service interface.

The second category is input from subsystems. Web service applications often communicate with subsystems and receive input from these systems. An example of such subsystems can be a database. To ensure that all input data is valid, web services should also validate database input. This validation can be challenging, especially when several applications write to the same database.

4.2 Defining Valid Data

When designing a validation scheme, one should begin with defining what should be accepted as valid data. As is to be seen in this section, this should be done carefully. This section will describe two approaches to defining valid data called "blacklisting" and "whitelisting".

Alternative One: Blacklisting

The first, and probably most intuitive, approach to the challenge of defining valid data is called blacklisting. Blacklisting is based on maintaining a list of invalid data and compare input data against this list. If input data matches an item in the blacklist, it is considered as invalid.

If one forgets adding an invalid item to the list, the validation will be less strict than intended because invalid data will pass as valid.

Alternative Two: Whitelisting

The second approach is called whitelisting, and is the opposite of blacklisting. Instead of maintaining a list of invalid data, this approach maintains a list of valid data. When validating against such a whitelist one defines all input data which matches an entry in the whitelist as valid.

If one forgets to list an item in the whitelist, the validation will be stricter than intended. Valid data which should have been on the list, but is forgotten, will not pass the validator. This property makes whitelisting more suitable to human developers than blacklisting. Humans will sooner or later forget listing something. Thus, it is often desirable to use whitelisting because the consequences of forgetting are often less serious with this approach.

4.3 Validation in Code Versus Validation Out of Code

The web service developer is faced with a design choice when deciding on where to put the validation logic. The first alternative is writing validation code inside each method that accepts input. Given that the programmer is able and willing to handle validation when programming, doing validation in code is a solution to the input validation challenge. However, maintenance of validation code becomes more challenging.

The second alternative for validating input is building an automatic validation framework. The framework should handle all input to the web application and hide all validation-specific programming from the programmer. By using such a framework, validation is moved out of the application.

Huseby [40] lists several benefits from moving the validation out of the application:

- The application code becomes "cleaner" and the programmer does not need to think of validation when programming.

- The programmer does what he is good at (programming) and leaves security to the security division.

- If an automated out-of-code-framework existed, adding input validation to existing applications becomes easy.
- Maintenance of the blacklist/whitelist is much easier. By centralizing input validation, the blacklist/whitelist can be put into one single file.
- Using dedicated processing recourses for validation will improve the application performance.

4.4 Summary

This chapter introduced some key concepts associated with input validation. Two opposite approaches to validation, called whitelisting and blacklisting, were presented.

When implementing validation, one should consider where to write validation code. The first alternative is writing validation code directly in the application code. This might be sufficient for small applications but does not scale well. The second alternative is making an automated validation framework which can be configured out of code.

The next chapter is the first chapter describing the system development part of this master's thesis work. The chapter specifies system requirements for the web service input validation system.

Part II

System Development

Introduction

This part of the thesis provides a detailed description of a web service input validator. The validator mechanism presented here is developed for use by system administrators or other security staff, and allows adding input validation to a web service at deployment time. Thus, developers are able to focus on implementing business logic rather than considering input validation during programming.

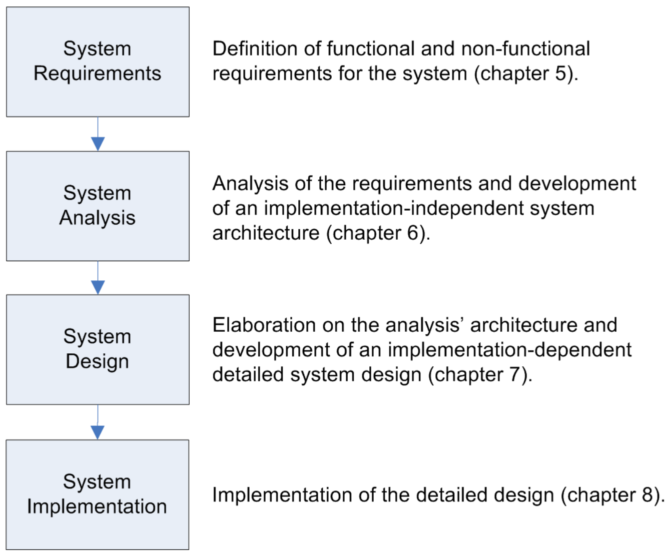

System Requirements
Definition of functional and non-functional requirements for the system (chapter 5).
System Analysis
Analysis of the requirements and development of an implementation-independent system architecture (chapter 6).
System Design
Elaboration on the analysis' architecture and development of an implementation-dependent detailed system design (chapter 7).
System Implementation
Implementation of the detailed design (chapter 8).

Figure 4.1: System Development Approach

An overview of our system development approach is shown in figure 4.1. As indicated in the figure, each of the following four chapters maps to one of the four steps.

Before delving into the solution's requirements, it may be helpful to clarify the context in which our solution will operate. Figure 4.2 shows a principle drawing of a web service and two web service clients accessing the service using IP and a set of web services-related standards. Note that our solution will intercept the incoming messages to the web service, and validate these before letting them through to the web service.

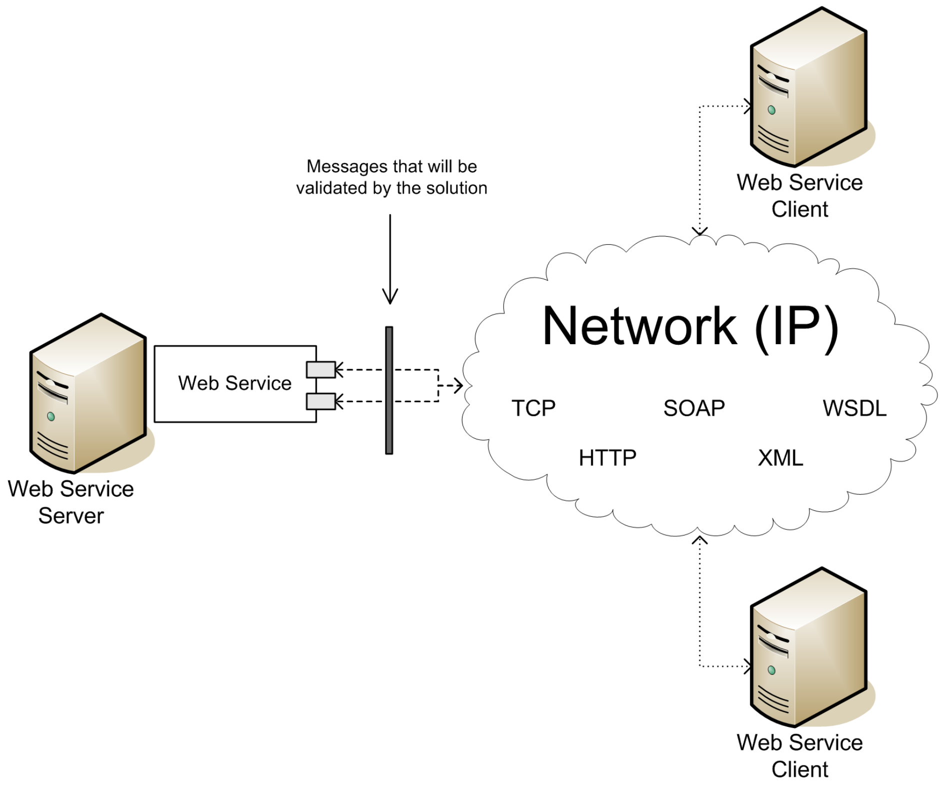


Figure 4.2: Solution Context

Chapter 5

System Requirements

In this chapter the problem statement is used as a starting point for defining the system's functional and non-functional requirements, and for convenience the problem statement from the introduction is repeated:

We want to design and implement an automatic, verifiable, and centralized mechanism for web services input validation that can be added to and configured for a web service at deployment time.

Properties explicitly mentioned in the problem statement are mapped into solution requirements. Additionally, we also add non-functional requirements that are considered important for the solution to function satisfactory.

5.1 Functional Requirements

5.1.1 Requirement 1: Automatic Policy Enforcement

A policy-based approach to specifying web services protection has been chosen mainly because existing web services security standards and specifications (e.g. WS-SecurityPolicy, see Chapter 3) are policy-based.

The system will enforce specified input policies for a defined set of web services and contained operations, called ports. Policy enforcement should be automatic, and should not depend on manual intervention [6]. This principle is necessary for the system to be usable in a practical application; manual enforcement would make the system both prohibitively slow and increase the risk of errors.

5.1.2 Requirement 2: Verifiable Protection

An important aspect of the system is the ability to verify its operation on two levels; coverage and correctness.

Requirement 2.1: Coverage

It must be possible to ensure that all web services that the system administrator intends to protect are in fact protected.

Requirement 2.2: Correctness

Additionally, the system administrator must be able to verify that the specified policy definitions work as intended.

5.2 Non-functional Requirements

5.2.1 Requirement 3: Maintainability

Requirement 3.1: Separation of Policy Definition and Enforcement

Separation of security policy and enforcement is a generally accepted principle [41]. This allows the policy to be modified without having to change the enforcement mechanism. The solution should therefore separate policy definition and enforcement.

Requirement 3.2: Separation of Web Service and Policy Definition

Leaving the definition of security policies as a deployment consideration is valuable for several reasons. For instance, developers can continue developing web services as before. Security policies can then be added by others at deployment time. The input validation solution should therefore support separation of web service and policy definition.

Requirement 3.3: Centralized Policy Definition

In order to simplify policy updates and facilitate reuse of policy definitions, a centralization of policy definitions is called for. This way, common policy definitions can be shared between several web services. As each definition is defined only once, there is no problem with keeping different definition instances synchronized.

5.2.2 Requirement 4: Security

Requirement 4.1: Non-Bypassability

For the policy mechanism to be effective, it is important that it cannot be bypassed. One must endure that every web service request is checked for policy compliance.

Requirement 4.2: Secure Policy Storage

It is vital that the security policy is protected against tampering. The validation mechanism will be compromised if an attacker succeeds in modifying the policy. The policy should be protected against adding of new entries and modification of existing entries.

Unauthorized reading of the policy file also represents a security threat and should be denied. The main threat associated with giving attackers read-access to the policy file is that the attacker becomes able to discover configuration weaknesses introduced by system administrators. Thus, the system would no longer be a black box from the attacker's point of view.

It must be ensured that the policy itself is stored by a trusted party. Also, unauthorized replacement of the policy must not be allowed.

Requirement 4.3: Whitelisting

In accordance with the theory on whitelisting and blacklisting, mentioned in section 4.2, we want our solution to employ the whitelisting technique for validating input.

Requirement 4.4: Logging

The solution must be able to log important events, such as program errors and security incidents, to persistent storage.

For the log to be used for non-repudiation purposes, it must be tamper-proof.

5.2.3 Requirement 5: Platform Independence

The solution must be platform independent in order to support heterogeneous server environments. It is not uncommon that enterprises maintain code written years ago [42]. As time goes by, new platforms emerge. Thus, due to different time of development, the applications are most probably developed for, and deployed on, different platforms. Companies tend to be reluctant to rewriting old code because they already have a working, debugged solution. Platform independent solutions are therefore valuable.

Web services are widely used for integrating new and existing systems. By using the XML message format and standardized protocols, web services are designed to be platform independent. Accordingly, the same should apply for the input validation mechanism.

5.2.4 Requirement 6: Modularity and Encapsulation

Encapsulation on the object-level is a combination of two aspects: grouping of object state and operations, and limiting access to an object's data only through its interface [43]. In addition to using object-level encapsulation, we also want grouping (also called modularity) and encapsulation on the the module-level in our solution. In general, encapsulation and modularity make both development and testing easier.

5.2.5 Requirement 7: Flexibility and Extensibility

The system must be flexible enough to allow for a wide range of different policy definitions. Furthermore, it is important that the system can be extended when needed. This property goes hand-in-hand with modularity and encapsulation, as they make it easier to replace and add functionality by replacing or adding modules.

5.2.6 Requirement 8: Reusability

Requirement 8.1: Existing Standards and Specifications

When possible, the solution should use existing standards and interfaces. This requirement is tightly connected to compatibility and platform independence, but some additional remarks should be made. Reuse reduces development costs by avoiding rethinking and reimplementing others' thoughts. Furthermore, support from existing development environments and tools saves time and money.

Training costs will also be reduced when existing standards, protocols, and interfaces are used. If developers have experience with the standards, the threshold for making use of the solution should be low.

Developers should also be able to verify that the given security solution provides the intended security. Verification of the solution becomes easier when well-known and trusted standards are used.

Requirement 8.2: Existing Functionality

In addition to reusing existing standards and specifications, the solution should reuse existing implementations whenever possible. This is particularly important in the case of validation functionality, in which a programming error could severely threaten the security of the entire solution.

Although not typically non-functional, this requirement is listed here because it is closely related to the non-functional reusability requirement.

5.2.7 Requirement 9: Performance

The verbose XML format generates overhead compared to more compressed formats (e.g. binary encoded data). There is no doubt that other middleware platforms outperform web services when it comes to performance. Simulations have found that the XML based SOAP protocol used for web services performs seven times poorer than the CORBA platform [44]. The simulations state that delay associated with SOAP stems mainly from parsers and the implementation of the communication handlers.

In order to avoid adding significant delays to the web service, all communication messages for the input validation solution should run internally on the server.

5.3 Summary

In this chapter, the problem statement was used as a basis for defining the system's requirements.

The functional requirements say that the system should protect web services using automatic and verifiable policy enforcement. Additionally, the non-functional requirements tell us that the system should be easily maintained, secure, platform independent, flexible, extensible, and modular. Also, it was stated that existing functionality should be reused when possible, and that performance - although not our prime concern - should not be prohibitively poor.

The stated requirements will serve as the basis from which decisions are made in the next chapter's analysis.

Chapter 6

System Analysis

In this chapter we analyze the requirements that were listed in chapter 5, and outline a solution that satisfies them. In order to check that all system requirements are satisfied, a system requirements traceability matrix is developed. The blank matrix is shown in table 6, and the completed matrix is provided in the last section of the chapter.

The solution architecture is developed step-by-step throughout this chapter. The architecture will then be used for creating the implementation-specific design in the next chapter.

6.1 The Initial System

We start the analysis by showing an illustration of the system to which we intend to add our solution. For a summary of the solution context, the reader is referred to the introduction of the system development process on page 33.

Figure 6.1 shows a set of web services, each with web service ports, situated on a given web server. When a web service request arrives on the web server, it is forwarded to the correct web service and port.

	Component 1	Component 2	. . .	Component n
Req 1.0: Automatic Policy Enforcement				
Req 2.1: Coverage				
Req 2.2: Correctness				
Req 3.1: Sep. of Policy Def. and Enforcement				
Req 3.2: Sep. of Web Service and Policy Def.				
Req 3.3: Centralized Policy Def.				
Req 4.1: Non-Bypassability				
Req 4.2: Secure Policy Storage				
Req 4.3: Whitelisting				
Req 4.4: Logging				
Req 5.0: Platform Independence				
Req 6.0: Modularity and Encapsulation				
Req 7.0: Flexibility and Extensibility				
Req 8.1: Existing Standards and Specs.				
Req 8.2: Existing Functionality				
Req 9.0: Performance				

Table 6.1: Blank Requirements Traceability Matrix

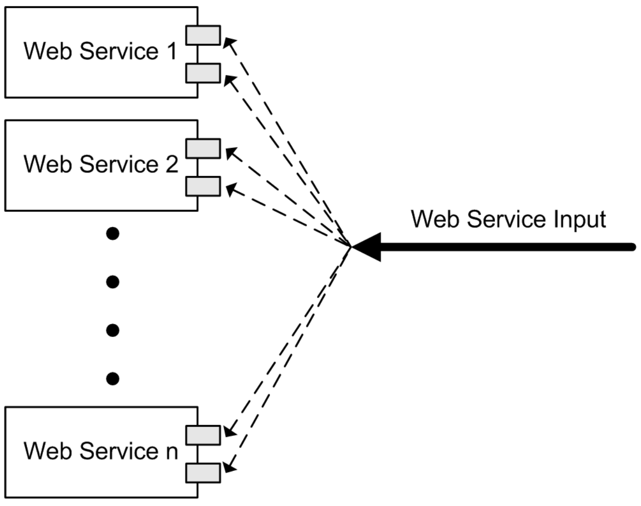


Figure 6.1: Initial System Architecture

Our goal is to add a mechanism that validates that the values of the parameters within each web service request are in accordance with a specified security policy.

