UNIVERSITY OF OSLO
Department of informatics

A message-level security approach for RESTful services

Master Thesis

60 Credits

Omar Slomic

01 August 2011

Acknowledgements

First and foremost, I want to thank my supervisor, Professor Audun Jøsang, for giving me support and new ideas. By discussing with Audun I became aware of the complexity of the security mechanisms but he also showed me that security can be provided by smart and simple solutions. I would also like to thank my co-supervisor, Professor Frank Eliassen, for giving me feedback even though he was away from work. I would also like to thank my family and friends, but foremost my brother and his wife, for their support and encouragement during the work with this thesis. At last but not least, I would also like to send special thanks to my good friend Aida Golic her time and much support.

Oslo, 31 July 2011 Omar Slomic

Abstract

In the past ten years Web Services have positioned themselves to be one of the leading distributed technologies. The technology, supported by major IT companies, offers specifications to many challenges in a distributed environment like strong interface and message contacts, service discovery, reliable message exchange and advanced security mechanisms. On the other hand, all these specifications have made Web Services very complex and the industry is struggling to implement those in a standardized manner.

REST based services, also known as RESTful services, are based on pure HTTP and have risen as competitors to Web Services, mainly because of their simplicity. Now they are being adopted by the majority of the big industry corporations including Microsoft, Yahoo and Google, who have deprecated or passed on Web Services in favor of RESTful services. However, RESTful services have been criticized for lacking functionality offered by Web Services, especially message-level security. Since security is an important functionality which may tip the scale in a negative direction for REST based services, this thesis proposes a prototype solution for message-level security for RESTful services. The solution is for the most part technical and utilizes well-known, cross-platform mechanisms which are composed together while a smaller part of the solution discusses a non-technical approach regarding the token distribution. During the development of the prototype, much of the focus was to adapt the solution according to the REST principals and guidelines, such are multi-format support (XML or JSON) and light-weight, human readable messages.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AES	Advanced Encryption Standard
AS	Authentication Service
CA	Certificate Authority
CIA	Confidentiality, Integrity and Availability
CMS	Cryptographic Message Syntax
CS	Client-Server
DES	Data Encryption Standard
DNS	Domain Name Service
DOS	Denial-Of-Service
DPRL	Digital Property Rights Language
DRM	Digital Right Management
EAI	Enterprise Application Integration
ESB	Enterprise Service Bus
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
IDEA	International Data Encryption Algorithm
IETF	Internet Engineering Task Force
JMS	Java Message Service
JSON	JavaScript Object Notation
KDC	Key Distribution Center
MD5	Message-Digest Algorithm
MSMQ	Microsoft Message Queuing
OASIS	Organization for the Advancement of Structured Information Standards
OAuth	Open Authorization
P2P	Peer-to-Peer
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PGP	Pretty Good Privacy
PIP	Policy Information Point
PKI	Public Key Infrastructure
RC4	Rivest Cipher
RDF	Resource Description Framework
REL	Rights Expression Language
REST	Representational State Transfer
RPC	Remote Procedure Call
RSA	Rivest, Shamir and Adleman
RSTR	Request Security Token Response
RTS	Request Security Token
SAML	Security Assertion Markup Language

SGT	Service-Granting Ticket
SHA	Secure Hash Algorithm
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
STS	Security Token Service
TGS	Ticket-Granting Service
TLS	Transport Layer Security
UDDI	Universal Description Discovery and Integration
URI	Uniform Resource Identifier
VB.NET	Visual Basic .NET
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WCF	Windows Communication Foundation
WS	Web Services
WSDL	Web Services Description Language
XACML	eXtensible Access Control Markup Language
X-KISS	XML Key Information Service
XKMS	XML Key Management Specification
X-KRSS	XML Key Registration Service
XML	Extensible Markup Language
XPath	XML Path Language
XrML	eXtensible Right Mark-up Language
XSLT	Extensible Stylesheet Language Transformations

1 Introduction

Chapter 1 represents a starting point for this thesis and starts by introducing the motivation for the work. The chapter continues with presenting research goals, scope, research method and ends with a description of the way the thesis is structured.

1.1 Motivation and background

Distributed computing enables data exchange across computers, regardless of their geographical localization. Data that is exchanged is often business related, meaning that one company subscribes on business critical data delivered by another company. Through a contract both companies agree upon a format and a structure of the data and how often data exchange should occur. From the beginning of a relatively short history of distributed computing, many distributed systems emerged presenting alternative approaches of accomplishing the data exchange across different parties. Newer alternatives offered easier programming interface, more functionality and better performance. One of those approaches was Web Services which came into the market in the late 1990s. They were specified and driven by the big software vendors like Microsoft and IBM. Web Services offered not only data exchange, but also tried to accomplish interoperability between different programming languages basing the entire data definition and data exchange on the well-known technologies, XML and HTTP. Since two major software companies began standardizing on Web Services many others followed, and in the early 2000s Web Services framework, originally consisting of three separate specifications, begun to position itself as a de-facto distributed technology. As soon as the industry started using Web Services new requirements became reality. One of those requirements was the end-to-end security or message-based security, which means securing messages until they reach their final destination regardless of the amount of intermediaries or how well the network is secured. In the years to follow many security related specifications for Web Services appeared, extending the Web Services and enabling advanced security mechanisms, like cryptography, trust negotiation and single sign-on. But all those extensions did not just solve the challenges that the industry posed, they also made the Web Services one of the most complex distributed systems of the modern time. Today, creating a Web Service without the use of third-party tools is almost impossible, especially when it comes to extensions for message-based security.

REST, an architectural style based on existing HTTP functionality, was described around the same time as the Web Services originated but since it was not backed by big corporations its inception was almost unnoticed. Several years after its original description, services based on REST, also called RESTful services, began to gain more popularity in the developer community, mainly due to their simplicity. As the time moved on the industry started to pay more attention to the developer communities and started to offer services based on REST. Since RESTful services gained momentum it became the target of comparison against Web Services. One of the functionalities RESTful services are missing is the ability to ensure end-to-end security. While standard REST security is defined on the transport level by enabling TLS/SSL it does not impose security on the messages directly but instead it secures the transport layer. Once an intermediary party receives the data from the sender, the message becomes unsecured and its content visible in its original format. This is in contrast to end-to-end security principals which ensure secured message content until its final destination. End-to-end security is also known to protect the content of certain parts of the message while other parts may be left intact. This is often done to allow message routing based on those unprotected parts that are understandable. Such processes are often handled by intermediate systems like enterprise application integration systems (EAI) or enterprise service bus (ESB), which are implemented in many larger companies and responsible for routing and transformation of the incoming and outgoing messages. The messages are often routed to their final destination systems based on certain parts of the message content.

In our opinion, not offering end-to-end security may be the biggest limitation regarding REST architecture. This absent functionality may lead to discouragement of choosing RESTful services when working with sensitive data. In some cases this may also lead to creation of unstandardized workarounds. Therefore, in this thesis we want to propose a solution to end-to-end security for RESTful services. The thesis itself is a result of the author's curiosity for the REST architecture and is not written in context of any ongoing project.

The case study that will be used to test the solution is based on an application for customer registration. Even though the case study is fictive, we believe that we are covering many test scenarios and through those we are able to relate to the real life security challenges.

1.2 Research goals

The main objective of this thesis is to propose a solution for enabling message-based security for RESTful services. This includes message integrity and confidentiality, user authentication and token distribution. Before the design process we saw the need to investigate message-level security solutions and approaches implemented on similar technologies so to be more inspired and accumulate new knowledge. In order to support message-level security in RESTful services we knew we had to solve challenges regarding some basic RESTful capabilities like format flexibility and simple, human readable messages. Format flexibility is a feature where a message may be defined in multiple formats, thus our implementation had to be adoptable to this behavior and be able to ensure integrity and confidentiality on multiple formats. Simple and human readable messages is a REST architecture trademark where the message content is defined in a plain manner and as such is in contrast to complicated multi-schema constrained formats like SOAP or RDF. Our goal was to keep the messages in a simple manner even after integrity and confidentiality has been applied.

Finally, the ultimate goal was to create a new library that would solve many of the challenges regarded message-level security and the library was to be developed in a widely-used REST API in order to target bigger developer audience.

1.3 Scope

The investigation of an existing approach for message-level security was based on Web Services. Although there are many differences between the Web Services and RESTful services, we found them to be similar in certain areas since both support XML as message format and utilize HTTP to exchange data. The investigation part is described in chapter 3.

While RESTful messages may be exposed through multiple formats, in this thesis there was a focus on protecting XML and JSON formats since they are some of the most used ones. XML is a standard format for representing data in a structured manner[5](p.8). JavaScript Object Notation (JSON) is also used for representation of structured data but its design goals are to be “minimal, portable, textual, and a subset of JavaScript”[81]. Because of this JSON offers simpler structure and size than XML.

Further on, we found WCF framework to be the most used one for the creation of the RESTful services on the .NET platform. The .NET platform is a Microsoft developed software platform containing libraries, tools and runtimes to develop and execute software. The platform allows software to be developed in several programming languages such as C#, VB.NET and J#[53]. WCF is an acronym for Windows Communication Foundation and is a universal framework for building distributed services on the .NET platform[82](p.1). It is also probably one of the most used APIs for RESTful services in general. This claim is based on forum discussions and by searching for *books* and *RESTful* on www.amazon.com where the search engine produced many WCF related books[107]. That is why we used .NET and WCF to develop the security library prototype. Additionally, C# was chosen as the programming language.

So to create a smoother implementation on the existing RESTful projects and avoid create dependencies upon additional libraries, our new security library was developed solely by libraries included in the .NET platform and WCF, i.e. no third-party libraries were used. This was done intentionally to ease the adoption of the new library.

1.4 Research method

For this thesis a technological research method was applied. The research method is described in Technology Research Explained, written by Solheim and Stølen[98]. As stated by the authors this research method is “concerned about how to make new artefacts or improve existing”[98](p.7), which is what we were trying to achieve in this work.

1.4.1 Description

The research process starts by collecting requirements for the artefact. When requirements are in place the process continues by designing and creating an artefact. This is the innovative phase requiring creativity and technical insight of the researcher. At the end, the produced artefact has to demonstrate that it actually fulfills the specified requirements and satisfies the need on which it is based. The overall hypothesis is: *The artefact satisfies the need*. The overall hypothesis can be evaluated positively if the falsifiable predictions derived from the posed requirements converge. If the evaluation results diverge, the researcher has to repeat the whole process resulting in adjusting the requirements, possibly build a new artefact and evaluate it. Figure 1.1 shows this iterative activity.

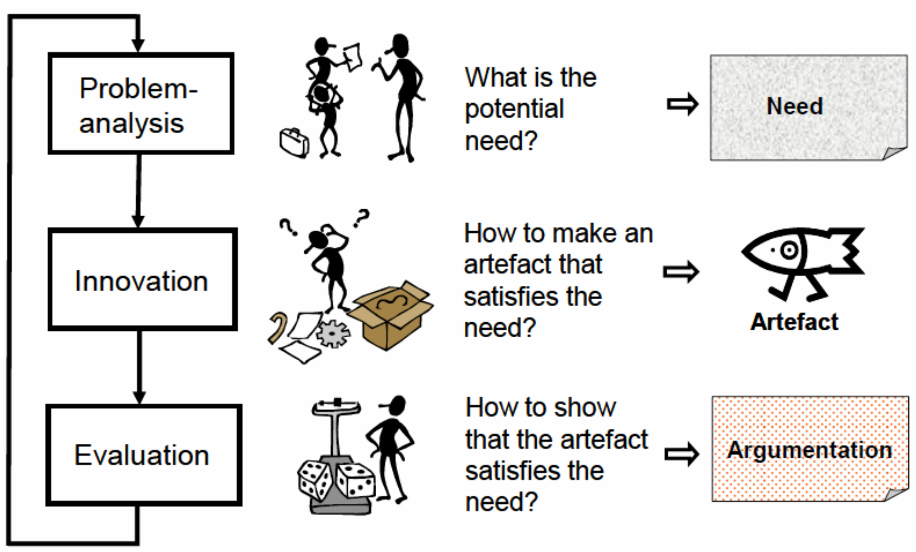

Problem-analysis
What is the potential need?
Need
Innovation
How to make an artefact that satisfies the need?
Artefact
Evaluation
How to show that the artefact satisfies the need?
Argumentation

Figure 1.1: Technology research steps [98]

1.4.2 Problem Analysis

Our thesis is concerned with offering a complete solution for message-level security for RESTful services. Before we started with planning of the solution we studied similar functionalities offered by Web Services. We realized that our solution will need to address three separate modules:

1. Authentication
2. Token distribution
3. Message protection

In this thesis authentication and token distribution are modules that did not require development of any new artefacts, as shown in the sections 4.3.2, 4.3.3 and 4.3.4, but they did require creativity and technical insight, token distribution in particular.

The only module that required development of new artefacts was the message protection module. Message protection module is the one responsible for encrypting and applying digital signatures on XML and JSON messages in the context of WCF REST API. At the end, message protection module resulted in what we refer to as the new security library for RESTful services.

1.4.3 Innovation

Before we started on the design and development of the message protection module we needed to identify requirements for it. Those requirements are presented in section 2.1. After requirements were specified, we did a small proof of concept on WCF, SOAP and REST API to learn more about those implementations and look for reusable components. The ultimate goal of proof of concept was to prepare us better for the planning process and this is presented in section 4.2.

While the innovation part which includes design and implementation was well-planned in our opinion, there were times when we had to reconsider the design due to the problems related to the implementation. Design and implementation are presented in chapter 4 and 5 respectively.

1.4.4 Evaluation

The evaluation part is the one confirming if the expected predictions meet our expectations. As mentioned earlier, predictions are based on requirements and both are presented in the chapter 2 while the evaluation of the predictions is presented in the chapter 6.

1.5 Outline

The rest of this thesis is organized as follows. Chapter 2 provides success criteria in form of requirements, predictions and hypothesis which the final solution is evaluated by. Chapter 3 presents an overview of the distributed architectures discussed in this thesis, Web Services and RESTful services, and their security repertoire. The solution design is thoroughly discussed in chapter 4, whereas the implementation, technical details and test results regarding message size, are presented in chapter 5. Chapter 6 provides an evaluation to predictions and hypothesis made in chapter 2 while chapter 7 sums up the work done and presents further work. Additionally, the thesis includes two appendices where Appendix A explains the DVD content and the structure of the solution, while Appendix B explains the sample code found on the DVD.

2 Criteria

Following sections describe requirements, predictions and hypothesis that will be used to guide us through the development process and evaluate the final product.

2.1 Requirements

There is a set of absolute requirements that we feel should be fulfilled in order to support real-life security scenarios. These absolute requirements form the basis and reflect the purpose of our solution.

R1: The very first requirement relates to the formats of RESTful services. By developing a service in WCF, a RESTful service may offer either XML or JSON format. Therefore, ***the solution needs to support encryption and digital signatures for XML and JSON formats.***

R2: The second requirement is about full and partial message security. Regarding the nature of systems responsible for message routing, some properties or elements must be in clear text so that the message may be routed to the specific end system. In some cases the requirement can be to encrypt certain properties and sign those. In other cases the requirement can be to sign a combination of both encrypted and unencrypted properties. Therefore, ***the solution needs to support different combination schemes of encryption and digital signatures for the complete message or selected message properties.***

R3: The third requirement is concerned with the overhead upon a RESTful message. Since a RESTful message is a pure representation of the domain object in one of the supported formats, the message is easy to read for the humans, especially when compared to SOAP messages. Therefore, ***when encryption and signature are***

applied, the message should not become much larger and should be simple to read.

R4: The fourth requirement relates to the existing RESTful projects. The idea is to design and implement our solution in such way that the existing projects developed in WCF may enable message-based security with least effort. Therefore, ***the solution must be adaptable with the current RESTful projects developed in WCF.***

R5: The fifth requirement is about the interoperability between different platforms and programming languages. RESTful services are based on HTTP which is heavily supported on all popular platforms. Although our solution is to be created on .NET platform, the secured messages may still be downloaded by another platform but the messages may not be understandable. That is why we should strive to secure and structure messages in that way so a solution created on a different platform may understand those messages without much effort. In other words we should make it easier for a developer developing on a non-.NET platform to create a RESTful client or service, and exchange protected messages with RESTful client or services developed on .NET and WCF. This can be achieved by utilizing well-known security mechanisms, implemented on most of the platforms. Therefore, ***secured messages should implement cross-platform encryption and digital signature mechanisms in order to be more interoperable.***

Finally, it is worth mentioning areas we will not consider in this thesis. The first one is caching which we acknowledge to be important but due to the complexity and time limit this functionality will be ignored. The second area is performance which in our case may be measured by the time it takes to protect a message or the time is takes to accomplish the message exchange. Although an important parameter in every distributed environment this will not be on our highest priority list but we will always keep it our mind while developing the solution.

2.2 Hypothesis

This thesis defines following hypothesis:

H1: Proposed solution will enable message-level security for RESTful messages on .NET platform.

2.3 Predictions

Given that the requirements are specified and the hypothesis is stated we proceed with the definition of the predictions or success criteria. These predictions will be used to test the hypothesis.

P1: The new artefact, message protection library, later referred to as the new security library, will enable encryption and digital signatures on XML and JSON messages.
P2: The new artefact will support partial encryption and partial digital signature.
P3: The new artefact will enable messages compression and decompression on protected messages so to decrease message size.
P4: The new security solution will be easily adoptable by existing and standard WCF RESTful services.

Since P4 is relatively hard to falsify because it is composed by two relative statements, “easily adoptable” and “standard”, we feel the need to explain those in more details.

“Easily adoptable” in this context refers to the usage of the new artefact to protect the message content which will not require existing service code to be changed but will require new code to be added to the existing service solution. In addition it is worth mentioning that the authentication and token distribution will be handled outside the service code so it will not require any structural changes but will require additional configuration to be specified.

“Standard” in this context refers to the service code which defines a service, its service contract and where none of the HTTP security features are enabled. Those “standard” solutions are generally found on the Internet and are often used for demonstration purposes. Following link from Microsoft demonstrates what we consider to be a standard solution[97].

3 Related work

3.1 Intro

This chapter introduces a description of distributed technologies used in this thesis and their security mechanisms. It is assumed that the reader has basic knowledge of concepts related to programming and XML.

The chapter starts with section 3.2 presenting the basic concepts and characteristics of a distributed system. The section continues with describing Web Services and RESTful services in 3.2.1 and 3.2.2 respectively. Section 3.3 presents general security concepts as well as the security of Web Services and Restful services. The first three subsections of 3.3 are introduction to security systems and security technology. Subsection 3.3.1 starts with introducing basic security goals and security terminology. 3.3.2 subsection is about cryptography and is divided in three subsections presenting encryption, hash functions and digital signatures, each starting by a short historical overview, continuing with description, pros and cons, and ending with their field of usage. The subsection 3.3.3 is about authentication services, giving introduction about some of the most used authentication systems. Security mechanisms in RESTful services are presented in subsection 3.4 while security in XML and Web Services are presented in subsections 3.5 and 3.6 respectively. We end this chapter by discussing couple of community efforts regarding REST security 3.7.

3.2 Distributed systems

According to Coulouris et al., "a distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages"[64](p.1) Computers that are part of the network may be geographically located in the same room or far from each other, like on different continents. Probably the most famous example of a distributed system is the Internet where millions of computers are connected with each other and where data is exchanged between people, applications and machines.

This section introduces the most important and widely accepted concepts and ideas behind distributed systems. Moreover, a brief overview of Web Services and RESTful services will be given, also pointing to their strengths and weaknesses.

One of the simplest but also common models of a distributed system is the client-server model composed of a server and a client. A server is a machine hosting an application that offers resources valuable to the clients. A client is a machine that runs a client application able to communicate with the server application and collect or update the resource. A distributed system offers many challenges that generally do not apply to standalone systems. These challenges are described in Table 3.1.

Table 3.1: Challenges in a distributed system [64](p.16-24)

Heterogenity:	A distributed system is composed of the modules that runs on different hardware, networks, operating systems and are made in different programming languages
Openness:	A distributed system should be extensible for its consumers by publicly publishing its interfaces
Security:	Communication between multiple computers must be protected
Scalability:	A distributed system should avoid performance bottlenecks when a new user is introduced
Failure handling:	A distributed system must avoid single point of failure, meaning that if a failure should occur in a process, a machine or a network, it should not bring the whole system down
Concurrency:	A distributed system must ensure multiple operations at a time. In addition it must ensure that all those operations are independent of each other
Transparency:	A distributed system should make certain aspects invisible to make it easier for the programmer to concentrate on the application development rather than details about the distributed system implementation.

Through the history of computer science many distributed systems organized as middleware emerged. A middleware is term for a software providing a programming model, transparency and independence above the building blocks like communication protocols, operating systems and hardware [64](p.166). The middleware can be divided into four categories by the way they function[65]. Those categories are: remote procedure call, object oriented middleware, transaction oriented middleware and message oriented middleware.

Table 3.2: Middleware categories

Remote procedure call (RPC):	Enables remote invoking of the procedures as if they were internal procedure calls by using interface definition language (IDL). Much like traditional procedure calls, remote procedure calls are synchronous which means that the caller waits for a response from the procedure[61]. Examples of RPC are Sun RPC and Isis.
Object-oriented middleware:	Beyond RPC, IDL supports object types as parameters. In addition it supports inheritance and failure handling[62]. Examples of object oriented middleware are CORBA, Java RMI and DCOM.
Transactional-oriented middleware:	Support transactions across different distributed database systems by implementing two-phase commit protocol[63]. An example of transaction oriented middleware is ODTP XA.
Message-oriented middleware:	Sends and retrieves messages to and from a queue system by encompassing publish/subscribe and message queuing communication patterns[61]. Asynchronous communication is the natural choice of this category. Examples of the message oriented middleware are JMS and Microsoft MSMQ.

3.2.1 Web Services

The World Wide Web Consortium (W3C) defines a Web Service as “a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.”[66]. Rajendran & Balasubramanie has a slightly different definition where “Web Service refers to a modular, self-contained piece of code with a well-defined functionality that you can access across the network.”[67].

Web Services, also known as SOAP based services, are an industry standard used by many companies and organizations[86]. They are formed out of three separate specifications, each defining its own domain of usage. The first specification is SOAP which is a protocol for message exchange, described in XML[72]. The second specification, WSDL, is also described in XML and provides information about network services, their requirements and returning values[73]. The third specification is UDDI which is described in XML as the other two specifications, and which provides a way to register service descriptions, find service descriptions by specifying a criteria and retrieve descriptions[74]. All these specifications are required to be implemented in order to deliver a complete distributed system. Today, multiple programming languages have implemented these specifications consequently enabling cross-platform message exchange, service description and service discovery[88].

By default, Web Services builds on the Internet protocols TCP/IP and HTTP to send and receive messages. HTTP is not the only application level protocol supported but it is the mostly chosen one since firewalls often do not prevent HTTP communication on port 80 as it might be the case with other protocols or ports.

Web Services may be seen as part of RPC middleware category since their communication involves calling procedures residing on another computer. The communication is initiated by client sending request messages to a service hosted on a server. The server processes the request and sends a respond message back to the client. Since Web Service provides abstracted failure handling and allow objects to be passed as parameters when making remote calls, we may say that the Web Services mirrors similarities found in the object-oriented middleware. However, there exist additional specifications that extend the original Web Services capabilities and make them somehow compatible with the other two middleware categories. These additional specifications are known as WS-*. A WS-* specification like WS-AtomicTransaction[70] extends the SOAP protocol and enables two-phase protocol and distributed transactions between multiple parties, thus fulfilling one of the main requirements for transaction-oriented middleware. Message-oriented middleware requires asynchronous message exchange based on publish/subscribe pattern where message delivery is ensured. Web Services achieve similar qualifications by implementing WS-Eventing[68], another WS-* specification, enabling asynchronous communication and publish/subscribe pattern, while WS-ReliableMessaging[69] specification can be used to ensure message delivery.

3.2.1.1 SOAP

SOAP, a W3C recommendation, is a platform independent protocol for message exchange described in XML. Abbreviation originally stood for Simple Object Access Protocol but of version 1.2 the acronym was dropped[72].

In 1998 a group of developers from Microsoft, Userland Software and DevelopMentor Incorporated started on the SOAP project. Later on, other companies like IBM contributed on the project which formed SOAP version 1.1[75]. The basic blocks of a SOAP message have been consistent since its first version. A SOAP message starts with of a top XML element called Envelope and a specific namespace, uniquely defining for the whole world that the XML document is a SOAP message. The namespace may either have the value of http://schemas.xmlsoap.org/soap/envelope describing SOAP 1.1 or http://www.w3.org/2003/05/soap-envelope describing SOAP 1.2. Enevelope-element contains two sub elements called Header and Body. Header-element is optional and contains additional information about the message exchange like security mechanism used to protect the message. Body-element contains the message itself where the message is either an operation described in XML or a returning XML type. If it is a request message from a client then the message will contain the operation name and its parameters if required. On the server the operation will be mapped to an executable method and executed. After its execution, the service will create a SOAP response message containing the returning type from the operation.

```
POST /Service/Soap11 HTTP/1.1
Content-Type: text/xml; charset=utf-8
SOAPAction: "urn:ICustomerService/GetListOfCustomersByName"
Host: ana2
Content-Length: 163
Expect: 100-continue

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
		<s:Body>
				<GetListOfCustomersByName>
						<name>Ola</name>
				</GetListOfCustomersByName>
		</s:Body>
</s:Envelope>
```

Figure 3.1: SOAP request

```
HTTP/1.1 200 OK
Content-Length: 693
Content-Type: text/xml; charset=utf-8
Server: Microsoft-HTTPAPI/1.0
Date: Fri, 01 Apr 2011 13:49:41 GMT

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
   <ActivityId CorrelationId="26ab7785-dd9f-432a-a57a-7a29d0270a3e"
Xmlns="http://schemas.microsoft.com/2004/09/ServiceModel/Diagnostics">
		353d7ea9-7603-4ecc-a1e7-53d3e9338f66
   </ActivityId>
 </s:Header>
 <s:Body>
  <GetListOfCustomersByNameResponse>
   <GetListOfCustomersByNameResult xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
    <Customer>
     <Id>14</Id>
     <LastName>Normann</LastName>
     <Name>Ola</Name>
     <Address>
       <Id>0</Id>
       <PostalCode>1567</PostalCode>
       <Street>Christian Michelsens gate 6</Street>
       <City>OSLO</City>
     </Address>
    </Customer>
   </GetListOfCustomersByNameResult>
  </GetListOfCustomersByNameResponse>
 </s:Body>
</s:Envelope>
```

Figure 3.2: SOAP response

The previous two figures demonstrate SOAP request and response on the HTTP. SOAP messages are wrapped in the Envelope-element and belong to the HTTP body part while the data prior to Envelope defines the HTTP header. The SOAP request specifies an operation called GetCustomerByName with a parameter Name and parameter value Ola.

The service is found on the location, also referred to as URI, http://ana2/ICustomerService/GetListOfCustomersByName which in this case is defined concatenating values found in the Host and SOAPAction header properties. The SOAP response message contains Customer-objects where each customer contains members like name, last name and an Address-object containing street name, postal code and city name.

Web Service API defined by many programming languages offers transparency for operation mapping, creation of the SOAP messages and parsing of the SOAP messages. The API is also responsible for creating proxy classes so the programmer does not need to know the details about where the service is or how to connect to it. The process of transforming types defined in a programming language to and from XML types, also known as the serialization and deserialization, and in most cases this transformation is handled automatically. The API makes it possible for a programmer, even with poor SOAP skills, to start creating and exchanging messages. Figure 3.3 shows C# types before being serialized to SOAP types as demonstrated in Figure 3.2.

```
public class Customer
{
        public int Id { get; set; }
        public string Name { get; set; }
        public string LastName { get; set; }
        public Address Address { get; set; }
}

public class Address
{
        public int Id { get; set; }
        public string Street { get; set; }
        public int PostalCode { get; set; }
        public string City { get; set; }
}
```

Figure 3.3: C# types

As already mentioned, SOAP messages are often transported on the HTTP but many other application protocols or even distributed middleware systems may be used to accomplish the same task. For instance, message-oriented middleware like JMS may be used to transport the messages from a party to the another[77].

3.2.1.2 Web Services Description Language (WSDL)

WSDL , also a W3C recommendation, provides detailed description about services specified in a specific XML format[73]. Its history dates back to year 2000 when the SOAP gained wider acceptance and there was a need to describe services and their requirements. In the fall of 2000 IBM and Microsoft announced WSDL as a joint accomplishment[75].

A WSDL is often hosted on the same machine as the service. It presents a description about operations, operation parameters and returning types to the client. By reading the WSDL the Web Services API on the client side knows how to contact the service and what may be expected from it. Nevertheless, the WSDL is optional and is not required if the clients know how to send and retrieve messages from the Web Service or the definition has been documented in another way.

```xml
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions targetNamespace="" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:wsu="http://docs.oasis-
        open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" >
  <wsdl:types>
    <xsd:schema targetNamespace="/Imports">
      <xsd:import schemaLocation="http://ana2/Service?xsd=xsd0"/>
      <xsd:import schemaLocation="http://ana2/Service?xsd=xsd1"
              namespace="http://schemas.microsoft.com/2003/10/Serialization/"/>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="ICustomerService_GetListOfCustomersByName_InputMessage">
    <wsdl:part name="parameters" element="GetListOfCustomersByName"/>
  </wsdl:message>
  <wsdl:message name="ICustomerService_GetListOfCustomersByName_OutputMessage">
    <wsdl:part name="parameters" element="GetListOfCustomersByNameResponse"/>
  </wsdl:message>
  <wsdl:message name="ICustomerService_InsertCustomer_InputMessage">
    <wsdl:part name="parameters" element="InsertCustomer"/>
  </wsdl:message>
  <wsdl:message name="ICustomerService_InsertCustomer_OutputMessage">
    <wsdl:part name="parameters" element="InsertCustomerResponse"/>
  </wsdl:message>
  <wsdl:message name="ICustomerService_Version_InputMessage">
    <wsdl:part name="parameters" element="Version"/>
  </wsdl:message>
  <wsdl:message name="ICustomerService_Version_OutputMessage">
    <wsdl:part name="parameters" element="VersionResponse"/>
  </wsdl:message>
  <wsdl:portType name="ICustomerService">
    <wsdl:operation name="GetListOfCustomersByName">
      <wsdl:input wsaw:Action="urn:ICustomerService/GetListOfCustomersByName"
              message="ICustomerService_GetListOfCustomersByName_InputMessage"/>
      <wsdl:output wsaw:Action="urn:ICustomerService/GetListOfCustomersByNameResponse"
              message="ICustomerService_GetListOfCustomersByName_OutputMessage"/>
    </wsdl:operation>
    <wsdl:operation name="InsertCustomer">
      <wsdl:input wsaw:Action="urn:ICustomerService/InsertCustomer"
              message="ICustomerService_InsertCustomer_InputMessage"/>
      <wsdl:output wsaw:Action="urn:ICustomerService/InsertCustomerResponse"
              message="ICustomerService_InsertCustomer_OutputMessage"/>
    </wsdl:operation>
    <wsdl:operation name="Version">
      <wsdl:input wsaw:Action="urn:ICustomerService/Version"
             message="ICustomerService_Version_InputMessage"/>
      <wsdl:output wsaw:Action="urn:ICustomerService/VersionResponse"
             message="ICustomerService_Version_OutputMessage"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

Figure 3.4: WSDL

3.2.1.3 Universal Description, Discovery and Integration (UDDI)

UDDI is an OASIS standard defining a scheme to publish and discover information about Web Services[76]. The history behind UDDI was created by the need for public advertisement and location of the Web Services. This was seen as the final missing piece to the Web Services framework back in the March 2000. In this fashion service discovery and communication could be accomplished across different parties independent of their geographical location. IBM, Microsoft and Ariba started working on a solution together and formed the first version of the UDDI in September 2000[75].

UDDI, as an implemented software product, offers a repository for service descriptions. A UDDI Web Service runs above the UDDI repository and is responsible for registration and discovery of the service descriptions. Even though UDDI is treated as a vital part of the Web Service framework, it might also be used to handle other kind of service descriptions than the standard Web Service description, WSDL. Therefore, all service descriptions published to UDDI are mapped to UDDI description definitions. UDDI tModel is an example of such definition. In a typical Web Service registration scenario, a service provider would send the WSDL to the UDDI by using the UDDI Web Service and the UDDI service will map the WSDL to the tModel description before storing it in the UDDI repository[78].

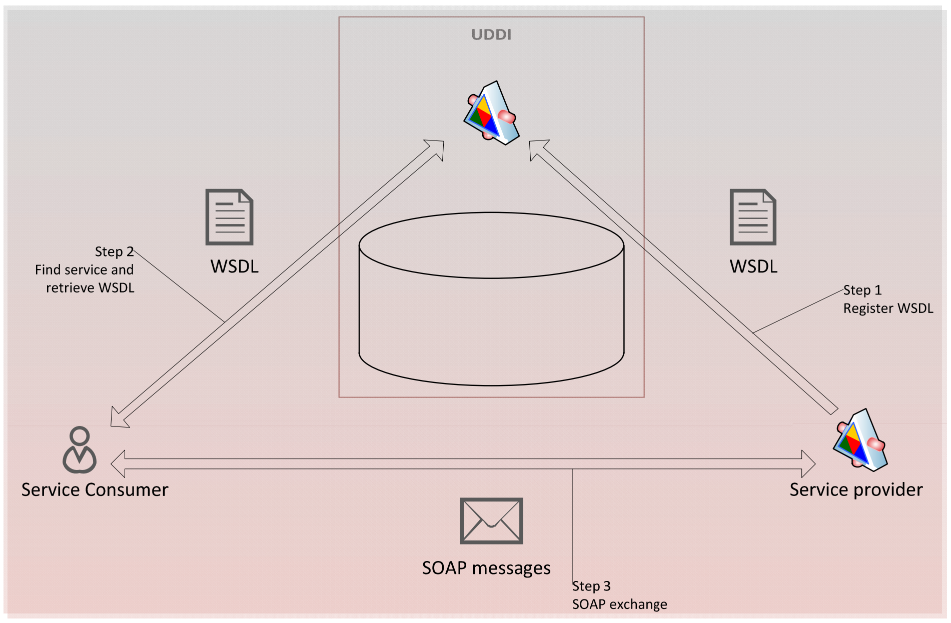


Figure 3.5: UDDI registry and discovery process

In the world of network based services, the client is often referred to as the service consumer while the server is referred to as the service provider. Figure 3.5 shows how a service provider makes its Web Services known to the public. Firstly, the service provider publishes its WSDL to the UDDI registry by sending a SOAP containing a UDDI message to the UDDI service. Since UDDI is universal service repository it will map the WSDL to tModel. In addition to WSDL, other properties describing both the service and the service provider should be registered so the service is discoverable by organization properties like business type or company name. In another organization a service consumer looks up for this specific service to accomplish a certain task. By some criteria given by the service consumer UDDI is able to find a matching service so the UDDI passes the WSDL URI to the service consumer wrapped in a SOAP containing UDDI response message. The service consumer receives the SOAP message, extracts the URI of the WSDL and finds the WSDL on the network. By now the consumer have all necessary knowledge about the service and is able to establish the connection with it.

3.2.2 RESTful Services

Representational State Transfer (REST) according to its creator Dr. Roy Fielding is a “coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations.”[79]. Indeed, REST is not a new standard nor a new distributed system but an architectural style for building services on the Web, fully exploiting the functionality of the HTTP. It does not add any new specifications but utilizes those already defined in the HTTP.

Back in 2000 REST was firstly described in Roy Fielding’s Ph.D. dissertation, the same man who also co-authored on the HTTP standard[85]. He acknowledged the big success of the Internet and believed that all the resources on the HTTP should be identified in one common way. Every single resource should be identified by its own unique URI so that it may be referenced from other resources. Actions upon the resource should be decided by HTTP methods and communication between a client and a server should always be stateless, just like the regular HTTP communication. In the HTTP world the term resource is used for any type of document hosted on the Internet, like HTML, XML or JPEG.

When it comes to services based on REST architecture, popularly called RESTful services[3](p.5) or RESTful web services[1](p.8), the same principals apply as for the other resources. A RESTful service is also referred to as a resource, and a request or a response is just a plain HTTP exchange with the URI to that resource. The URI is used to download the state of the resource to the requestor, potentially change the state and uploaded it to the same URI[87]. The HTTP exchange may also contain a message in the HTTP body presented in a chosen format. Some of the typical message formats are XML, XHTML, JSON and RSS/Atom where XML is probably the most typical one[3](p.9-10). Since RESTful messages are tied to the HTTP they are not neutral regarding the transporting protocol like SOAP messages. In contrary to SOAP, they do make use of the HTTP mechanisms like caching which makes them quite scalable[3](p.2).

The RESTful service architecture is dependent of the HTTP methods[1](p.8), also referred to as HTTP verbs[3](p.7). There are eight methods defined by the HTTP 1.1 standard[80] but only four of them are actively used by the RESTful services. Those four are GET, POST, PUT, and DELETE, where GET is used to retrieve messages, POST is used for creation of new messages, PUT is used for both creation and modification, and DELETE is used to delete the data[3](p.8). These verbs are specified by the client in a HTTP request and in combination with the resource URI it tells the service what type of action is expected from it. SOAP HTTP requests on the other hand, are always specified with the POST method, irrelevant of the action type[3](p.9). Following two figures demonstrates two HTTP requests formatted accordingly to REST guidelines with the same URI but different HTTP methods. The first one call for data retrieval specified by an identifier equals 1 while the second one deletes the data with the same value.

```
GET RestService/Customer/1 HTTP/1.1
HOST: ana2
```

Figure 3.6: HTTP request with GET

```
DELETE RestService/Customer/1 HTTP/1.1
HOST: ana2
```

Figure 3.7: HTTP request with DELETE

The messages representing objects are given in their original form. For instance, the Customer-element from Figure 3.3 will be serialized to XML and no additional elements will be added. Compared to the SOAP message in Figure 3.2, the REST version in Figure 3.8 contains pure XML object, is smaller and easier to read by humans. Smaller message size and simpler processing require lower resources in term of power and hardware, hence making RESTful services more suitable on smaller devices then Web Services[84].

```
HTTP/1.1 200 OK
Content-Length: 193
Content-Type: text/xml; charset=utf-8
Server: Microsoft-HTTPAPI/1.0
Date: Fri, 01 Apr 2011 13:49:41 GMT

<Customer>
    <Id>14</Id>
    <LastName>Normann</LastName>
    <Name>Ola</Name>
    <Address>
      <Id>0</Id>
      <PostalCode>1567</PostalCode>
      <Street>Christian Michelsens gate 6</Street>
      <City>OSLO</City>
    </Address>
</Customer>
```
Figure 3.8: HTTP response with XML in the body

The REST API is heavily based on HTTP API, something all major programming languages support. It is not as abstracted as the Web Services API and generally requires little language support beyond sending and receiving the HTTP streams. Because of this, the REST API is very simple. Through WSDL 2.0 and WADL service definition is offered but these are not widely supported and are target of concerns that such functionality conflicts with the core flexibility of REST services[88]. Because of the description lacking a client needs to gain knowledge about the service in another way, like reading the documentation.

Many companies and organizations have embraced REST and offer services based on its principals. Many of them including Google, Microsoft[3](p.251), Yahoo and Amazon[1](p.49) offer different services and even technologies on top of their REST API.

3.3 General Security

3.3.1 Security goals

One of the primary goals of any company in the world should be to protect their assets. Assets may be of both tangible and intangible character. Tangible can be grouped into buildings, computers and other equipment while intangible assets may stand for product and business secrets. Both product and business secrets stand for inexpressible values, both from the monetary perspective as well as from the competitive perspective. For instance in pharmaceutical industry, a chemical composition of a unique medicine may create such an advantage for a company that the company may overcome competition in a certain field thus gaining contracts, reputation and huge economic boost. Cerezyme, a medicine for rare Gaucher disease, cost $200000 for the average patient and annual sales are above $1000000000[6] because there is no real alternative.

The Web Services and the RESTful services are also some of the intangible assets that may represent the portal to valuable business processes. When company's data are shared with customers through these services it becomes their responsibility to provide correct data in secure manner at agreed time period. Several models have been proposed to describe the goals of a security policy. According to Hollar & Murphy these goals can be divided into classic security goals and transactional security goals[5](p.53).

3.3.1.1 Classic security goals

Security concerning IT and information is traditionally defined by the "CIA triad" or "CIA triangle" where the acronym CIA stands for confidentiality, integrity and availability[8](p.98). These three aspects define the heart of any secure system and most systems cover all of them[5](p.54). There exist conflicts between those three goals but the area in the center is the place where all three work together and define security.

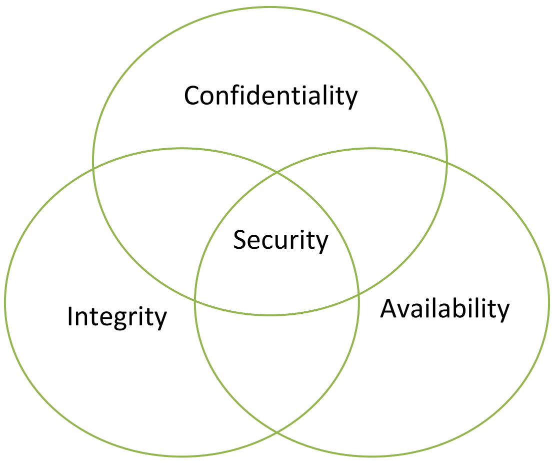


Figure 3.9: CIA Triad

Confidentiality

Confidentiality of information is occupied with concerns of ensuring that the information is read and understood by trusted, authorized parties[5](p.54). Any other party will be prevented from accessing the information, sometimes even to the extent of keeping unauthorized parties from knowing that the information exists[5](p.54). Data is often classified at varying levels of sensitivity making it sometimes available to all authorized users and other times to a smaller group of users. Confidentiality is often ensured by the use of cryptography. Although privacy and confidentiality are very much related, privacy seeks to protect the information from being read while confidentiality covers all forms of access like reading, copying, printing, and so on[5](p.54).

Integrity

Information integrity is occupied with keeping the information in correct form and not allow modification by parties or artefacts without control[8](p.99). The main concern of information integrity is to protect the information against attempts to tamper with it and make sure it is always correct. In a banking system we would like the system to detect if our information is changed by someone else while we are about to commit a payment. There exists a conflict between confidentiality and integrity where confidentiality is concerned with concealing the information while integrity doesn't care if the information is read as long as it remains correct[5](p.55).

Availability

Availability is occupied with that information can be accessed or used by authorized parties and artefacts when required[8](p.99). Any information that is not available when required is of a limited value. In the banking system example, while we are about to pay for a set of cheap airplane tickets and the banking transaction service is down due to an error, most probably those tickets will have a higher price next time the service is up. It is important to understand that this policy is concerning authorized requests since an unauthorized user with the knowledge about the resource may flood the system with requests so that the system would not be able to serve authorized users (DOS attack)[5](p.55). As with integrity, there is a conflict between confidentiality and availability where confidentiality tries to make the information unavailable or hidden for an unauthorized person while availability tries to be available at any time[5](p.55).

It is important to understand that security objectives between organizations vary in the sense that confidentiality may be top prioritized in an organization like a hospital, while availability may be the most prioritized goal in an organization offering products and services[8](p.100).

3.3.1.2 Transaction security goals

Distributed transaction-based systems have additional security requirements that are not completely covered by the traditional CIA triad. Those systems represent a more complicated security scheme than standalone systems, not only considering data transmission over a network but also considering who and under what privileges will be allowed to run different components[5](p.60). These additional goals are authentication, scalability and non-repudiation.

Authentication

Authentication is a security goal seeking to validate a user’s identity[5](p.57). There are several ways of authenticating a user on a network. A common web site authentication mechanism is done by username and password. In other cases a user may provide a unique binary value to the server and in that way the server will know who the user is.

The insecurity of network communication, client side software and various other security issues pushes for a more secure exchange of identity information. Protocols must be carefully chosen or created as well as strong cryptography is needed to ensure the true identity of a user[5](p.57). In cryptography, encryption is a process that enables transformation of human readable text to non-readable text (ciphertext) and decryption is the inverse operation[7]. The encrypted text can only be understood by the parties or artefacts sharing an appropriate digital secret key meaning that parties without the correct secret key will not be able to decipher the encrypted text.

Scalability

Scalability is a transactional goal regarding system growth or adaption as demand increases[5](p.58). It is a goal indirectly linked to security through issues related to system bottlenecks and single point of failure. System bottlenecks are weaknesses that limit the full potential of a system[9]. The bottlenecks are often related to hardware limitations but they may also be related to software as well[9]. An example of a bottleneck may be a process that operates a huge amount of transactions. If the system is not able to handle all those transactions in a reasonable amount of time then the system is not working properly. Because of the longer processing time these bottlenecks will eventually make the customers unhappy and that may lead to quick fixes, which again may lead to poorer security. Single point of failure is a critical part of a system which may take down the whole system if it fails. The term refers also to network components or systems that may take down the entire network in case of failure. A good example of this is a wireless network switch. If it fails all computers using that switch will lose internet connection. Domain Name Service (DNS) on the other hand, is an example of a scalable system. DNS replicates its data across multiple computers so in case of an unexpected failure on a one machine others will have the exact same data and continue to handle requests.

Non-repudiation

Non-repudiation, sometimes written as nonrepudiation, means that a receiver can prove to everyone that the sender did indeed send a piece of information, that is, the sender can't deny sent information[10](p.3). Non-repudiation has a goal to bind different parties with a contract and make them sign the contract with a signature and a timestamp. This is a crucial requirement of an online transaction because digital signatures together with a timestamp will provide both accountability and integrity, and through these capabilities parties are able to identify each other and guaranty that the transaction has not been modified from the beginning.

3.3.2 Cryptography

Cryptography is the science and an ancient art of writing secret code, first documented in 1900 B.C. when an Egyptian scribe used non-standard hieroglyphs in an inscription[55]. It has been extensively used in war times to make original messages unreadable by the enemies and some famous cryptographic schemes are Caesar’s cipher and Wehrmacht Enigma machine, used by the Romans and Nazis respectively. Keeping the message unreadable is an essential requirement in the world of network communication, especially the untrusted ones like the Internet. In this section we will have a look at different algorithms used to produce the ciphertext. This section is divided into three major subsections based on cryptographic capabilities of the specific mechanism.

3.3.2.1 Encryption

Encryption is a process of obscuring plaintext, making it unreadable without some sort of special knowledge[7]. Plaintext or clear text is a text understandable by humans and for confidentiality purposes this text can be transformed to ciphertext so it becomes unreadable. The transformation is carried by the use of an encryption algorithm and an encryption key. Encryption key is a digital key expressed in byte code. On the other hand, decryption is the reverse process of encryption, transforming the ciphertext to plaintext by the use of decryption algorithm and the decryption key. The algorithm used to decrypt ciphertext must be the same one used in the encryption process while decryption key may vary depending on the class of encryption. In fact, the encryption class, either symmetric or asymmetric, determines what type of keys to use in encryption and decryption. Another important feature regarding the keys is the key size. The key size is defined as bit length and together with the encryption algorithm decides how strong encryption will be. Larger key size within the specific encryption algorithm ensures stronger encryption while smaller key size provides faster encryption[5](p.233).

There are two types of encryption algorithms, block ciphers and stream ciphers. Block cipher algorithms works on one block of input data at time where block size may vary, usually between 64 and 512 bits. Plaintext is sent to the buffer and when the buffer gets full it is passed for encryption resulting in a block of data, usually of the same size as the input block size. Stream cipher algorithm waits for a stream of data to fill a block before the block is used in encryption process. If there is not enough data to fill the block then extra bits are used to fill the empty space[5](p.228).

Symmetric encryption

The encryption class where the same key is used for encryption and decryption is called symmetric encryption. This sort of key is often referred to as symmetric key, shared key[5](p.230) or secret key[55]. Symmetric encryption keys are easily understandable and manageable since there is only one key involved in both processes. The biggest drawback of this encryption class is the key distribution where multiple users require a separate key in order to communicate to one another. For instance, when a person A wants to communicate with a person B they both share the same key. When A wants to exchange messages with C and A does not want messages to be understood by B then A and C need to share a new symmetric key. Following the formula n*(n-1)/2, a small environment consisting of 10 users will require 45 unique symmetric keys. In addition to the issue regarding keys distribution, all keys need to be exchanged in a secret and secure manner.

Some of the most common symmetric keys algorithms include DES, 3DES, AES, IDEA, Blowfish and Twofish[54]. DES algorithm was the most used algorithm of the ones mentioned but has been cracked and considered unsafe partially due to its relatively small key size[5](p.233). Generally symmetric keys sizes are considered relatively small, starting from 64 bits. AES is the US government standard and its key size range from 128 to 256 bits[54] where the 256 variant ensures the stronger encryption. Symmetric encryption is usually preferred in operations where performance is a crucial requirement and for larger data volumes[54]. For instance, BitLocker is a disk encryption software product built-in in Windows, based on AES[57]. TrueCrypt i an open source variant similar to BitLocker, based on multiple symmetric encryption algorithms like AES, Twofish and Serpent[56].

Asymmetric encryption

The encryption class where two separate keys are used, one for encryption and one for decryption, is called asymmetric encryption. The philosophy behind asymmetric encryption is having a single key pair identifying a party on the network. The key pair is defined by a public key and a private key where the public key is used for encryption while the private one is used for decryption. As the name reflects, the public key is intended for the public so that anybody can encrypt information destined for the original party. The private key is to be protected by the original party since it is the only key able to decrypt the information encrypted by the original party's public key. In this fashion the party is identified by a single public key to all other parties, eliminating the need for multiple keys as it was the case with the symmetric keys. Additionally the distribution of the public keys does not have to be secretive or protected since encryption and signature validation are only operations public key may perform. However, asymmetric keys are much larger than symmetric keys, starting from 1024 bits[5](p.233). Although key size is larger, the asymmetric algorithms are generally weaker than their counterparts. For instance, RSA is one of the asymmetric encryption algorithms and together with its 1024 bits key it is still weaker than AES with 254 bits key[5](p.233). Another issue regarding asymmetric encryption is the speed. Compared to the symmetric encryption, asymmetric is up to 1000 times slower and requires far more processing power to do both encryption and decryption[54]. RSA, together with Diffie-Hellman, ElGamal and Elliptic curve cryptography are some of the more known asymmetric algorithms today[54]. Their field of usage is within key exchange, typically symmetric keys, where the asymmetric encryption is used to encrypt the symmetric keys before being sent to the other party[55]. Another example of the usage is digital signatures which are used to identify a party, ensure integrity upon the information being exchanged and provide non-repudiation.

3.3.2.2 Hash functions

Hash functions, also known as message digests and one-way encryption[55], are the algorithms used to create ciphertext without additional items. It is a one-way transform of the input which means that it should not be possible to compute the original input out of the output[5](p.248). Digest or hash value are the terms describing the ciphertext produced when working with the hash functions[54]. A particular hash function input object will have same digest value every time it gets processed but if the smallest change occurs, like replacing one single character, the hash value will change drastically. These drastic changes are crucial for preventing any sort of pattern guessing of the input value and are sometimes characterized as the avalanche effect[54]. Some of the famous hash functions include MD5 and SHA. Both MD5 and SHA-1 has been proven to produce equal digests for two different inputs[60]. SHA-2, the successor of SHA-1, consisting of the SHA-256, SHA-384 and SHA-512 which are defined by their bit size, is still regarded secure[60]. The digests are used for integrity checks ensuring that no change has occurred on the object since last digest calculation. For instance, the string “hello world” ran through the SHA-256 hash algorithm produce digest b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9 while “hello world!” produce 7509e5bda0c762d2bac7f90d758b5b2263fa01ccbc542ab5e3df163be08e6ca9.

3.3.2.3 Digital signatures

Digital signatures are not an additional mechanism to cryptography but rather the result of combining both asymmetric encryption and hash functions to provide authentication, non-repudiation and integrity on digital documents. Digital signature was a concept publicly described by Diffie and Hellman in their 1976 paper “New directions in Cryptography” suggesting that it is a computer based equivalent of physical written signature[21]. In contrast to ciphertext, digital signatures are created using a hash function and party’s private key while the signature is validated using party’s public key. Before the private key is used to create the signature a digest value needs to be calculated out of the input message using a hash function. When the digest is created it will be encrypted by the private key completing the digital signature. This signature will then be appended to the original message and the message is ready to be sent to the other party. The other party uses the public key to decrypt the signature and reveal the digest. At this point the receiving party can conclude that the message was signed by the alleged party. The reveled digest will be compared to a new digest recalculated out of the message using the same hash function. If both digests match then the receiving party can conclude that no alteration has occurred since the message was signed which completes the process[5](p.247). Digital signatures are well used mechanism implemented in many security products like Oracle

Security Server[59] and CoSign[58]. XML Signature[22] is a security standard that has its mindset built on digital signatures.

3.3.3 Authentication services

According to Hollar & Murphy, “An authentication service’s primary role is to establish a client’s, or a server’s, identity”[5](p.198). They follow two basic patterns of authenticating a client; the secure channel and the secure format. A secured communication channel is needed when credentials are sent over the network in plaintext. The secure channel will encrypt all the information from the client to the server. Secure format, on the other hand, stores credentials in a message which gets encrypted before being sent to the server and thus does not require a secured communication channel. Another difference between authenticating services lies in the trust relationship that exists between the parties. The trust relationship simply means that both parties know and trust in each other’s identity[5](p.193). There are different types of trust relationships and those can roughly be split in direct, brokered and delegated trust relationships. In the case of the direct trust relationship, credentials are exchanged between the parties and in that manner parties are directly identified to one another. In the case of a brokered trust relationship a server verifies the client’s credentials with a third-party broker. In a delegated trust relationship, the client impersonates another party’s identity in order to gain access privileges to a resource[5](p.193).

3.3.3.1 User Ids and passwords

User ID and password is one of the simplest ways to identify a party. This type of scheme assumes that every single party is identified by a unique ID and password combination. Since a party have to send its credentials as text, user ID and password scheme needs to be secured on the channel level. Some of the most often used mechanisms for securing the communication channel is a SSL/TLS protocol[5](p.199). Web servers do support hiding of the credentials in HTTP by creating a digest, using hash algorithms like MD5 or SHA-1. This type of HTTP authentication is known as HTTP Digest authentication[2].

3.3.3.2 Kerberos

Kerberos is a network authentication protocol designed to provide strong authentication for client/server applications by using symmetric cryptography[35]. The protocol was originated at the Massachusetts Institute of Technology (MIT) in the 1980s as part of a project called Athena. Athena focused on offering single sign-on by integrating computers on the MIT campus that ran on different operating systems[38].

Kerberos Key Distribution Center (KDC) authenticates users to servers and servers to users. Since KDC is vouching for different parties it is also known as a trusted third party. KDC also maintains a list of all users, servers, their passwords and server specific secret keys.

The Kerberos is implemented using four sub protocols; authentication service (AS) protocol, ticket-granting service (TGS) protocol, service-granting ticket (SGT) protocol and client-server (CS) protocol[5](p.203).

When a client is about to log on a system it will send a request message to a server running an AS. The request message will contain client's ID, server's ID and a timestamp. Then the server will respond with a ticket-granting ticket that contains client and AS details, timestamp, ticket lifetime and session key to the client. The client submits the ticket-granting ticket to the TGS to get authenticated. The TGS creates a service ticket that contains a session key with a timestamp. The service ticket is encrypted by a secret key derived from the client's password and then sent to the client. The client receives the service ticket and decrypts it by generating a secret key using its own password. All the subsequent communication tells the KDC that the client was able to decrypt the service-ticket and that it really is who it claimed to be[5](p.204).

Kerberos runs on multiple operating systems such as Linux, Unix and is the default authentication mechanism on Windows 2000 and later[36][37][38]. Although symmetric encryption is used in Kerberos, Microsoft added public key encryption in their implementation[5](p.205).

3.3.3.3 X.509 Public Key Authentication

A digital certificate is a digital identity that uniquely identifies a person or a machine on the network. Digital certificates replace Ids and passwords and are issued by a trusted third-party certificate authority (CA). CA is responsible for ensuring authenticity of issued digital certificates by signing each certificate with its own private key[18]. Popular CA entities are VeriSign and Thawte. The ISO X.509 standard and Public Key Cryptography Standards (PKCS) define a certificate type X.509 and its usage[39]. CA issues a new certificate together with a private and a public key, where the public key and the additional information are attached to the certificate. Before any business transaction between a client and a server starts, a mutual authentication of one another has to be in place. The mutual authentication process starts after both the client and the server have exchanged their respective certificates with each other and validated each other's CA by CA's signature. Now the client generates a transaction ID (tran Id1), attaches it to a message and encrypts the message with the server's public key. The server receives the message and decrypts it using its own private key. The server generates its own transaction Id (tran Id2) and includes it in the returning message together with tran Id1. The message is encrypted with the client's public key before it is sent to the client. After receiving the message from the server, the client decrypts the message, finds its own tran Id1 in it and thus understands that the server was capable of decrypting its previous message. Finally the client takes the tran Id 2 and includes it in a message before encrypting it and sending it to the server. The server receives the message, opens it and founds its tran Id2, which tells it that the client was able to decrypt its previous message. The server creates a session key which is encrypted by client's public key and sent to the client. The session key will be used to encrypt all the subsequent message exchange between the client and the server[5](p.200). The main reason behind the usage of session keys is their efficiency.

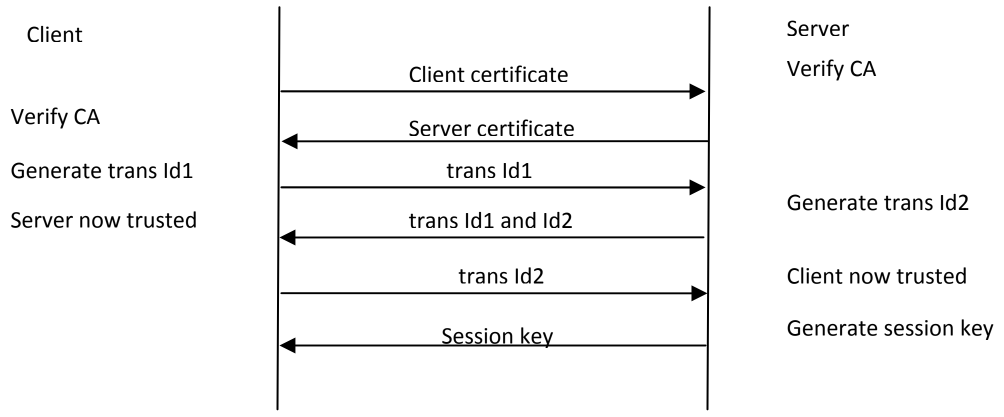


Figure 3.10: X.509 authentication process[5](p.201)

3.4 Security in RESTful services

HTTP offers three types of security mechanisms; HTTP Basic authentication, HTTP Digest authentication and HTTPS (HTTP with SSL) [2]. Since RESTful services are just HTTP endpoints all of the security mechanisms mentioned above do apply to them as well.

The first two features are used to authenticate the client offering no additional protection mechanisms. HTTPS is the only complete security feature used to ensure trust, authentication and message protection in an exchange, and is widely used in e-commerce[11].

3.4.1 HTTP Basic authentication

HTTP Basic authentication is based on username and password exchange which is made when a HTTP client makes a request to a HTTP server application. HTTP Basic authentication was first introduced as part of HTTP 1.0, defined by Internet Engineering Task Force (IETF)[13]. This scheme is used for authentication purposes only and provides no protection of the credentials sent to the server[4]. It is also the most used HTTP authentication mechanism[1](p.64).

When a client is about to send a HTTP request to a RESTful service, it’s credentials will also be sent in that same request. Username and password will then be a part of the HTTP header field called ‘Authorization’[1](p.146). When the service receives the request it will examine the credentials and identify the client as a user, and check its permissions. This means that both authentication and authorization needs to be confirmed in order to proceed with the actual request. If one of the conditions is not met whether credentials are missing, invalid or not good enough to provide authorization, then the server sends a ‘401 Unauthorized’ response code and sets HTTP header field ‘WWWAuthenticate’ with instructions about how to send correct credentials next time[1](p.146). Those instructions will tell us what type of authentication is being used on the particular service, for instance the following line tells us that HTTP Basic Authentication is used and the name of realm:

```
401 Unauthorized
WWW-Authenticate: Basic realm="My Area"
```

Figure 3.11: HTTP Basic authentication response

The realm is typically used to identify a collection of resources on a site and can be of any name [1](p.238). This pattern of communication is referred to as challenge/response[2] since the server challenges the client with an authentication type when the client is about to use a resource. After the challenge, the client must answer with an appropriate response which must contain its credentials. Credentials exchanged between the client and the service are encoded as Base-64 string sent in clear text[2] [1](p.238). Base-64 is an encoding format for transforming binary data into text, used for instance in transmitting binary data over Web Services[5](p.133). Since the credentials are transmitted this way it makes them vulnerable to reply attacks[2]. A reply attack is a way of deceiving the server by recording a previously made request between a particular client and the server. When the copy of a successful request is recorded then an attacker is able to send this request again to server and succeed at mimicking another client's identity[2].

3.4.2 HTTP Digest authentication

As HTTP Basic authentication, HTTP Digest authentication is used for authentication purposes over HTTP. HTTP Digest authentication is designed to be more secure than HTTP Basic authentication in the sense that passwords are not sent over the network directly but a MD5 hash is used instead[2]. It is a more complex way of authentication and follows a communication pattern of request/challenge/response[1](p.239) where a HTTP client sends a request and receives a challenge that may look like the one in Figure 3.12.