The first step is to separate web services, policy, and policy enforcement, described in section 6.2. The high-level model described in 6.2 is further extended for allowing reusability in 6.3. Section 6.4 explains the solution more in depth by dividing each of the components into subcomponents. In 6.5 we introduce logging and security to the system. In section 6.6, the remaining components, called correctness and coverage, are described. After each of the above-mentioned steps, a depiction of the solution at that point is provided.

6.2 Step 1: Separating Web Services, Policy, and Policy Enforcement

The requirements section dictates separation of the web service from the policy and policy enforcement. In order to fulfill this requirement, we want to divide the system into three parts. The first part contains the web services, the second holds the policy, and the third enforces the policy. These parts will be further explained below. The result of this step is depicted in figure 6.2.

6.2.1 Web Services

This part of the system contains the web services that will be protected. Each web service contains one or more ports. The web service ports represent the publicly available web service methods, where anyone in possession of the web service address is able to inject data. The web service name must be unique, and within each web service all port names must be unique. All

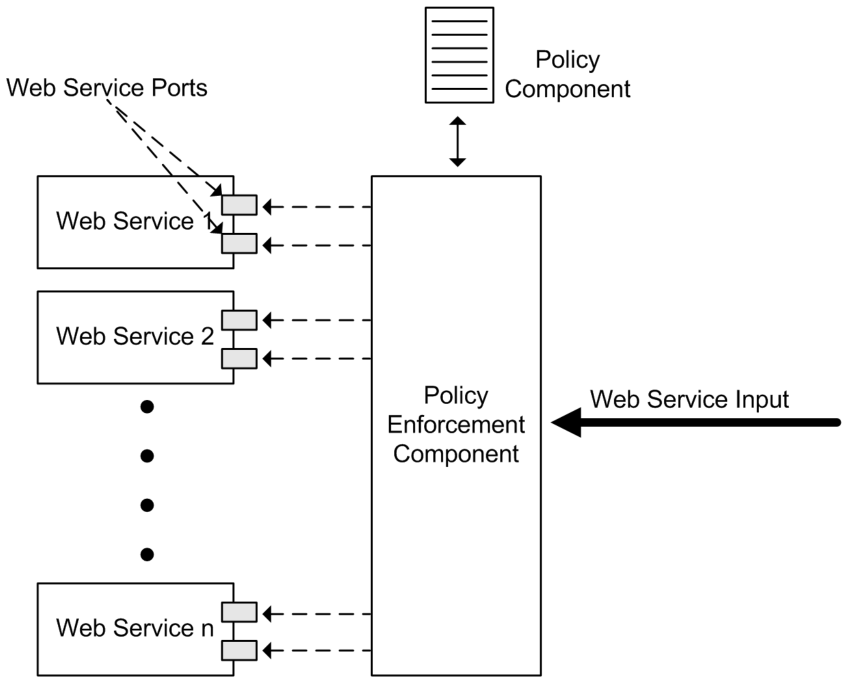


Figure 6.2: System Architecture After Step 1

input to the web service ports should be validated against the policy definitions held by the policy component described below.

6.2.2 The Policy Component

The policy component holds a repository of the policy definitions. For now we do not care about secure storage of the repository. The policy component must have the ability to browse through the content of the repository. Further, it must accept incoming policy definition requests and respond with the requested policy. It is important that only authorized requests - those made by the policy enforcement component - must be accepted. A typical request will ask for the policy definition belonging to a given web service and port. The policy component must therefore accept web service name and port name as input parameters. Upon receiving input, the component must look up the policy definition matching these parameters and return it to the sender.

6.2.3 The Policy Enforcement Component

The policy enforcement component performs the validation and is therefore considered the main component in this project. According to the requirements, policy enforcement should be executed automatically. Thus, the policy enforcement component should not require human decisions or user interactions when deciding whether input is valid or not.

Another important requirement associated with the enforcement component is non-bypassability. In order to meet this requirement, the component must intercept and validate every incoming web service request before it is handed over to the web service.

Each incoming request has the name of the web service and port name embedded in the request. For each incoming request, the policy enforcement component must send a request to the policy component in order to get the correct policy definition. After receiving the policy definition from the policy component, the policy enforcement component validates the incoming data. If validation succeeds, the incoming web service request should be handed over to the correct web service. Otherwise, the request should be rejected. For security reasons we do not want to provide the message sender with more details than necessary, so the policy enforcement component only replies with a message stating that validation failed.

6.3 Step 2: Reuse

Reuse of existing standards, specifications, and functionality was emphasized in the requirements. The proposed solution will incorporate reuse in several ways; this is further described in the sections below. The system architecture after adding reuse functionality is shown in figure 6.3.

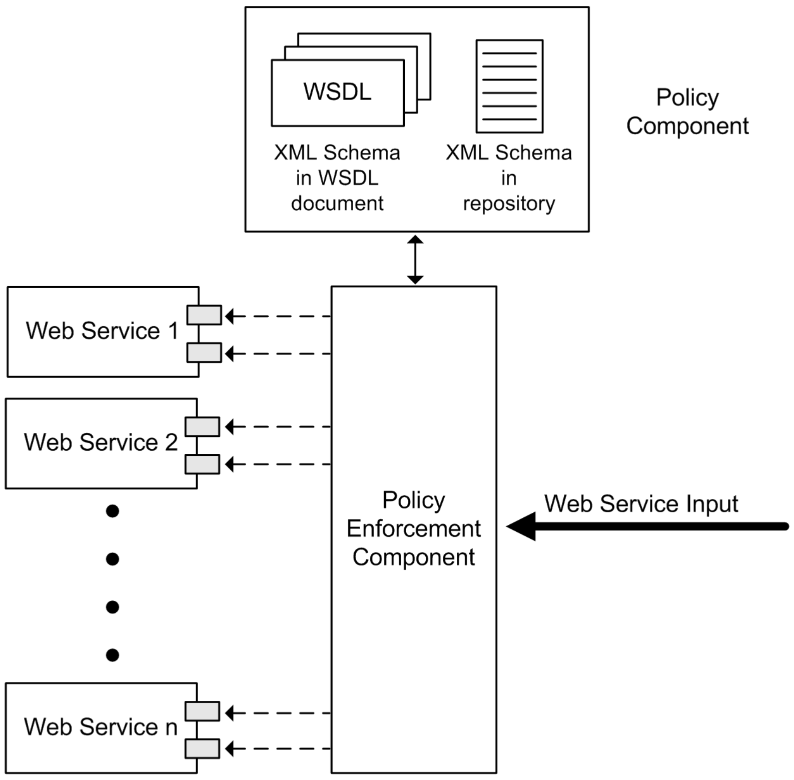


Figure 6.3: System Architecture After Step 2

6.3.1 Using XML Schema for Validation

In our solution, an XML Schema, introduced in section 2.1.4, is used for validation of incoming web service requests. In addition to the benefits of reusing an existing standard (for details, please refer to section 5.2.6), this means that existing XML validation functionality can be employed.

6.3.2 Policy Definition Reuse

The second level on which we want to enable reuse is by creating a repository of policy definitions that can be shared by several web services. This way, already-existing, mature, and well-tested definitions can easily be reused by new web services.

6.3.3 Policy Definition Flexibility

As mentioned above, we validate input against XML schemas. The requirements dictate that existing functionality should be reused. Our solution should make use of the fact that WSDL documents can contain important validation information in XML Schema format. Most web services have a WSDL document associated with them and, thus, making use of the WSDL document XML schemas can be a time-saving factor for developers. It should not be necessary to rewrite the XML Schema in the policy component's repository if a proper schema already exists in the WSDL document. Before describing the XML schema reuse logic, we discuss the need for the possibility to retrieve XML schemas from both WSDL documents and the repository.

Why not Use Only WSDL?

It might seem like retrieving the XML Schema from the WSDL document would be sufficient, and that the repository therefore is unnecessary. We want to emphasize that retrieving the XML schema from a repository is needed in several cases and for several reasons:

- Not all web services have associated WSDL documents. One reason is that not all enterprises publish their WSDL documents due to fear of exposing infrastructure.
- Basing a solution on WSDL definitions alone would result in limited flexibility because policy reuse becomes difficult.
- WSDL definitions are tied up to the WSDL standard. It would therefore be challenging to extend the policy definitions.

Due to the above-mentioned reasons, we have chosen to retrieve XML schemas from both the repository and the WSDL document. Thus, the policy component is extended with functionality that makes validation against the WSDL XML schema possible.

WSDL XML Schema Reuse Logic

The logic behind allowing existing WSDL XML Schema definitions to be reused is illustrated in figure 6.4. Note that the web service request heading for a given port is intercepted by the policy enforcement module. Then, both WSDL and repository schema definitions for this port are retrieved from the policy component. Next, all input parameters for this port are validated against either the WSDL or the repository schema. To allow for maximum flexibility, the solution should allow some parameters to have WSDL validation, whereas the rest are validated against the repository schema. If all input parameters are valid, the web service request is forwarded. If not, the request is rejected.

As stated in the requirements, the solution should be composed of encapsulated modules and be extensible. Extending the policy component should

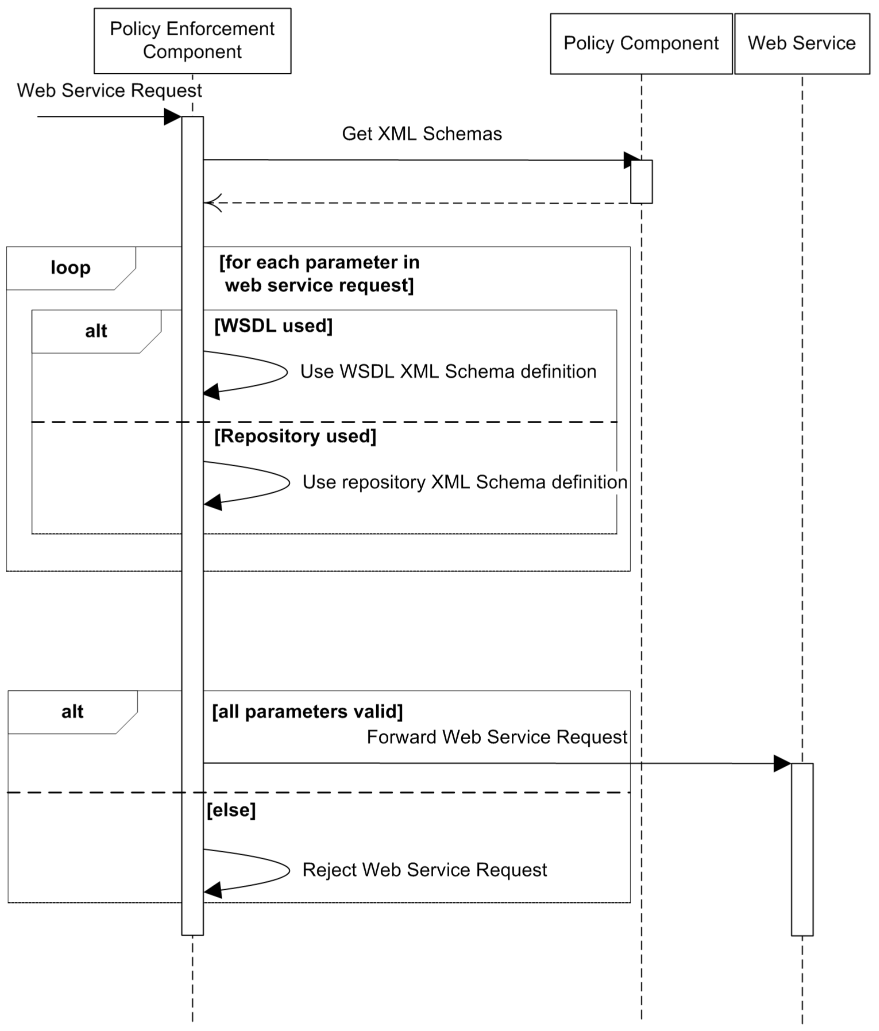

Policy Enforcement Component
Policy Component
Web Service
Web Service Request
Get XML Schemas
loop
[for each parameter in web service request]
alt
[WSDL used]
Use WSDL XML Schema definition
[Repository used]
Use repository XML Schema definition
alt
[all parameters valid]
Forward Web Service Request
[else]
Reject Web Service Request

Figure 6.4: Sequence Diagram Illustrating the Flexibility of Reusing Existing WSDL Schema Definitions

therefore be easy. If, for any reason, future use of the system should require getting the XML schema from other locations than yet supported, it should be convenient to add such extensions.

6.4 Step 3: Modularity

In this step, the modularity requirements are taken into account. An illustration of the result can be seen in figure 6.5.

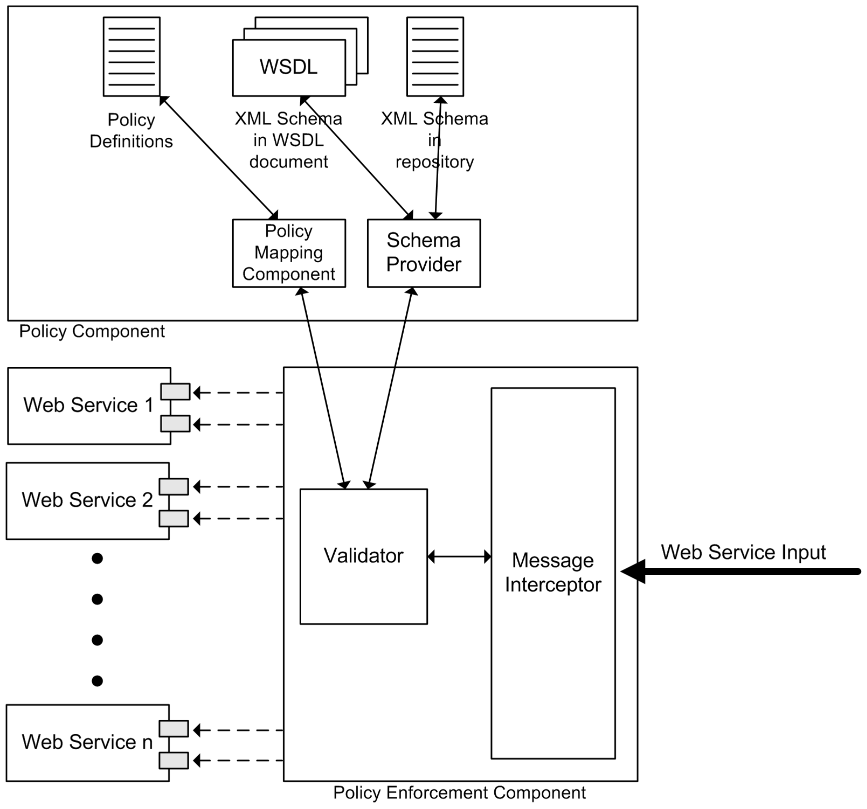


Figure 6.5: System Architecture After Step 3

6.4.1 Modularization of the Policy Component

In order to meet the requirements regarding encapsulation and modularization of the system, we split the policy component into several subcomponents. During the discussion in this chapter it has become clear that the policy component must support retrieving data from a repository as well as the WSDL document that is associated with a given web service. In the previous section, we added a subcomponent for handling each of these ways to retrieve an XML schema.

Now, we add two more subcomponents in the policy component in order to make reuse of policy definitions possible. The two components are described below.

Policy Mapping Component

The first subcomponent is called the policy mapping component and should handle the mapping between web services, ports and policy. The policy enforcement component will request this subcomponent for the policy to use for the incoming web service request. Thus, the policy mapping component takes care of the look-up functionality described in section 6.2.2.

Schema Provider Component

The second subcomponent which is added to the policy component is called the schema provider. Providing the enforcement component with the correct XML schemas will be the main purpose of this subcomponent. For each port's input parameters, the policy enforcement component must request the XML schema provider for a schema. Upon requests, the XML schema provider must look up the XML Schema in either the WSDL document or the repository.

6.4.2 Modularization of the Policy Enforcement Component

To satisfy the requirements, the policy enforcement component should also be modularized. The component has two main purposes. First, it should intercept all incoming web service requests. Second, it should do the validation of the input parameters. We therefore split its functionality into two separate subcomponents.

The first subcomponent is the message interceptor. One must ensure that every incoming web service request is intercepted by the message interceptor. There should be no way to request the web service without going through the message interceptor. It is thus clear that this component handles the non-bypassability requirement.

The second component represents the main logic in the system and is called the validator. The validator component interacts with almost every other component in the system and decides on whether the incoming input is valid or not.