```
401 Unauthorized
WWW-Authenticate: Digest realm="My Area",
qop="auth",
nonce="1aa176b9c0f1b6a641c399e269772777",
opaque="16ec5ffee6132fec3ad71c77753157c6"
```

Figure 3.12: HTTP Digest authentication response

The challenge describes the authentication type, realm and three other pieces of information. One of those three pieces is a nonce which is a random string that is changed on every initial request[2]. After receiving the challenge, the client generates its own nonce and a sequence number[1](p.239). Then the client creates a single digest string out of the following pieces of the information: the HTTP method and path from the request, all four pieces of the information from the server challenge, username and password, its own nonce and its sequence number[1](p.239). The digested value variable, varHa3, is calculated as demonstrated in Figure 3.13.

```
varHa1 = MD5("{USERNAME}:{REALM}:{PASSWORD}")
varHa2 = MD5("{METHOD}:{PATH}")
varHa3 = MD5("{varHa1}:{NONCE}:{NC}:{CNONCE}:{QOP}:{varHa2}")
```

Figure 3.13: HTTP Digest authenitcation, digest value calculation

When the digest is calculated, the client resends the request by passing the server challenge information, its own nonce as well as the digested string [1](p.240). It may look like the following figure.

```
GET /resource.html HTTP/1.1
Host: www.fantasiaislanders.com
Authorization: Digest username="John",
realm="My Area",
nonce="1aa176b9c0f1b6a641c399e269772777",
uri="/default.html",
qop=auth,
nc=00000001,
cnonce="4123c6512a1945abc12333112121333",
response="1111039ff8a9fb83b4293210b22253d1",
opaque=”16ec5ffee6132fec3ad71c77753157c6”
```

Figure 3.14: HTTP Digest authenitcation request

The server receives the response from the client and uses its own username and password stored as a MD5 hash (varHa1) and the other values in order to compute a final digest value (varHa3) that must match the value from the client[1](p.240). If they do not match then the authentication fails and the client receives ‘401 Unauthorized’ response once again. The HTTP Digest authentication is in many ways a better alternative to HTTP Basic authentication, not only because it prevents reply attacks by offering a different nonce on every request but also because it stores the credentials on the server into an irreversible hash value ensuring that both username and password stays hidden even at the source system.

3.4.3 HTTP Secure (HTTPS)

HTTPS or HTTP with SSL/TLS is a way to secure communication between a client and a web server. SSL (Secure Socket Layer) and TLS (Transport Layer Security) are two protocols at application layer mostly utilized to protect HTTP transactions but those have also been used with IMAP and POP3[12]. SSL was originally developed by Netscape reaching version 3 already in 1996[5](p.192). TLS, developed by the IETF, is based on SSL version 3 and later became the successor of SSL as well. The latest version of TLS is 1.2 and is very similar to the earlier versions of TLS and SSL version 2 and 3. There is even a built-in mechanism for version negotiation which makes different TLS versions and SSL compatible with each other[17](p.87). SSL/TLS is supporting applications exclusively running over TCP[12]. This security protocol offers confidentiality, authentication and integrity protection of the data[11][15] by creating an encrypted tunnel between two computers. The information is generally in the clear text at both endpoints but is protected by the encrypted tunnel which the information travel through[5](p.191). This type of security is referred to as point-to-point security since the information is secured between the two nodes only[14]. SSL/TLS is composed of several protocols: Record protocol, Handshake protocol, Change Cipher Spec Protocol and Alert Protocol. The Record protocol which is the lowest one, provides privacy and data integrity to higher level protocols[11][15]. The Handshake protocol is responsible for the authentication of both the client and the server, and to exchange cryptographic keys and negotiate an encryption algorithm between them[11][12]. The Change Cipher Spec Protocol consists of a single message used to indicate that the chosen keys will be used by the Record Layer[12][15]. The Alert Protocol is used to alert the peers about the current state like errors and closure of the session[12].

The way communication between the client and the server is done is by a client issuing a simple request to server’s port 443 which is the default HTTPS port. Then the client decides cryptography type it wants to use in order to establish a secure communication with the server. Secure communication in this context stands for authentication, encryption and data integrity[5](p.192). There are several mechanisms available for accomplishing authentication like RSA cryptography algorithm, Diffie-Hellman key agreement algorithm and Fortezza[16] crypto cards[5](p.192). DES, IDEA, RC2 and RC4 are cryptography algorithms that might be chosen for data encryption and decryption. Digest algorithms like SHA and MD5 might be chosen for data integrity [5](p.192). After negotiation is done the server sends its certificate to the client. The certificate contains information about the server and a certificate authority (CA) signature used to identify the CA. Once the client receives the certificate from the server, it uses CA's public key to identify certificate's signature. After that, the client generates a session key and encrypts it using server's public key. Then the encrypted session key is sent to the server who decrypts the session key using its own private key. The server now encrypts a message with the session key and marks the end of the whole negotiation by sending it to the client[5](p.192). This shows that both client and server have agreed upon a session key and from now on data exchange between the client and the server will be secured.

3.4.4 ATOM format security

Previously it was mentioned that the focus of this thesis is to offer protection for XML and JSON formatted messages since they are the most used ones and selectable by default in WCF. However, it should be said that according to the IETF Atom Syndication Format specification[20], ATOM format may be signed and encrypted using XML Signature and XML Encryption. Unfortunately, the only implementation we found supporting encryption and signature of the ATOM formatted messages is the Apache Abdera project[99] which is a java library. It is also important to understand that the IETF specification discusses how to protect a single message, unrelated to any specific transport protocol.

3.5 Security in XML

XML got growing acceptance in the industry as the document standard and as the protocol. In order to meet the demand for classic and transaction security goals, a number of specifications and standards have emerged through the years. These standards use well-known cryptographic and security technologies and many of them provide the basis of Web Services security. In this chapter we will look at how basic security requirements like authentication, confidentiality and integrity are met, but also present specifications targeting larger implementations and solving issues regarding key management, access policies and single sign-on.

3.5.1 XML Signature

XML Signature is a W3C recommendation[22] and is based on XML syntax. It is primarily used to provide digital signatures on XML but may also be used on other types of objects, whether they are in textual or in binary format[19](p.10). As the digital signature, XML Signature will ensure message integrity, authentication and non-repudiation[22]. A key feature offered by XML Signature is that it may provide a single signature covering multiple resources. These resources may be multiple XML documents, a XML document or its elements, or another type of an object[19]. Since the elements may be signed separately, the document may be signed with multiple digital signatures thus allowing different senders to contribute on a single document.

The way XML Signature works on XML documents is by extending the document with the additional elements where Signature-element being the root node.

```
<?xml version=1.0" ?>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <SignatureMethod></SignatureMethod>
    <SignatureValue></SignatureValue>
    …
  </SignedInfo>
</Signature>
```

Figure 3.15: XML Signature Element

The Signature-element sub elements describes the information about how the signature was generated, the actual signature, information about the key used in the process of generating the signature, optional objects included in creating the signature and additional information about the process of signing[5](p. 273). The process of creating a digital signature is handled in the following way. First step is to create a message digest out the chosen XML document, elements or the object by using a hash function such as SHA-1 or SHA-2[19]. Although SHA-1 is the only algorithm required to be supported, there are several implementations that support SHA-2 as well[19]. The second step completes the creation of the digital signature by encrypting the message digest using sender’s private encryption key [5](p.271). Encryption algorithms like DSA or RSA might be chosen, or even and encryption technology like PGP[5](p.274). After this step digital signature is appended to the document.

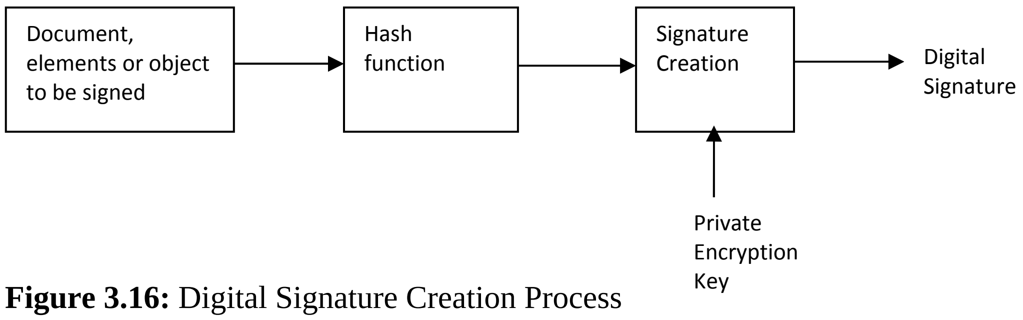


Figure 3.16: Digital Signature Creation Process

The validation process of the signature on the recipient side may be seen as reverse of the signing process. First the recipient calculates the digest value using the exact same hashing algorithm as the sender did. Then the newly calculated digest value is fed into a verification function along with the sender’s digest value and sender’s public key[5](p.271). If the values don’t match then the document has been changed since the last time it was signed. In case of the XML, sometimes the verification process may fail even if the XML document has not been changed since it was signed. This might be due to the differences in the physical representation of the document on the sender’s and recipient’s side. In order to minimize the differences in their physical appearance there is an important feature called canonicalization. Canonicalization can be specified inside the CanonicalizationMethod-element which allows us to choose an algorithm that will convert the document into a common, standardized format before digest is calculated[5](p.273). This operation is important because two logically equivalent XML documents may differ in their physical representation where the difference may be in whitespaces, line endings or character encodings. For instance, ASCII text widely uses three different line ending sequences and every major operating system family incorporate different line ending by default. Line ending in Windows is defined by ‘/r/n’, Linux by ‘/n’ and the classic Mac OS by ‘/r’. Digital signature validation will not break if the verification digest calculations are performed on exactly same bits as the original signature digest is based on. The issue around canonicalization is well-documented in the paper “What you see is Not Always What You Sign” by Jøsang et al[21].

```
<?xml version=1.0” ?>
<car model="Think" electric="true">

<?xml version=1.0” ?>
<car
electric=  "true"
           model='Think'
>
```

Figure 3.17: Two logically equivalent XML documents

XML Signature is a widely used standard and an important building block many other security related standards like WS-Security[19] and Security Assertion Markup Language (SAML)[24].

3.5.2 XML Encryption

XML Encryption is, like XML Signature, a WC3 recommendation[23] and its purpose is to ensure confidentiality by encrypting the data. The syntax of XML Encryption is based on XML. It also shares other similarities with XML Signature in the way it operates on the objects. For instance it allows XML documents, selected XML elements and non-XML resources to be encrypted[19]. The advantage of allowing certain parts of the XML document to be encrypted may be convenient in the case of a workflow process. For instance, routing-centric elements in a XML document may not be business sensitive like some other elements and thus does not require to be encrypted since both encryption and decryption are costly processes. If those elements do not become encrypted then a system for handling incoming documents will be able to determine document’s end system without decrypting it. This type of flexibility is one of the advantages of XML Encryption over technologies like SSL/TLS.

When XML Encryption is implemented on a XML document it will extend it with an EncryptedData-element. EncryptedData-element and its sub-elements will contain all the information about how and what elements are encrypted. The placement of the EncryptedData -element on a XML document depends upon encryption requirements. If a whole document is to be encrypted, then EncryptedData will become the root node of the document. Otherwise, the element will replace selected elements only. Figure 3.18 and Figure 3.19 demonstrates how an element, OrderDetails, gets encrypted.

```
<Order xmlns=”http://sample.uio.no/order”>
  <Name>Ola Norman</Name>
  <OrderDetails>
    <Article>Learn XML in 24 hours</Article>
    <Price>200$</Price>
  </OrderDetails>
</Order>
```

Figure 3.18: Order-element containg OrderDetails

```
<Order xmlns=”http://sample.uio.no/order”>
  <Name>Ola Norman</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#tripledes-cbc'/>
       <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
         <KeyName>Ola Norman</KeyName>
       </KeyInfo>
       <CipherData>
         <CipherValue>oqUYdLlNNa...</CipherValue>
       </CipherData>
  </EncryptedData>
</Order>
```

Figure 3.19: OrderDetails-element encrypted by XML Encrypt

While there are several sub-elements inside EncryptedData-element there is only one mandatory element called CipherData which contains or provides a reference to the encrypted data in the cyphertext format[19]. XML Encryption supports several encryption algorithms like AES, Triple-DES and RSA[5](p.266) and chosen algorithm may be specified in the EncryptionMethod-element[19]. In some cases the algorithm may not specified because both the sender and the receiver have a mutual contract on the algorithm type[19]. In addition XML Encryption supports so-called superencryption which enables already encrypted elements to be encrypted multiple times in order to strengthen the encryption[19][5](p. 266).

In the world of Web Services, XML Encryption is a widely used standard, just like the XML Signature[5](p.265). For instance, security standards like SAML version 2.0[24] and XML Key Management Specification (XKMS)[26] incorporates XML Encryption.

3.5.3 XML Key Management Specification (XKMS)

XKMS is WC3 recommendation providing an unified way for registration and distribution of public keys and digital certificates for use suitable with XML Signature and

XML Encryption[26]. XKMS itself builds on XML Signature and XML Encryption, and the specification is meant to hide key management complexity on a larger scale usage and provide centralized storage of the keys[19]. While both XML Signature and XML Encryption utilize encryption keys, an appropriate method for key management is needed in order to employ both standards in a scalable manner. XKMS solves this issue by providing two Web Services, the XML Key Information Service (X-KISS) and the XML Key Registration Service (X-KRSS)[25](p.61). X-KISS service defines a protocol for location and validation of the public keys while X-KRSS service defines a protocol for registering, reissuing, revoking and recovering of the keys[25](p.62). In order to illustrate a simple encryption operation involving XKMS let us imagine there are two people, A and B. When the person A want to make its public key publicly available so others may contact it in a confidential manner, it needs to register its key by using the X-KRSS Web Service. Since key pair generation may be generated by A on the client side then A needs to prove the possession of the private key before public key is registered[19]. Registration allows A to associate its public key with several attributes like name, an ID or anything that may help others uniquely identify A[5](335). When the person B wants to send an encrypted document to A it is now able to find A's public key by using X-KISS Web Service locate function. X-KISS Web Service will be able to locate A's key by its attributes defined by the X-KRSS service. Figure 3.20 describes the process.

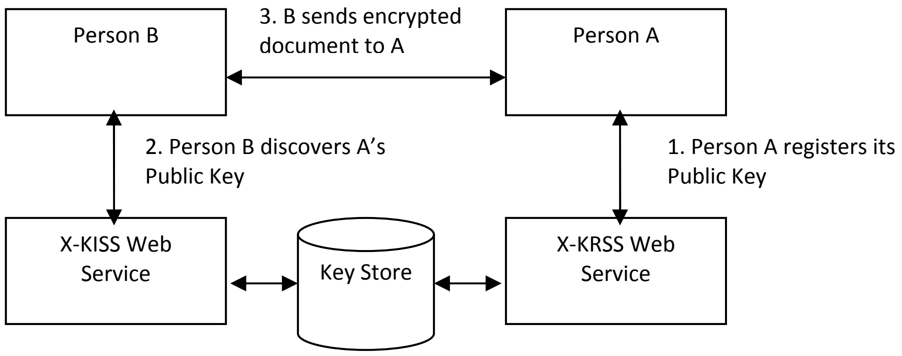

Person B
3. B sends encrypted document to A
Person A
2. Person B discovers A's Public Key
1. Person A registers its Public Key
X-KISS Web Service
Key Store
X-KRSS Web Service

Figure 3.20: The XKSM model[5](p.336)

The X-KISS locate function offers additional functionality, for instance it allows a recipient of a signed document to obtain the key used for signature if the recipient does not know what key was used[25](p.62). It also allows retrieval of the key by parsing X.509 v3

certificates[5](p.336). It is important to notice that XKMS offers no mechanisms for client authentication[19], meaning that everybody may ask for someone’s public key.
XKSM may be used across multiple domains. This means that a XKMS service handling XKMS requests in an organization may forward same requests to another XKMS service in another organization[19].

As the X-KRSS Web Service is designed to handle key registration one at a time there is an additional specification called XML Key Management Specification Bulk Operation (X-BULK) that allows multiple key registrations to happen through a single Web Service call. X-BULK build on X-KRSS and provides all the necessary protocols in order to support bulk type of operations, for instance ability to correlate batch requests and responses[5](p.337).

3.5.4 Security Assertion Markup Language (SAML)

The Security Assertion Markup Language (SAML) 2.0 is an OASIS standard specification approved on 15 March 2005[25](p.53). It is a framework for the exchange of security-related information expressed as assertions in XML syntax, between asserting parties, i.e. SAML authorities[19]. By having its syntax defined in XML it is also platform independent. Current version of the SAML is 2.0 and it is not backwards compatible with the previous versions[24]. The major driver behind SAML was a demand of providing single sign-on functionality[5](p.337), meaning provide cross domain authentication and authorization of parties by authenticating at one common place.

There exist several ways of providing single sign-on. Hollar & Murphy states the following, “Single sign-on has historically been an identity management problem with three traditional solution approaches”[5](p.337). These approaches are specified in the following table.

Table 3.3: Single sign-on approaches[5](p.337)

Trusted Tickets:	Users are authenticated by a ticket granting service. From then on all re-authentication is eliminated since all the services recognize the ticket issued by the ticket granting service
Synchronized Credentials:	Credentials are synchronized from one system or domain to other systems or domains.
Pseudonym Services:	A proxy service provides authentication information by maintaining a repository of credentials and credential mappings.

SAML will provide possibility of single sign-on functionality through assertions rather than the traditional cross domain mechanisms as mentioned above. The way these assertions works is by offering multiple attributes describing a subject. A subject could be a person or a computer based device, and assertions may describe subject’s email address, information about authentications previously performed by the subject, authorization details and decisions as to whether the subject is allowed to access certain resources[25](p.53). There are three different assertion statement types offered by SAML; authentication, authorization and attribute statements. An authentication statement is concerned with the authentication of a subject and specifies how and when a subject was authenticated [19]. An authorization statement is concerned with the access rules and specifies if a subject should be granted access to resources, alternatively if the authorization could not be decided [19]. An attribute statement may possess additional information about a subject that may be useful to a request[5](p.338).

A typical authentication and authorization process starts by subject requesting a set of assertion references from SAML services by using its credentials. A service is one of the three types of statements mentioned above. The services grant the assertion references after validating the credentials of the subject and comparing the requested permissions against policy. If the subject is authorized then the assertion references extend the original message by adding an Assertion-element, for instance, extending a SOAP request[5](p.339). That SOAP message then becomes a policy enforcement point (PEP), meaning a “logical entity or place on a server that enforces policies for admission control and policy decisions in response to a request from a user wanting to access a resource on a computer or network server”[28]. Figure 3.21 shows a SOAP message extended with the Assertion-element. In a SAML model PEP is responsible for enforcing actual access control decisions by calling a policy decision point (PDP). The job of the PDP is to decide whether or not to authorize the user based on the assertions provided by the PEP[5](p.339). An assertion, in addition to the authentication, authorization and attribute statements, may also contain a Conditions-element. Conditions-element is used so to place restrictions on the use of an assertion. Some of those restrictions may apply to the assertion validity, i.e. the assertion should be relied upon only once because it will change next time[19]. Another conditions may specify that the assertion targets a particular audience or the time period when the assertion is valid[27](p.21). The integrity of the assertions is provided by a signature specified in the Signature-element and the value of it is on XML Signature format[27](p.38). Another important features are single logout protocol and protocols for managing the identifiers used between identity providers[19](p.33).

```
<?xml version=”1.0”>
<soap:Envelope …>
  <soap:Header …>
    <soap:Security …>
      <saml:Assertion
        xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" Version="2.0" IssueInstant="2011-01-
01T09:30:00Z"
       …>
        <saml:Issuer>https://sample.org/SAML2</saml:Issuer>
        <ds:Signature  xmlns:ds="http://www.w3.org/2000/09/xmldsig#">...</ds:Signature>
        <saml:Subject>… </saml:Subject>
        <saml:Conditions
          NotBefore="2011-01-01T09:31:00Z"
          NotOnOrAfter="2011-01-01T09:33:00Z">
           <saml:AudienceRestriction>
              <saml:Audience>https://sample.com/ChosenAudience</saml:Audience>
           </saml:AudienceRestriction>
        </saml:Conditions>
       <saml:AuthnStatement AuthnInstant="2011-01-01T09:30:00Z"  …>
          <saml:AuthnContext>
            <saml:AuthnContextClassRef>
              urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport
            </saml:AuthnContextClassRef>
         </saml:AuthnContext>
       </saml:AuthnStatement>
     </saml:Assertion>
   </soap:Security>
  </soap:Header>
  <soap:Body>
  </soap:Body>
</soap:Envelope>
```

Figure 3.21: SAML structure in a SOAP message

Today SAML is used within Web Services as part of the SOAP message-level security. Being a framework that is independent of a centralized authority, it has the potential in organizations where centralized authority may become an issue, either geographically, politically or technically.

3.5.5 eXtensible Access Control Markup Language (XACML)

The eXtensible Access Control Markup Language version 2.0 became an OASIS standard in February 2005 and is a specification for defining access control policies[25](p.67). The syntax used for describing the policies is in XML format making this specification platform independent as well. The goal of XACML is to solve issues with enforcing access control policies at multiple points in a distributed environment since altering policies at multiple points is an expensive and unreliable operation[25](p.67). Therefore, XACML can be used as a centralized policy store for all applications in a domain, including providing access control to Web Services. Other issues XACML is solving relates to the standardized way of making policies rather than specifying them differently depending on the platform and language[25](p.67). XACML is structured in a hierarchical manner where policy set represented by the PolicySet-element may be on the root node of a XML document and will contain one or several policies. A policy set may be unnecessary in cases where there exists only one policy and in those cases the policy, represented by a Policy-element, may be the root node of a document[29]. Further on, a policy contains one or several rules. Each rule, represented by a Rule-element, decides on either permitting or denying a target. A target, defined by Target-element, may be a subject, resource, action or environment[9](p.27). Conditions can also be a part of the rule enforcing comparative, arithmetical or Boolean operators applied upon subjects, resources, actions, and environments[9](p.28). A condition may for instance compare the role of a subject to a static value. In cases where there are multiple rules inside of a policy, each of which may evaluate to a different access control decision, a rule-combining algorithm will be applied and create a single decision. There are seven different algorithms to choose from and it is also possible to define custom algorithms[29]. The same combining algorithms may be applied at the policy set level deciding on a single policy. An example of such algorithm is the Deny Override Algorithm that will return deny if any evaluation returns deny or if no evaluation returns permit[29]. Figure 3.22 shows a XACML policy containing a rule with a condition to only allow logins from 9am to 5pm.

```
<Policy PolicyId="SamplePolicy"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
   <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <ResourceMatch …>
        <AttributeValue…>SampleServer</AttributeValue>
        <ResourceAttributeDesignator…/>
      </ResourceMatch>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>
  <!-- Rule to see if we should allow the Subject to login -->
  <Rule RuleId="LoginRule" Effect="Permit">
    <!-- Only use this Rule if the action is login -->
    <Target>
      <Subjects>
        <AnySubject/>
      </Subjects>
      <Resources>
        <AnyResource/>
      </Resources>
      <Actions>
        <ActionMatch…>
          <AttributeValue…>login</AttributeValue>
          <ActionAttributeDesignator… AttributeId="ServerAction"/>
        </ActionMatch>
      </Actions>
    </Target>
    <!-- Only allow logins from 9am to 5pm -->
    <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-greater-than-or-equal"
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-one-and-only">
          <EnvironmentAttributeSelector
AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time"/>
        </Apply>
        <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#time">09:00:00</AttributeValue>
      </Apply>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-less-than-or-equal"
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-one-and-only">
          <EnvironmentAttributeSelector…
AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time"/>
        </Apply>
        <AttributeValue…>17:00:00</AttributeValue>
      </Apply>
    </Condition>
  </Rule>
  <Rule RuleId="FinalRule" Effect="Deny"/>
 </Policy>
```

Figure 3.22: XACML policy

The XACML model may be combined with the SAML model so that XACML can provide a standardized way of access control decision making[19]. In the SAML process example described in the SAML section, a SOAP message became a SAML policy enforcement point (PEP) when assertion references were included as the part of that SOAP message. SAML PEP then called SAML policy decision point (PDP) in order to decide if the subject is authorized or not. XACML complements the process when SAML PDP receives a call by SAML PEP so that SAML PDP routes a decision request to XACML context handler. XACML context handler calls than an attribute authority, also known as XACML policy information point (PIP), which will collect attributes about subject, resources and environment and return them to the XACML context handler[5](p.341). When the context handler receives the attributes it will pass them on to a XACML PDP service asking for a decision. Then XACML PDP will query a PolicySet store where it will invoke the appropriate rules and policies and return an authorization decision to the context handler which will in turn provide those to SAML PDP and on to the SAML PEP[5](p.341)

3.5.6 eXtensible Right Mark-up Language (XrML)

eXtensible Right Mark-up Language addresses how to express and enforce access control and information dissemination policies. XrML is a rights expression language (REL) meaning that it is a language which specifies how to describe rights, conditions and, in XrML case, fees for using digital contents with message integrity and entity authentication[25](p.72). XrML syntax is based on XML and thus is platform independent. Latest version of XrML is 2.0, released in November 2001. Its first version was released in April 2000 by Content Guard Inc, a Xerox spin-off company, but prior to that it has existed under the name Digital Property Rights Language (DPRL) developed by Xerox. The history behind DPRL goes way back in time when Digital Right Management (DRM) originated from the music industry with the goal of preventing illegal copying of protected digital music. DRM systems were controlling what a user can and cannot do with the digital media based on a description that traveled with the digital media. The description has a similar role of an access policy and with it DRM systems can determine what a user can do with the digital content on any device where DRM system exists. Since DRM systems could exist in different versions based on operating systems, mobile phones, portable players and be able to support previous versions, there was a need to standardize policy descriptions. The DPRL emerged in 1994 as the solution to that issue, providing a standardized language for describing rights, conditions, and fees[25](p.71).

The way XrML works is by having an issuer granting rights to a principal, to use a resource. A principal is a subject identified by either a secret key typically represented by XML Signature or by providing several credentials that are validated at the same time. A resource is an object which a principal manipulates through principal's rights. Resources can be digital media, documents and also right expressions. Conditions may also be set and for instance specify how many times a media file may be played before it becomes unusable. A common name for a container containing issuers, principals, rights, resources and conditions is called license[30](p.7). Figure 3.23 shows a simple XrML license structure. Since all the details about a license are shown in XML and since business requirements may demand to hide those for unauthenticated parties, there is a possibility of encrypting parts or the whole license by enabling XML Encryption[30](p.22).

```
<license xmlns="http://www.xrml.org/schema/2001/11/xrml2core"
xmlns:sx="http://www.xrml.org/schema/2001/11/xrml2sx"
xmlns:dsig="http://www.w3.org/2000/09/xmldsig#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.xrml.org/schema/2001/11/xrml2cx..\schemas\xr
ml2cx.xsd">
   <!-- Certify that the following key holder has the common name "Alice Richardson"-->
  <grant>
   <keyHolder>
   <info>
   <dsig:KeyValue>
     <dsig:RSAKeyValue>
       <dsig:Modulus>Fa7wo6NYfmvGqy4ACSWcNmuQfbejSZx
       7aCibIgkYswUeTCrmS0h27GJrA15SS7TYZzSfaS0xR9lZdUEF0ThO4w==
       </dsig:Modulus>
       <dsig:Exponent>AQABAA==</dsig:Exponent>
    </dsig:RSAKeyValue>
   </dsig:KeyValue>
   </info>
  </keyHolder>
  <possessProperty />
  <sx:commonName>Alice Richardson</sx:commonName>
 </grant>
</license>
```

Figure 3.23: XrML license[30](p.11)

XrML is not an official standard although a technical committee at OASIS was formed but was disbanded before taking any decision around standardization, probably due to the issues regarding patents held by ContentGuard[19]. As its predecessor DPRL, XrML is still used in digital media and is basis for MPEG-21 REL[5](p.380). It can also be included as part of SOAP messages, defining right expressions[19]. As shown in SAML section, SAML may also be included in SOAP messages to accomplish similar tasks and appears to be more widely supported by Web Service implementations than XrML[19].

3.6 Security in Web Services

Although previous section provided security standards targeting XML documents, this section will focus on standards and specifications written for Web Services. However, as Web Services technology is based on XML, many of the XML security standards are used within Web Services security as well. Previously we have mentioned that there has been added multiple specifications over the years which extend the Web Services framework. All these extending specifications are known as WS-* and many of them are specified to provide advanced security mechanisms. Although IPsec and SSL/TLS can be used with Web Services, WS-* security related specification are specifically made for message-level security and in this section there will be focus on those that are in use today. Figure n shows WS-* security related specifications in its early stages. WS-Security, WS-Policy, WS-Trust, WS-SecureConversation and WS-Federation are the ones still used while WS-Authorization and WS-Privacy became obsolete and were replaced by other specifications[5](p.49).

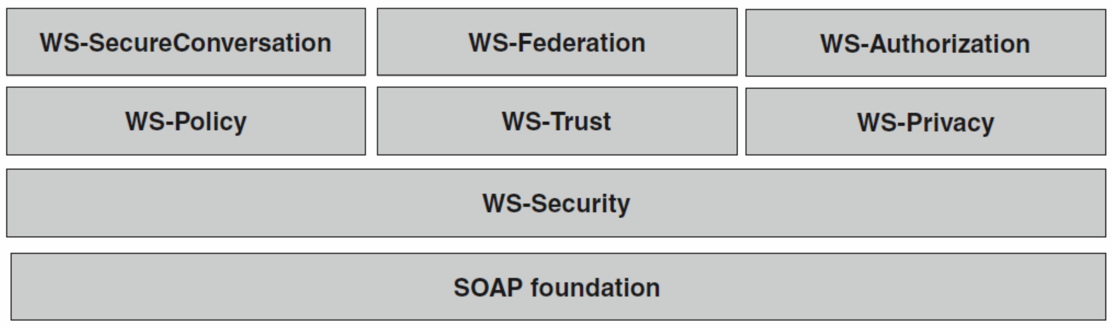


Figure 3.24: Web Services security standards framework [25](p.48)

3.6.1 Web Services Security (WS-Security or WSS)

WS-Security is a standard for securing a single SOAP message by enabling both confidentiality and integrity[19]. WS-Security was first proposed in April 2002 by Microsoft and IBM[40] and became the standard in June 2002[25](p.56). Confidentiality is enabled by XML Encryption while integrity is enabled by XML Signature. As in XML Encryption and XML Signature cases, we are allowed to encrypt and sign the whole message or selected elements[25](p.57), thus enabling different parts of the message to be encrypted and signed for different parties. All WS-* specifications, including WS-Security, are defined in XML format. When implemented on a single SOAP message, WS-Security will extend the SOAP-header adding a new element called Security[42]. Security-element will then keep references to the encrypted and signed elements in the whole SOAP message. WS-Security specifies an attribute that may contain a copy of the signature from the requested message and that copy will be a part of the response message from the server[42]. The response message will firstly be signed by the server and sent to the client. In such manner the response will be tied to the original request message, a very useful feature in message correlation process[19]. It is also possible to specify target party or role for whom the security is intended for. The actor-attribute and role-attribute describes the party and the role respectively. The SOAP header may contain multiple Security-elements but only one Security-element is permitted per party or role[5](p.257). A Timestamp-element is also specified helping in prevention of replay attacks[19]. Figure 3.25 shows a simple SOAP message with WS-Security enabled. The SOAP Body-element is encrypted and is similar to Figure 3.19 which demonstrated XML Encryption.

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
                                        xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
  <soap:Header  xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext"
    xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">
    <wsu:Timestamp>
      <wsu:Created  wsu:Id="Id-3beeb885-16a4-4b65-b14c-0cfe6ad26800">2002-08-
        22T00:26:15Z</wsu:Created>
      <wsu:Expires wsu:Id="Id-10c46143-cb53-4a8e-9e83-ef374e40aa54">2002-08-
        22T00:31:15Z</wsu:Expires>
    </wsu:Timestamp>
    <wsse:Security soap:mustUnderstand="1" >
      <xenc:ReferenceList>
        <xenc:DataReference URI="#EncryptedContent-f6f50b24-3458-41d3-aac4-390f476f2e51" />
      </xenc:ReferenceList>
```

```
       <xenc:ReferenceList>
         <xenc:DataReference URI="#EncryptedContent-666b184a-a388-46cc-a9e3-06583b9d43b6" />
       </xenc:ReferenceList>
     </wsse:Security>
   </soap:Header>
   <soap:Body>
     <xenc:EncryptedData Id="EncryptedContent-f6f50b24-3458-41d3-aac4-390f476f2e51"
       Type="http://www.w3.org/2001/04/xmlenc#Content">
       <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />
       <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
         <KeyName>Symmetric Key</KeyName>
       </KeyInfo>
       <xenc:CipherData>
         <xenc:CipherValue>InmSSXQcBV5UiT...  Y7RVZQqnPpZYMg==</xenc:CipherValue>
       </xenc:CipherData>
     </xenc:EncryptedData>
   </soap:Body>
</soap:Envelope>
```

Figure 3.25: WS-Security enabled on a SOAP message[41]

In addition to message confidentiality and integrity, WS-Security specifies the third mechanism which is the ability to send security tokens as part of the SOAP header. Security tokens are used to prove one’s identity by representing a set of claims where a claim may be an encryption key, identity, assertion, digital signature or a set of attributes[25](p.57). WS-Security specifies five different token profiles; Username token profile, X.509 certificate token profile, Kerberos token profile, SAML token profile and XrML/REL token profile[19]. Furthermore, custom security profiles may be specified in order to support new types of security tokens[41], but that may lead to non-standardization. WS-Security is neutral with respect to the type of any security token and specifies a general mechanism for including or referencing them inside a SOAP message[25](p.57). When implemented in a SOAP message the security token will be a sub-element of the Security-element.

3.6.1.1 Username Token Profile

This token profile specifies how to include username and either a password or a secret digest value inside a Security-element so to identify a party[19]. Username token is defined by a UsernameToken-element and contains Username- and Password-elements. Username-element will contain a username in plaintext while Password-element may either contain a password in plaintext (default) or a digest created by a hash algorithm. The profile recommends that the digest is created in a similar manner as in the HTTP Digest authentication scenario which means that the output is computed using a nonce, timestamp and a password[5](p.257).

3.6.1.2 X.509 Certificate Token Profile

X.509 certificate token profile specifies how to include X.509 certificates in the Security-element. BinarySecurityToken-element is an element that can contain X.509 certificates and other binary data encoded in Base64 or hexadecimal format, since the XML can only be specified in a textual format[5](p.258). ValueType-attribute contains the information about how certificates are included or referenced into the BinarySecurityToken-element. A certificate may be inserted into the element, represented by value #X509v3, or a certificate path can be used to represent the certificate or multiple certificates represented by values #X509PKIPathv1 or #PKCS7. It is however recommended that if certificates are to be referenced then the value of #X509PKIPathv1 should be used since it represents the ordered list of certificates[43]. The certificates may be used to validate the public key used for signing of the incoming message or to define the public key used for encryption of the message.

3.6.1.3 Kerberos Certificate Token Profile

As with X.509, Kerberos tickets are also represented in binary format and must be stored in BinarySecurityToken-element. ValueType-attribute may only contain one value when a Kerberos token is used and that value is #KerberosV5_AP_REQ representing AP-REQ messages that allows a client to authenticate to a Kerberos service[44]. Earlier version of WS-Security allowed ticket-granting and service-granting tickets to be specified[5](p.259).

3.6.1.4 SAML Token Profile

SAML token profile specifies how to add SAML assertions in the Security-element. SAML assertions are identified with Assertion-element and this element is inserted directly into Security-element. An example of a SAML token inside a SOAP header has been demonstrated by Figure 3.21. So to establish a relation between a SAML token and a SOAP message, the SOAP message can be signed with a key specified within the SAML assertion. As an alternative, a trusted third-party can be used to vouch for the message on behalf of the sender for whom assertions apply. This can be done by applying a trusted third-party signature on the message[19]. In the section 3.6.3 regarding the specification WS-Trust, we will look at an example where a third-party is used to provide SAML assertions to the service in order to authorize the client.

3.6.1.5 XrML/REL Token Profile

The XrML/REL token profile specifies how to add ISO/IEC 21000-5 Rights Expressions, also known as XrML expressions, in a SOAP header[19]. As SAML assertions, XrML expressions are added directly under the Security-element, hence the license-element becomes a sub element of the Security. Licenses can also be referenced instead of added directly. We can use either location or a license ID for referencing the XrML license[45].

3.6.2 WS-Policy

A policy is generally used to place requirements and conditions upon a requestor before he or she can consume an artefact or a service. To be effective, policies must be communicated and enforced. If a policy is not written down or communicated in some ways it is like not having a policy at all. WS-Policy is one of the key components of the Web Services architecture, providing a grammar for describing policies as a set of expressions made up of individual assertions[5](p.130). It is used to place general requirements and is not only tied to security requirements. However, in order to place security specific requirements there is a policy specification called WS-SecurityPolicy that adds security specific assertions to WS-Policy for communicating security related constraints, like security tokens, confidentiality and integrity[47]. It is interesting to note that WS-Policy is a W3C recommendation since 04 September 2007[46] while WS-SecurityPolicy became an OASIS standard on 01 July 2007[47]. WS-PolicyAttachment, a W3C recommendation, is the final policy specification which binds policies defined by WS-Policy and WS-SecurityPolicy to a policy subject through the Web Services Description Language (WSDL) or the Universal Description, Discovery and Integration (UDDI)[48]. A policy subject may be a service, a message, a provider, an endpoint or an interaction that can be constrained by a Web Service provider[5](p.146). WS-Policy offers a flexible way of defining policy expressions where a Web Service provider may present a service consumer with a choice to fulfill all or parts of the policy. Different choices may be defined so that the client can provide information that it owns or have knowledge of, potentially saving client’s time and economy to gather new information. In addition, the policies can be specified in two forms, compact and normal form where the latest is the most verbose one[49]. The following two figures shows the same policy specified in both forms. We see that the service provider demands WS-Addressing[71] to be part of the SOAP message and that the following SOAP exchanges must be secured either by transport-level security or the message-level security.

```
<Policy>
  <All>
    <wsap:UsingAddressing />
    <ExactlyOne>
      <sp:TransportBinding>...</sp:TransportBinding>
      <sp:AsymmetricBinding>...</sp:AsymmetricBinding >
    </ExactlyOne>
  </All>
</Policy>
```

Figure 3.26: WS-Policy and WS-SecurityPolicy in compact form[49]

```
<Policy>
  <ExactlyOne>
    <All>
       <wsap:UsingAddressing />
       <sp:TransportBinding>...</sp:TransportBinding>
    </All>
    <All>
      <wsap:UsingAddressing />
      <sp:AsymmetricBinding>...</sp:AsymmetricBinding >
    </All>
</ExactlyOne>
</Policy>
```

Figure 3.27: WS-Policy and WS-SecurityPolicy in normal form[49]

Detailed description of the Figure 3.26 follows. WS-Policy is contained inside the Policy-element. All-element demands that all of policy expressions are to be fulfilled. Inside All-element a provider is demanding a WS-Addressing defined by wsap-namespace prefix and two WS-SecurityPolicy assertions requiring either transport-level security or message-level security, defined by TransportBinding and AsymmetricBinding-elements respectively.

Since both of the WS-SecurityPolicy assertions are inside an ExactlyOne-element, the client can choose between one of them. Figure 3.27 contains the same elements as Figure 3.26, only specified in normal form. In the section about WSDL it was mentioned that the WSDL is optional as well as the UDDI is. Since policy requirements are a part of the WSDL or UDDI, service providers have the option of not showing WSDL and UDDI but instead convey requirements through word-of-mouth or documentation to the service consumers, thus hiding the descriptions from the public. The possibility of not providing this description is in itself security percussion where unknown users will never know how to establish connection with the business service or if it even exists.

3.6.3 WS-Trust

WS-Trust is an extension of WS-Security providing methods for requesting, issuing, renewing, cancelling and validating security tokens[19][33]. In addition it defines ways to establish the presence and broker trust relationship[33]. It became an OASIS standard on March 2007[32] and its latest version is 1.4[33]. The central part of this standard is security token service (STS) which is a Web Service providing a client with methods that issues, renews, cancels or validates different types of tokens[19]. STS may also be seen as the authentication broker providing a common access control infrastructure and is responsible for negotiating trust between a client and a Web Service.

In a typical WS-Trust scenario the only communication to STS is done by the client, so it becomes client's responsibility to retrieve the correct security token from the STS and send it to the business Web Service. In order to speak to STS Web Service, the client must add an extra element called RequestSecurityToken (RTS) in its SOAP request[5](p.325). RTS-element will contain all the information that STS needs to understand, like issue a new Kerberos token. When a token is ready to be delivered back to the client, STS creates a SOAP response and extends it with a RequestSecurityTokenResponse (RSTR). RSTR-element either hosts or provides a reference to the token element[5](p.326), and may also provide a session key so that all subsequent communication between the client and the service is encrypted using this key. The session key is temporary key and lasts as long as the session between the two parties is active.

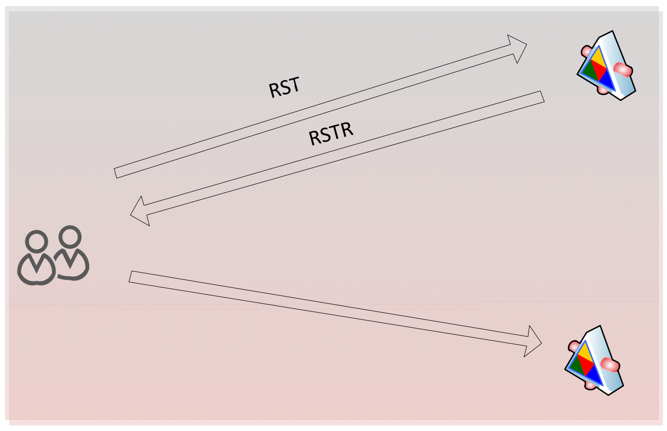


Figure 3.28: WS-Trust process

STS is clearly useful in situation when there is no direct trust between a service and its consumer but both of them trust STS in some way. The reason why the trust between STS and other parties does not have to be total is because some tokens, like X.509 certificate tokens, may prove its trustworthiness by the certificate authority that issued it. WS-Policy and WS-SecurityPolicy are policy specifications used by the business service and STS that tells the client what type of security mechanisms are needed in order to establish the communication with both of them[19].

The way it all works together is well-described by the scenario presented in the video by a Microsoft architect, Vittorio Bertocci[34]. Imagine that a client wants to buy wine on the internet by calling wine Web Service. Before proceeding with the transaction, the wine Web Service demands that the client provides its age verified by a trusted authority, specifically Driver and Vehicle Licensing Agency. Further on, the wine Web Service requires a SAML token that will contain the client’s age and the token must be signed by the Driver and Vehicle Licensing Agency. Driver and Vehicle Licensing Agency acts as a STS since it will vouch for client’s identity by presenting it with a required token. In order for all this to take place, different parties need to store public keys of each other so they all know who they are communicating with by validating each other’s signatures. Although public keys could be distributed by a XKMS service, we make an assumption that the Driver and Vehicle Licensing Agency, the wine service and the client have each other’s public keys stored locally, for simplicity sake. The first step (Step 1) is for the client to obtain the WS-Policy of the wine Web Service so it knows what type of security token the wine service requires. The client finds out that the SAML token is required and it must be signed by the Driver and Vehicle Licensing Agency (Step 2). Now the client prepares a RST message in which it requests a SAML token representing the age assertion. Before RST is sent to Driver and Vehicle Licensing Agency (STS), the RST message gets encrypted by the STS public key and signed by clients own private key (Step 3). When the STS receives the RST it verifies the client by its signature and issues the desired SAML token containing age equals to 22 years. The message that gets sent back to the client is of type RSTR (Step 4) and will contain both the SAML token and a proof token. In addition to the assertion, SAML token will also contain a session key and the whole SAML token will be encrypted and signed for the wine service so the client won't be able to read the SAML details. Proof token will contain the same session key that SAML token got but the proof token will be encrypted and signed for the client. The client receives the RSTR and now possesses the session key that will be used for all subsequent communication with the wine service. The client also possesses the SAML token which is going to be inserted into the order request SOAP message. The SOAP message will be encrypted and signed for the wine service and shipped to it (Step 5). The wine service verifies that the order comes from the client and verifies that the SAML token comes from the Driver and Vehicle Licensing Agency. Now the wine service opens the SAML token, reads that the client is 22 years old and is in possession of the session key as well. From now on all SOAP message exchange between the client and the wine service are encrypted by the session key they both possess (Step 6). The whole WS-Trust scenario is presented by Figure 3.29.

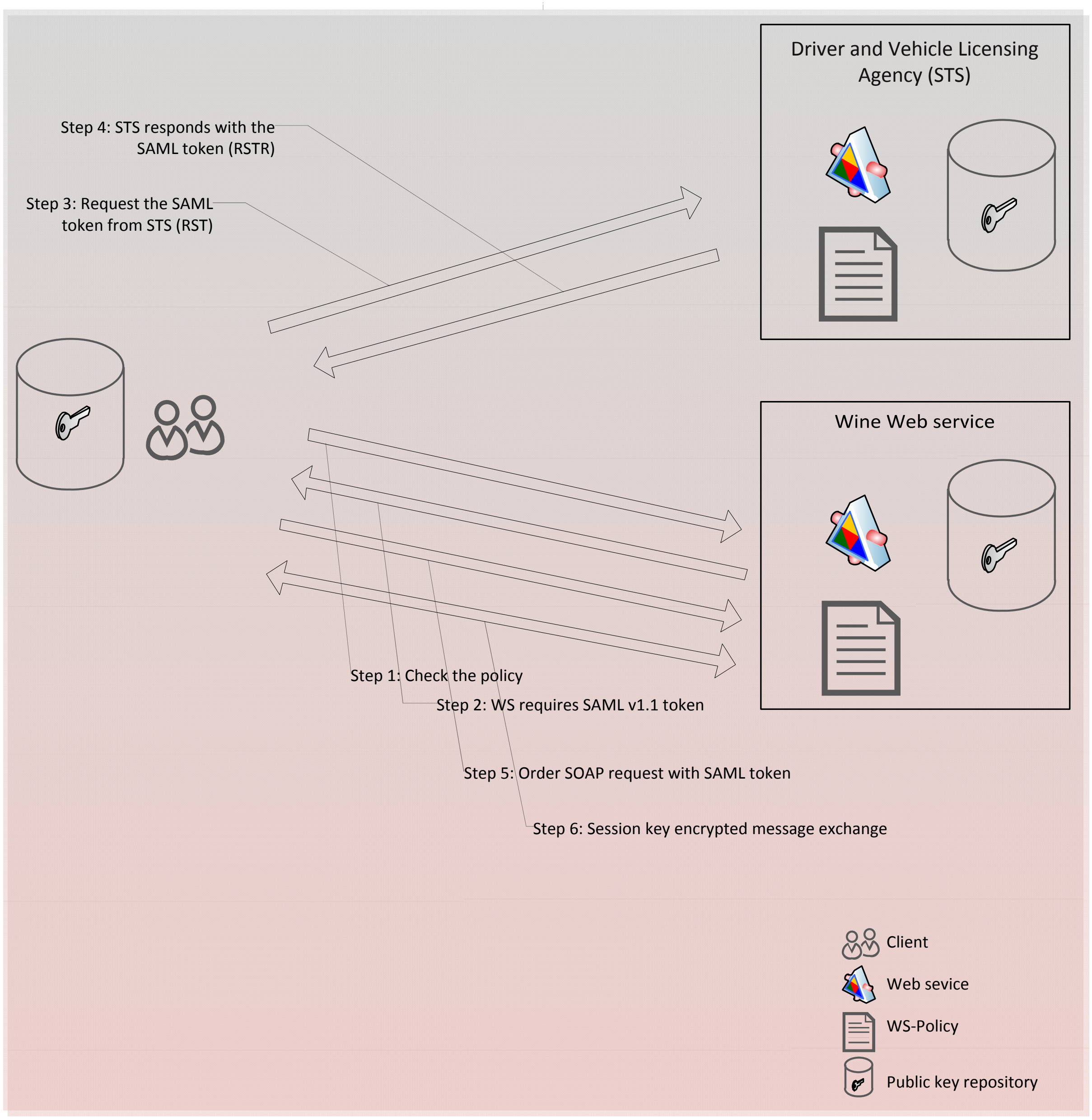

Driver and Vehicle Licensing Agency (STS)
Step 4: STS responds with the SAML token (RSTR)
Step 3: Request the SAML token from STS (RST)
Wine Web service
Step 1: Check the policy
Step 2: WS requires SAML v1.1 token
Step 5: Order SOAP request with SAML token
Step 6: Session key encrypted message exchange
Client
Web sevice
WS-Policy
Public key repository

Figure 3.29: Wine service with WS-Trust

3.6.4 WS-Federation

Through the history of information management there have been several proposals and approaches to identity management models. Today, those models are divided into three categories; distinguished isolated, centralized and distributed federated management model[25](p.81). The isolated model is the oldest approach and specifies that each service provider has its own identity provider meaning that a subject needs to provide separate credentials per service provider. The centralize model specifies a single identity provider managing multiple service providers and their identities providers. The weakness of this approach is issues regarding performance and availability since this model opens up for potential bottlenecks and single point of failure. The distributed federated management model defines a federation of identity providers where every identity provider manages identities within its own domain and vouches for those identities to another identity provider. This scenario enables co-existence of multiple security technologies and eliminates any technology changes for new members of the federation. The only requirement for a new identity provider is to establish trust with the existing members of the federation and provide a potential conversion between its own security technology used for authentication and authorization and the other security technologies used by other members[25](p.81). An important benefit of the federated identity management system is that it facilitates single sign-on.

WS-Federation falls into distributed federated management group. It is one of the newest OASIS standards reaching this status on 29 May 2009[51]. This standard builds on other standards like WS-Security, WS-SecurityPolicy, WS-Policy, WS-Trust, and provides federated identity architecture for Web Services architecture[50].

The start of a WS-Federation identity process is based on WS-Trust. Before a client use a service in another domain it needs to ask for the security token from its own identity provider. Identity provider is just a STS with additional extensions[25](p.87). The identity provider in the client's domain is a trusted third party for the client and the service and is responsible to provide any token type necessary to the client. In order to switch client's domain token to the security token required by the service, a pseudonym service is needed. The pseudonym service provides the mechanism for saving and obtaining alternate information about an identity, and offers identity or token change in a cross-domain scenario, for instance changing a X.509 certificate to a Kerberos token[5](p.333). In addition to identity provider and pseudonym service, WS-Federation defines attribute service and validation service[31]. Attribute service is responsible for saving information about an identity, for instance attribute information from a X.509 certificate. An UDDI service may become an attribute service[5](p.333) offering information about services, businesses, etc. Validation services is another feature of WS-Federation were special Web Services are used for validation of tokens with the purpose of determining the level of trust[5](p.331).

3.6.5 WS-SecureConversation

While WS-Security is used to ensure confidentiality and integrity of a single SOAP message its model becomes too costly when there is a need for multiple request/responses exchanges between a client and the server. WS-SecureConversation introduces the notion of security context token composed of a shared secret between the two parties, used to enforce both the confidentiality and the integrity upon multiple message exchanges in a given session period[52].

WS-SecureConversation became an OASIS standard on 01 March 2007[52] and builds on WS-Security and WS-Trust to provide functionality for establishing and identifying a security context[19]. The security context token needs to be created and exchanged between the parties before a session-like SOAP message exchange may begin. There are three ways to obtain and distribute security context tokens among parties and all three utilize WS-Trust[19]. The first way of token distribution is handled by a party creating the token by itself and sending it to the other party. The second way is by requesting the token from STS which will then distribute the token to both parties. The third way describes how parties may negotiate a security context token by using WS-Trust's four step negotiation protocol[5](p.334).

Full key exchange and authentication is only required when establishing security context token. After this phase every message is encrypted or signed with the information inside of the security context token. The secret which is a part of the security context token may be used to encrypt and sign the messages but the standard recommends the use of derived keys created from the secret context token. There can also be created several derived keys depending on the requirements. For instance, we may use one key to sign and other to encrypt the SOAP message[52]. Since all SOAP messages belonging to the same message exchange will reference the same security context token, there will be achieved increase in the overall performance[25](p.59).

3.7 REST alternative security efforts

It was already mentioned that there is no solution for message-level security for RESTful services but there exist some community supported efforts that address RESTful security which we feel should be mentioned. Since one of the efforts is based on WS-* security specification, it is appropriate to describe it now, after Web Services security introduction. The first effort is a thesis written by Dan R. Olsen at Brigham Young University[108]. The thesis discusses how elements from WS-Security specification can be included in the HTTP header. This is demonstrated by two different examples where the first shows how a URI can be signed and how that URI signature can be stored in the header as WS-Security-element. Another example shows UsernameToken-element of WS-Security stored in header as well. This means that the thesis does not address protection of the message in the HTTP body at all. Even though it is possible to use certain WS-* specifications to enable security functionality mentioned in Dan's thesis, we do not feel it is a right way to go since big and complex XML structures will be stored in the HTTP Header and decrease readability.

The second effort is an article written by Dan Forsberg who at that time was a researcher at Nokia Research Center[109]. The article deliberates about encryption of all kinds of HTTP content like pictures, files, etc. The same author has a strong focus on HTTP caching and discusses a way for providing decryption keys over TLS sessions. Although an interested article, there is no emphasis on signatures or partial encryption of the content. Since this was the most promising article we found on the Internet even if it did not address all of our requirements, we decided to find out more about it. We tried to search for a more descriptive paper or sample code but we were unsuccessful in finding it. At last, we decided to approached the author by mail and ask for more details. Dan was very responsive and told us that there is no sample code. We also got an impression that the article was never preceded by any other material since we could not find anything related to it on the Internet.

3.8 Summary

RESTful services and Web Services offer advanced mechanisms to ensure secure message exchange between multiple parties. While REST utilize traditional HTTP security mechanisms like SSL/TLS thus limiting itself to transport-level security, Web Services may be secured using both transport-level and message-level security. Message-level security is achieved by WS-Security, WS-Policy and WS-SecureConversation specifications which are based on well-known XML standards, like XML Signature and XML Encryption. WS-Security also supports different authentication tokens, a useful option that lets service providers standardize on their existing security infrastructure. Through other WS-* specification Web Services offer additional security schemes used to define advanced policies, negotiate trust through third-parties and accomplish single sign-on.

Although Web Services seem to be superior in their security repertoire, they have been labeled for being complex for both service consumers and providers[25](p.74). Industry is also struggling to implement WS-* security related specifications fully and in a standardized manner which has resulted in WS-I, an industry consortium aiming to provide guidelines for different specification implementers so that interoperability across separate platforms is achieved[25](p.76). We ended this chapter by highlighting some work done by the community although none of them fulfilled our requirements or were implemented.