We now draw the whole picture of the validation process from the validator's point of view. The following steps are included:

1. The validator receives the incoming web service request from the message interceptor and reads through the request in order to find the web service name and port name to which the request is addressed.

2. The validator then requests the policy mapping policy component for the proper policy to use for this request.

3. When the validator receives the policy mapping, it requests the XML Schema provider for the correct XML Schema to use for each input parameter in the request.

4. The validator then validates each input parameter against the received schema.

5. At last, the validator decides on whether the input is valid or not. Every input parameter must be valid if the input should be considered valid.

6. When input is valid it should be forwarded to the web service, else it is rejected and should under no circumstances be forwarded. Upon rejection, a simple message stating that validation failed is returned to the sender.

An illustration of the process was shown in figure 6.4.

6.5 Step 4: Security

In this step, we specify how to secure the policy definitions and how to ensure that exceptions are logged. The solution after this step is depicted in figure 6.6.

6.5.1 Secure Policy Against Tampering

As stated in the requirements, it is vital that the policy cannot be changed or read by unauthorized users. On the server, this can be achieved by employing operating system security in such a way that only authorized users are allowed to read, edit, remove, or replace the policy. In addition to protecting the file locally on the server, it is also necessary that there exists no other mechanism for unauthorized users to view or modify the file (e.g. through the web server).

6.5.2 Logging

In order to fulfill the logging requirement, the solution will contain a centralized logging component whose responsibility is to log important events to persistent storage. It is desirable to reuse an existing and stable logging mechanism, but the exact choice will depend on several implementation decisions made in the next chapter.

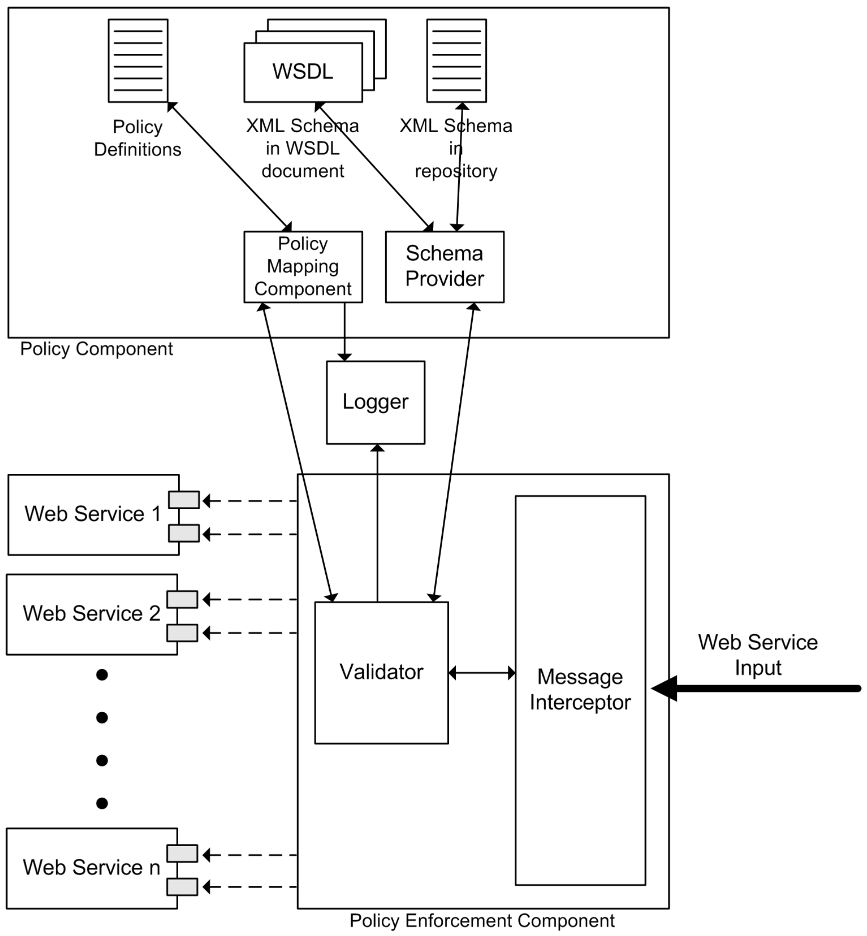


Figure 6.6: System Architecture After Step 4

6.6 Step 5: Verifiable Operation

One of the functional requirements is that the system is verifiable with respect to coverage and correctness. The solution with the verifier added is shown in figure 6.7. The functionality of the verifier will now be discussed.

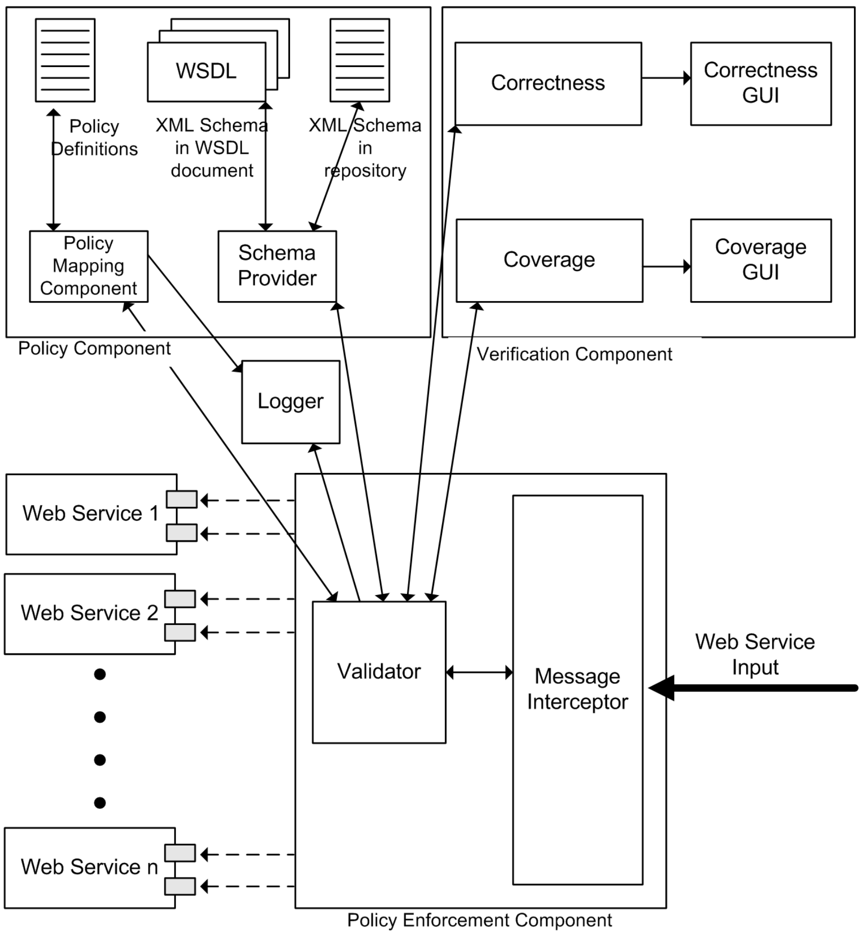


Figure 6.7: System Architecture After Step 5

6.6.1 Coverage

To satisfy the coverage requirement, the solution must let the system administrator verify that input validation applies for all the web services, ports, and parameters for which protection is wanted. This can be done by listing the relevant contents of the input validation security policy in an easily readable format where any unprotected items are clearly highlighted.

6.6.2 Correctness

As specified in the requirements, it should be possible to verify that the input validation is performed as expected. This will be done by allowing the system administrator to specify a set of valid and invalid input for each web service port. To check for correctness, a verifying component will execute the set of input and make sure that the valid input is considered valid and invalid input considered invalid by the validator.

6.7 Requirements Traceability Matrix

In this section we provide a requirements traceability matrix which can be found in table 6.2. The analysis in this chapter has derived a system architecture (figure 6.7) consisting of seven main components. The requirements traceability matrix maps the components into the system requirements. As can be seen from the matrix, all requirements are satisfied by the system architecture.

6.8 Summary

The analysis was a gradual process that involved several steps, where the solution was extended and further specified with each step. The final illustration in figure 6.7 is rather involved, and this complexity is not needed in most of the coming discussions. We therefore increase the level of abstraction and show a higher-level illustration of the solution in figure 6.8.

In this chapter, the requirements were analyzed, but as few decisions as possible where made with regard to implementation details. This was important in order to make the solution generic and implementation-independent.

	PolicyMappingManager	SchemaProvider	Correctness	Coverage	Validator	MessageInterceptor	Logger
Req 1.0: Automatic Policy Enforcement	X	X			X	X	
Req 2.1: Coverage				X			
Req 2.2: Correctness			X				
Req 3.1: Sep. of Policy Def. and Enforcement	X	X			X		
Req 3.2: Sep. of Web Service and Policy Def.	X	X					
Req 3.3: Centralized Policy Def.	X	X					
Req 4.1: Non-Bypassability						X	
Req 4.2: Secure Policy Storage	X	X					
Req 4.3: Whitelisting		X			X		
Req 4.4: Logging							X
Req 5.0: Platform Independence	X	X			X		
Req 6.0: Modularity and Encapsulation	X	X	X	X	X	X	X
Req 7.0: Flexibility and Extensibility		X					
Req 8.1: Existing Standards and Specs.	X	X			X	X	X
Req 8.2: Existing Functionality		X			X	X	X
Req 9.0: Performance	X	X			X	X	

Table 6.2: Completed Requirements Traceability Matrix

In the next chapter, we take the results from the analysis one step further and decide on all the details that are missing in order for the solution to be implemented later in the thesis.

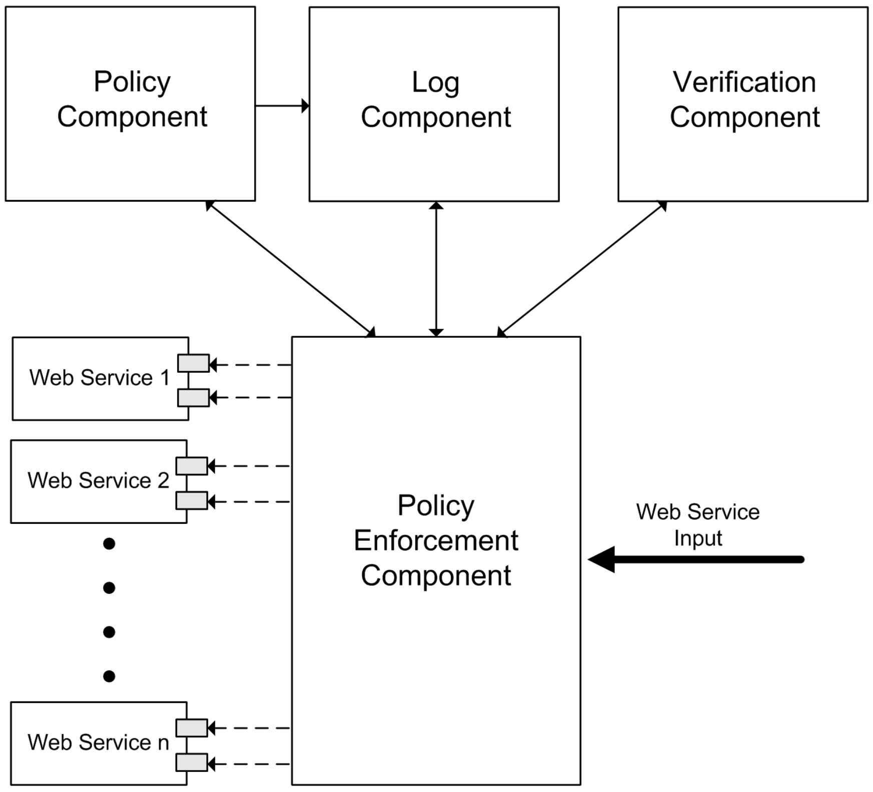


Figure 6.8: A High-Level System Architecture

Chapter 7

System Design

In this chapter, we elaborate on the analysis that was performed in the previous chapter, make the necessary implementation decisions, and present the complete solution design.

The first section presents our choice of implementation platform. It is important to note that, in principle, the system can be implemented on all platforms that support web services. The platform choice presented in the coming section is therefore a matter of personal preference.

The second part of this chapter presents specific class-level designs for each of the components introduced in chapter 6. UML class diagrams are used and a textual explanation to each of the UML diagrams is provided.

7.1 Implementation Platform

Many platforms are available for implementation. So far, we have focused on developing a platform-independent solution, but the detailed class-level design must be targeted for a specific platform and programming language.

We decided to implement the system on the Microsoft .NET platform [45]. There are several benefits from developing web services applications on this platform, and we justify the choice in the coming section.

7.1.1 Broad XML Support

Our system is based on extensive use of XML. It is therefore important that the platform has implemented functionality for consuming and manipulating XML data. The .NET platform implements this functionality. The following list shows more specifically the XML functionality provided by .NET and the classes implementing this functionality.

- **Query an XML document by means of XPath expressions.** This property is needed by several components in order to extract parts of an XML document. .NET provides this functionality in the XmlDocument and XPathNavigator classes.

- **XML Schema Validation.** We wanted a platform with XML schema validation implemented. XmlDocument and XmlSchema implement this.

- **Read and Write XML.** In order to read from the XML-based repository, we need a class that can read XML from file. .NET has this functionality implemented in the XmlReader class. XML writing functionality is performed by the XmlWriter class.

7.1.2 Broad Web Services Support

The platform must also implement, and have support for, web services and the continually evolving web services standards. The .NET Framework contains extensive web services support.

7.1.3 Implementation of WS-Security and WS-Policy

.NET provides an implementation of these standards in a package called web services enhancements (WSE).

7.1.4 Logging Facilities

The last argument for choosing the .NET platform is its logging facilities. The framework offers several ways of logging system events, for instance using the Windows Event Log.

7.2 Detailed Design

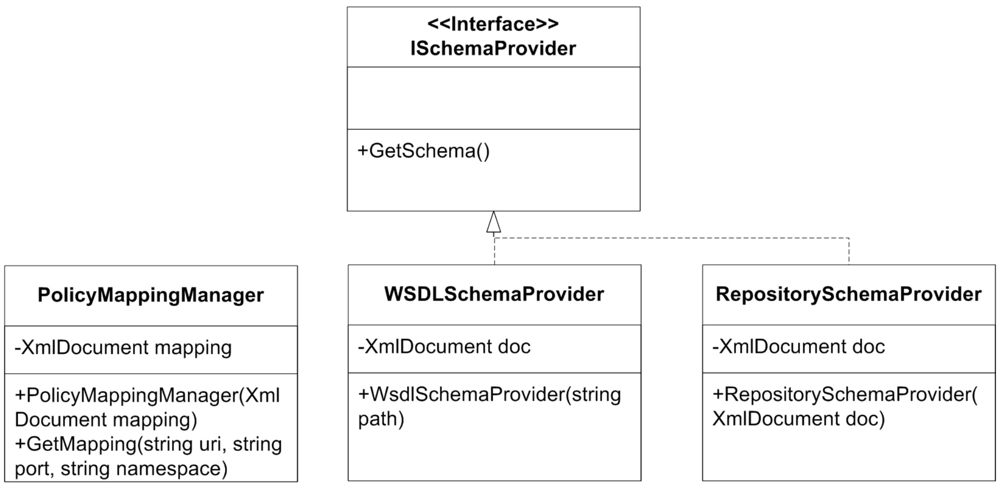
Having made the implementational decisions, we now move on to the detailed design of the solution. In the following, the Unified Modeling Language (UML) version 2.0 [46] is used to illustrate the design. Fowler's *UML Distilled* [47] has been selected as the main reference for UML notation.

We refer the reader to figures 6.7 and 6.8 (pages 57 and 60) in the previous chapter for illustrations of the system architecture which will now be examined closely.

First, we focus on the policy component and decide what classes and relations that are needed for implementation. Next, we design the main part of the system, the policy enforcement component. This component frequently uses the policy component during message validation. After the main functionality has been described, we discuss the log component, and finally we examine the verification component.

7.2.1 The Policy Component

The policy component is designed for handling requests of two types. Firstly, it responds to requests for policy mappings. Secondly, it responds to requests for XML schemas, as indicated in figure 6.7. The results derived from this section are depicted in figure 7.1. All important methods are shown in the UML diagram.