4 Design

4.1 Intro

According to our research goals we reached our first objective which was studying a similar distributed technology and investigating mechanisms related to its message-level security. That was done in the chapter 3. This chapter is about the design of the complete solution, both from the coding perspective and the perspective of deciding on authentication mechanism, cryptography tokens and their distribution. Before the design is discussed we will present results from the WCF proof of concept where we give a short explanation of the framework, description of how RESTful messages are generated and demonstrate some unexpected behavior related to SOAP message-level security in WCF. The chapter will then, based on the requirements and WCF knowledge, continue with the discussion of the design where we challenge ourselves and argue for the optimal solution.

4.2 WCF proof of concept

Even though it may seem a bit odd to run proof of concept before designing the solution, it showed to be of significant use and great importance regarding the design. Since the developer of our solution is not very familiar with the WCF and its REST API, we felt the need to accumulate more knowledge before starting on the design. WCF is one of the most used frameworks and APIs for developing RESTful services. It is an acronym for Windows Communication Foundation and is a universal framework for building distributed services on the .NET platform and Microsoft operating systems [82](p.1). The WCF supports exposing .NET objects as Web Services, RESTful services, MSMQ, P2P based services and many others[82](p.1). A .NET object is exposed in chosen distributed technology mostly by configuration which does not demand deep knowledge of any distributed technology.

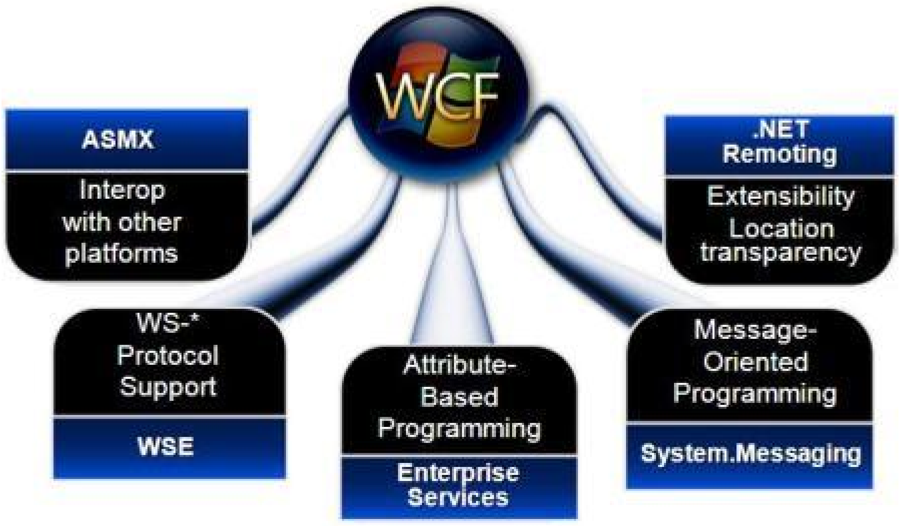

WCF
ASMX
Interop with other platforms
.NET Remoting
Extensibility Location transparency
WS-* Protocol Support
WSE
Attribute-Based Programming
Enterprise Services
Message-Oriented Programming
System.Messaging

Figure 4.1: WCF framework overview[106]

Through the proof of concept we learned how to develop Web Services and RESTful services, and we also studied how security mechanisms related to Web Services were enabled. The latter part was of big use, especially the way Web Services enable full and partial encryption and digital signatures in their messages. Figure 4.2 shows a class which will be serialized to SOAP XML and in which the Salary-member is marked for encryption and signature while the Name-member is marked for signature only.

```
using System.Net.Security;

[MessageContract]
public class Employee
{
  [MessageBodyMember(ProtectionLevel=ProtectionLevel.Sign)]
  public string Name;
  [MessageBodyMember(ProtectionLevel=ProtectionLevel.EncryptAndSign)]
  public int Salary;

}
```

Figure 4.2: Partial SOAP message-level protection in WCF

Any class marked with *MessageContract*-attribute will be serialized to SOAP message. We can also serialize members by setting a *MessageBodyMember*-attribute over any member we wish to expose. *ProtectionLevel* is an enumeration that might be a part of a type or any of its members, giving service provider a possibility to choose different protection levels per different scenarios. Unfortunately, we never succeeded to partially encrypt or sign a message by following a procedure found on the Microsoft Developer Network[90], which is a site for the .NET developers. Because of this we started to look for a solution on the internet. According to a post posted by an experienced security developer on a forum in which we also participated [89], it is not possible to have different protection levels for each body element even though it is syntactically allowed to set different ProtectionLevel-enumeration on them. Nevertheless, we found the idea behind this solution to be a very elegant one and a good candidate for our solution.

Another important discovery relates to the REST API of the WCF. Early in the beginning of the WCF proof of concept we wanted to explore how WCF auto-serialization works and localize messages just when the serialization process ends. We discovered that while the framework seems very user-friendly when offering standard functionality, it also hides majority of its complexity deep inside its code and demands serious work when a developer tries to accomplish a bit more advanced functionality. When we finally localized the code where we could study the structure of serialized messages, we discovered that WCF always serialize .NET types to XML, even in cases where JSON is chosen as the message format. This particular discovery was proven to be crucial regarding cryptography mechanisms for our solution.

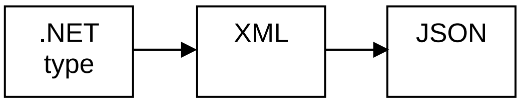


Figure 4.3: WCF JSON serialization steps

4.3 Design ideas

The lessons learned from the Web Services are that they offer multiple security mechanisms and support many security scenarios. Yet, these mechanisms have made Web Services even more complex and difficult to manage. But in every technology there are some positive and some negative characteristics and in this section we will concentrate on some security components used in SOAP messages and argue for their usage with the RESTful services.

While Web Services offer solution to advanced security scenarios, our scope is directed toward securing messages by introducing confidentiality and integrity provided by encryption and digital signatures respectively. In section 3.6.1 we saw this being implemented in Web Services using WS-Security specification. WS-Security specification ultimately extended the SOAP message with the additional security related elements that made SOAP messages larger, but it also made them transport neutral since all the security information like security tokens, encryption algorithm and digital signature were defined in the message itself and were not a part of its transporting protocol. WS-Security is also very flexible, allowing service providers to choose between several identity tokens, encryption algorithms and hash functions. Finally, WS-Security enables partial message encryption and partial signature which are crucial requirements for our solution. However, it is important to remember that WS-Security does not specify its own encryption and digital signature mechanisms but instead delegates that responsibility to the XML Encryption and XML Signature.

Contrary to the Web Services, RESTful services are highly dependent of HTTP and their messages cannot be transported neutral. Their architecture is light and their messages are simple. To keep both the architecture and the messages light and simple, the cryptography mechanism that we want to implement should not change those basic characteristics.

4.3.1 Locate implementation

In the first stage of our design process it is important to decide where to enable cryptography in order to support message-level security. In our case, message-level security can either be enabled on the .NET objects or their serialized representations (XML or JSON). This decision, as we will soon realize, will highly impact design and implementation timeframe.

Our first option is to secure .NET objects before they become serialized. While this seems feasible, there are several serious issues that prevent us from carrying out this sort of solution. The first issue is about interoperability. If a .NET object is fully encrypted, encoded and then serialized to XML or JSON, it will not be understood by another platform on the other side when it gets decrypted, simply because .NET objects can only be interpreted by a .NET environment. The second issue is about partial encryption which is practically impossible to implement on the objects because types, like an integer, cannot keep the ciphertext value.

Our second option is to secure serialized representation. By studying both formats, one would assume that in order to support XML and JSON two separate implementations should exist, because enabling partial encryption on both would be different due to their semantic structure. Creating and maintaining two implementations is very time consuming and offer several challenges. One of the great challenges is related to the partial cryptography and being able to navigate to the selected members in order to secure them. However, navigating a JSON message is not possible with the built-in classes. For instance, navigating to a member *Window* of a type *House* is not possible without creating a custom JSON navigator. Fortunately, based on our newly acquired knowledge presented in WCF proof of concept, we discovered that the WCF stores JSON temporarily as XML so we may secure the temporary XML before it becomes serialized to JSON. This approach will also work with the partial encryption and signature. If we enable cryptography at the XML level then the same logic can be implemented to secure regular XML formatted messages. In this fashion only one logic will exist for implementing encryption, decryption, digital signatures and validation, thus making it more maintainable and shortening our development time drastically.

4.3.2 Authentication

At message-level security, party identification is the initial step. Every party has to provide some sort of a unique and valid identification in order to proceed with the rest of the security process. Related to our case study, customer registry RESTful service may reuse well-known authentication mechanisms provided by HTTP or implement support to relatively new authentication delegation mechanisms like OpenID[95] and OAuth[83]. However, the HTTP mechanisms rely on the HTTP server configuration while OpenID and OAuth are dependent on third-party libraries and because of that we decided to apply one of the HTTP mechanisms. Since the HTTP Digest authentication seems to be a better option than the HTTP Basic authentication, we choose the HTTP Digest as the authentication mechanism for our solution.

4.3.3 Cryptography token

Although a token may be used for authentication and cryptography, we decided to reuse the existing HTTP authentication mechanism as stated in the previous section. In the case of cryptography there should be chosen a token that is widely used. A widespread cryptography token in the corporate world is the digital certificate based on X.509. CA's are responsible for binding the keys to the certificate which is also used as the party identification on the internet. In addition, the fact that the CA vouches for someone's certificate solves the issue regarding the trust of the certificates exchanged. Nevertheless, digital certificates provided by the CAs are expensive are probably not affordable by many. For instance, per May 2011, VeriSign offered digital certificates from $399 up to $1499, for one year validity[91]. There are also free tools that can create digital certificates and one of the famous ones is Open SSL[102]. The certificate that is created by the Open SSL is self-signed and its validity period can be set to a desired date.

4.3.4 Certificate distribution

An important part of the PKI is certificate distribution. After the study related to the XKMS in section 3.5.3, there were concerns of how to implement something of these proportions to RESTful services. In our opinion certificate distribution is such a complex process as well as a solution, that it will jeopardize the simplicity of the RESTful services. What we consider to be more appropriate for our solution, primarily from the perspective of the complexity but also based on the limited development resources, is a non-technical, social resolution. The resolution will be particularly useful when the certificate is self-signed and its origin cannot be vouched for. Our idea is based on certificate distribution fulfilled by mail exchange while being on a phone with a targeted party. The phone is used to ensure that the correct party is contacted but it is also used as a starting point of the transaction. When the targeted party answers the call, a short time window is created until certificates are exchanged through the mail. Due to the limited time, both parties can trust the incoming mail to contain the original certificate and not to be a fraud mail sent by someone else. Even if our idea may be original, phones, especially mobile phones, are sometimes used to verify already provided information or as an accompanying token. This is particularly common in an authentication process. For instance, Citrix, a well-known virtualization system, automatically sends each client a session password by SMS which the client has to provide together with its credentials in order to be successfully authenticated[103]. This type of authentication approach is referred to as two-factor authentication[103].

To describe our idea of the certificate exchange let us think of two parties, A and B. Party A contacts party B by the phone and asks B to send its certificate by mail. B sends its certificate to A and vice versa while they both are on the phone line. Since A and B are on the phone simultaneously they are able to establish a trusted relationship and follow the progress of the exchange. When certificates are exchanged both parties will be aware of it and the communication may end. It is a very light solution but we acknowledge that it can also be very unpractical on a larger scale.

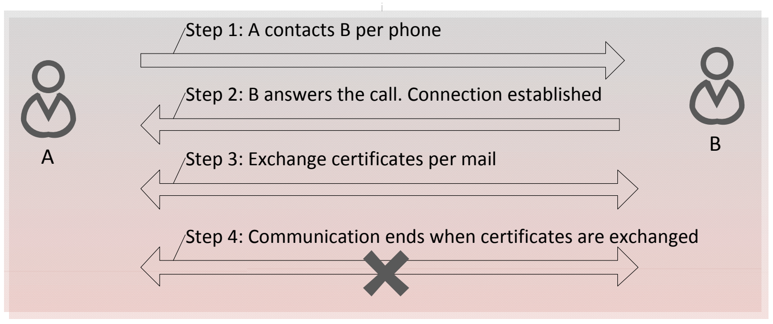

Step 1: A contacts B per phone
Step 2: B answers the call. Connection established
Step 3: Exchange certificates per mail
Step 4: Communication ends when certificates are exchanged
A
B

Figure 4.4: Certification distribution process

4.3.5 Cryptography mechanism

In our case, in order to implement a PKI based solution, we should use a standardized end-to-end security mechanism and not handle necessary cryptography steps separately. To be more precise, in a PKI solution when enabling cryptography such as encryption, there are multiple steps that have to be taken.

1. Generate a session key
2. Use the session key to encrypt a message
3. Use the public key to encrypt the session key
4. Provide the encrypted message together with the encrypted session key to the other party so it can decrypt the message

By using a standardized cross-platform cryptography mechanism we encourage interoperability. There are several security mechanisms that fulfill PKI goals and our requirement of being a cross-platform. Some of the most famous ones are Pretty Good Privacy (PGP), Cryptographic Message Syntax (CMS) and XML Signature and XML Encryption.

PGP is a security mechanism mainly used to encrypt and sign e-mails and files [93](p.3). It is a well-used and relatively light-weight security mechanism that has been used for two decades[93](p.10). In addition, PGP supports both public keys and digital certificates. Unfortunately, there are no .NET built-in classes supporting the PGP cryptography but there are several .NET open source libraries, such as Bounty Castle[92], offering this kind of cryptography. By implementing an external library we would breach our principle of not using any external APIs so the PGP will not be considered for our solution.

Cryptographic Message Syntax (CMS) is a specification that may be used to sign and encrypt any digital content[94]. It is well-supported in .NET and many other platforms and it supports a variety of encryption and hashing algorithms. The down side of this specification, in regard to our requirements, is that the partial signing and encryption are not supported by default meaning that the selected message parts must be extracted, signed, encrypted and then added back to the original message replacing the clear text. The receiving party is then responsible to extract the ciphertext, decrypt, validate and replace the ciphertext with the clear text. Another down side is the issues regarding backward-compatibility between newer and older versions[94].

XML Encryption and XML Signature can also be used to achieve our goals. Both specifications are a fundamental part of Web Services and XML security specifications and because of that, probably less prone to cardinal changes which could make them less backward-compatible in the future. Although not a formal requirement, less changes and backward-compatibility are important characteristics when building applications that should last for longer periods. The main reason why those two specifications seem to be better alternatives are because both are made primarily for the XML documents and are well-supported on multiple platforms. In addition, partial signing and encryption is possible without introducing custom code. Therefore, we conclude that the most suitable cryptography mechanism for our solution is the combination of XML Encryption and XML Signature.

4.3.6 Message size and readability

By standardizing cryptography on XML Encryption and XML Signature we are ensured to be using optimal mechanisms for securing XML. But every choice comes with a price and in this case the price is readability. From the simple examples showed in sections 3.5.1 and 3.5.2 we have seen that both standards produce a lot of information when they are enabled upon a XML document. All this information is important for both security standards and cannot be removed from the XML. In addition, a lot of the information produced by both security mechanisms is repeated throughout the document. For instance, if we take a closer look to the information produced by XML Encryption we will notice that a lot of it is repeating. This may be seen in Figure 4.6 which shows a XML message containing customer data related to our case study.

```
<ArrayOfSerCustomer xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <SerCustomer>
    <Id>14</Id>
    <LastName>Normann</LastName>
    <Navn>Ola</Navn>
    <SerAddress>
      <City>OSLO</City>
      <PostalCode>1567</PostalCode>
      <Street>Christian Michelsens gate 16</Street>
    </SerAddress>
  </SerCustomer>
  <SerCustomer>
    <Id>15</Id>
    <LastName>Olsen</LastName>
    <Navn>Kari</Navn>
    <SerAddress>
      <City>OSLO</City>
      <PostalCode>1567</PostalCode>
      <Street>Christian Michelsens gate 16</Street>
    </SerAddress>
  </SerCustomer>
</ArrayOfSerCustomer>
```

Figure 4.5: XML message containing two customers

```
<ArrayOfSerCustomer xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <!--##################################-->
  <!-- CUSTOMER with last name Normann  -->
  <!--##################################-->
  <SerCustomer>
    <Id>14</Id>
    <LastName>Normann</LastName>
    <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#">
            <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
            <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
              <X509Data>
<X509Certificate>MIIDzTCCArmgAwIBAgIQ9le+j1LyAoJJDZTI7QaWhzAJBgUrDgMCHQUAMFsxCz
AJBgNVBAYTAk5PMQ0wCwYDVQQHEwRPc2xvMREwDwYDVQQLEwhPcmcgVW5pdDEVMBMG
A1UEChMMT3JnYW5pemF0aW9uMRMwEQYDVQQDEwpNeSBSb290IENBMB4XDTEwMTAxMTE3
Mzk1NloXDTE4MTAxMTE3Mzk1NVowWzELMAkGA1UEBhMCTk8xDTALBgNVBAcTBE9zbG8xET
APBgNVBAsTCE9yZyBVbml0MRUwEwYDVQQKEwxPcmdhbml6YXRpb24xEzARBgNVBAMTCk15I
FJvb3QgQ0EwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCqcVtqOLwn4JinDKMHu16
2/bxstk6I1ObSp/FLxuatquJuNJncqxNfqzGzeBg/VURRw8RMDa5laWLV6KjbX2VpHg8A4m1MsXX83zU
516RkHWzMlrr6wlnbN72VrioIDf24fwhnCtjfOJ1CsEJuxTkDFvtSFUevsbGq5J0taJrPI4eIMehrEEtnAcoX8
7GNRj3FLUSEuyTGX2pwHHkF01cSuSjl08UX1E3JwBDt1WyHFB4fHG51soczIIG9aS/vYAlIp7LNKhmE
yCv2zd+dQgS645NaFrFL7yM9LPMXA2TCJVDejh2FRwkXSDtIvojZt1fxk+Q52+5zRi7mlGzVFtf1AgMB
AAGjgZQwgZEwgY4GA1UdAQSBhjCBg4AQFGRX3nk6+CwkKk6pTLjB9qFdMFsxCzAJBgNVBAYTA
k5PMQ0wCwYDVQQHEwRPc2xvMREwDwYDVQQLEwhPcmcgVW5pdDEVMBMGA1UEChMMT3Jn
YW5pemF0aW9uMRMwEQYDVQQDEwpNeSBSb290IENBghD2V76PUvICgkkNlMjtBpaHMAkGBSsO
AwIdBQADggEBAEcjZF70sBpy3gPVv/GYeggs1pURuXxefPODLgZBT74vY0TGJvicGWTfgCjj+Tx/zXT
zW8zM6fTFLYIoYA4ZryRM11aiYfdHwJfd6dAgoSIBwEMEZspJYZ7B4g7KuQEweeJticOXOK4aK6pqT
oxMWvpLwUcBGXQfMp6Uz2zYjQw8n5XFYpe3kIiEDDH/9XRMGIteuYBAo/DvMjhbhsv9MUotK3v1oq
6tIXTjoDeDmmx6j45ZO8cRQ2wfbpjDXlJXmDkQDQD2J39dytmX+fBxDg5QHCmg62Y+rvjRoqVsth/Asi
```

```
vN16BQ0mhhvI06Zh0ORE2GQUso5ohoqdQJXDNDyVg=</X509Certificate>
              </X509Data>
            </KeyInfo>
            <CipherData>
<CipherValue>Lfc4eGugyFAifvm1KnppO1kaMcDbgL0Av5nJsIjHK8CbsN5ySfmm7Ipu9tRLUBXTb63+ZZq5FrvCfsmhR+SGSted1QjX1ajZ4RgPk26bN1Xuuld9QRUK/O2o0h+Zk590GFR5R3KZ9Be2zMocSW5Jm+TQH4Vtei/lJgYHaRChW+aHekZlV2RASU4FBdsBxQKq5j9QSltl0GbUy0zSfB1IvQn6vbumOfC8oVE+/XyTH0+1x7zA2XwjWgzYz17fCKUV1WJQgsc4jD+Ct1mqC5b67I+80FPx77lW9lz0Sj7ILBTTNJ+7uIGoG8meSuQQUboi50KMKaNPYii7QgC5NQhO9A==</CipherValue>
            </CipherData>
          </EncryptedKey>
        </KeyInfo>
        <CipherData>

<CipherValue>J62XtOHX7f0An8DckXbDNYdAoADTdGq01FclGcjEwHfx2aoPu2zIBrvOKpEO2Oz/</CipherValue>
        </CipherData>
      </EncryptedData>
    </Name>
    <SerAddress>
      <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#">
            <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
            <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
              <X509Data>
<X509Certificate>MIIDzTCCArmgAwIBAgIQ9le+j1LyAoJJDZTI7QaWhzAJBgUrDgMCHQUAMFsxCzAJBgNVBAYTAk5PMQ0wCwYDVQQHEwRPc2xvMREwDwYDVQQLEwhPcmcgVW5pdDEVMBMGA1UEChMMT3JnYW5pemF0aW9uMRMwEQYDVQQDEwpNeSBSb290IENBMB4XDTEwMTAxMTE3Mzk1NloXDTE4MTAxMTE3Mzk1NVowWzELMAkGA1UEBhMCTk8xDTALBgNVBAcTBE9zbG8xETAPBgNVBAsTCE9yZyBVbml0MRUwEwYDVQQKEwxPcmdhbml6YXRpb24xEzARBgNVBAMTCk15IFJvb3QgQ0EwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCqcVtqOLwn4JinDKMHu162/bxstk6I1ObSp/FLxuatquJuNJncqxNfqzGzeBg/VURRw8RMDa5laWLV6KjbX2VpHg8A4m1MsXX83zU516RkHWzMlrr6wlnbN72VrioIDf24fwhnCtjfOJ1CsEJuxTkDFvtSFUevsbGq5J0taJrPI4eIMehrEEtnAcoX87GNRj3FLUSEuyTGX2pwHHkF01cSuSjl08UX1E3JwBDt1WyHFB4fHG51soczIIG9aS/vYAlIp7LNKhmEyCv2zd+dQgS645NaFrFL7yM9LPMXA2TCJVDejh2FRwkXSDtIvojZt1fxk+Q52+5zRi7mlGzVFtf1AgMBAAGjgZQwgZEwgY4GA1UdAQSBhjCBg4AQFGRX3nk6+CwkKk6pTLjB9qFdMFsxCzAJBgNVBAYTAk5PMQ0wCwYDVQQHEwRPc2xvMREwDwYDVQQLEwhPcmcgVW5pdDEVMBMGA1UEChMMT3JnYW5pemF0aW9uMRMwEQYDVQQDEwpNeSBSb290IENBghD2V76PUvICgkkNlMjtBpaHMAkGBSsOAwIdBQADggEBAEcjZF70sBpy3gPVv/GYeggs1pURuXxefPODLgZBT74vY0TGJvicGWTfgCjj+Tx/zXTzW8zM6fTFLYIoYA4ZryRM11aiYfdHwJfd6dAgoSIBwEMEZspJYZ7B4g7KuQEweeJticOXOK4aK6pqToxMWvpLwUcBGXQfMp6Uz2zYjQw8n5XFYpe3kIiEDDH/9XRMGIteuYBAo/DvMjhbhsv9MUotK3v1oq6tIXTjoDeDmmx6j45ZO8cRQ2wfbpjDXlJXmDkQDQD2J39dytmX+fBxDg5QHCmg62Y+rvjRoqVsth/AsivN16BQ0mhhvI06Zh0ORE2GQUso5ohoqdQJXDNDyVg=</X509Certificate>
              </X509Data>
            </KeyInfo>
            <CipherData>
<CipherValue>nsBgcwWvf8kqWLoDtnLqArDDsCl/m65+z8CxJuwiDC7eXTC7T6FIlfpIHC9Q5nPYS8iKSuiyOEKx2ym2ZGcz2vrpJ9CA76UDW097Yui3YAFWPPNZWw8Y82etgogEkos8o5oyiYC9SLcJqtBBC4dbuRLmRx4uP0VMpZO0vEfyJI79dKAPy/BSVpmZBztLyeiz4SgndRL164xHuYcC9w56k0WptwHS6f58cMP3lGwR7Y0F9MG2c8Xes2fbb9usmj9Fu+djY1oSe4Q/9mOokUTzC3TqqQWy0zse7Pwn685t5Ujp7wTovGxw7SJDFugO36Ydd4ka1T4fpXXag9CLI93KyA==</CipherValue>
            </CipherData>
          </EncryptedKey>
        </KeyInfo>
        <CipherData>
```

```
<CipherValue>LhfD5A2kMxNcaeTlj4o1bpULxMlf4U64D1PyLc1NGIduNO60gwhn8OXd50UQpqnywKqc
rY39ZpFaLdtwpIYE/eSbyDtexJWtNtAyKR6fDreH0nEdH8ED51stOxVDm0J07HfEsS9pzxUCcUHHW4xL
J0T2V6J7gbmHFjt9AiWtL+aM85qcN2O9h8IoIj/LdHKH</CipherValue>
        </CipherData>
      </EncryptedData>
    </SerAddress>
  </SerCustomer>
  <!--###################################-->
  <!-- CUSTOMER with last name Olsen  -->
  <!--###################################-->
  <SerCustomer>
    <Id>15</Id>
    <LastName>Olsen</LastName>
    <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#">
            <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
            <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
              <X509Data>
<X509Certificate>MIIDzTCCArmgAwIBAgIQ9le+j1LyAoJJDZTI7QaWhzAJBgUrDgMCHQUAMFsxCz
AJBgNVBAYTAk5PMQ0wCwYDVQQHEwRPc2xvMREwDwYDVQQLEwhPcmcgVW5pdDEVMBMG
A1UEChMMT3JnYW5pemF0aW9uMRMwEQYDVQQDEwpNeSBSb290IENBMB4XDTEwMTAxMTE3
Mzk1NloXDTE4MTAxMTE3Mzk1NVowWzELMAkGA1UEBhMCTk8xDTALBgNVBAcTBE9zbG8xET
APBgNVBAsTCE9yZyBVbml0MRUwEwYDVQQKEwxPcmdhbml6YXRpb24xEzARBgNVBAMTCk15I
FJvb3QgQ0EwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCqcVtqOLwn4JinDKMHu16
2/bxstk6I1ObSp/FLxuatquJuNJncqxNfqzGzeBg/VURRw8RMDa5laWLV6KjbX2VpHg8A4m1MsXX83zU
516RkHWzMlrr6wlnbN72VrioIDf24fwhnCtjfOJ1CsEJuxTkDFvtSFUevsbGq5J0taJrPI4eIMehrEEtnAcoX8
7GNRj3FLUSEuyTGX2pwHHkF01cSuSjl08UX1E3JwBDt1WyHFB4fHG51soczIIG9aS/vYAlIp7LNKhmE
yCv2zd+dQgS645NaFrFL7yM9LPMXA2TCJVDejh2FRwkXSDtIvojZt1fxk+Q52+5zRi7mlGzVFtf1AgMB
AAGjgZQwgZEwgY4GA1UdAQSBhjCBg4AQFGRX3nk6+CwkKk6pTLjB9qFdMFsxCzAJBgNVBAYTA
k5PMQ0wCwYDVQQHEwRPc2xvMREwDwYDVQQLEwhPcmcgVW5pdDEVMBMGA1UEChMMT3Jn
YW5pemF0aW9uMRMwEQYDVQQDEwpNeSBSb290IENBghD2V76PUvICgkkNlMjtBpaHMAkGBSsO
AwIdBQADggEBAEcjZF70sBpy3gPVv/GYeggs1pURuXxefPODLgZBT74vY0TGJvicGWTfgCjj+Tx/zXT
zW8zM6fTFLYIoYA4ZryRM11aiYfdHwJfd6dAgoSIBwEMEZspJYZ7B4g7KuQEweeJticOXOK4aK6pqT
oxMWvpLwUcBGXQfMp6Uz2zYjQw8n5XFYpe3kIiEDDH/9XRMGIteuYBAo/DvMjhbhsv9MUotK3v1oq
6tIXTjoDeDmmx6j45ZO8cRQ2wfbpjDXlJXmDkQDQD2J39dytmX+fBxDg5QHCmg62Y+rvjRoqVsth/Asi
vN16BQ0mhhvI06Zh0ORE2GQUso5ohoqdQJXDNDyVg=</X509Certificate>
              </X509Data>
            </KeyInfo>
            <CipherData>
<CipherValue>Woeue9Lh3Na5rOnBAi60VADt4n7xQWmpIfPhlQ8zabBoox3ZuDCBK+bD8Ss4gkrt0wNet
Ncb8e+Cve1zh/GZ4SqfdAWEXtMnUIZbFioHGf7qdmGcaXS3AFTAq/wahCdHh3qq44Bjgt4KY72iBImF7
WOTqYL1Trb6+YlzprrocnH2EloTvNUG8y5Aer0UvzanEG5FZOgLqcuzEw/2W9mvUQ+HBo9QGrhGvT
HW/w+KJ9iFCreF9MkdE+6pqHSkotXH/88Yggs6fd7/jkxUmqxVVNoeckZ7X0X+GQ10ItnOcWO5swcl7V
1oE/OBNOYNIpSkUv6BSXP5vFl2hC4DHPkdIw==</CipherValue>
            </CipherData>
          </EncryptedKey>
        </KeyInfo>
        <CipherData>
<CipherValue>hXAYOlAtG3dQGUil64BVB/NGOrsHhs2XbEbmrusM5bIz8FGAChM8rEUpRs7+prSh</Ci
pherValue>
        </CipherData>
      </EncryptedData>
    </Name>
    <SerAddress>
```

```
   <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#">
      <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
      <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
       <X509Data>
<X509Certificate>MIIDzTCCArmgAwIBAgIQ9le+j1LyAoJJDZTI7QaWhzAJBgUrDgMCHQUAMFsxCz
AJBgNVBAYTAk5PMQ0wCwYDVQQHEwRPc2xvMREwDwYDVQQLEwhPcmcgVW5pdDEVMBMG
A1UEChMMT3JnYW5pemF0aW9uMRMwEQYDVQQDEwpNeSBSb290IENBMB4XDTEwMTAxMTE3
Mzk1NloXDTE4MTAxMTE3Mzk1NVowWzELMAkGA1UEBhMCTk8xDTALBgNVBAcTBE9zbG8xET
APBgNVBAsTCE9yZyBVbml0MRUwEwYDVQQKEwxPcmdhbml6YXRpb24xEzARBgNVBAMTCk15I
FJvb3QgQ0EwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCqcVtqOLwn4JinDKMHu16
2/bxstk6I1ObSp/FLxuatquJuNJncqxNfqzGzeBg/VURRw8RMDa5laWLV6KjbX2VpHg8A4m1MsXX83zU
516RkHWzMlrr6wlnbN72VrioIDf24fwhnCtjfOJ1CsEJuxTkDFvtSFUevsbGq5J0taJrPI4eIMehrEEtnAcoX8
7GNRj3FLUSEuyTGX2pwHHkF01cSuSjl08UX1E3JwBDt1WyHFB4fHG51soczIIG9aS/vYAlIp7LNKhmE
yCv2zd+dQgS645NaFrFL7yM9LPMXA2TCJVDejh2FRwkXSDtIvojZt1fxk+Q52+5zRi7mlGzVFtf1AgMB
AAGjgZQwgZEwgY4GA1UdAQSBhjCBg4AQFGRX3nk6+CwkKk6pTLjB9qFdMFsxCzAJBgNVBAYTA
k5PMQ0wCwYDVQQHEwRPc2xvMREwDwYDVQQLEwhPcmcgVW5pdDEVMBMGA1UEChMMT3Jn
YW5pemF0aW9uMRMwEQYDVQQDEwpNeSBSb290IENBghD2V76PUvICgkkNlMjtBpaHMAkGBSsO
AwIdBQADggEBAEcjZF70sBpy3gPVv/GYeggs1pURuXxefPODLgZBT74vY0TGJvicGWTfgCjj+Tx/zXT
zW8zM6fTFLYIoYA4ZryRM11aiYfdHwJfd6dAgoSIBwEMEZspJYZ7B4g7KuQEweeJticOXOK4aK6pqT
oxMWvpLwUcBGXQfMp6Uz2zYjQw8n5XFYpe3kIiEDDH/9XRMGIteuYBAo/DvMjhbhsv9MUotK3v1oq
6tIXTjoDeDmmx6j45ZO8cRQ2wfbpjDXlJXmDkQDQD2J39dytmX+fBxDg5QHCmg62Y+rvjRoqVsth/Asi
vN16BQ0mhhvI06Zh0ORE2GQUso5ohoqdQJXDNDyVg=</X509Certificate>
       </X509Data>
      </KeyInfo>
      <CipherData>
<CipherValue>JQua/+gmrZeauoJ51acPrKSzII72aFvEbD091z9MdrJ5dGjAz9c+QllQoa6b3dJvjvaosI0+XeQ
mq97h3hJyoeQU6PDVDNTj4vfkYFupXT3zJcHkzx4HNfYgKFK03wVRqAQ9GhOSBKq0ECllltuUhTny+
W3z0RrM1+aQu6s1z+32gpGOgQwwYntf1tG0VgxAKnl/KOyYMIAEg3L4MksFkt/Yrc/QsBN7TCcLwoP2
2nrX/8sUCyjzXV7TG1yH0iQL2vgmFsc+Y51cIFVOIWQNdXr1tppzd/rmwvunmOjEFIQO4dYdn7H3dprO
cQQphXTLF1/BHWrC7CxWGZ/sCB+6JXQ==</CipherValue>
      </CipherData>
     </EncryptedKey>
    </KeyInfo>
    <CipherData>
<CipherValue>nYMjAiI+ZARr20O54Laa66wrvnuNw9nKi0V9olTDDfZ6doGNbKzwhIK+x89FbgJvDN1Y
M1XMuWnVvK0EoxfoV5OkLMekcISgpKAlOfGHKd8JyL5nl25IJbl3qAZKeDK+gYdmK+u0m59UnwbM
80llVrYBwdNz7LA98+xaIBsCQj/+/45T46SjutpstegzV7kB</CipherValue>
    </CipherData>
   </EncryptedData>
  </SerAddress>
 </SerCustomer>
</ArrayOfSerCustomer>
```