<<Interface>>
ISchemaProvider
+GetSchema()
PolicyMappingManager
-XmlDocument mapping
+PolicyMappingManager(Xml Document mapping)
+GetMapping(string uri, string port, string namespace)
WSDLSchemaProvider
-XmlDocument doc
+WsdlSchemaProvider(string path)
RepositorySchemaProvider
-XmlDocument doc
+RepositorySchemaProvider(XmlDocument doc)

Figure 7.1: Detailed Design for the Policy Component

Mapping a given Web Service onto a Given Policy

Each incoming web service request is validated against the policy belonging to the web service's policy. The policies are defined on the server side. Thus, incoming web service requests do not contain information about which policy they map to.

Since incoming web service requests contain information only about the web service to which they are destined, we need a mechanism for mapping the web service to a policy on the server. This mapping function should also make it possible for several web services to share the same policy.

The class implemented for this purpose is called PolicyMappingManager. The PolicyMappingManager constructor accepts an XML document holding the mapping information. Further, the class has one important method called GetMapping, which takes the web service URI, port number, and namespace as input. Based on this input, the PolicyMappingManager does a look-up in order to find the policy associated with the web service. This lookup is done by using an advanced XPath expression that queries the mapping document. When the policy root node is found, the document formed by its subelements is extracted to a policy XML document. The policy XML document is then returned.

Handling XML Schema Requests

For handling requests for XML schemas we have decided to make use of a simple interface called ISchemaProvider. ISchemaProvider defines one single method called GetSchema, which returns an XML schema. The main reason for implementing this interface is making our solution more flexible with respect to future extensions. If designers of future extensions for instance want to store XML schemas in a database, this feature can be built into the system by having a database handling class that implements ISchemaProvider. We have two classes implementing this ISchemaProvider in our solution. These are called RepositorySchemaProvider and WsdlSchemaProvider, and are discussed next.

Handling XML Schema Requests with Repository

RepositorySchemaProvider assumes that XML schemas are stored in an XML document. The RepositorySchemaProvider has a constructor that takes an XML document as input parameter. The constructor input parameter contains an XML Schema. The only purpose of the RepositorySchemaProvider is to hold the XML document and return the schema via the GetSchema method.

Handling XML Schema Requests with WSDL document

The second class implemented in our system that implements the interface is WsdlSchemaProvider. This class is used for extracting XML schemas from a WSDL document. The constructor of the class takes a path string as input parameter. Depending on the format of the input path string, the class retrieves the WSDL schema from file or from the Internet.

7.2.2 Policy Enforcement Component

The policy enforcement component, also called the “enforcer”, is the main component of the solution. Its responsibility is to intercept all incoming SOAP envelopes, and make sure that the envelope’s contents are in accordance with the specified policy, using the Policy Component.

Detailed Design

The enforcer is also the component that interacts with Web Services Enhancements (WSE) and the .NET web services pipeline. Mainly, WSE is used for two things:

- To hold the policy mapping definitions and XML Schema definitions in the WSE policy file
- To pass MessageInterceptor all SOAP envelopes that are received on the server

The detailed design of the enforcer can be seen in figure 7.2. Only the important methods and parameters are included. We now go through each part of the figure and explain how they all fit together.

In WSE, policy assertions are used by applications that want to control the security-related aspects of the message flow in and out of the web service. We have created such an assertion, ValidationAssertion, and register it with WSE.

WSE lets assertions specify their own configuration options in XML-format in a central security policy file. In this file we have chosen to store the policy mapping definitions and the XML Schema repository. The PolicyFileHelper class has been created in order to encapsulate the logic of extracting this information from the policy file.

WSE asks the registered policy assertions for any SOAP filters that the assertion wants to connect to the web service pipeline, so that it can get

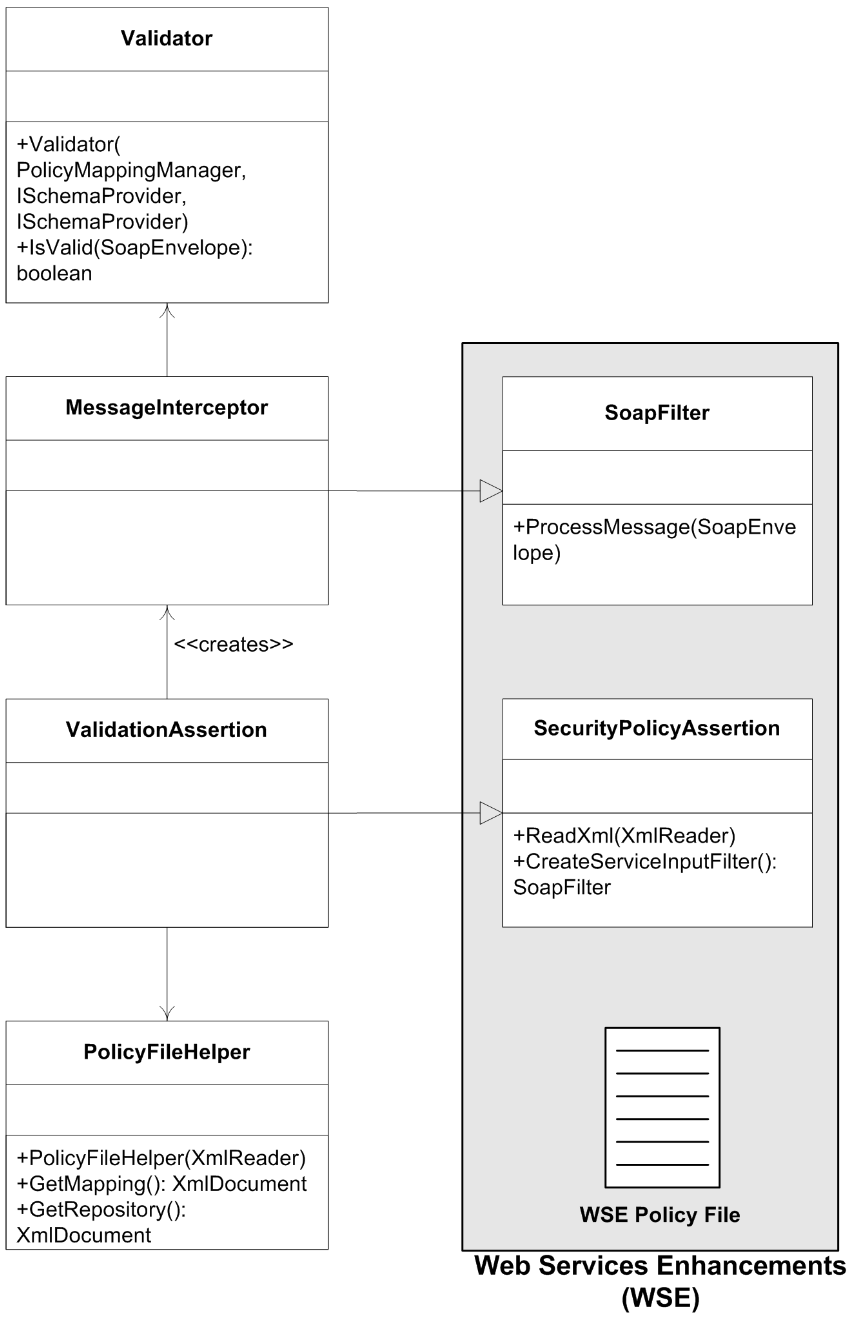


Figure 7.2: Detailed Design for the Policy Enforcement Component

hold of incoming and outgoing web service requests. The proposed solution includes one such filter, MessageInterceptor, which is configured to intercept all incoming messages on the web server.

MessageInterceptor delegates the task of validating incoming messages to a class called Validator. Validator makes use of the Policy Component in order to achieve this.

Behavior

The behavior of the policy enforcement component can be summarized in the following steps:

1. Because ValidationAssertion has been registered with WSE, an instance of ValidationAssertion is created automatically when WSE is initialized.

2. The ReadXml message in ValidationAssertion is called by WSE. A handler to the WS-Policy configuration file used in WSE is passed to ValidationAssertion in this method call.

3. Using PolicyFileHelper, ValidationAssertion retrieves the policy mapping definition and the XML Schema repository, which are stored in the configuration file. This information will later be passed to MessageInterceptor.

4. WSE then calls CreateServiceInputFilter in ValidationAssertion, asking for an instance of the SoapFilter class. ValidationAssertion then initializes a MessageInterceptor object with the information from the previous step.

5. MessageInterceptor creates a Validator and passes it instances of PolicyMappingManager, RepositorySchemaProvider, and WsdlSchemaProvider, which it also creates.

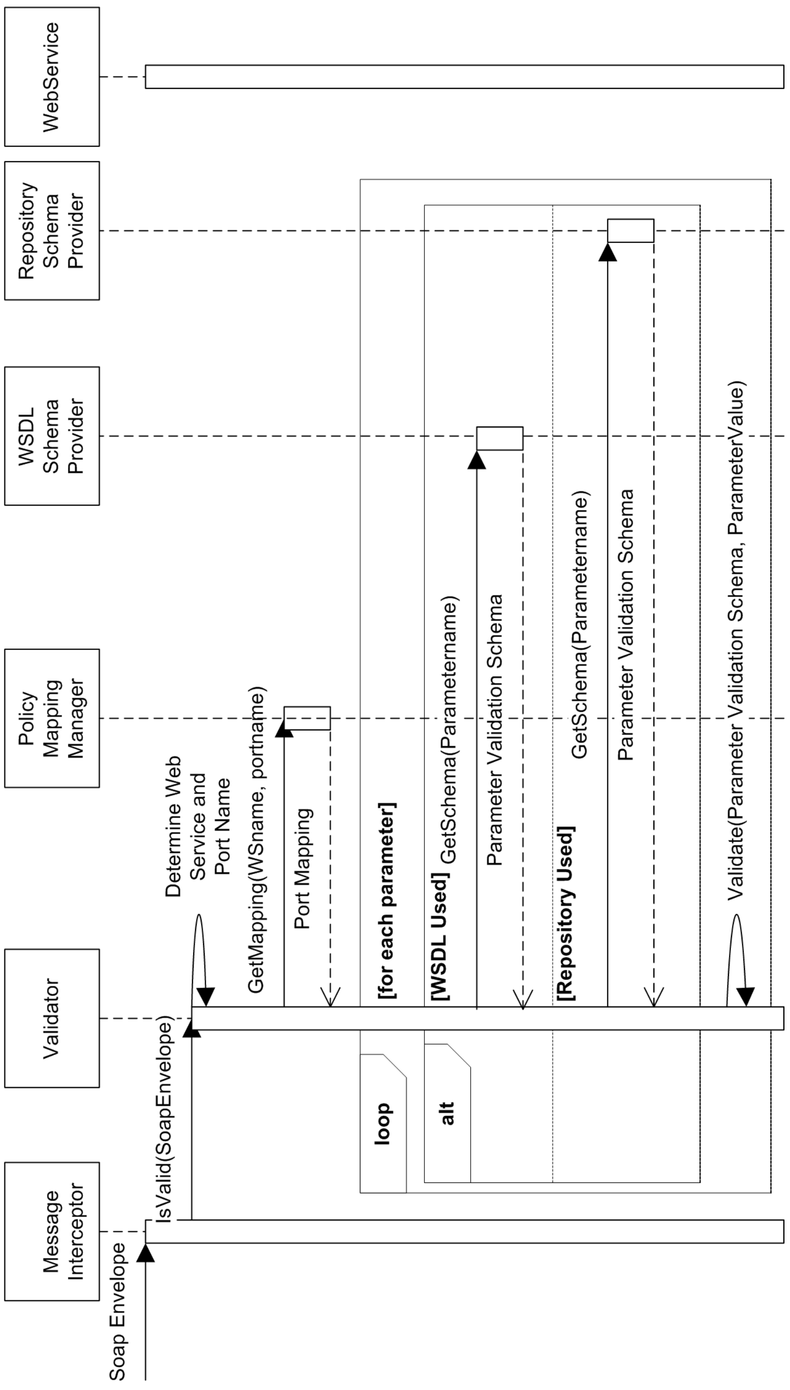


Figure 7.3: Sequence Diagram for Validation of a SOAP Envelope, Part I

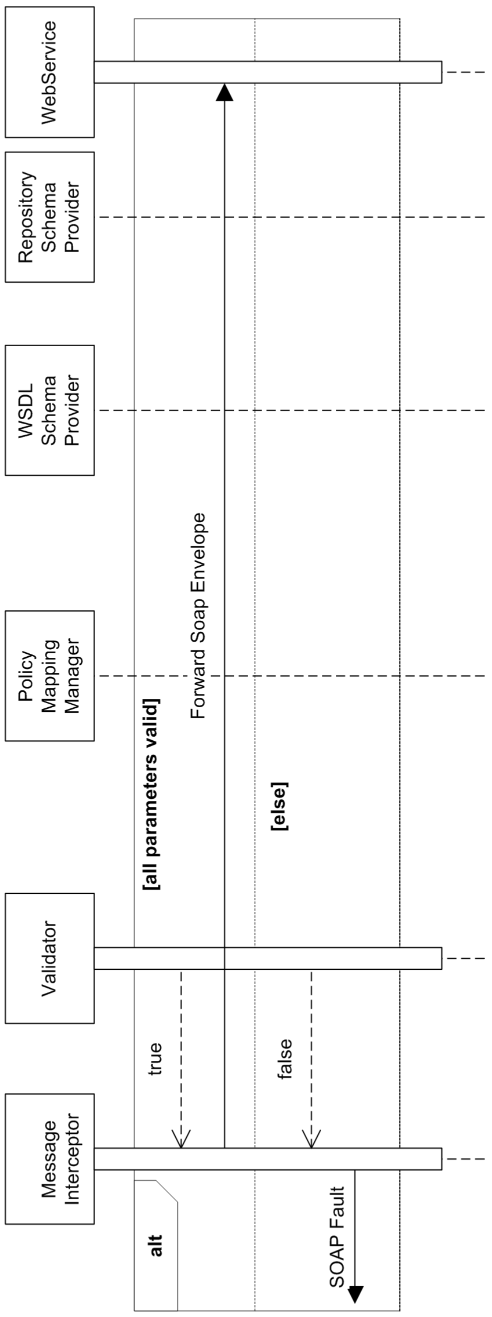


Figure 7.4: Sequence Diagram for Validation of a SOAP Envelope, Part II

6. The MessageInterceptor instance is returned to WSE

7. When a SOAP envelope is received, it is passed by WSE to the MessageInterceptor instance for inspection. MessageInterceptor then asks its Validator instance to validate the message, using the IsValid method call.

8. The logic that takes place in Validator upon the IsValid method call is illustrated in the sequence diagram in figures 7.3 and 7.4.

9. If IsValid returns true, MessageInterceptor lets the message pass. If false is returned, a SOAP fault stating that verification failed is returned to the sender.

7.2.3 Log Component

We have chosen to make use of the Windows Event Log. In order to reuse as much functionality as possible, we choose to inherit from the EventLog class which is included in the .NET class library. An illustration of our class, ValidationLog, and the EventLog class is shown in figure 7.5.

7.2.4 Verification Component

We choose to use the .NET Framework's web interface component, ASP.NET, for the user interfaces for the verification component. Figure 7.6 illustrates the design of the verification component.

Coverage

The Coverage GUI asks the system administrator for the address to the WSE security policy as well as the name of the name of the section that contains the input validation policy. Then, it initializes an instance of the CoverageVerificator class and calls the Verify method. CoverageVerificator

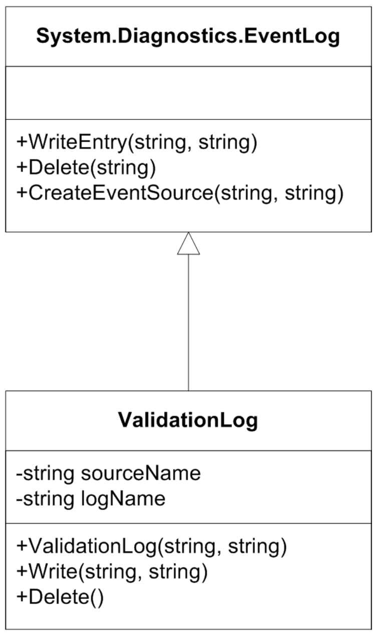

System.Diagnostics.EventLog
+WriteEntry(string, string)
+Delete(string)
+CreateEventSource(string, string)
ValidationLog
-string sourceName
-string logName
+ValidationLog(string, string)
+Write(string, string)
+Delete()

Figure 7.5: Detailed Design for the Log Component

then analyzes the security policy, looking for unprotected ports and parameters, and returns the results to the GUI in the form of an XML document. Finally, the Coverage GUI displays the results to the system administrator.

Correctness

The Correctness GUI asks the system administrator for the message test set, the address to the WSE security policy, and the name of the section that contains the input validation policy. Then, it initializes an instance of the CorrectnessVerificator class and calls the Verify method. CorrectnessVerificator then uses the messages specified in the message test set and asks the Validator to verify each of these. Messages that are specified as valid in the test set should be verified, and the messages that are defined as invalid should not. The result of this process is specified in an XML file and returned to the GUI. Finally, the Correctness GUI displays the results to the system administrator.

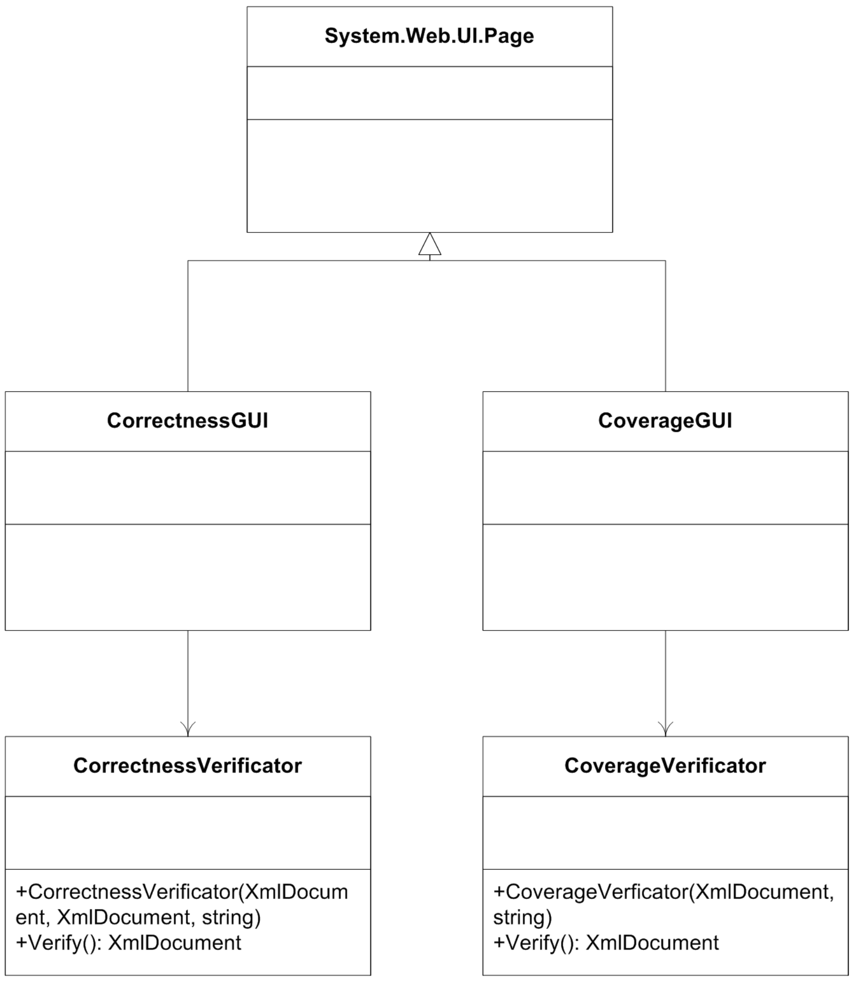


Figure 7.6: Detailed Design for the Verification Component

7.2.5 Combining the Components

After having designed all four components, it is now time to combine them in one diagram. This is done in figure 7.7. In order to keep it simple, only class names are shown.

7.3 Summary

This chapter completed the design of the input validation system. First, platform and infrastructure design choices were made. Then, based on these choices, the detailed design of the components was performed. The next chapter will describe our implementation of the design presented in this chapter.

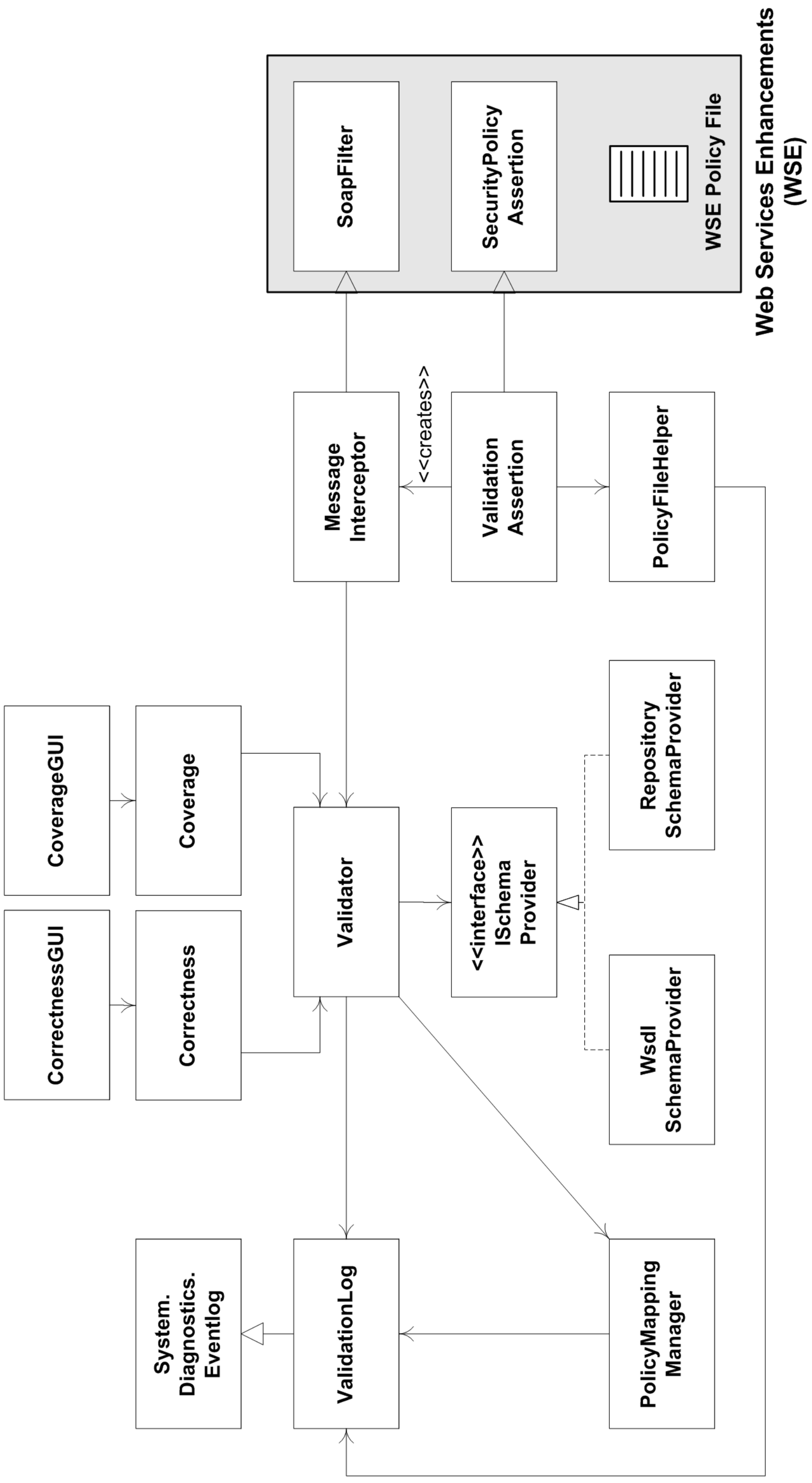


Figure 7.7: Detailed Design for the Input Validator

Chapter 8

System Implementation

The design in the previous chapter has been implemented and the implementation results are presented in this chapter.

The design has resulted in three different tools: the input validator, the correctness tool, and the coverage tool. In this chapter, all of them are described and some screenshots are provided.

The main purpose of this chapter is to give the reader an overview of the different implemented tools before we introduce an example application in the following chapter.

8.1 The Input Validator

The input validator represents the core of our work. It intercepts web service input and validates it against a given policy. The input validator consists of several components. These components were described in detail in chapters 6 and 7.

By adding a reference to the input validator class library, input validation is effectively added to an application. The input validator has no GUI. It runs as a background process which listens for incoming web service requests, validates these requests, and forwards them to the web service if validation succeeds. Otherwise, if validation fails, an exception is sent to the calling application.

8.2 The Correctness Tool

The correctness tool was created to give the system administrator an opportunity to test if his configurations work as expected. This function is called for because it might be difficult to see the full meaning of XML schema expressions by examining the textual representation of the schema.

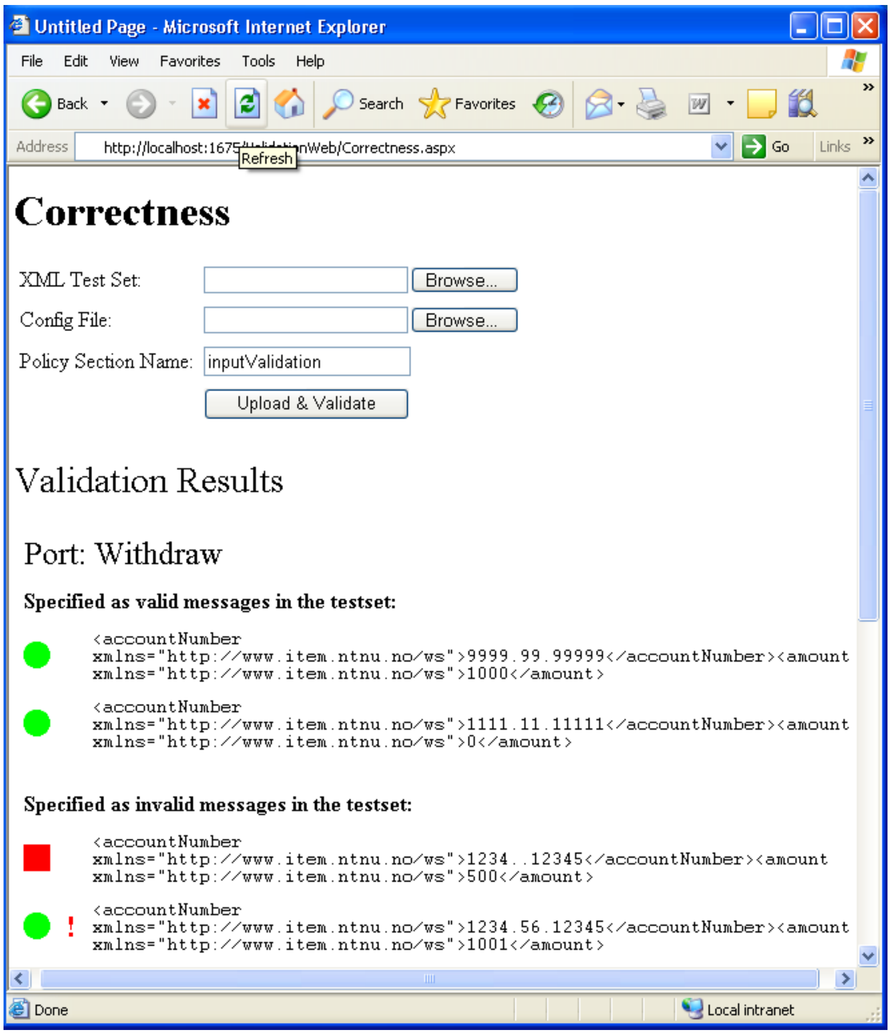
The correctness tool lets the system administrator define a set of messages that are supposed to be valid and another set of messages supposed to be invalid. These messages are uploaded through a web-based GUI. The GUI is depicted in figures 8.1 and 8.2. The upload section of the GUI is seen in the upper part of the GUI screenshots.

When the messages are uploaded, the correctness tool runs the messages through the validator and displays the status of each of the messages. If the test fails (e.g. messages supposed to be invalid are considered valid by the validator), this is indicated in the GUI.

The correctness tool uses green circles to depict valid input and red squares to depict invalid input. Figure 8.1 shows an example of a correctness report where input supposed to be invalid has been recognized as valid by the validator. As can be seen from the screenshot, an exclamation mark is used to notify the system administrator about this anomaly. Additionally, it can be seen that a green circle is depicted where a red square was expected.

8.3 The Coverage Tool

The coverage tool was developed in order to allow the system administrator to verify that he has protected all ports.


Untitled Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:1675/ValidationWeb/Correctness.aspx
Refresh
Go Links
Done
Local intranet

Correctness

XML Test Set:	Browse...
Config File:	Browse...
Policy Section Name:	inputValidation

Upload & Validate

Validation Results

Port: Withdraw

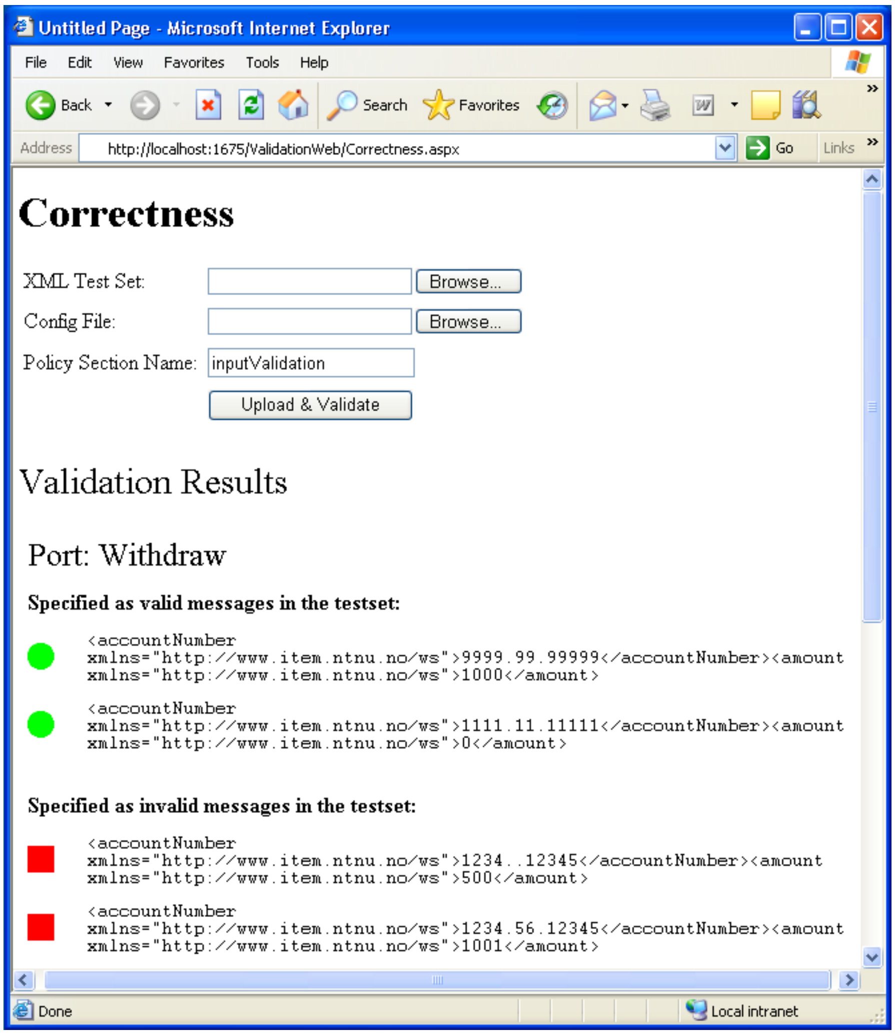
Specified as valid messages in the testset:

```
● <accountNumber
  xmlns="http://www.item.ntnu.no/ws">9999.99.99999</accountNumber><amount
  xmlns="http://www.item.ntnu.no/ws">1000</amount>

● <accountNumber
  xmlns="http://www.item.ntnu.no/ws">1111.11.11111</accountNumber><amount
  xmlns="http://www.item.ntnu.no/ws">0</amount>
```

Specified as invalid messages in the testset:

```
🟥 <accountNumber
   xmlns="http://www.item.ntnu.no/ws">1234..12345</accountNumber><amount
   xmlns="http://www.item.ntnu.no/ws">500</amount>

● ! <accountNumber
    xmlns="http://www.item.ntnu.no/ws">1234.56.12345</accountNumber><amount
    xmlns="http://www.item.ntnu.no/ws">1001</amount>
```

Figure 8.1: Correctness Report with Invalid Policy


Untitled Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:1675/ValidationWeb/Correctness.aspx
Go Links
Browse...
Upload & Validate
Done
Local intranet

Correctness

XML Test Set:

Config File:

Policy Section Name: inputValidation

Validation Results

Port: Withdraw

Specified as valid messages in the testset:

```
● <accountNumber xmlns="http://www.item.ntnu.no/ws">9999.99.99999</accountNumber><amount xmlns="http://www.item.ntnu.no/ws">1000</amount>
● <accountNumber xmlns="http://www.item.ntnu.no/ws">1111.11.11111</accountNumber><amount xmlns="http://www.item.ntnu.no/ws">0</amount>
```

Specified as invalid messages in the testset:

```
■ <accountNumber xmlns="http://www.item.ntnu.no/ws">1234..12345</accountNumber><amount xmlns="http://www.item.ntnu.no/ws">500</amount>
■ <accountNumber xmlns="http://www.item.ntnu.no/ws">1234.56.12345</accountNumber><amount xmlns="http://www.item.ntnu.no/ws">1001</amount>
```

Figure 8.2: Correctness Report with Valid Policy

It might be hard to determine whether all web services are protected by manually examining the configuration file. The coverage tool automates this process and provides a GUI for displaying the results.