Figure 4.6: XML message containing two customers where Name and SerAddress elements are encrypted

By looking at the previous figure we may notice how elements like EncryptedData/EncryptionMethod, EncryptedData/KeyInfo/EncryptedKey/EncryptionMethod and EncryptedData/KeyInfo/EncryptedKey/KeyInfo contains exactly same values for both customers. In fact all three elements are repeated four times because *Name* and

SerAddress-elements are encrypted for each customer. These repetitions occur due to the fact that elements may be protected by using multiple keys. Although XML Encryption is a highly flexible standard with multiple configuration options, there is no logic for deprecating duplicate elements like the ones we mentioned. The message in this state is not very readable and it is too large as well. The encrypted message is much larger than its original. To be more precise, the original message is approximately 508 bytes in size while the encrypted one is 9,66 KB. In order to make the encrypted message more readable as well as to decrease its size, we could implement a logic that will eliminate duplicates after encryption process but also restore eliminated elements when the message reaches its final destination. We need to be careful when restoring the original message state because if it is not restored properly then the signature validation process or decryption process will fail.

Now, if the logic is implemented in .NET then it will tie our solution even more to one specific platform. Fortunately, there are cross-platform technologies like XPath and XSLT that can be used to achieve the transformation logic[5](p.96-97). Since XPath and XSLT are optimized for XML manipulation and use XML syntax, we find the idea of implementing logic that will remove and recreate elements very feasible. By following this idea we ensure that our solution is less platform-dependent while keeping the .NET code clearly separated from the transformation logic. The latter is a part of a design principle known as *separation of concerns* which goal is to create systems so that separated layers can be developed independently, thus making it easier to understand, design and manage[104]. In the chapter 5 we will see how this principle was implemented on the rest of our solution. Following figure summarizes the design discussion.

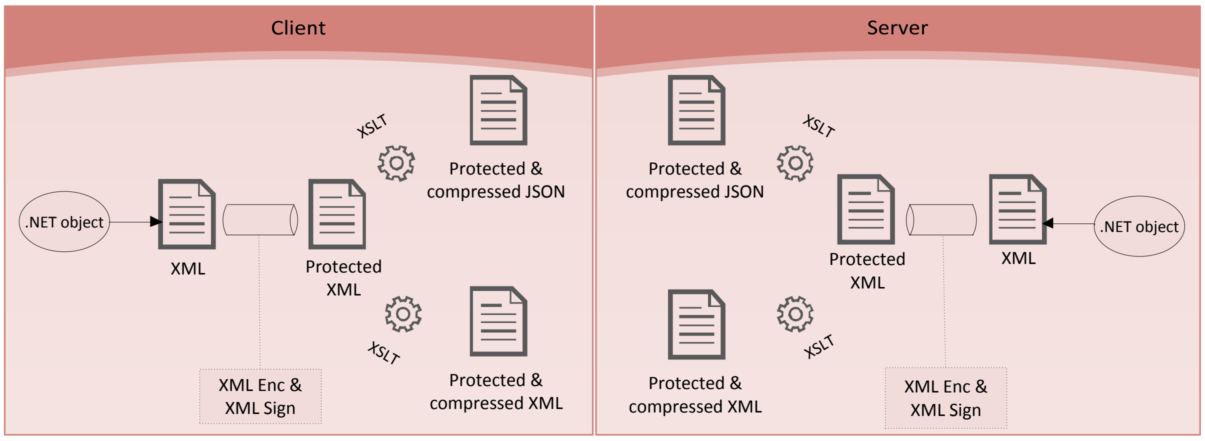


Figure 4.7: Process of protecting and unprotecting a RESTful message

4.4 Summary

In the previous sections of this chapter we described the architecture and design of our forthcoming solution. We concluded that HTTP Digest authentication is to be used as the authentication mechanism while digital certificates will be used as the cryptography token due to its widespread usage. We also proposed a new way to distribute certificates, especially self-signed ones, by the use of email and phone. This is done in order to keep the security architecture simple. Cryptography operations cannot take place at the .NET type level since it is very difficult to enable partial encryption on the programing language types. Instead cryptography will be implemented at the XML level only as this will impact both XML and JSON because WCF use XML as a temporary format when serializing and deserializing objects to and from JSON. As a result of that it will be implemented common code to secure both formats. After studying different cryptography mechanisms we found XML Signature and XML Encryption to be optimal choices for our solution. Still, since both standards generate a lot of information we will use XSLT to create simple and readable messages before shipping them to its destination. This way we create more interoperable artifacts so that different platforms may understand those and reuse them.

Here we conclude the architecture and design chapter and leave the description of the realization and implementation details to the next.

5 Implementation

5.1 Intro

This chapter is about code implementation from the design discussed in the previous chapter. The customer registry system case study scenario will be used so as to demonstrate a particular scenario although the solution is intended to be useful on many other scenarios.

We start the chapter by presenting a software development methodology which we followed during the development of our solution. Furthermore, we will give a presentation of the case study system and the RESTful service that is based on it. Then we move onto the implementation of the new security library and message compression library. Finally, the chapter ends by presenting couple of results related to comparison of protected and compressed messages versus protected and uncompressed messages.

5.2 Software development methodology

In order to have a more structured and possibly successful software development, a specific project methodology should be followed. There exist many project methodologies and some of the most famous are related to agile software development[105]. Examples of agile software development methods are Extreme Programming, Feature Driven Development and SCRUM. The characteristics of such methodologies are their responsiveness to changing requirements. This is for instance important in project based development where technology is new and unfamiliar to the developers. With the intension of following a specific agile methodology we found SCRUM to be the optimal one for our development process.

The strong points of SCRUM are that many development processes cannot be predicted hence an exact development description is not required because it will likely change during the development[101]. The methodology also gives space for change of the initial plan as we make progress. According to SCRUM, prior to starting on the development we should have created a product backlog[101]. Product backlog is a list of prioritized requirements or functionalities which will keep a team or a developer focused on the primary work. In our case we needed to base primary requirements on the use case chosen for the project and keep them in the product backlog. Our initial product backlog had following requirements for the functionality.

Table 5.1: Product backlog

Item no.	**Description**	**Estimation (hours)**
	Create customer registry system	
1	Design customer database	1
2	Create customer database	1
3	Design customer registry system	1
4	Develop & test customer registry system	8
	Create RESTful services for the customer registry system	
5	Design RESTful service	1,5
6	Develop & test service	12
7	Develop & test a client	6
	Create .NET library for signature & encryption	
8	Design encryption and signature implementation	5
9	Develop the library	20
	Create library for dynamic XPath generation used for finding types & their properties prior to signature and encryption	
10	Design of the library	3
11	Develop & test library for XPath generation	12
	Create XSLT transformations for message compression and decompression	
12	Understand the JSON XML format	4
13	Design compression/decompression process	7
14	Create transformation from secured plain XML to its compressed variant	15
15	Create transformation from compressed XML message to its original variant	15
16	Create transformation from secured JSON XML message to its compressed variant	10
17	Create transformation from compressed JSON XML message to its original variant	10
	Finalize the solution	
18	Adapt new libraries with the service and the client	6
19	Tie the whole solution together and run final tests	25
Sum:		**162,5**

It is important to have in mind that this product backlog was created in a time period where we did not have much knowledge about most of the technologies used in this project and estimations were likely to change on the way. The next step was to create a sprint backlog which is a list of concrete development tasks based on the items defined in the product backlog. After breaking the items into development tasks we managed to define 85 tasks in total. When it comes to the development process itself, SCRUM splits it into several smaller development periods called sprints. A sprint usually last for two to four weeks and at its end a team should have a piece of code that is fully functional and tested. During a sprint we are not allowed to add new change requests but we are allowed to refactor the code as much as it is needed. It means that after the first sprint we should have created and tested customer registry system, RESTful services for the registry system, new C# library for signature and encryption, and library for generating dynamic XPath expressions. However, since there is only one developer involved in the coding process it was convenient to have each sprint last for one week. In this manner we encouraged ourselves to have functional code available more often and in a testable state. Since we calculated with a seven hour workdays we initially created five sprints so to complete the development.

5.3 Customer registry system

The customer registry system is a business system that our use case is based on. It was developed as part of sprint 1 and uses a SQL Server database to store and query customer data. The system is very simple offering basic operations for maintaining customers. Class diagram in Figure 5.2 demonstrates relationships between two entities, *Customer* and *Address* where Address is part of Customer class. The whole system is designed as an n-tier application where each tier or layer is responsible to handle separate logic. On top of model classes there is a data access layer class *CustomerDal* and an interface *ICustomedDal* which are responsible to store and retrieve data from the database. Data access layer is consumed by business logic layer which involves a class, *CustomerBll* and an interface *ICustumerBll*. Business logic layer is the layer used for communication with other applications and offers six operations; DeleteCustomer, GetCustomerById, GetListOfCustomers, GetListOfCustomersByName, InsertCustomer and UpdateCustomer. Figure 5.1 shows how the system is layered.

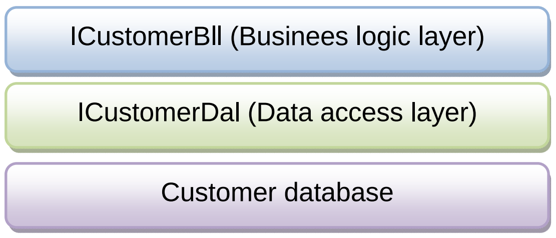

ICustomerBll (Businees logic layer)
ICustomerDal (Data access layer)
Customer database

Figure 5.1: Customer registry system layers

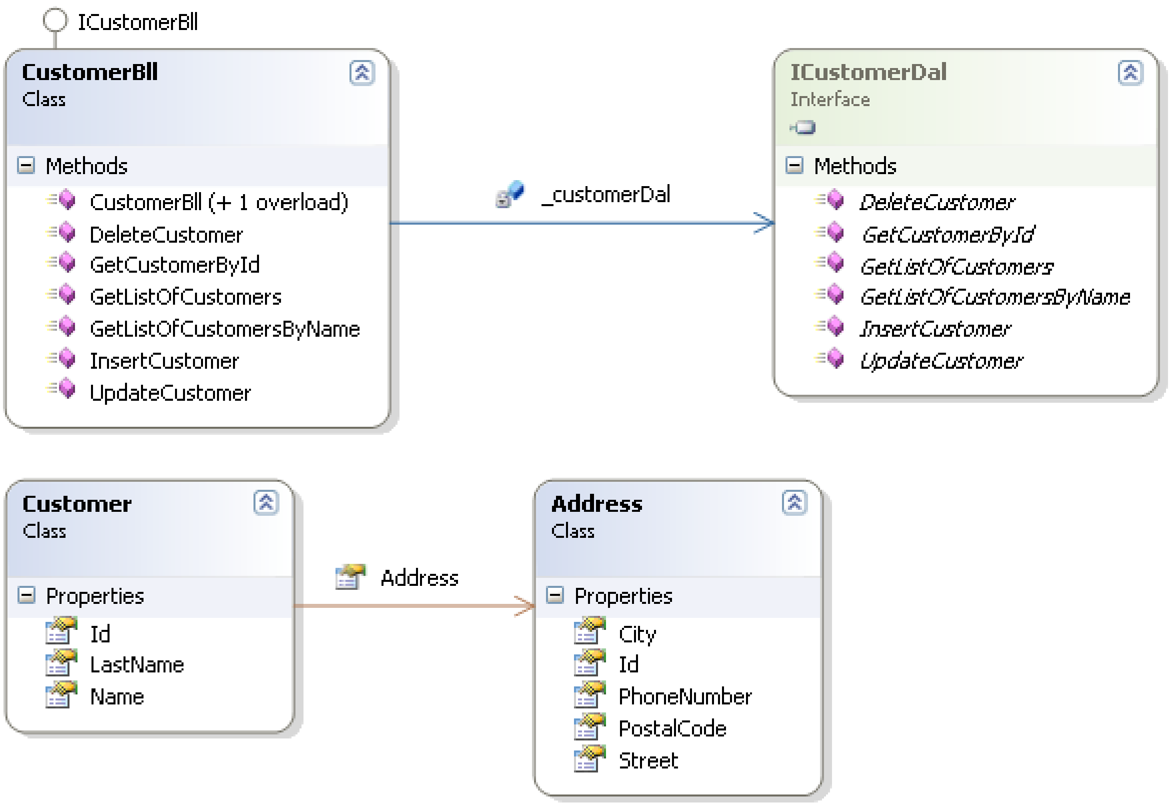


Figure 5.2: Customer registry class diagram

5.4 Customer registry RESTful services

Customer registry RESTful service, developed using Windows Communication Foundation (WCF), resides on top of the customer registry business logic layer and exposes almost all of business logic layer's operations. Interface *ICustomerService* and the class *CustomerService* define the service layer on top of ICustomerBll while the *SerCustomer* and *SerAddress* represent server entities which values are mapped to and from customer registry system entities, Customer and Address.

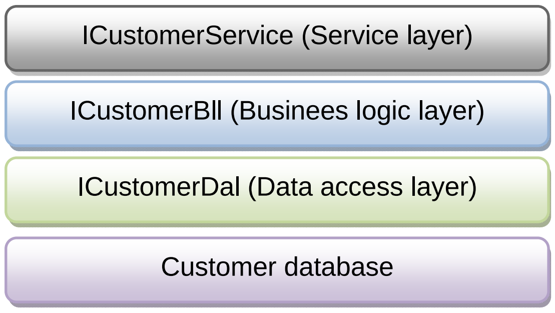

ICustomerService (Service layer)
ICustomerBll (Businees logic layer)
ICustomerDal (Data access layer)
Customer database

Figure 5.3: Customer registry RESTful service layers

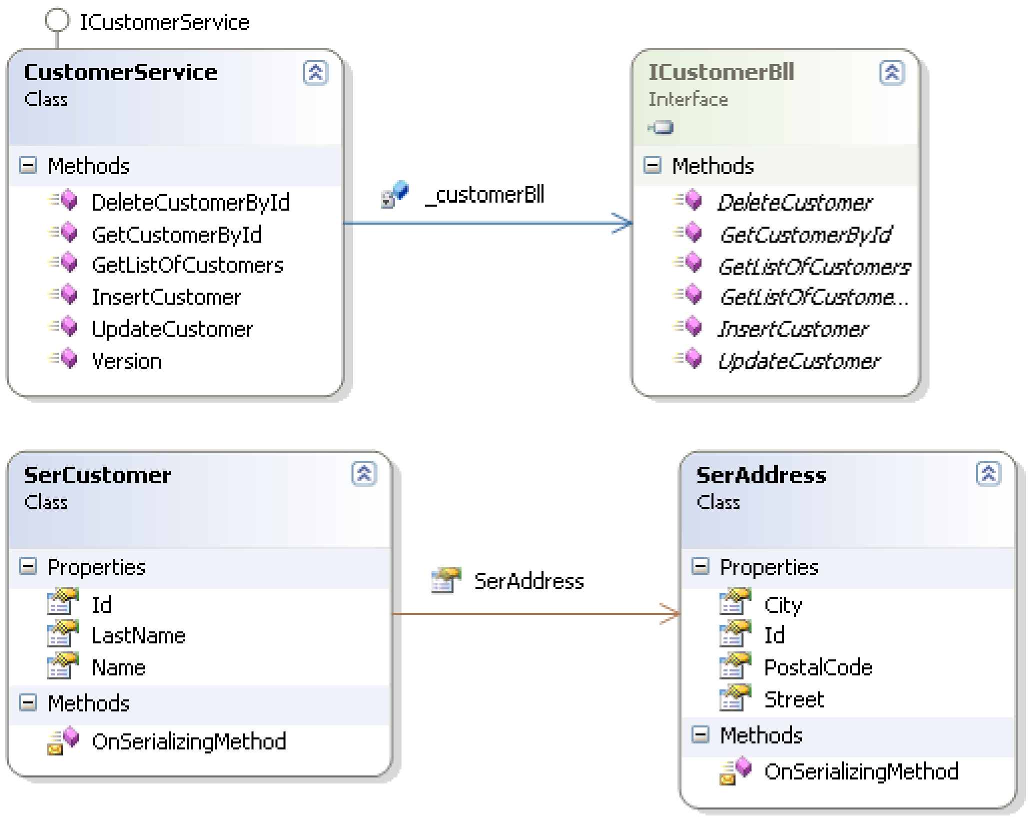


Figure 5.4: Customer registry RESTful service diagram

For a client those interfaces and classes are irrelevant as the only way for it to speak with the service is through the URI and HTTP methods. The binding between ICustomerService and the URI is presented in the following table.

Table 5.2: Customer RESTful service details

.NET operation	URI	Method	Input	Output
DeleteCustomerById	http://localhost:8090/Service/Customers/{id}	DELETE	n/a	n/a
GetCustomerById	http://localhost:8090/Service/Customers/{id}	GET	n/a	customer
GetListOfCustomers	http://localhost:8090/Service/Customers	GET	n/a	customers array
InsertCustomer	http://localhost:8090/Service/Customers	POST	customer	customer
UpdateCustomer	http://localhost:8090/Service/Customers	PUT	customer	customer

From Table 5.2 we see that the service is hosted on the HTTP address http://localhost:8090/Service. *Customers* is the relative URI which is used to invoke customer related operations. Some operations like the ones for insertion and updating require *SerCustomer* serialized object as part of the HTTP body in the request and will also produce an updated SerCustomer object. Other operations like the ones to get a SerCustomer object or a list of objects require no inputs while delete operation neither requires an input nor produces an output. Indeed, deletion operation is in very contrast to the SOAP counterpart which demands input and produces an output thus justifying message-level security. In our case the customer ID is part of the URI which is used to delete a specific customer based on its identification key so there is no reason of sending a serialized object to the operation. One may say that by sending an unencrypted customer ID to the other side, we expose critical data unprotected. The answer is that only an authorized party may send such unprotected requests. Additionally, the customer ID will represent nothing to a sniffing third party since it is the only customer item visible. In order to make the ID more puzzling, it can be randomly created containing a mix of randomized characters which will help even more in concealing the details about the ID itself.

We believe that our service operations cover enough scenarios in order to test the message-level security. Through *insert* and *update* operations we are able to test fully secured message interchange. Operations to *retrieve* customer data are testing secured message exchange from one party while the *delete* operation is a scenario where the message-level security is unnecessary.

5.5 Security library

The new security library is much based on the idea available for the SOAP messages in WCF. The functionality was discussed in section 4.2 where we shortly described how types and its members are supposed to be separately protected by using ProtectionLevel-enumeration. The final protection of the selected types and members happens after serialization, i.e. on the SOAP-message itself. Unfortunately, as already mentioned earlier, many others including ourselves, did not succeed to make it work but we found the idea to be very elegant. Finally, we borrowed the idea and made it work for RESTful services. In fact, since RESTful Services and Web Services share same set of attributes to mark different types and members for serialization, our code should also work on SOAP messages although this was never tested. Following demonstrates a code snippet used to mark *Street* and *PostalCode*-members for encryption and signature, and mark *SerAddress*-type for signature only.

```
namespace Samples.Rest.Model
{
   [ProtectionMember(ProtectionLevel.Sign)]
   [DataContract((Name=”Address”, Namespace = "")]
   public class SerAddress
   {
      [DataMember]
      public int Id { get; set; }
      [DataMember, ProtectionMember(ProtectionLevel.EncryptAndSign)]
      public string Street { get; set; }
      [DataMember, ProtectionMember(ProtectionLevel.EncryptAndSign)]
      public int PostalCode { get; set; }
      [DataMember]
      public string City { get; set; }

   }
}
```

Figure 5.5: Adding protection on an existing entity

In the good spirit of programming we were encouraged to reuse existing functionality whenever possible. The only new code we introduced in Figure 5.5 is the *ProtectionMember*-attribute which is used to mark the types and members for protection. We reused ProtectionLevel-enumeration for specifying type of protection we want to apply on the different parts of the message. If we again look closer at Figure 5.5 we see that we are using ProtectionLevel-enumeration together with *DataContract* and *DataMember*-

attributes while the code in Figure 4.2 is using ProtectionLevel with MessageContract and MessageMember-attributes. The truth is that the latter two attributes are supposed to be used in connection with the serialization of the SOAP messages only while the former ones are generic for RESTful and SOAP messages. That may explain why DataContract and DataMembers-attributes did not have any protection capabilities prior to our code.

The way our new library operates is by intercepting the automatic serialization process to read the names of types and members with DataContract and DataMember-attributes, read their ProtectionMember-attribute and stores the read data in a temporary structure. When the object becomes serialized the data in temporary structure is then translated to a structure containing XPath expressions. XPath expressions are created dynamically and used to navigate to the XML representation of types and members marked for protection. When specified elements are found they will be passed on to the XML Encryption and XML Signature. XPath expressions will also handle DataContract with namespaces and DataContract and DataMember with serialized name aliases.

Now, when the ProtectionLevel is sat on higher level member, what should be default behavior for the lower level members with the same or similar ProtectionLevel? For instance, should we have to re-sign Street and PostalCode-members after signing SerAddress-type? We do not see any logical reason for encrypting or signing an element as long as the parent has the same protection level. Yes, there might be cases where the signature from a party should be applied on elements already signed by another party, but to keep it simple, our solution does not support multiple signatures. Neither does it support superencryption. To be able to support our preferred logic we created a functionality that always checks root type’s ProtectionLevel and then its child members for the same value. Looking at the Figure 5.5 it means that when the code discovers *ProtectionLevel.Sign* on the *SerAddress* it will search for the same or similar value on its members and discover *ProtectionLevel.EncryptAndSign* on *Street* and *PostalCode*-elements. Then it will create a protection plan based on the findings which will result in Street and PostalCode being encrypted and then SerAddress signed. In such manner we will avoid creating extra computational and data overhead as Street and PostalCode will not be separately signed. This logic will also apply on complex hierarchical types with multiple levels of non-primitive members and collection structures.

The process of unprotecting messages or their elements start when the message is received. Our code will intercept the deserialization process, validate the signature, find all encrypted elements and then try to decrypt those. Once the message is in clear text the process of deserialization will continue.

5.6 Compressing protected messages

According to our design, when the message is protected by the new security library, the next step in the process is to decrease its size by grouping and removing repeating elements. When the message is received by the other party the compressed messages needs to be restored into its original form. Before we can start describing the compression process we need to understand the XML code produced by both the XML Signature and the XML Encryption. For the introduction on XML Signature and XML Encryption please refer to sections 3.5.1 and 3.5.2.

When XML Encryption encrypts each element it will replace it with *EncryptedData*-element which will contain the ciphertext and other metadata information that describe the way encryption was implemented. The metadata information may vary depending on the several properties such as the encryption token. In our case, since we are using X.509 certificate as the encryption token, data related to the certificate is generated as well as metadata related to the asymmetric algorithm, session key used for data encryption and session key algorithm. As a result of the one certificate constraint that we imposed, we realize that the certificate metadata will be redundant in each encrypted element. Additionally, since there is a little point of specifying different types of asymmetric and symmetric algorithms for every element, we found the algorithm metadata to be unnecessary repeatable. That leaves us with the only one metadata that is truly unique per each element and that is the symmetric key used for the encryption. While we are aware that one session key could be used for the encryption of all elements, it would require certain custom coding. The combination of X.509 certificate and custom generated session key would make our solution less standardized so that other platforms would have difficulties communicating with the service. Furthermore, XML Encryption would still generate all the metadata mentioned above even if we did use only one session key.