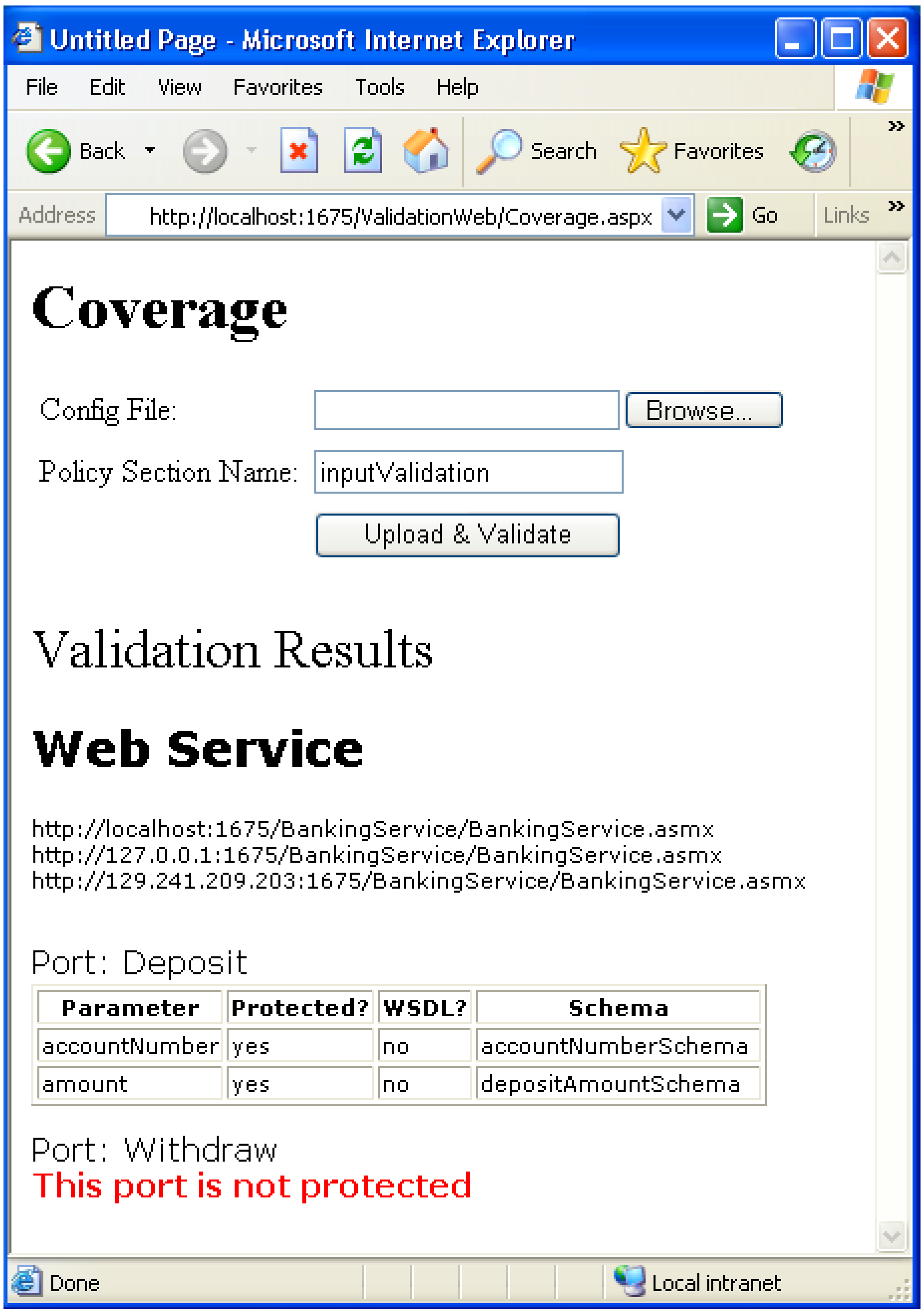
Figure 8.3 and 8.4 shows the coverage tool GUI. Figure 8.3 shows an example where wrong configuration is revealed. The system administrator has most likely thought that all web service ports are protected. However, the coverage tool states that the port called Withdraw is not protected. The system administrator should then change the configuration file in order to protect the port.

Further, it can be seen from the figure whether the parameters uses repository or WSDL XML schemas. This functionality was thoroughly explained in section 6.3.3.

8.4 Summary

This chapter has described the main functionality implemented in our work with the master's thesis. Mainly, three tools have been implemented. These tools are the input validator, the coverage tool, and the correctness tool.

The next chapter will demonstrate the use of the implementation presented here by means of an example application.


Untitled Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:1675/ValidationWeb/Coverage.aspx Go Links
Browse...
Upload & Validate
Done Local intranet

Coverage

Config File:

Policy Section Name: inputValidation

Validation Results

Web Service

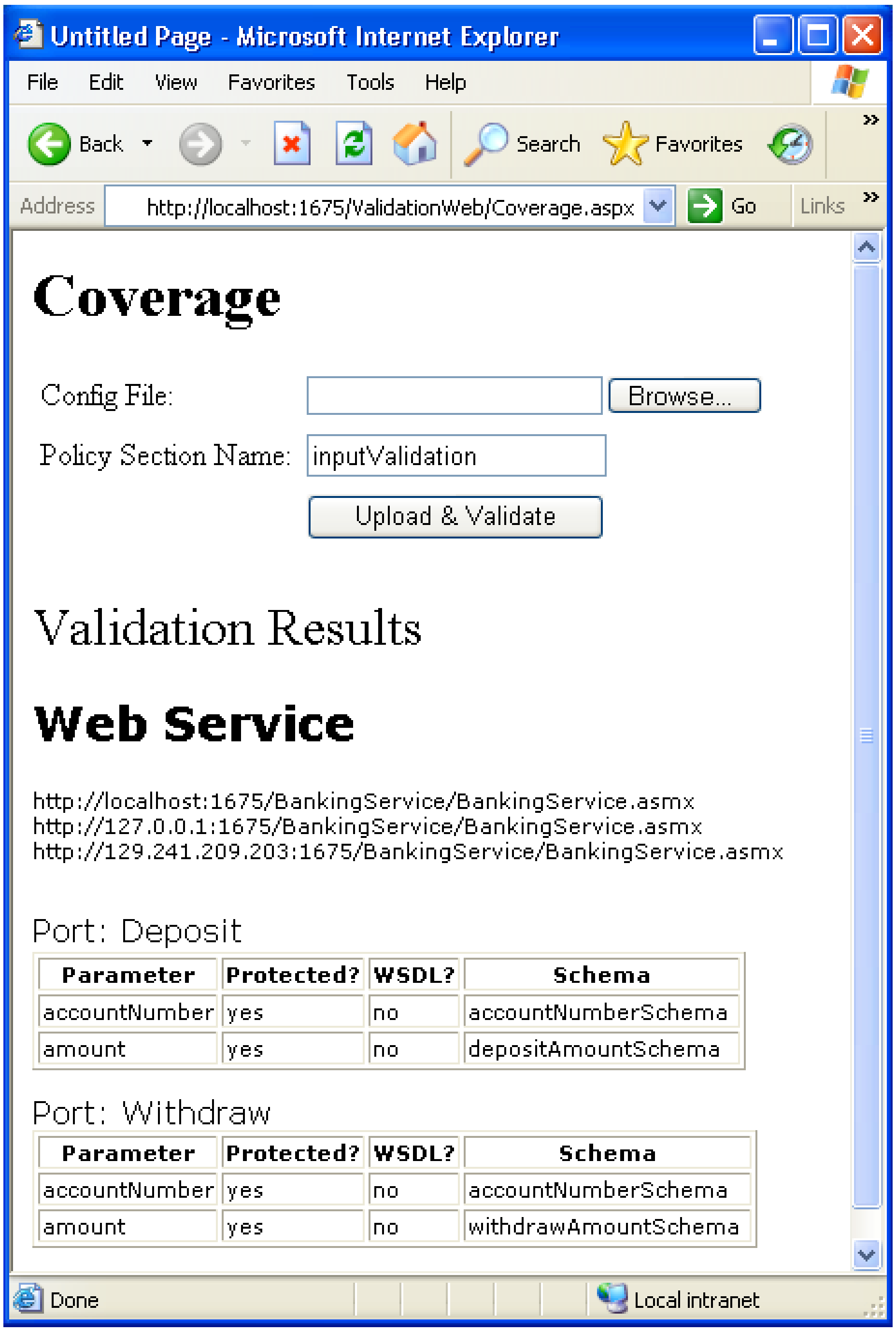
http://localhost:1675/BankingService/BankingService.asmx
http://127.0.0.1:1675/BankingService/BankingService.asmx
http://129.241.209.203:1675/BankingService/BankingService.asmx

Port: Deposit

Parameter	Protected?	WSDL?	Schema
accountNumber	yes	no	accountNumberSchema
amount	yes	no	depositAmountSchema

Port: Withdraw
This port is not protected

Figure 8.3: Coverage Report with Invalid Configuration


Untitled Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
Go
Links
Browse...
Upload & Validate
Done
Local intranet

http://localhost:1675/ValidationWeb/Coverage.aspx

Coverage

Config File:

Policy Section Name: inputValidation

Validation Results

Web Service

http://localhost:1675/BankingService/BankingService.asmx
http://127.0.0.1:1675/BankingService/BankingService.asmx
http://129.241.209.203:1675/BankingService/BankingService.asmx

Port: Deposit

Parameter	Protected?	WSDL?	Schema
accountNumber	yes	no	accountNumberSchema
amount	yes	no	depositAmountSchema

Port: Withdraw

Parameter	Protected?	WSDL?	Schema
accountNumber	yes	no	accountNumberSchema
amount	yes	no	withdrawAmountSchema

Figure 8.4: Coverage Report with Valid Configuration

Chapter 9

Adding Input Validation to an Example Application

The detailed design of the input validation solution was completed in chapter 7, and in the previous chapter we described the resulting implementation. In this chapter, we describe an example application that makes use of the solution for adding input validation functionality.

9.1 Example Application Description

The example application is a simple banking application for deposit and withdrawal of money. The application consists of two parts. The first part resides on the server side and is a web service with two ports. One port enables withdrawals and the other deposits. The second part of the application is a Windows client application that provides a GUI for invoking the web service. Using this Windows application, bank customers can withdraw and deposit money via the web service.

The business rules for the application are that an amount between 0 and 1000 can be withdrawn, and that any positive amount can be deposited.

The web service is implemented without thinking of input validation. In code, there is no check on whether deposit or withdrawal amounts are positive or negative. Further, there is no check on amount limit. Before adding input validation, the application therefore accepts all input.

9.1.1 Adding Encryption, Integrity and Authentication

After developing the above-described, naive implementation of the web service, we add security. By using the Microsoft .NET implementation of WS-Policy and WS-Security, called Web Services Enhancements (WSE), we add encryption, integrity, and authentication.

All security added by WSE can be configured in an XML configuration file. We also use the XML configuration file to add input validation at a later stage.

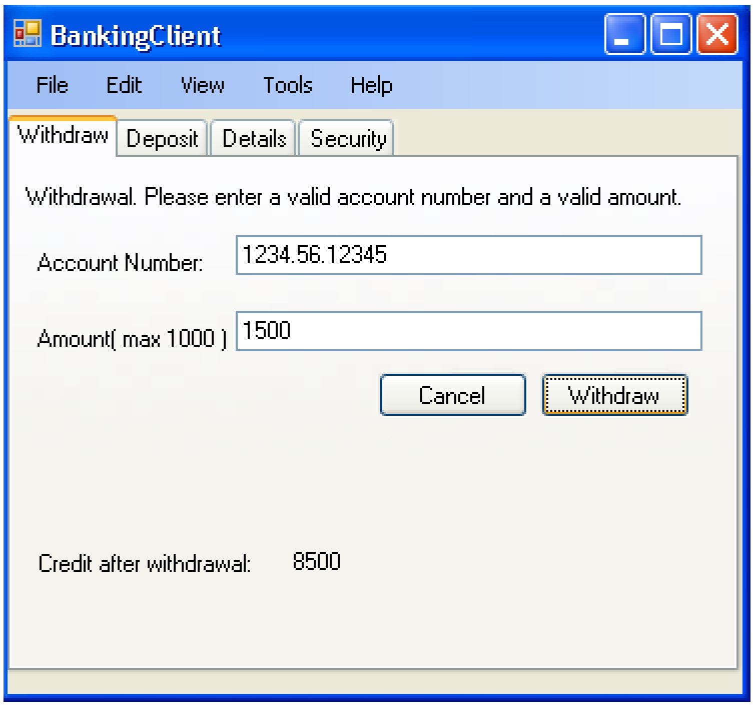
9.1.2 Withdrawing Money Without Input Validation

Before adding input validation functionality, the application is run and invalid values are entered. In figure 9.1, we see that the application does not prevent the user from withdrawing more than 1000.

9.2 Adding Input Validation

We now demonstrate how the solution described in this thesis can be employed to add input validation functionality to the example application. This process consists of three steps:

1. **Step 1:** Define input validation policy
2. **Step 2:** Create mapping definition
3. **Step 3:** Verify policy and mapping using the web-based validation tools



Withdrawal. Please enter a valid account number and a valid amount.

Account Number: 1234.56.12345

Amount(max 1000) 1500

Credit after withdrawal: 8500

Figure 9.1: Withdrawing Money without Input Validation

We now go through each of these steps.

9.2.1 Step 1: Define Input Validation Policy

As both web service ports receive an account number as one of their input parameters, we want to reuse the validation logic for account numbers. The restriction is specified using a regular expression that says that an account number must be on the form “dddd.dd.ddddd”, where each “d” symbolizes a digit.

We are unable to reuse the logic for the amounts that are deposited and withdrawn as they have different restrictions; the withdraw amount must be between 0 and 1000, while the only restriction for the deposit amount is that it must be positive.

The input validation policy for account number, withdraw amount, and deposit amount is shown below.

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

   <!-- the withdrawn amount must be
   between 0 and 1000, inclusive -->
    <xsd:element name="withdrawAmountSchema">
      <xsd:simpleType>
        <xsd:restriction base="xsd:double">
          <xsd:minInclusive value="0"/>
          <xsd:maxInclusive value="1000"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>

    <!-- the deposited amount must be positive -->
    <xsd:element name="depositAmountSchema">
      <xsd:simpleType>
        <xsd:restriction base="xsd:double">
          <xsd:minInclusive value="0"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>

    <!-- an account number must be on the form
    1234.12.12345 -->
    <xsd:element name="accountNumberSchema">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="\d{4}.\d{2}.\d{5}" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:schema>
</repository>
```

9.2.2 Step 2: Create Mapping Definition

After having defined the policy elements that we need, it is time to define the mapping between the web service and policy. For our web service, we will allow SOAP envelopes addressed to three different URIs, which all point to the same web service.

Further, we define mappings for both the Deposit and Withdraw ports. The account numbers both use the same policy element, and the amounts use individual elements, as described above.

The resulting mapping is shown below.