When repeatable and unique elements are identified, they will be grouped accordingly. The repeatable elements will then be deleted until there are no duplicates left. Similar logic exists for the XML Signature as well. The compressing and decompressing work is done by XSLT. Per today there exist four XSLT transformations and they are all JSON XML specific. The first two transforms encrypted and signed JSON XML to compressed JSON XML while the last two reverse the process. The same transformations do not exist for the plain XML messages because of the limited development period but the transformations for the plain XML should be easier than those for JSON XML because JSON XML have to follow a certain structure in order to be processed correctly by the serialization procedure. Figure 5.6 shows a segment from a JSON XML document where we see encrypted PostalCode and Street-elements before compression. Figure 5.7 shows the compressed version of the same segment.

```
<PostalCode type="number">
        <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#">
              <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
              <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
                <X509Data>
                  <X509Certificate>MIIDzTCCAr…</X509Certificate>
                </X509Data>
              </KeyInfo>
              <CipherData>
                <CipherValue>WAzSJM6E…</CipherValue>
              </CipherData>
            </EncryptedKey>
          </KeyInfo>
          <CipherData>
            <CipherValue>l3ipRPA…CipherValue>
          </CipherData>
        </EncryptedData>
</PostalCode>
<Street>
        <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#">
              <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
              <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
                <X509Data>
                  <X509Certificate>MIIDzTCCAr…</X509Certificate>
                </X509Data>
              </KeyInfo>
              <CipherData>
                <CipherValue>Pk8Qd5jukK…CipherValue>
              </CipherData>
            </EncryptedKey>
          </KeyInfo>
          <CipherData>
            <CipherValue>gOcrtwamn…CipherValue>
          </CipherData>
        </EncryptedData>
</Street>
```

Figure 5.6: Encrypted PostalCode and Street-elements

```
<PostalCode type="object">
        <EncryptedData type="object">
          <JsonType>number</JsonType>
          <DataCipherValue>dPVPQqvzsdvlaJJ+… </DataCipherValue>
          <KeyCipherValue>e4rAm/…</KeyCipherValue>
        </EncryptedData>
 </PostalCode>
<Street type="object">
        <EncryptedData type="object">
          <JsonType>string</JsonType>
          <DataCipherValue>rngHVBWig…</DataCipherValue>
          <KeyCipherValue>XbeWZLp3AI…</ KeyCipherValue>
        </EncryptedData>
      </Street>
</SerAddress>
<EncryptedDataRef type="object">
      <X509Certificate> MIIDzTCCAr…</X509Certificate>
      <DataEncrytionMethod>http://www.w3.org/2001/04/xmlenc#aes256-cbc</DataEncrytionMethod>
      <KeyEncrytionMethod>http://www.w3.org/2001/04/xmlenc#rsa-1_5</KeyEncrytionMethod>
</EncryptedDataRef>
```

Figure 5.7: Encrypted and compressed PostalCode and Street-elements

The implemented process of protecting and unprotecting a message is summarized by the following activity diagram.

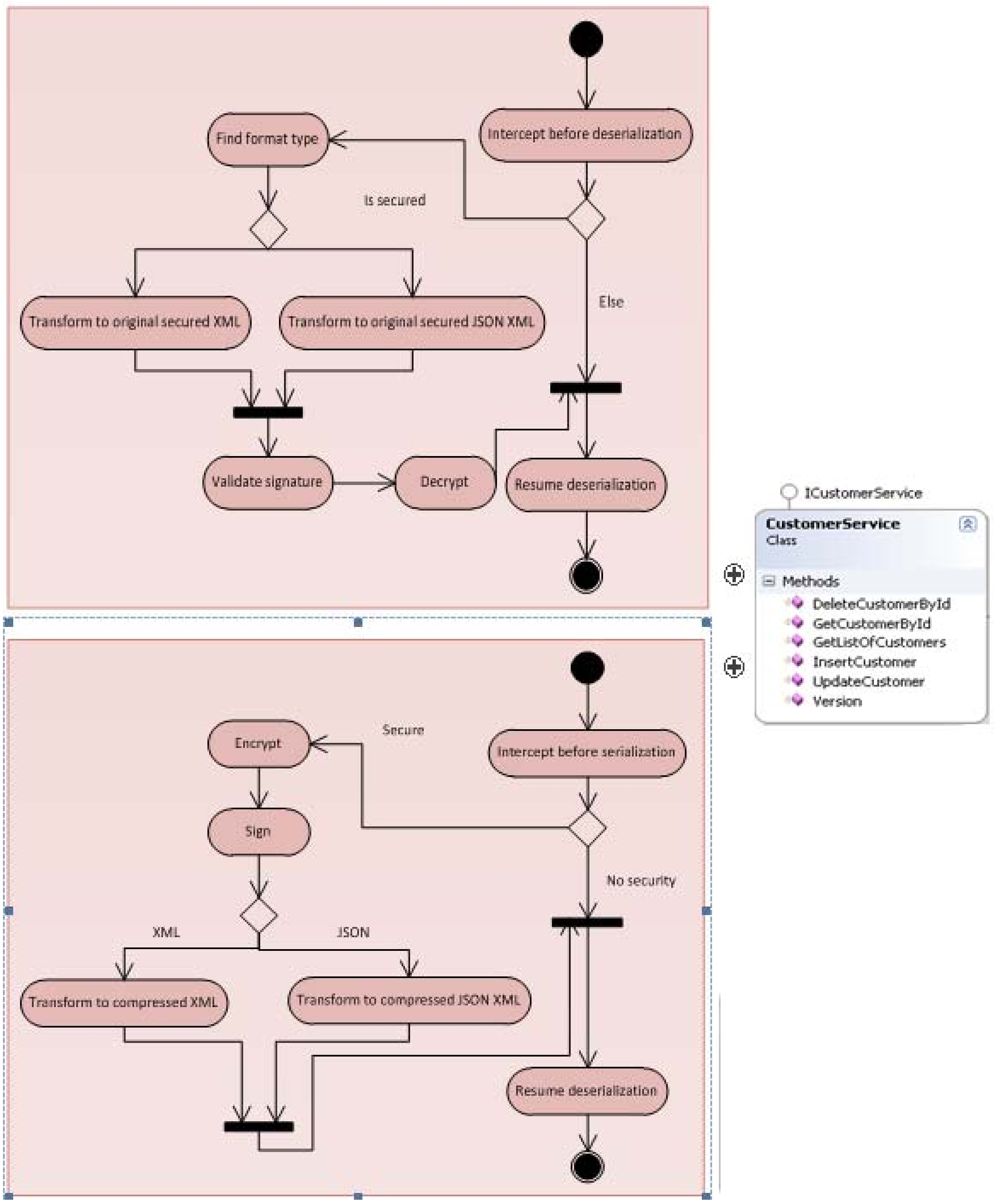


Figure 5.8: Customer registry RESTful service activity diagram

5.6.1 Original versus compressed

The difference in size between the original and compressed message will get even more drastic as the amount of encrypted elements increases. *EncryptedDataRef*-element in the compressed version of the message will always contain only *three metadata sub-elements per message*, unconstrained by the amount of the encrypted elements. Each encrypted element will be replaced by EncryptedData-element with *three metadata sub-elements*. Compared to the EncryptedData-element of the original, uncompressed message, the original one will always contain *eleven sub-elements per encrypted element*. Table 5.3 shows the difference in amount of metadata elements generated in the original and the compressed message when amount of encrypted elements varies.

Table 5.3: Amount of metadata elements per amount of encrypted elements

Amount of encrypted elements	Original message (EncryptedData & 11 sub-elements per encrypted element)	Compressed message (EncryptedData & 3 sub-elements per encrypted element + EncryptedDataRef & 3 sub-elements per message)
1	12	8
3	36	16
5	60	24
7	84	32
9	108	40

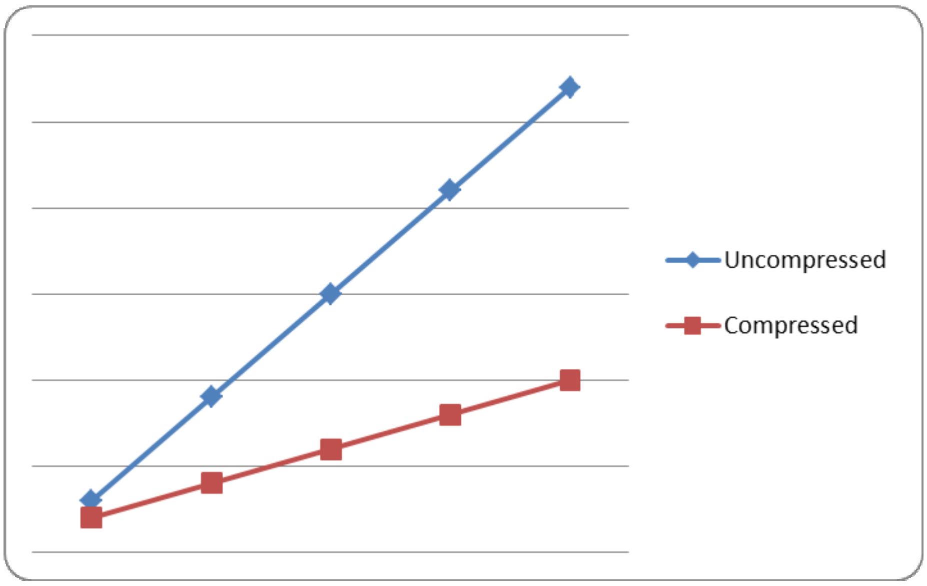


Figure 5.9: Data from Table 5.3

When it comes to the file size, the difference between uncompressed and compressed messages is even bigger. The big difference is caused by XML attributes which are present in the redundant elements, in the original message. Another reason is redundant elements values. Following table and figure demonstrates this difference.

Table 5.4: File size where each customer has Street and PostalCode encrypted and signed

Amount of customers (Street and PostalCode-elements are encrypted and signed)	Uncompressed	Compressed
1	12 KB	5 KB
2	21 KB	6 KB
3	31 KB	8 KB
4	40 KB	9 KB
5	50 KB	10 KB

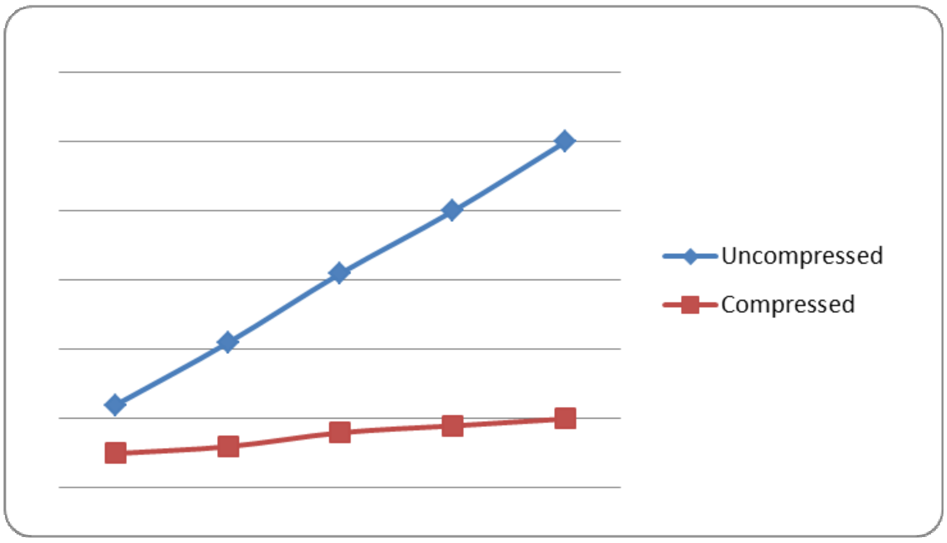


Figure 5.10: Data from Table 5.4

Looking at Figure 5.9 and Figure 5.10 we can confirm that compressed messages will be even more compact than the uncompressed ones when the amount of protected elements increases. According to Figure 5.9, while a compressed message with one encrypted element contained 33,3% less metadata, the percentage was 63% less with nine encrypted elements. When it comes to the message size, a compressed message containing a single customer with two encrypted and signed elements was 58% smaller than the uncompressed message while a compressed message with five customers was 1/5 of the size compared to the original. In addition to being much smaller we also believe that the compressed messages are much more human readable then their original counterparts.

Nevertheless, in fairness to the uncompressed message we should mention that our solution implements XML Signature with the *Object*-element instead of *Reference*-element. The big difference between those two elements is in the way they store information about signed objects. While the Object-element will contain a copy of the signed object, Reference-element will contain only the reference to it. Since the new library eliminates duplicate metadata, it will also remove copied object in the Object-element and create a reference to the signed element. Using built-in reference functionality would be a better solution because original uncompressed messages would be smaller and we would avoid creating and recreating signed elements as the situation is today.

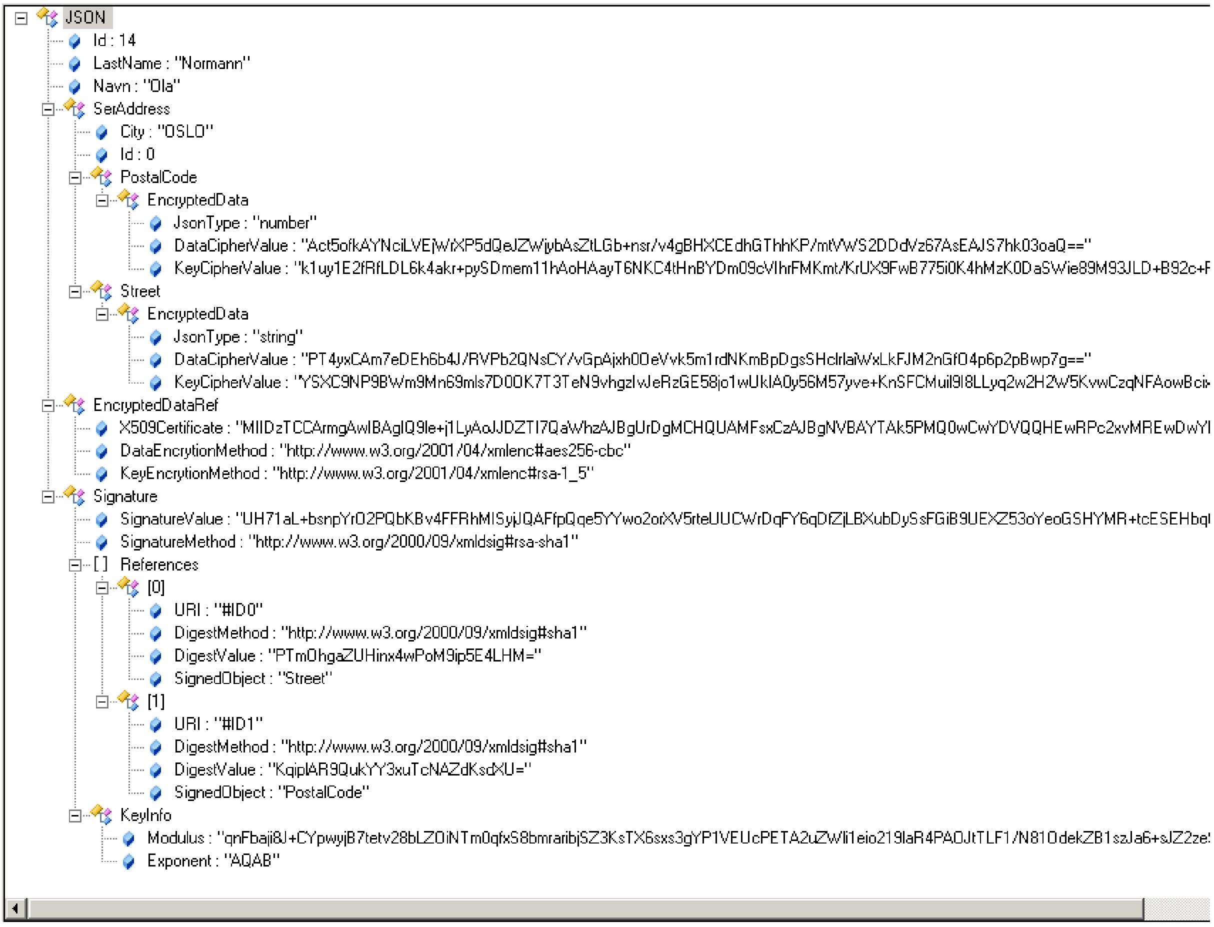

```
JSON
  Id : 14
  LastName : "Normann"
  Navn : "Ola"
  SerAddress
    City : "OSLO"
    Id : 0
    PostalCode
      EncryptedData
        JsonType : "number"
        DataCipherValue : "Act5ofkAYNciLVEjWrXP5dQeJZWjybAsZtLGb+nsr/v4gBHXCEdhGThhKP/mtVWS2DDdVz67AsEAJS7hk03oaQ=="
        KeyCipherValue : "k1uy1E2fRfLDL6k4akr+pySDmem11hAoHAayT6NKC4tHnBYDm09cVIhrFMKmt/KrUX9FwB775i0K4hMzK0DaSWie89M93JLD+B92c+
    Street
      EncryptedData
        JsonType : "string"
        DataCipherValue : "PT4yxCAm7eDEh6b4J/RVPb2QNsCY/vGpAjxh0OeVvk5m1rdNKmBpDgsSHclrlaiWxLkFJM2nGf04p6p2pBwp7g=="
        KeyCipherValue : "YSXC9NP9BWm9Mn69mls7D0OK7T3TeN9vhgzIvJeRzGE58jo1wUkIA0y56M57yve+KnSFCMuil9l8LLyq2w2H2W5KvwCzqNFAowBci
  EncryptedDataRef
    X509Certificate : "MIIDzTCCArmgAwIBAgIQ9le+j1LyAoJJDZTI7QaWhzAJBgUrDgMCHQUAMFsxCzAJBgNVBAYTAk5PMQ0wCwYDVQQHEwRPc2xvMREwDwY
    DataEncryptionMethod : "http://www.w3.org/2001/04/xmlenc#aes256-cbc"
    KeyEncrytionMethod : "http://www.w3.org/2001/04/xmlenc#rsa-1_5"
  Signature
    SignatureValue : "UH71aL+bsnpYr02PQbKBv4FFRhMISyjJQAFfpQqe5YYwo2orXV5rteUUCWrDqFY6qDfZjLBXubDySsFGiB9UEXZ53oYeoGSHYMR+tcESEHbq
    SignatureMethod : "http://www.w3.org/2000/09/xmldsig#rsa-sha1"
    [] References
      [0]
        URI : "#ID0"
        DigestMethod : "http://www.w3.org/2000/09/xmldsig#sha1"
        DigestValue : "PTm0hgaZUHinx4wPoM9ip5E4LHM="
        SignedObject : "Street"
      [1]
        URI : "#ID1"
        DigestMethod : "http://www.w3.org/2000/09/xmldsig#sha1"
        DigestValue : "KqiplAR9QukYY3xuTcNAZdKsdXU="
        SignedObject : "PostalCode"
    KeyInfo
      Modulus : "qnFbaji8J+CYpwyjB7tetv28bLZ0iNTm0qfxS8bmraribjSZ3KsTX6sxs3gYP1VEUcPETA2uZWli1eio219laR4PA0JtTLF1/N810dekZB1szJa6+sJZ2ze
      Exponent : "AQAB"
```

Figure 5.11: A compressed JSON message with encrypted and signed Street and PostalCode-elements, shown in JSON Viewer

5.7 Summary

This chapter gave an explanation of the work done regarding the implementation of the design. We showed how case scenario and the RESTful service were created and relation between the two. Further on, we demonstrated code snippets from the security library and gave an explanation on how library works. The chapter ended by explaining how message compression and decompression were implemented in order to provide light and human readable protected messages.

6 Evaluation

The final question is, did we fulfill predictions as stated in section 2.3 and what is the result of the hypothesis? Let us review the predictions and give a short explanation for each of them.

P1: The new artifact, message protection library, will enable encryption and digital signatures on XML and JSON messages.

Yes, the security library does support encryption and digital signature for both formats. We consider this prediction to be fulfilled.

P2: The new artifact will support partial encryption and partial digital signature.

Yes, the message can be fully and partially encrypted and signed, and does support different combination schemes like encrypting a single element and then sign the complete message. We consider this prediction to be fulfilled.

P3: The new artifact will enable message compression and decompression on protected messages so to decrease message size.

When XML Encryption and XML Signature are utilized messages become significantly large. Yet, by compressing the message it becomes smaller in size and easier to read. We succeed in compressing the temporary JSON XML format which reflects JSON format itself. Plain XML were not compressed due to the lack of development time. That is why we consider this prediction to be partially fulfilled.

P4: The new security solution will be easily adoptable by existing and standard WCF RESTful services.

We strongly believe that the security code can be added on the existing and standard WCF projects without having to rewrite the code. This is what we did in our case study where we enabled security <u>after</u> the RESTful service was fully developed. We consider this prediction to be fulfilled.

Through the evaluation of the predictions we are now able to conclude the hypothesis H1. Three out of four predictions were fulfilled while P3 was partially fulfilled. Yet, we have stated that the task regarding compression of the plain XMLs was not finished due to the lack of development time, i.e. task should be solved if we had more time. As clearly stated by the research methodology, “Technology research does not always produce artifacts that are complete, regarded from a user’s point of view.”[98](p.8). It continues with “If the prototype looks promising, it can later on be elaborated to a complete, saleable product. Such finalization is typically done by other people than researchers.”[98](p.8). Therefore, we conclude P3 to be accepted as fulfilled which results to the conclusion that H1 is true.

Hence we state that: ***Solution proposed does enable message-level security for RESTful services on .NET platform.***

7 Conclusion and further work

7.1 Conclusion

The purpose of this thesis was to present a complete solution for the message-level security for RESTful services. The solution we presented is primarily consisting of the newly added security library but we also recommended authentication mechanism and demonstrated a non-technical approach for token distribution which we consider appropriate because of its simplicity. The solution was developed because no other similar solutions seem to exist. The new security library and our recommendations were tested with a case scenario although the solution is intended to be generic and not tailored for any specific application. Through the design and implementation process we focused strongly on the interoperability and multi-platform adoption. That is why the security library is based on well-known standards, XML Encryption and XML Signature, while the message compression and decompression is based on transformations developed in XSLT.

7.2 Contribution

This thesis has successfully demonstrated a new security mechanism for RESTful services. Through this work following has been achieved:

- Designed and implemented probably the very first prototype for message-level security for RESTful services.
- Identified reusable security components and still kept protected messages small and readable.
- Message protection with platform-independent components which enables easier adoption by solutions developed in other development platforms.

- Proposed a feasible certificate distribution procedure.

7.3 Further work

As many other developers, we would like our proposed solution to be considered useful to others. If the solution seems to be useful then it might be actually used in other scenario and provide better tests cases then we have today. Perhaps a group of people may be formed to maintain the code by fixing the bugs and consider possible extensions to that code. Someone will not consider the complete solution to be useful but may consider parts of it to be relevant for their requirements. At last, if the solution seems inappropriate we hope that someone may be inspired by the ideas behind this work and start on their own projects.

The following sections describe certain fields which could require further investigation and development.

7.3.1 Message compression

Although we did a lot of work protecting the message content and working on the compressing process, we did not complete transformation of protected XML messages into their compressed variant. There is not much work left on finishing this task but then we could ask ourselves if those transformations could be further optimized. The answer to that question is probably yes, even though we spent days on figuring out how to make the XSLT code more efficient. Optimization is a process that takes a lot of time and testing before one can conclude to have a solid performing piece of code.

7.3.2 Caching

Caching, if is used correctly, will highly improve performance for HTTP resources including RESTful services. In our work we did not concentrate on this area because it can be very challenging to enable both cache and message-level security together since there are many places on the Internet that might cache old response messages. In our opinion caching is an important feature that should be studied further so as to find a way for it to co-exist with protected messages without side effects.

7.3.3 Authentication

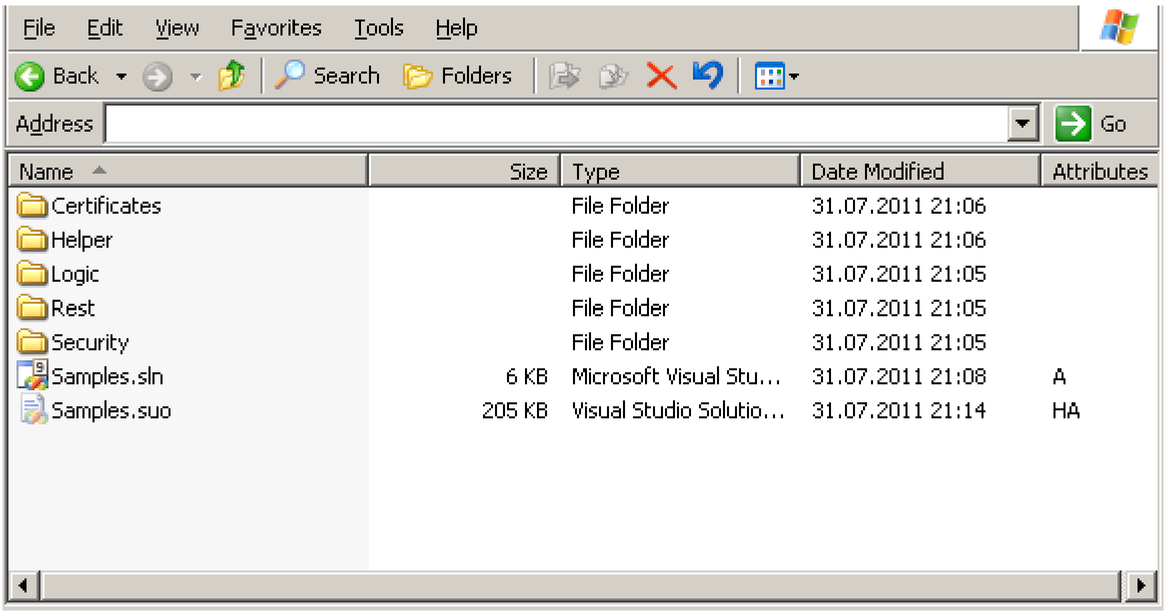
In this thesis we recommended a solution for authentication that is based on the existing HTTP authentication mechanism. However, we feel that authentication could be solved in a more secure and modern way by utilizing OAuth which we shortly mentioned in section 4.3.2. The main reason why we feel OAuth to be more appropriate than HTTP Digest is that through OAuth a user sends tokens instead of credentials. Those tokens will then be validated by an external service and the service will make tokens last for a predefined time period only[100]. HTTP Digest hashed passwords on the other hand, could be intercepted and used in offline password guessing attacks.

7.3.4 Certificate distribution

Our solution for certificate distribution is intended for a smaller group of people. It is a relatively secure solution but a manual one. In the world of certificates there is nothing wrong with that. There are many other cases where manual certificate distribution is required, for instance, European Association for the Streamlining of Energy Exchange offers certificates of all of its members by downloading them from their member site[96]. Yet, we feel that an “automatic” solution could be more appropriate. XKMS, as described in section 3.5.3, is such a solution but it is very complex for REST architecture and it highly depends on Web Services for registration and distribution of keys or certificates. A similar solution for RESTful services could be designed according to REST principles and strive to be simple.

Appendix A DVD content

This chapter describes the contents of the DVD appended to this thesis.
The DVD consists of a folder *Customer registry system* which contains folders as shown in Figure A1.



Name	Size	Type	Date Modified	Attributes
Certificates		File Folder	31.07.2011 21:06	
Helper		File Folder	31.07.2011 21:06	
Logic		File Folder	31.07.2011 21:05	
Rest		File Folder	31.07.2011 21:05	
Security		File Folder	31.07.2011 21:05	
Samples.sln	6 KB	Microsoft Visual Stu...	31.07.2011 21:08	A
Samples.suo	205 KB	Visual Studio Solutio...	31.07.2011 21:14	HA

Figure A1: Customer registry system folder structure

The folder contains a .NET solution called *Samples.sln*. This .NET solution includes all the .NET projects that are inside Customer registry system-folder. The solution can be used directly to run the case study scenario application. Following shows the .NET solution and its projects when opened in Visual Studio 2008.

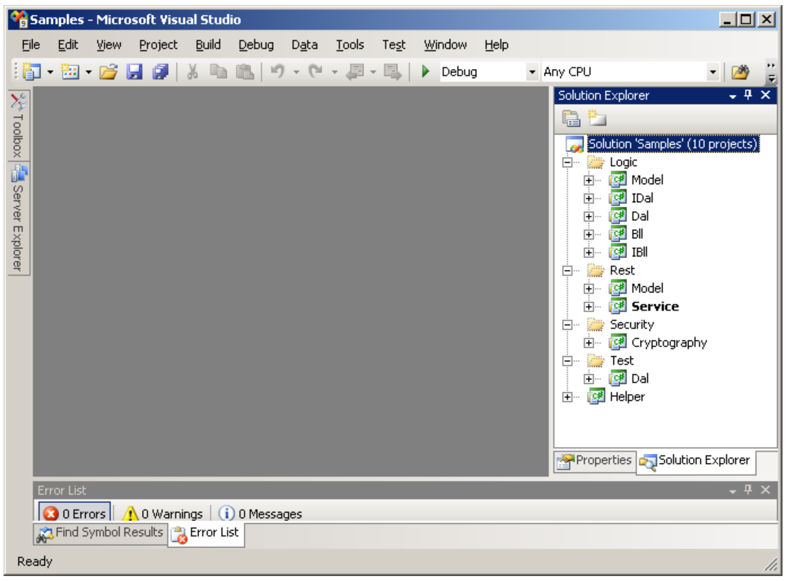


Figure A2: Samples.sln

Description of the subfolders

Certificates-folder

Contains a *test certificate* that was used to test the solution. May be further used for testing purposes. Must be imported to the certificate store on desired test machine. Importing process depends on the operating system.

Helper-folder

Contains a *Helper*-project and three helper-classes related to XML handling. Helper-project may be used by all projects.

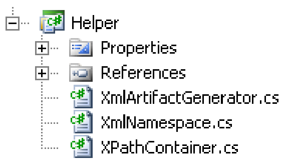
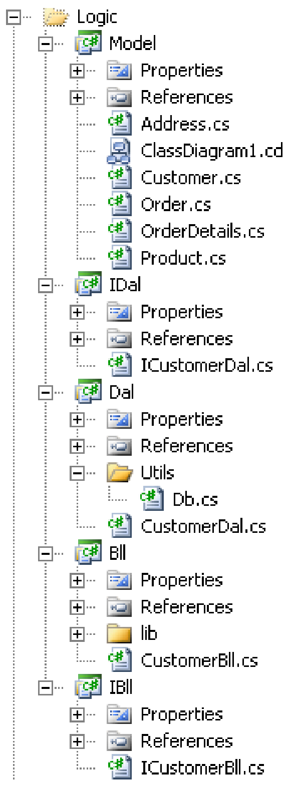


Figure A3: Helper-project

Logic-folder

Logic-folder represents Customer registry standalone system. It contains five projects which contains an entity project (*Model*), data access layer projects (*IDal* and *Dal*) and business logic layer projects (*IBll* and *Bll*).

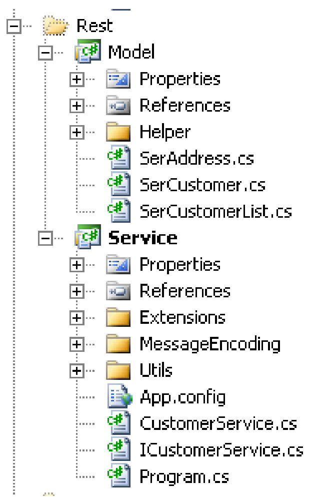


```
Logic
  Model
    Properties
    References
    Address.cs
    ClassDiagram1.cd
    Customer.cs
    Order.cs
    OrderDetails.cs
    Product.cs
  IDal
    Properties
    References
    ICustomerDal.cs
  Dal
    Properties
    References
    Utils
      Db.cs
    CustomerDal.cs
  Bll
    Properties
    References
    lib
    CustomerBll.cs
  IBll
    Properties
    References
    ICustomerBll.cs
```

Figure A4: Customer registry standalone system project structure

Rest-folder

Contains two projects representing the Customer registry RESTful service. RESTful service is dependent on business logic layer in the Logic-folder, as described in section 5.4. The project *Model* contains server entities and project *Service* contains the service logic. There exist references to IBll, Bll and Model-projects from the Service-project.

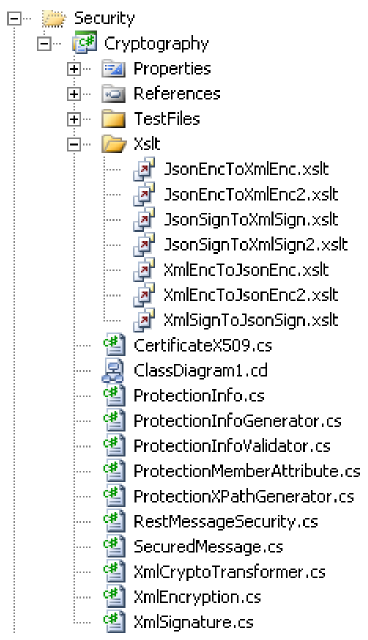


```
Rest
    Model
        Properties
        References
        Helper
        SerAddress.cs
        SerCustomer.cs
        SerCustomerList.cs
    Service
        Properties
        References
        Extensions
        MessageEncoding
        Utils
        App.config
        CustomerService.cs
        ICustomerService.cs
        Program.cs
```

Figure A5: RESTful service project structure

Security-folder

Security folder contains a .NET project called *Cryptography* which includes all classes and XSLT-files to enable message-level security upon a Message-object. Message-type is a WCF specific type containing all the context and content related to SOAP, plain XML, JSON or other messages. This project is referred to as the new security library or the new artefact.



```
Security
    Cryptography
        Properties
        References
        TestFiles
        Xslt
            JsonEncToXmlEnc.xslt
            JsonEncToXmlEnc2.xslt
            JsonSignToXmlSign.xslt
            JsonSignToXmlSign2.xslt
            XmlEncToJsonEnc.xslt
            XmlEncToJsonEnc2.xslt
            XmlSignToJsonSign.xslt
        CertificateX509.cs
        ClassDiagram1.cd
        ProtectionInfo.cs
        ProtectionInfoGenerator.cs
        ProtectionInfoValidator.cs
        ProtectionMemberAttribute.cs
        ProtectionXPathGenerator.cs
        RestMessageSecurity.cs
        SecuredMessage.cs
        XmlCryptoTransformer.cs
        XmlEncryption.cs
        XmlSignature.cs
```

Figure A6: The new security library project structure

Test-folder

Contains a .NET-project *Dal* which contains a single class, *CustomerDalTest.cs*, that will recreate the tables in the database (but it will not create database) and populate the data in the tables. The class will also perform various data logic layer tests as such as to retrieve customers from the database.

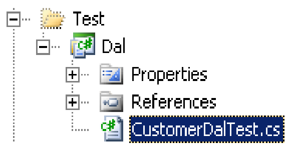


Figure A7: Test project

Appendix B Sample code explanation

In order to implement the new security library on the existing project we may study the code in Customer registry RESTful service, Service-project. The process of securing the existing code can be summarized in following way:

A. Set Protection level on entities

Set desired protection level on your entity types or/and their members and create OnSerializingMethod in exactly same way as shown in the following figure.

```
[DataContract(Namespace = "")]
   public class SerAddress
   {
      [DataMember]
      public int Id { get; set; }
      [DataMember, ProtectionMember(ProtectionLevel.EncryptAndSign)]
      public string Street { get; set; }
      [DataMember, ProtectionMember(ProtectionLevel.EncryptAndSign)]
      public int PostalCode { get; set; }
      [DataMember]
      public string City { get; set; }


      [OnSerializing()]
      internal void OnSerializingMethod(StreamingContext context)
      {
         IEnumerable<ProtectionInfo> listOfSecuredElements = new
         ProtectionInfoGenerator(this.GetType()).CreateListOfProtectionMembers();

         foreach (ProtectionInfo securedElement in  listOfSecuredElements)
         {
           OperationContext.Current.OutgoingMessageProperties
          .Add(AutoId.GetNext().ToString(), securedElement);
         }
      }
   }
```

Figure A8: Type with different protection levels

OnSerializingMethod-method is used to remember the protection level state of the type and its members. It is a static piece of code and must be added to all types where protection is requested.

B. Implement the IDispatchMessageInspector and call the security library

IDispatchMessageInspectore is a WCF interface used to inspect the messages before they are serialized or deserialized on the server side. When the interface is implemented, call the new security library from the interface methods *BeforeSendReply* or *AfterReceiveRequest.* Following demonstrates the code from the BeforeSendReply-method where types will be protected before being sent.

```
public void BeforeSendReply(ref Message reply, object correlationState)
{
    try
    {
       MessageProperties messageProperties = OperationContext.Current.OutgoingMessageProperties;
       string contentType = WebOperationContext.Current.IncomingRequest.ContentType;
       CertificateX509 certX509 = new CertificateX509("CN=My Root CA, O=Organization, OU=Org
                                                   Unit, L=Oslo, C=NO");
       RestMessageSecurity restSecurity = new RestMessageSecurity(reply, contentType);
       restSecurity.ContentOnly = true;
       reply = restSecurity.GetSecuredMessage(messageProperties, certX509);
    }
    catch (Exception ex)
    {
       OutgoingWebResponseContext response = WebOperationContext.Current.OutgoingResponse;
       response.StatusCode = HttpStatusCode.BadRequest; // or anything you want
       response.StatusDescription = ex.Message;
       reply = null;
    }
}
```

Figure A9: Calling the new security library

The process in Figure A9 starts by collecting all the types that are marked for protection (messageProperties). contentType is a string variable that describes which format should the type be serialized to. CertificateX509 is a custom class and part of the new security library. It is used for validation and retrieving of the certificate from the certificate store. RestMessageSecurity is also part of the new library and is used to protect and unprotect messages. In addition, it is also responsible to compress and decompress the messages. Finally, GetSecuredMessage is a method responsible to trigger the protection and compression process, and return the message in XML format. This concludes the protection process and after this stage the message is in control of the WCF serialization. When serialization is done WCF will send the message to its requestor.

Bibliography

[1] Richardson, Ruby. *RESTful Web Services*. O'Reilly Media, 1st Edition, 15 May 2007.

[2] Peng, Li, Huo. An Extended UsernameToken-based Approach for REST-style Web Service Security Authentication. In *2nd IEEE International Conference on Computer Science and Information Technology*. pages 582-586, Beijing, China, 2009.

[3] Jon Flanders. *RESTful .NET*, O'Reilly Media, 1st Edition, 28 November 2008.

[4] The Internet Engineering Task Force. *HTTP Authentication: Basic and Digest Access Authentication*, June 1999 [online]. Available: http://tools.ietf.org/html/rfc2617 [cited 30 September 2010].

[5] Hollar, Murphy. *Enterprise Web Services Security*. Charles River Media, 1st Edition, 27 September 2005.

[6] Matthew Herper. *The World's Most Expensive Drugs*, 22 February 2010 [online]. Available: http://www.forbes.com/2010/02/19/expensive-drugs-cost-business-healthcare-rare-diseases_2.html [cited 09 October 2010].

[7] Kostaszotos, Litke. *Cryptography and Encryption*, Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece, 2005.

[8] Per Oscarson. *Actual and Perceived Information Systems Security*. PhD dissertation, Linköping University, Department of Management and Engineering, Linköping, Sweden, 2007.

[9] Ron Lee. *Performance Analysis: Isolating the Real Bottleneck in a System*, 01 January 1996 [online]. Available: http://support.novell.com/techcenter/articles/ana19960104.html [cited 09 December 2010].

[10] Furht, Socek. A Survey of Multimedia Security. In *Comprehensive Report on Information Security*, International Engineering Consortium, Chicago, USA, 2004.

[11] Berbecaru. On Measuring SSL-based Secure Data Transfer with Handheld Devices. In *Wireless Communication Systems, 2005. 2nd International Symposium*, Digital Object Identifier 10.1109/ISWCS.2005.1547731, pages 409 – 413, 05 December 2005.

[12] Alshamsi, Saito. Technical Comparison of IPSec and SSL. In *Proceedings of the 19th International Conference on Advanced Information Networking and Applications - Volume 2*. IEEE Computer Society, pages 395 – 398, Washington DC, USA, 2005.

[13] The Internet Engineering Task Force. *Hypertext Transfer Protocol -- HTTP/1.0*, May 1996 [online]. Available: http://tools.ietf.org/html/rfc1945 [cited 15 December 2010].

[14] Tang, Chen, Levy, Zic, Yan. A Performance Evaluation of Web Services Security. In *Enterprise Distributed Object Computing Conference, EDOC '06*. Digital Object Identifier 10.1109/EDOC.2006.12, pages 67 – 74, 2006.

[15] Portman, Seneviratne. Selective Security for TLS. In *ICON '01 Proceedings of the 9th IEEE International Conference on Network.* IEEE Computer Society, pages 216 - 221, Washington DC, USA, October 2001.

[16] Microsoft TechNet. *FIPS-140-1 Security and FORTEZZA Crypto Cards* [online]. Available: http://technet.microsoft.com/en-us/library/cc962054.aspx [cited 19 December 2010].

[17] The Internet Engineering Task Force. *The Transport Layer Security (TLS) Protocol Version 1.2*, August 2008 [online]. Available: http://tools.ietf.org/html/rfc5246 [cited 06 January 2011].

[18] Dhillon, Randhawa, Wang, Lamont. Implementing a Fully Distributed Certificate Authority in an OLSR MANET. In *Wireless Communications and Networking Conference*, IEEE volume 2, Digital Object Identifier 10.1109/WCNC.2004.1311268, pages 682 – 688, 2004.

[19] Nils Agne Nordbotten. *XML and Web Services Security*. FFI-rapport 2008/00413, Norwegian Defence Research Establishment (FFI), 18 February 2008.

[20] The Internet Engineering Task Force. *The Atom Syndication Format*, December 2005 [online]. Available: http://www.ietf.org/rfc/rfc4287.txt [cited 07 January 2011]

[21] Jøsang, Povey, Ho. What You See is Not Always What You Sign. In *Proceedings of AUUG2002*, page 25-35, Melbourne, Australia, September 2002.

[22] W3C. *XML Signature Syntax and Processing (Second Edition)*, 20 June 2008 [online]. Available: http://www.w3.org/TR/xmldsig-core/ [cited 09 January 2011].

[23] W3C. *XML Encryption Syntax and Processing*, 10 December 2002 [online]. Available: http://www.w3.org/TR/xmlenc-core/ [cited 09 January 2011].

[24] SAML XML.org. *Differences between SAML 2.0 and 1.1*, 23 January 2008 [online]. Available: http://saml.xml.org/differences-between-saml-2-0-and-1-1 [cited 20 January 2011].

[25] Bertino, Martino, Paci, Squicciarini. *Security for Web Services and Service-Oriented Architecture.* Springer; 1st Edition, 10 November 2009.

[26] W3C. *XML Key Management Specification (XKMS 2.0)*, 28 June 2005 [online]. Available: http://www.w3.org/TR/2005/REC-xkms2-20050628/ [cited 22 January 2011].

[27] OASIS. *Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0*, 15 March 2005 [online]. Available: http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf [cited 24 January 2011].

[28] Christine Martz. *PEP - Policy Enforcement Point*, [online]. Available: http://www.birds-eye.net/definition/p/pep-policy_enforcement_point.shtml [cited 24 January 2011].

[29] OASIS. *A Brief Introduction to XACML*, 14 March 2003 [online]. Available: http://www.oasis-open.org/committees/download.php/2713/Brief_Introduction_to_XACML.html#xacml-example [cited 25 January 2011].

[30] ContentGuard. *XrML 2.0 Technical Overview*, 08 March 2002 [online]. Available: http://www.xrml.org/Reference/XrMLTechnicalOverviewV1.pdf [cited 26 January 2011].

[31] OASIS. *Web Services Federation Language (WS-Federation) Version 1.2*, 22 May 2009 [online]. Available: http://docs.oasis-open.org/wsfed/federation/v1.2/ws-federation.doc [cited 28 January 2011].

[32] OASIS. *Members Approve WS-SecureConversation and WS-Trust as OASIS Standards*, 27 March 2007 [online]. Available: http://lists.oasis-open.org/archives/announce/200703/msg00004.html
[cited 28 January 2011].

[33] OASIS. *WS-Trust 1.4*, 02 February 2009 [online]. Available: http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html [cited 30 January 2011].

[34] Vittorio Bertocci. *WS-Trust - Under the Hood*, 04 October 2006 [video]. Available: http://channel9.msdn.com/Shows/Going+Deep/Vittorio-Bertocci-WS-Trust-Under-the-Hood
[cited 30 January 2011].

[35] The Internet Engineering Task Force. *The Kerberos Network Authentication Service (V5)*, July 2005 [online]. Available: http://www.ietf.org/rfc/rfc4120.txt [cited 02 February 2011].

[36] Free Software Foundation. *GNU Shishi* [online]. Available: http://www.gnu.org/software/shishi/
[cited 02 February 2011].

[37] The FreeBSD Foundation. *FreeBSD Handbook, Chapter 14 Security* [online book]. Available: http://www.freebsd.org/doc/en/books/handbook/kerberos5.html [cited 02 February 2011].

[38] WindowSecurity.com. *How to use Kerberos Authentication in a Mixed (Windows and UNIX) Environment*, 19 April 2006 [online]. Available: http://www.windowsecurity.com/articles/Kerberos-Authentication-Mixed-Windows-UNIX-Environment.html [cited 02 February 2011].

[39] The Internet Engineering Task Force. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*, January 1999 [online]. Available: http://www.ietf.org/rfc/rfc2459.txt [cited 02 February 2011].

[40] IBM, Microsoft. *Security in a Web Services World: A Proposed Architecture and Roadmap*, 01 Apr 2002 [online]. Available: http://www.ibm.com/developerworks/library/specification/ws-secmap/
[cited 03 February 2011].

[41] Scott Seely. *Understanding WS-Security*, October 2002 [online]. Available: http://msdn.microsoft.com/en-us/library/ms977327.aspx [cited 05 February 2011].

[42] OASIS. *Web Services Security: SOAP Message Security 1.1*, 01 February 2006 [online]. Available: http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf [cited 05 February 2011].

[43] OASIS. *Web Services Security X.509 Certificate Token Profile*, March 2004 [online]. Available: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf [cited 07 February 2011].

[44] OASIS. *Web Services Security Kerberos Token Profile 1.1*, 01 November 2006 [online]. Available: http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-errata-os-KerberosTokenProfile.pdf [cited 07 February 2011].

[45] OASIS. *Web Services Security Rights Expression Language (REL) Token Profile 1.1*, 01 February 2006 [online]. Available: http://www.oasis-open.org/committees/download.php/16687/oasis-wss-rel-token-profile-1.1.pdf [cited 08 February 2011].

[46] W3C. *Web Services Policy 1.5 – Framework*, 04 September 2007 [online]. Available: http://www.w3.org/TR/ws-policy/ [cited 09 February 2011].

[47] OASIS. *WS-SecurityPolicy 1.2*, 01 July 2007 [online]. Available: http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html#_Toc161826495 [cited 09 February 2011].

[48] W3C. *Web Services Policy 1.5 – Attachment*, 04 September 2007 [online]. Available: http://www.w3.org/TR/ws-policy-attach/ [cited 10 February 2011].

[49] Vedamuthu, Roth. *Understanding Web Services Policy*, 06 July 2006 [online]. Available: http://msdn.microsoft.com/en-us/library/ms996497.aspx [cited 10 February 2011].

[50] Goodner et al. *Understanding WS-Federation*, 28 May 2007 [online]. Available: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-fed/WS-FederationSpec05282007.pdf?S_TACT=105AGX04&S_CMP=LP [cited 17 February 2011].

[51] OASIS. *Web Services Federation Language (WS-Federation) Version 1.2*, 22 May 2009 [online]. Available: http://docs.oasis-open.org/wsfed/federation/v1.2/ws-federation.html [cited 17 February 2011].

[52] OASIS. *WS-SecureConversation 1.3,* 01 March 2007 [online]. Available: http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.html [cited 18 February 2011].

[53] Wiley, *The .NET Framework , Part 1* [online]. Available: http://media.wiley.com/product_data/excerpt/98/07645482/0764548298.pdf [cited 19 February 2011].

[54] CBT Nuggets, *CompTIA Security+ 2008 Certification Package* [video]. Available: http://www.cbtnuggets.com/series?id=454 [cited 20 February 2011].

[55] Gary C. Kessler. *An Overview of Cryptography* [online]. Available: http://www.garykessler.net/library/crypto.html [cited 23 February 2011]

[56] TrueCrypt. *TrueCrypt – Free Open-Source Disk Encryption Software* [online]. Available: http://www.truecrypt.org/docs/ [cited 24 March 2011].

[57] Microsoft TechNet. *BitLocker Drive Encryption Overview* [online]. Available: http://technet.microsoft.com/en-us/library/cc732774.aspx [cited 24 March 2011].

[58] ARX. *CoSign Digital Signature*[online]. Available: http://www.arx.com/digital-signature-how-it-works [cited 28 March 2011].

[59] Oracle. *Oracle Security Server Guide Release 2.0.3, Oracle Security Server Concepts* [online]. Available: http://download.oracle.com/docs/cd/A58617_01/network.804/a54088/conc1.htm [cited 29 March 2011].

[60] Gutmann, Naccache, Palmer, C.C. When Hashes Collide. In *IEEE Security & Privacy*. volume 3, issue 3, Digital Object Identifier 10.1109/MSP.2005.84, pages 68 – 71, 2005.

[61] Steve Vinoski. Where is Middleware? In *IEEE Internet Computing.* volume 6, issue 2, Digital Object Identifier 10.1109/4236.991448, pages 83-85, 07 August 2002.

[62] Ingeniørhøjskolen i Århus. *Presentation 4: Principles of Object-Oriented Middleware* [online]. Available: http://kurser.iha.dk/eit/onk/slides_2007/4-principles.ppt [cited 30 March 2011].

[63] Illinois Institute of Technology. *II. Middleware for Distributed Systems* [online]. Available: http://www.cs.iit.edu/~ren/cs447/lectures/ooMiddleware-3.ppt [cited 30 March 2011].

[64] Coulouris, Dollimore, Kindberg. *Distributed Systems: Concepts and Design.* Addison Wesley, 3rd Edition, 21 August 2000.

[65] Frank Eliassen. *Introduction to Distributed Systems*, 2009 [online]. Available: http://www.uio.no/studier/emner/matnat/ifi/INF5040/h09/lectures/2009-08-25_IntroDS.pdf [cited 01 April 2011].

[66] W3C. *Web Services Architecture Requirements*, 11 February 2004 [online]. Available: http://www.w3.org/TR/2004/NOTE-wsa-reqs-20040211/#id2604831 [cited 01 April 2011].

[67] An optimal agent-based architecture for dynamic Web service discovery with QoS. In *Second International conference on Computing, Communication and Networking Technologies, 2010.* Digital Object Identifier 10.1109/ICCCNT.2010.5591673, pages 1 – 7, Karur, India, 30 September 2010.

[68] W3C. *WS-Eventing*, March 2006 [online]. Available: http://www.w3.org/Submission/WS-Eventing/ [cited 09 March 2011].

[69] Bilorusets et. al, *WS-ReliableMessaging*, Febryary 2005 [online]. Available: http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf [cited 09 March 2011].

[70] OASIS. *Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.2*, 02 February 2009 [online]. Available: http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec.html [cited 10 March 2011].

[71] W3C. *Web Services Addressing 1.0 – Core*, 09 May 2006 [online]. Available: http://www.w3.org/TR/ws-addr-core/ [cited 10 March].

[72] W3C. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*, 27 April 2007 [online]. Available: http://www.w3.org/TR/soap12-part1/ [cited 12 March 2011].

[73] W3C. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, 26 June 2007 [online]. Available: http://www.w3.org/TR/wsdl20/ [cited 14 March 2011].

[74] OASIS. *UDDI Version 3.0.2*, 19 October 2004 [online]. Available: http://uddi.org/pubs/uddi_v3.htm [cited 15 March 2011].

[75] Jason Levitt. *From EDI To XML And UDDI: A Brief History Of Web Services*, 01 October 2001 [online]. Available: http://www.informationweek.com/news/development/tools/showArticle.jhtml?articleID=6506480 [cited 15 March 2011].

[76] OASIS. *UDDI v3.0 Ratified as OASIS Standard*, 03 February 2005 [online]. Available: http://www.oasis-open.org/news/pr/uddi-v3-0-ratified-as-oasis-standard [cited 15 March 2011].

[77] W3C. *SOAP over Java Message Service 1.0*, 26 October 2010 [online]. Available: http://www.w3.org/TR/soapjms/ [cited 22 March 2011].

[78] Brittenham et al. *Understanding WSDL in a UDDI registry*, 01 September 2001 [online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-wsdl/ [cited 12 May 2011].

[79] Fielding, Taylor. Principled design of the modern Web architecture. In *Journal*

ACM Transactions on Internet Technology (TOIT). volume 2, issue 2, pages 115 - 150, May 2002.

[80] W3C. *Hypertext Transfer Protocol -- HTTP/1.1* [online]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9 [cited 05 June 2011].

[81] The Internet Engineering Task Force. *The application/json Media Type for JavaScript Object Notation (JSON)*, July 2006 [online]. Available: http://www.ietf.org/rfc/rfc4627.txt [cited 27 June 2011].

[82] Juval Lowy. *Programming WCF Services*. O'Reilly Media, 1st Edition, 27 February 2007

[83] Pablo Cibraro. *OAuth channel for WCF RESTful services*, 14 November 2008 [online]. Available: http://weblogs.asp.net/cibrax/archive/2008/11/14/oauth-channel-for-wcf-restful-services.aspx [cited 01 July 2011].

[84] Chang, Mohd-Yasin, Mustapha. An Implementation of Embedded RESTful Web Services. In *Innovative Technologies in Intelligent Systems and Industrial Applications, 2009.* Digital Object Identifier 10.1109/CITISIA.2009.5224244, pages 45 – 50, 28 August 2009.

[85] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD dissertation, University of California, Irvine, USA, 2000.

[86] Michael Przybilski. *REST - REpresentational State Transfer*. [online]. Available: http://www.cs.helsinki.fi/u/chande/courses/cs/MWS/reports/MichaelPrzybilski_REST.pdf [cited 04 October 2010].

[87] Landre, Wesenberg. REST versus SOAP as Architectural Style for Web Services. In *5th International OOPSLA Workshop on SOA & Web services Best Practices*, 2007.

[88] Stockinger, Attwood, Chohan, Cote, Cudre-Mauroux, Falquet, Fernandes, Finn, Hupponen, Korpelainen, Labarga, Laugraud, Lima, Pafilis, Pagni, Pettifer, Phan, Rahman. Experience using web services for biological sequence analysis. In *Brief Bioinform*, volume 9, number 6, Digital Object Identifier 10.1093/bib/bbn029, pages 493-505, 01 November 2008.

[89] .NET Development Forums. *Multiple Protection Levels does not work in WCF* [online]. Available: http://social.msdn.microsoft.com/Forums/en-US/wcf/thread/dc82fd79-8836-4214-b541-720abba4c6b3/#0e3302b0-9896-43a6-970f-b04a9d63373e [cited 20 May 2011].

[90] Microsoft Developer Network. *How to: Set the ProtectionLevel Property* [online]. Available: http://msdn.microsoft.com/en-us/library/aa347791%28v=VS.90%29.aspx [cited 26 June 2011].

[91] VeriSign. *Compare SSL Certificates* [online]. Available: http://www.verisign.com/ssl/buy-ssl-certificates/compare-ssl-certificates/index.html [cited 28 June 2011].

[92] Bouncy Castle. *Bouncy Castle C# API* [online]. Available: http://www.bouncycastle.org/csharp/ [cited 27 June 2011].

[93] Simson Garfinkel. *PGP: Pretty Good Privacy*, O'Rielly Media, 1995.

[94] The Internet Engineering Task Force. *Cryptographic Message Syntax*, September 2009 [online]. Available: http://tools.ietf.org/html/rfc5652 [cited 28 June 2011].

[95] Recordon, Reed. OpenID 2.0: A Platform for User-Centric Identity Management. In *Proceedings of the second ACM workshop on Digital identity management.* Digital Object Identifier 10.1145/1179529.1179532, pages 11-16, Alexandria, Virginia, USA, 2006.

[96] EASEE. *EASEE-gas* [online]. Available: http://www.easee-gas.org/home.aspx [cited 01 July 2011].

[97] Kirk Evans. *Creating RESTful Services Using WCF*, 03 April 2008 [online]. Available: http://blogs.msdn.com/b/kaevans/archive/2008/04/03/creating-restful-services-using-wcf.aspx [cited 02 July 2011].

[98] Solheim, Stølen, *Technology Research Explained.* SINTEF A313, SINTEF, March 2007.

[99] The Apache Software Foundation. *Apache Abdera, An Open Source Atom Implementation*, [online]. Available: http://abdera.apache.org/index.html [cited 07 January 2011]

[100] The Internet Engineering Task Force. *The OAuth 1.0 Protocol*, April 2010 [online]. Available:
http://tools.ietf.org/html/rfc5849 [cited 01 July 2011]

[101] Vlaanderen, Jansen, Brinkkemper, Jaspers. The agile requirements refinery: Applying SCRUM principles to software product management. In *Third International Workshop on Software Product Management (IWSPM), 2009.* Digital Object Identifier 10.1109/IWSPM.2009.7, pages 1-10, 03 May 2010

[102] OpenSSL Project. *Welcome to the OpenSSL Project* [online]. Available: http://www.openssl.org/ [cited 28 June 2011]

[103] Citrix Systems. *Access Gateway Enterprise Edition 9.0 with Nordic Edge One TimePassword Server* [online]. Available: http://support.citrix.com/servlet/KbServlet/download/22820-102-642295/CTX122707_0050.pdf [cited 29 June 2011]

[104] Mili, Elkharraz, Mcheick. Understanding separation of concerns. In *Early Aspects workshop of the Aspect-Oriented Software Development – AOSD 2004 conference*, page 22-26, Lancaster, UK, March 2004.

[105] Gary Pollice. *Agile software development: A tour of its origins and authors*, 15 March 2007 [online]. Available: http://www.ibm.com/developerworks/rational/library/mar07/pollice/index.html [cited 21 June 2011]

[106] Murali Manohar Pareek. *WCF (Windows Communication Foundation) Introduction and Implementation*, 13 November 2008 [online]. Available: http://www.codeproject.com/KB/WCF/WCFServiceSample.aspx [cited 01 July 2011]

[107] Amazon.com. Search: Books, RESTful [online]. Available: http://www.amazon.com/gp/search/ref=sr_nr_p_n_feature_browse-b_mrr_0?rh=n%3A283155%2Ck%3ARESTful%2Cp_n_feature_browse-bin%3A2656022011&bbn=283155&keywords=RESTful&ie=UTF8&qid=1312026041&rnid=618072011 [cited 01 July 2011]

[108] Dan R. Olsen III. *Putting the Web Services Specifications to REST*, Master thesis, Brigham Young University, Provo, Utah, USA, April 2008.

[109] Dan Forsberg. *RESTful Security* [online]. Available: http://w2spconf.com/2009/papers/s4p3.pdf [cited 25 June 2011]