```
<mapping>
  <webservice>

   <!-- the URI synonyms that the web service can
   be contacted on -->
    <uris>
      <uri>http://localhost:1675/BankingService/
       BankingService.asmx</uri>
      <uri>http://127.0.0.1:1675/BankingService/
       BankingService.asmx</uri>
      <uri>http://129.241.209.203:1675/BankingService/
       BankingService.asmx</uri>
    </uris>

    <!-- the ports that the web service contains -->
    <ports>
      <port name="Deposit" namespace="http://tempuri.org/">
        <param name="accountNumber"
         schemaId="accountNumberSchema" />
        <param name="amount"
         schemaId="depositAmountSchema"/>
      </port>
      <port name="Withdraw" namespace="http://tempuri.org/">
        <param name="accountNumber"
         schemaId="accountNumberSchema" />
        <param name="amount"
         schemaId="withdrawAmountSchema"/>
      </port>
    </ports>

  </webservice>
</mapping>
```

9.2.3 Step 3: Verify the Configuration

Before employing the policy and mapping, it is important to verify that they are configured correctly. First, we use the correctness tool to make sure that the policy is defined as intended. Then, using the coverage tool we make sure that the mapping is correct. Both tools were depicted and described in chapter 8.

Correctness

The first step to test correctness is to create a set of valid and invalid messages for each web service port. An example of such a test set is shown below.

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions xmlns="http://www.item.ntnu.no/ws">
  <webservice>
   <!-- the URI synonyms that the web service will
   be contacted on -->
    <uris>
      <uri>http://localhost:1675/BankingService/
       BankingService.asmx</uri>
      <uri>http://127.0.0.1:1675/BankingService/
       BankingService.asmx</uri>
      <uri>http://129.241.209.203:1675/BankingService/
       BankingService.asmx</uri>
    </uris>

    <!-- the ports that will be checked -->

    <port name="Withdraw" namespace="http://tempuri.org/">

     <!-- the valid messages that are sent to the
     Withdraw port -->
      <validMessages>
        <message>
          <accountNumber>9999.99.99999</accountNumber>
          <amount>1000</amount>
        </message>
        <message>
      <accountNumber>1111.11.11111</accountNumber>
      <amount>0</amount>
    </message>
  </validMessages>

  <!-- the invalid messages that are sent to the
 Withdraw port -->
  <invalidMessages>
    <message>
      <accountNumber>1234..12345</accountNumber>
      <amount>500</amount>
    </message>
    <message>
      <accountNumber>1234.56.12345</accountNumber>
      <amount>1001</amount>
    </message>
  </invalidMessages>
</port>

<port name="Deposit" namespace="http://tempuri.org/">

 <!-- the valid messages that are sent to the
 Deposit port -->
  <validMessages>
    <message>
      <accountNumber>9999.99.99999</accountNumber>
      <amount>1000000</amount>
    </message>
    <message>
      <accountNumber>1111.11.11111</accountNumber>
      <amount>0</amount>
    </message>
  </validMessages>

  <!-- the invalid messages that are sent to the
 Deposit port -->
  <invalidMessages>
    <message>
      <accountNumber>1234..12345</accountNumber>
      <amount>500</amount>
    </message>
    <message>
      <accountNumber>bank</accountNumber>
      <amount>1001</amount>
```

```
        </message>
        <message>
          <accountNumber>1234.56.12345</accountNumber>
          <amount>-1</amount>
        </message>
      </invalidMessages>

    </port>

  </webservice>
</definitions>
```

To illustrate the operation of the correctness tool, we first apply it with an error in the security policy. The policy has been configured to allow withdrawals up to 10 000, instead of 1 000, which is the correct limit for the example application.

The result of the correctness test was shown in figure 8.1 (page 79). We see that a message specified as invalid in the test set has been found to be valid by the validator. In the message, the amount is set to 1 001, which is not allowed according to the intended policy. However, due to the misconfiguration, this message was found valid. This discrepancy is marked with a red exclamation mark in the report.

Then, we correct the policy by setting the maximum amount to 1 000 and re-run the test. The result, showing that all valid messages were found to be valid, and all invalid messages found to be invalid, was shown in figure 8.2 (page 80).

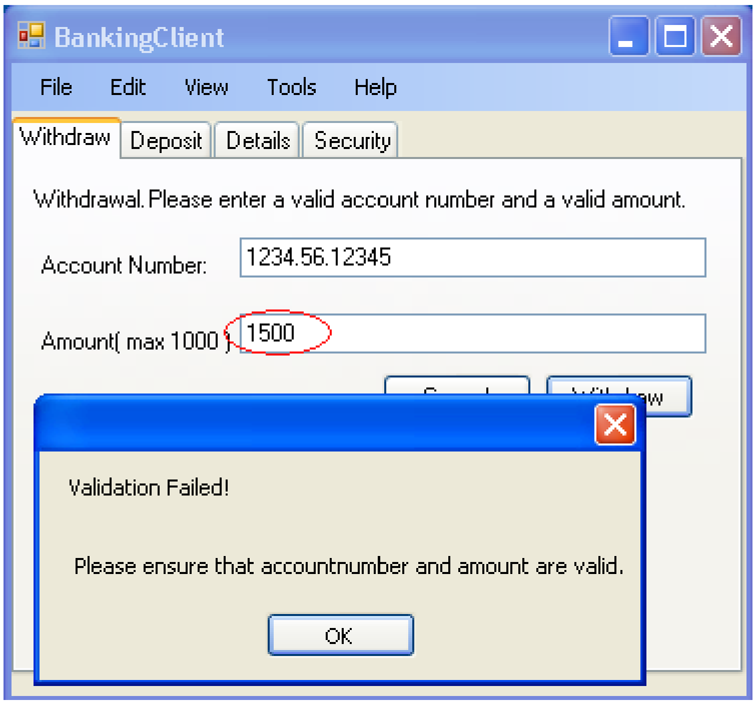
Coverage

Next, we want to make sure that the mapping has been defined correctly. To illustrate how the tool works, we have run the tool with an error in the mapping. The error was that validation was turned off for the Withdraw port. In figure 8.3 (page 82), we see that this is clearly pointed out in the report.

When the coverage tool is run with a correct mapping definition, the result is as shown in figure 8.4 (page 83).

9.2.4 Withdrawing Money With Input Validation

After having validated correctness and coverage of our input validation definitions, we run the banking application again. Using the same input as in the beginning of the chapter, we now experience that the input validation system rejects the input as invalid. The result is shown in figure 9.2.



Withdrawal. Please enter a valid account number and a valid amount.

Account Number: 1234.56.12345

Amount(max 1000): 1500

Validation Failed!

Please ensure that accountnumber and amount are valid.

Figure 9.2: Withdrawing Money with Input Validation

9.3 Summary

This chapter has demonstrated use of the input validation mechanism developed during this master's thesis work. We developed an example application and added security to this application. We showed that it was easy and straight-forward to add input validation to an existing web service, and to verify the validation policy and mapping using the web-based tools that have been implemented.

It should be noted that all security mechanisms are added out of code in an XML configuration file. Thus, the application code is not affected after adding our mechanisms.

Chapter 10

Conclusions

10.1 Conclusions

This thesis describes the design and implementation of a web service input validation mechanism. The mechanism is realized by extending existing web services security standards, such as WS-Security and WS-Policy.

To cover the requirements in the problem statement, the mechanism is centralized and operates without human intervention. Further, it can be verified for correctness and coverage using a web-based verification tool. Additionally, web services protection can be added at deployment time, without modifying existing code.

Last, the thesis demonstrates how the input validation mechanism can be added to an existing application, and how it can be configured and verified to ensure that it operates correctly.

10.2 Summary of Contributions

The major contributions of this work are:

- An examination of XML, web services, web services security, and input validation.

- A detailed design of an automatic, verifiable, and centralized web service input validation mechanism that can be added to and configured for a web service at deployment time.
- An implementation of the detailed design.
- Elaboration of the implementation by means of an example application.

10.3 Future Research

Suggestions for future research based on this thesis are:

- **More use of flexibility:** The design allows for much more flexibility than what has been explored in the described implementation. In regard to flexibility, at least four types of extensions can be envisioned:
  - An interesting extension of this work would be to make use of the security information available from WS-Security and WS-Policy in the security policies. One example would be to make a requirement that a given web service port can only be called by a certain set of users, who can then, in turn, be authenticated by digital certificates. This way, the mechanism would function like a web service firewall.
  - A second way to extend the current design would be to add functionality for auditing and billing. By using authentication provided by WS-Security or another source, it would be possible to record the users' service usage. Thus, one could use these records for billing or to determine which user performed a certain web service call in the past. This information could be used for non-repudiation purposes.

  - Third, the system could be extended to function as an intrusion detection system (IDS). Such a system could look for either statistical anomalies or for usage patterns that violate certain rules that have been specified [48].

  - Fourth, this thesis addresses only simple input validation. Parameters are validated independently of each other. We see opportunities for future work with extending this functionality by letting validation depend on the relationships between different parameters (e.g. the value of parameter A determines the valid range of parameter B).

- **Validation of output:** The current design only allows validation of input to web services. In certain situations it is also desirable to validate the output from a web service. Such validation could be useful to ensure that web services do not return unwanted information, for instance in the case of programming errors, or as an extra layer of security.

- **Formal verification:** The type of verification that has been designed and implemented in this thesis could be formalized. Then, it might be possible to mathematically prove the correct operation of the input validation mechanism. However, such formalization might also make the system less user friendly.

- **Graphical configuration:** A useful extension could be to develop graphical tools to assist with configuration of security policies and mappings. Such tools would make the job of configuring much more efficient and less error-prone.

- **Distributed policy definitions:** We see a potential benefit in developing a system for distributing policy definitions. Such a system could let web services on different locations share a common, distributed policy. This would allow for reuse on a larger scale than that which is possible between web services on the same server.

Bibliography

[1] IDC, “IDC Press Release: Consumption of Web Services Will Greatly Increase Through 2009.” http://www.idc.com/getdoc.jsp?containerId=prUS00190705, 2005.

[2] The Open Web Application Security Project (OWASP), “The OWASP Top Ten Project.” http://www.owasp.org/documentation/topten.html, 2005.

[3] M. R. Stytz and J. A. Whittaker, “Caution: this product contains security code,” *IEEE Security & Privacy Magazine*, vol. 1, no. 5, pp. 86–88, 2003.

[4] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti, “Using parse tree validation to prevent SQL injection attacks,” in *SEM ’05: Proceedings of the 5th international workshop on Software engineering and middleware*, (New York, NY, USA), pp. 106–113, ACM Press, 2005.

[5] Z. Su and G. Wassermann, “The essence of command injection attacks in web applications,” in *POPL ’06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 372–382, ACM Press, 2006.

[6] E. G. Sirer and K. Wang, “An access control language for web services,” in *SACMAT ’02: Proceedings of the seventh ACM symposium on Access

control models and technologies, (New York, NY, USA), pp. 23–30, ACM Press, 2002.

[7] C. A. Ardagna, E. Damiani, S. D. C. di Vimercati, and P. Samarati, “A Web Service Architecture for Enforcing Access Control Policies,” in *Proceedings of the First International Workshop on Views on Designing Complex Architectures (VODCA 2004)*, pp. 47–62, Elsevier B.V., 2004.

[8] C. Kruegel and G. Vigna, “Anomaly detection of web-based attacks,” in *CCS ’03: Proceedings of the 10th ACM conference on Computer and communications security*, (New York, NY, USA), pp. 251–261, ACM Press, 2003.

[9] L. Baresi, S. Guinea, and P. Plebani, “WS-Policy for Service Monitoring,” in *Technologies for E-Services: 6th International Workshop, TES 2005, Trondheim, Norway, September 2-3, 2005, Revised Selected Papers*, pp. 72–83, Springer Berlin / Heidelberg, 2005.

[10] K. Bhargavan, C. Fournet, and A. D. Gordon, “Verifying policy-based security for web services,” in *CCS ’04: Proceedings of the 11th ACM conference on Computer and communications security*, (New York, NY, USA), pp. 268–277, ACM Press, 2004.

[11] A. D. Gordon and R. Pucella, “Validating a web service security abstraction by typing,” in *XMLSEC ’02: Proceedings of the 2002 ACM workshop on XML security*, (New York, NY, USA), pp. 18–29, ACM Press, 2002.

[12] K. Bhargavan, C. Fournet, A. D. Gordon, and G. O’Shea, “An advisor for web services security policies,” in *SWS ’05: Proceedings of the 2005 workshop on Secure web services*, (New York, NY, USA), pp. 1–9, ACM Press, 2005.

[13] World Wide Web consortium (W3C), "Extensible Markup Language (XML) 1.0 (Third Edition)." http://www.w3.org/TR/2004/REC-xml-20040204/, February 2004.

[14] E. R. Harlond and W. S. Means, *XML In a Nutshell.* Sebastopol, CA, USA: O'Reilly, 2004.

[15] The Internet Society Network Working Group, "Uniform Resource Identifier (URI): Generic Syntax." http://www.gbiv.com/protocols/uri/rfc/rfc3986.html, January 2005.

[16] World Wide Web consortium (W3C), "XML Schema Second Edition." http://www.w3.org/TR/xmlschema-0/, October 2004.

[17] World Wide Web consortium (W3C), "XML Path Language (XPath)." http://www.w3.org/TR/xpath, November 1999.

[18] A. Kalani and P. Kalani, *MCAD/MCSD Developing XML Web Services and Server Components with Visual C# .NET and the .NET Framework.* Indianapolis, IN, USA: QUE Certification, 2003.

[19] E. Cerami, *Web Services Essentials.* Sebastopol, CA, USA: O'Reilly, 2002.

[20] C. Geuer-Pollmann and J. Claessens, "Web services and web service security standards," *Information Security Technical Report*, vol. 10, no. 1, pp. 15–24, 2005.

[21] Jesper Holgersson and Eva Soderstrom, "Web service security - vulnerabilities and threats within the context of WS-security," in *The 4th Conference on Standardization and Innovation in Information Technology*, pp. 138–146, September 2005.

[22] A. S. Tanenbaum, *Computer Networks.* Upper Saddle River, NJ, USA: Prentice Hall PTR, fourth ed., 2002.

[23] Internet Engineering Task Force (IETF), “The TLS Protocol v1.0.” http://www.ietf.org/rfc/rfc2246.txt, January 1999.

[24] P. Kearney, “Message level security for web services,” *Information Security Technical Report*, vol. 10, no. 1, pp. 41–50, 2005.

[25] W3C, “XML Encryption syntax and processing.” http://www.w3.org/TR/xmlenc-core/, December 2002.

[26] National Institute of Standards and Technology (NIST), “Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher.” http://csrc.nist.gov/publications/nistpubs/800-67/SP800-67.pdf, May 2004.

[27] National Institute of Standards and Technology (NIST), “Specification for the Advanced Encryption Standard (AES).” http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, November 2001.

[28] W3C, “XML Signature syntax and processing.” http://www.w3.org/TR/xmldsig-core/, February 2002.

[29] National Institute of Standards and Technology (NIST), “Secure Hash Standard.” http://www.itl.nist.gov/fipspubs/fip180-1.htm, April 1995.

[30] Internet Engineering Task Force (IETF), “HMAC: Keyed-Hashing for Message Authentication.” http://www.ietf.org/rfc/rfc2104.txt, February 1997.

[31] RSA Laboratories, “RSA Cryptography Standard.” ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf, June 2002.

[32] National Institute of Standards and Technology (NIST), “Digital Signature Standard (DSS).” http://www.itl.nist.gov/fipspubs/fip186.htm, May 1994.

[33] Microsoft, IBM, and VeriSign, “Web services security (WS-Security).” http://www-128.ibm.com/developerworks/webservices/library/ws-secure/, April 2002.

[34] K. Bhargavan, C. Fournet, and A. D. Gordon, “A semantics for web services authentication,” in *31st Annual Symposium on Principles of Programming Languages*, ACM SIGPLAN-SIGACT, January 2004.

[35] Organization for the Advancement of Structured Information Standards (OASIS), “Web Services Security: SOAP Message Security 1.0 (WS-Security 2004).” http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf, March 2004.

[36] The SANS Institute, “SANS Glossary of Terms Used in Security and Intrusion Detection.” http://www.sans.org/resources/glossary.php, 2006.

[37] R. Hollar and R. Murphy, *Enterprise Web Services Security.* Hingham, MA, USA: Charles River Media, Inc., 2006.

[38] Microsoft, IBM, VeriSign, BEA Systems, SAP AG, and Sonic Software, “Web services policy framework (WS-Policy).” http://www-128.ibm.com/developerworks/library/specification/ws-polfram/, September 2004.

[39] IBM, Microsoft, RSA Security, and VeriSign, “Web Services Security Policy Language.” http://www-128.ibm.com/developerworks/library/specification/ws-secpol/, July 2005.

[40] S. H. Huseby, *Innocent Code: a security wake up call for web programmers.* West Sussex, England: John Wiley & Sons, Ltd, 2004.

[41] S. D. Wolthusen, "Layered multipoint network defence and security policy enforcement," in *Proceedings of the IEEE Workshop on Information Assurance and Security*, June 2001.

[42] L. Wu, H. Sahraoui, and P. Valtchev, "Coping with Legacy System Migration Complexity," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 600–609, IEEE, 2005.

[43] J. R. Nicol, C. T. Wilkes, and F. A. Manola, "Object orientation in heterogeneous distributed computing systems," *Computer*, vol. 26, no. 6, pp. 57–67, 1993.

[44] Robert Elfwing, Ulf Paulsson, and Lars Lundberg, "Performance of SOAP in Web Service Environment Compared to CORBA," in *Ninth Asia-Pacific Software Engineering Conference*, pp. 84–93, December 2002.

[45] Microsoft, "Microsoft .NET Homepage."
http://www.microsoft.com/net/, 2006.

[46] Object Management Group, "Unified Modeling Language (UML)."
http://www.uml.org/, 2006.

[47] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Boston, MA, USA: Addison-Wesley, third ed., 2003.

[48] P. Porras, "STAT – a state transition analysis tool for intrusion detection," Master's thesis, University of California Santa Barbara, November 1993.

Part III

Appendices

Appendix A

Web Services Protocols

A.1 SOAP

A.1.1 SOAP Message Format

The Envelope Element

Versioning is the envelope's main purpose. Thus, the envelope element tag must include the SOAP version used. Versions are referred to by using XML namespaces, which are uniquely identified by an URI. The following example illustrates a typical envelope tag, which uses the SOAP 1.2 namespace (http://schemas.xmlsoap.org/soap/envelope/):

```
<SOAP-ENV:Envelope
   xmlns=http://schemas.xmlsoap.org/soap/envelope/>
```

The Header Element

The header element is optional and is often used for adding additional application specific information about the SOAP message. Such application information can typically be description of digital signature, authentication information or payment information.

The default namespace defines three attributes to the header element: actor, mustUnderstand and encodingStyle. These attributes contain information about how the SOAP message shall be processed.

The actor attribute defines the endpoint for which the message is intended. Use of this attribute is especially interesting when a message is to traverse several endpoint on it way to the intended endpoint. The actor attribute is set by an URI (e.g. soap:actor=http://www.item.ntnu.no/appl/).

The mustUnderstand attribute tells whether the entry is mandatory or optional for the receiver to process. The mustUnderstand attribute is set to a Boolean value (the syntax is soap:mustUnderstand = 1 (or 0)).If the mustUnderstand element is set to 1, the element must be recognized by the processing endpoint. If the element is not recognized, the processing of the header must stop.

The encodingStyle attribute defines the data types used in the SOAP message. encodingStyle can be applied to any element in a SOAP message and adds great flexibility to SOAP encoding. There is no standard SOAP encoding scheme. The encodingStyle attribute is set to a URI, e.g. soap:encodingStyle = http://www.item.ntnu.no/soap-enc/.

An example of an SOAP header element that illustrates use of all three attributes:

```
<soap:Header>
<ex:accountNumber
    xmlns:ex = http://www.item.ntnu.no/example
    soap:mustUnderstand=1
    soap:actor=http://www.item.ntnu.no/appl/
    soap:encodingStyle = http://www.item.ntnu.no/soap-enc/>
1615.70.25551
</ex:accountNumber>
</soap:Header>
```

The Body Element

The body element is mandatory and includes the data intended for the endpoint. Data requests and responses are typical contents of this element.

The following example illustrates a request body:

```
<soap:body>
<ex:GetCredit
    xmlns:m="http://www.item.ntnu.no/namespace/">
<ex:AccountType>
primaryAccount
</ex:AccountType>
</ex:GetCredit>
</soap:body>
```

This illustrates a response body for the request:

```
<soap:body>
<ex:GetCreditResponse
    xmlns:m="http://www.item.ntnu.no/namespace/>
<ex:Credit>
1298.15
</ex:Credit>
</ex:GetCredit>
</soap:body>
```

The Fault Element

The last element of the SOAP message is the Fault element. This element is used for error messages. Such messages can, as an example, be triggered by the processing endpoint if it does not recognize an element where the mustUnderstand attribute is set to 1. The standard defines several fault codes for use. When interested in more details on fault codes, the reader is referred to [19].

A.2 WSDL

A.2.1 WSDL elements

In order to get a better understanding of the standard we take a closer look at the different elements in the WSDL standard. The different elements are depicted in figure A.1We have made a simple example of a web service called LightwightCalculator. The LighweightCalculator webservicehas only one method called AddIntegers. AddIntegers takes two integers (addend_a and addend_b) as input parameters and returns the sum of the integers. Samples of the resulting WSDL code is used to illustrate use of the different WSDL elements.

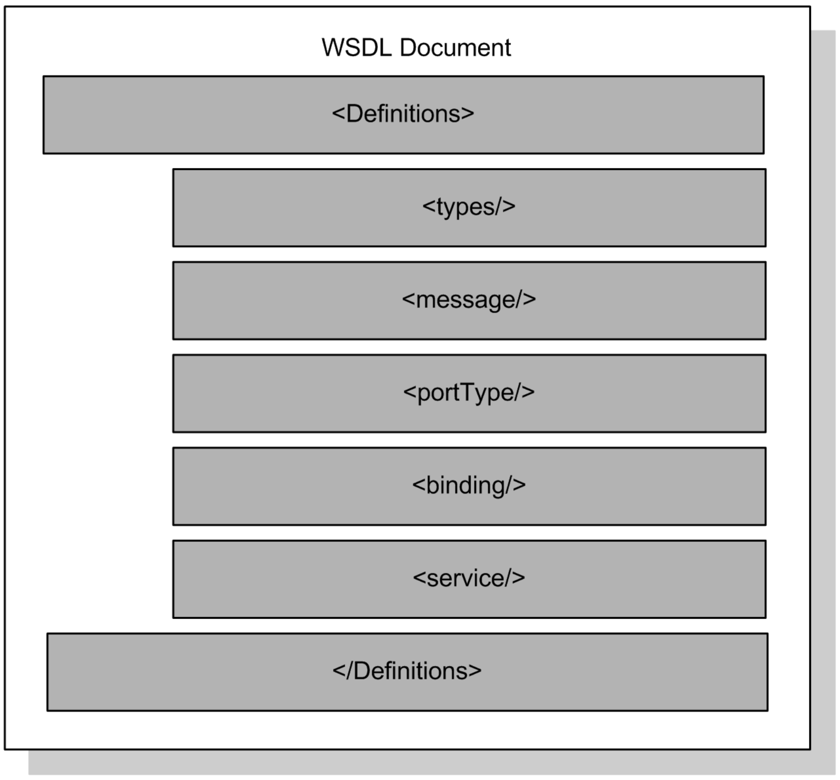


Figure A.1: WSDL Document Format

Definitions

The definitions element must be the root element in any WSDL document. The definitions tag specifies the name of the web service and namespaces for use later in the document. The example simply illustrates definitions of namespaces. Between definitions begin and end tags, all other WSDL elements are placed (this space is marked .. here).

```
<wsdl:definitions
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:tns="http://tempuri.org/"
    xmlns:s="http://www.w3.org/2001/XMLSchema"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    targetNamespace="http://tempuri.org/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

..

</wsdl:definitions>
```

Types

The type element specifies all data types used by sender and receiver. From the following example the reader can easily recognize the method name AddIntegers and the two input parameters addend_a and addend_b. Further one should observe that the response from the AddInteger method is defined as AddIntegersResult and that this response is of the type int.

```
<wsdl:types>
<s:schema
   elementFormDefault="qualified"
   targetNamespace="http://tempuri.org/">
<s:element name="AddIntegers">
<s:complexType>
<s:sequence>
<s:element minOccurs="1" maxOccurs="1"
    name="addend_a" type="s:int" />
<s:element minOccurs="1" maxOccurs="1"
    name="addend_b" type="s:int" />
   </s:sequence>
</s:complexType>
</s:element>
<s:element name="AddIntegersResponse">
<s:complexType>
<s:sequence>
```

```
<s:element minOccurs="1" maxOccurs="1"
    name="AddIntegersResult" type="s:int" />
</s:sequence>
</s:complexType>
</s:element>
</s:schema>
  </wsdl:types>
```

Message

When namespaces and types are defined, the next task is to define the SOAP message names. The messages must have unique names, and the messages must be defined only one way. As can be seen from the example, the AddInteger method results in two messages called AddIntegersSoapIn and AddIntegersSoapOut:

```
<wsdl:message name="AddIntegersSoapIn">
<wsdl:part name="parameters"
    element="tns:AddIntegers" />
</wsdl:message>
<wsdl:message name="AddIntegersSoapOut">
<wsdl:part name="parameters"
    element="tns:AddIntegersResponse" />
</wsdl:message>
```

PortType

The PortType field defines the functions available in the service, and tells which messages are associated with which methods. In the example, the AddInteger method and SOAP messages are linked together to one operation:

```
<wsdl:portType name="LightweightCalculatorSoap">
<wsdl:operation name="AddIntegers">
<wsdl:input message="tns:AddIntegersSoapIn" />
<wsdl:output message="tns:AddIntegersSoapOut" />
</wsdl:operation>
  </wsdl:portType>
```

Binding

The binding element has a reference to the PortType name, and defines how the PortType operation should be sent. HTTP is the typical choice for transport, but SMTP and FTP are other possible choices.

```
<wsdl:binding
    name="LightweightCalculatorSoap"
    type="tns:LightweightCalculatorSoap">
<soap:binding
    transport="http://schemas.xmlsoap.org/soap/http" />
<wsdl:operation name="AddIntegers">
<soap:operation
    soapAction="http://tempuri.org/AddIntegers"
    style="document" />
<wsdl:input>
<soap:body use="literal" />
</wsdl:input>
<wsdl:output>
<soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
```

Service

The final element, the service element, defines the service location. The example shows that the service named LightweightCalculator can be found on the given IP-address (localhost in this case), the given TCP port (1493) and at the given path (WebSite6/Service.asmx).Remark here that the service name LightweightCalculator is linked to the PortType name Lightweight-CalculatorSoap. The function described in the PortType is available in the Service.asmx file, which contains the actual web service.

```
<wsdl:service name="LightweightCalculator">
<wsdl:port
    name="LightweightCalculatorSoap"
    binding="tns:LightweightCalculatorSoap">
<soap:address
    location="http://localhost:1493/WebSite6/Service.asmx" />
</wsdl:port>
</wsdl:service>
```

A.3 UDDI

Companies that want to publish their services must register with the UDDI repository. By doing this, the companies make their services publicly available. Information is stored hierarchically in the repository. The XML structure of the stored data is shown below:

```
<publisherAssertion>
```

```
<businessEntity>
<businessService>
<bindingTemplate>
<tModel/>
<tModel/>
.
.
.
<tModel/>
</bindingTemplate>
</businessService>
<businessService>
.
.
.
</businessService>
.
.
.
<businessService>
.
.
.
</businessService>
</businessEntity>
</publisherAssertion>
```

For each business represented in the repository, there is a businessEntity element. The businessEntity element describes the business and the location of the business. Inside each businessEntity, there is at least one business service element. The businessService element holds the name and description of a given service. Inside the business service element, the bindingTemplate is found. The content of the bindingTemplate element is a pointer to the service. This pointer typically consists of a URI to the WSDL file. Furthermore, technical data is included in the tModel (technical model) element. Such technical data often concern invocation of the service and typical examples of technical data are information about input and output parameters.

Hollar [37, p109] compares the contents of an UDDI repository with that of a telephone book. The UDDI white pages contain plain contact information about the business. UDDI yellow pages provide a deeper description of the business, while the UDDI green pages describe the actual service and the URI where the service can be found. The observant reader can thus see that white pages correspond to the businessEntity element, the yellow pages correspond to the businessService element, and the green pages correspond to the bindingTemplate element,

As mentioned above, UDDI is intended used for both publishing and consumption of web services. In order to realize this, UDDI specifies two APIs. For consumers, the Inquiry API is defined. By using the Inquiry API, users get able to browse and search the tree structure of the repository. The Inquiry API does also include methods for getting the data when service consumers have found what they were looking for. Publishers have their own API, the Publish API. Through this API, service providers can register their services with the repository. After publishing, the providers may also edit and delete their services through this API.

Appendix B

XML Encryption and XML Signature

In this chapter we provide an illustration of the operation of XML Encryption and XML Signature. The following XML message will be used for illustration.

```
<?xml version="1.0" encoding="utf-8" ?>
<Order>
  <ItemId>7817</ItemId>
  <Amount>2</Amount>
  <Name>John Doe</Name>
  <CreditCardNumber>1234 5678 9012 3456</CreditCardNumber>
</Order>
```

B.1 XML Encryption

We will now demonstrate how XML Encryption works by encrypting the CreditCardNumber element in the XML document above.

We chose to encrypt the element using a session key that is encrypted with a shared key. As can be seen from the result below, the credit card details are replaced with an EncryptedData element, consisting of EncryptionMethod, KeyInfo, and CipherData elements. The EncryptionMethod element says that 256-bit AES was used in CBC-mode. The key described in the KeyInfo element (removed for brevity) was used for encryption, and in the CipherData element we find the encrypted version of the original XML element.

```
<?xml version="1.0" encoding="utf-8"?>
<Order>
<ItemId>7817</ItemId>
<Amount>2</Amount>
<Name>John Doe</Name>

<EncryptedData
Type="http://www.w3.org/2001/04/xmlenc#Element"
xmlns="http://www.w3.org/2001/04/xmlenc#">

<EncryptionMethod
Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc" />

<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
  <EncryptedKey
   xmlns="http://www.w3.org/2001/04/xmlenc#">
    [the encrypted key and associated information]
  </EncryptedKey>
</KeyInfo>

<CipherData>
  <CipherValue>
  [the encrypted CreditCardNumber element]
  </CipherValue>
</CipherData>

</EncryptedData>

</Order>
```

It is clear from the previous example that XML encryption introduces a significant amount of overhead compared to a cleartext XML document.

B.2 XML Signature

To illustrate the operation of XML Signatures, the original XML document presented in the beginning of this chapter will be signed. The resulting XML document is shown below.

```
<?xml version="1.0" encoding="utf-8" ?>
<Order>
<ItemId>7817</ItemId>
<Amount>2</Amount>
<Name>John Doe</Name>
<CreditCardNumber>1234 5678 9012 3456</CreditCardNumber>

<Signature xmlns="http://www.w3c.org/2000/09/xmldsig#">
```

```
<SignedInfo>
  <CanonicalizationMethod
    Algorithm="http://www.w3c.org/TR/2001/REC-xml-c14n-20010315"/>
  <SignatureMethod
    Algorithm="http://www.w3c.org/2000/09/xmldsig#rsa-sha1"/>
  <Reference URI="">
   <Transforms>
   <Transform
   Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
   </Transforms>
   <DigestMethod
     Algorithm="http://www.w3c.org/2000/09/xmldsig#sha1"/>
    <DigestValue>
    [hash of the information that was signed]
    </DigestValue>
  </Reference>
</SignedInfo>

<SignatureValue>
[the digital signature]
</SignatureValue>

</Signature>

</Order>
```

We see that a Signature element, consisting of a SignedInfo and a SignatureValue element, was added to the original document. SignedInfo specifies the method of canonicalization[1], specifies that signing is performed by creating a SHA-1 hash of the orignal document, found in the DigestValue element, and then signing this hash using RSA and an agreed-upon RSA key not specified in the document. The signature can be found in the SignatureValue element.

Just like XML encryption, XML signing introduces a significant overhead.

[1]Canonicalization is performed in order to ensure that two semantically identical XML documents are regarded as identical, e.g. when two elements have switched places.

Appendix C

Policy Mapping Format

The format that is used for mapping web services parameters to input policy definitions is shown below. For the attributes where several choices are possible (e.g. validate), the first alternative is the default one. This means that is the validate attribute is not included for a port, this is equivalent to including validate=”true”.

```
<mapping>
<webservice wsdlPath="path or URL to WSDL document, if used">
  <uris>
    <uri>http://synonym 1</uri>
    .
    .
    .
    <uri>http://synonym n</uri>
  </uris>
  <ports>
    <port name="port name A" namespace="namespace" useWsdl="false|true" validate="true|false">
      <param name="param 1" schemaId="schema 1" useWsdl="false|true" validate="true|false" />
      .
      .
      .
      <param name="param m" schemaId="schema t" useWsdl="false|true" validate="false|true" />
    </port>
    .
    .
    .
  </ports>
</webservice>
.
.
.
</mapping>
```