RewritingHealer: An approach for securing web service communication

Faisal Abdul Kadir

Master of Science Thesis
Stockholm, Sweden, 2007
ICT/ECS – 2007 - 20

RewritingHealer: An approach for securing web service communication

Faisal Abdul Kadir

Company Supervisor
Rits Erik Maarten
SAP Labs France

Examiner
Assoc. Prof. Vladimir Vlassov
KTH, Stockholm, Sweden

Master of Science Thesis
Stockholm, Sweden, 2007
ICT/ECS – 2007 - 20

Dedicated to
My mother Jahanara Begum

Abstract

Web Service is a distributed communication technology that can implement Service Oriented Architechture (SOA) to support the requirement of business process integration. SOAP (Simple object access protocol) is a lightweight protocol that standardized a framework for web service communication. WS-Security is a security standard for securing web service communication. WS Security adopted XML Digital Signature technology for providing authenticity and integrity verification capability for the receiver of a SOAP message. XML Digital Signature provides a mechanism for signing non-contiguous parts of a SOAP message. However, XML Digital signature does not provides any information regarding the location of a signed object. This feature of XML Digital Signature is used by an attacker to perform a class of attack known as XML Rewriting Attack where an attacker modifies the SOAP message in a way that does not violate the signature of the message.

Some works have already been done for the detection of XML Rewriting Attack in web service communication. However, none of the solutions proposed previously could eliminate XML Rewriting Attack totally from web service world. Our work tends to propose a solution for the detection of XML Rewriting Attack. We have analyzed different types of XML Rewriting Attack, identified the information that we can get after an attack takes place to provide a countermeasure against XML Rewriting Attack. We have demonstrated that our proposed solution can detect XML Rewriting Attack where all of the previous solutions fail. We have also demonstrated that our proposed approach has polynomial bounded processing complexity but it introduces some overhead in the total size of a SOAP message. We have proposed an optimization that can substantially reduce this overhead.

Table of Contents

Chapter 1: Introduction
- **1.1 Background**
- **1.2 Motivation**
- **1.3 Thesis Objective**
- **1.4 Thesis Structure**

Chapter 2: Background
- **2.1 Security Protocols**
  - **2.1.1 Cryptography**
  - **2.1.2 Symmetric Key Cryptography**
  - **2.1.3 Public Key Cryptography**
  - **2.1.4 Digital Signature**
  - **2.1.5 Digital Envelope**
  - **2.1.6 Hashing**
- **2.2 Overview of Web service**
  - **2.2.1 What is Web service?**
  - **2.2.2 Web service Architecture**
- **2.3 SOAP**
  - **2.3.1 Introduction**
  - **2.3.2 SOAP Message Construct**
  - **2.3.3 SOAP Processing Model**
  - **2.3.4 SOAP Extensibility Model**
  - **2.3.5 SOAP Protocol Binding**
- **2.4 Motivation of Web services Security**

Chapter 3: State of the Art
- **3.1 XML Digital Signature**
- **3.2 WS Security**
- **3.3 WS Policy**
- **3.4 WS Security Policy**
- **3.5 Formal methods and Web services Security**

Chapter 4: XML Rewriting Attack Scenarios
- **4.1 First Attack Scenario**
- **4.2 Second Attack Scenario**
- **4.3 Third Attack Scenario**

Chapter 5: Previous Solutions of XML Rewriting Attack
- **5.1 Using Xpath expression with WS-Security policy**
- **5.2 WSE Policy Advisor**
- **5.3 SOAP-Account approach**

Chapter 6: Proposed Method
- **6.1 Problem Analysis**
- **6.2 RewritingHealer**
- **6.3 Processing Rules for RewritingHealer**
- **6.4 Scenario with RewritingHealer**

Chapter 7: Implementation
- **7.1 Axis Overview**
- **7.2 Module Description**
- **7.3 A Simple Application**

7.4 Evaluation
Chapter 8: Conclusions and Future Works
Chapter 9: References

Table of Figures

Figure 1.1: The evolution of web services security specifications
Figure 2.1: Symmetric Key Cryptography
Figure 2.2: Public Key Cryptography
Figure 2.3: Digital Signature Process
Figure 2.4: Digital Envelope Creation Process
Figure 2.5: Digital Envelope Verification Process
Figure 2.6: Web service Roles
Figure 2.7: Protocol Stack for Web service
Figure 2.8: SOAP Message Structure
Figure 2.9: Soap Processing Model
Figure 2.10: Point to Point Security
Figure 2.11: End to End Security
Figure 3.1: Structure of XML Digital Signature
Figure 3.2: A Simple WS Policy
Figure 4.1: SOAP message before Replay Attack
Figure 4.2: SOAP message after Replay Attack
Figure 4.3: SOAP message before Redirection Attack
Figure 4.4: SOAP message after Redirection Attack
Figure 4.5: SOAP message before Multiple Security Header Exploitation Attack
Figure 4.6: SOAP message after Multiple Security Header Exploitation Attack
Figure 5.1: SOAP message with a ReplyTo header block signed and referenced using URI
Figure 5.2: Security Policy assertions for SOAP message of Figure 5.1
Figure 5.3: SOAP message where the attacker wrapped up the ReplyTo header block
Figure 5.4: SOAP message with a ReplyTo header block signed and referenced using XPath
Figure 5.5: Security Policy assertions for the SOAP message of Figure 5.4
Figure 5.6: SOAP message with a Timestamp element signed and referenced using XPath
Figure 5.7: Security Policy assertions for SOAP message of Figure 5.6
Figure 5.8: SOAP message where the attacker created a Security Header block with role attribute's value none
Figure 5.9: WSE Policy Advisor
Figure 5.10: Structure of SOAP Account
Figure 5.11: SOAP message with SOAP Account header block
Figure 5.12: SOAP Account attack detection
Figure 5.13: SOAP message with a SOAP Account header block and before modification by attacker
Figure 5.14: SOAP Account Vulnerability
Figure 6.1: Tree representation of the SOAP message of Figure 5.1
Figure 6.2: Rewriting attack without changing the depth of an element
Figure 6.3: Relocation of an element without changing its depth and parent Id
Figure 6.4: Pre-order traversal of a tree and its string representation
Figure 6.5: Tree representation of a SOAP message, each node is labelled with its role's integer value
Figure 6.6: Tree only representing nodes corresponding to elements targeted to the intermediary

Figure 6.7: Structure of RewritingHealer
Figure 6.8: SOAP message with RewritingHealer header block
Figure 6.9: SOAP message with RewritingHealer header block and signed elements reordered by attacker
Figure 6.10: SOAP message with two RewritingHealer header blocks
Figure 6.11 : SOAP message with two RewritingHealer header blocks and element created by attacker
Figure 7.1: Message flow through an Axis engine on server side
Figure 7.2: Message flow through the axis engine on the client side
Figure 7.3: Module Structure of RewritingHealer
Figure 7.4: The invoke method of the RewritingHealerVerifier handler
Figure 7.5: The verify method of RewritingHealerVerifier
Figure 7.6: The invoke method of RewritingHealerCreator library
Figure 7.7: Interface of the Math Service
Figure 7.8: MathService Client Interface
Figure 7.9: The RewritingHealerVerifier interface
Figure 7.10: RewritingHealerVerifier detecting XML Rewriting Attack
Figure 7.11: A simple tree representing a SOAP message
Figure 7.12: Tree of Figure 7.11 after elements order has been changed
Figure 7.13: Tree of Figure 7.11 after elements order has been changed

Table of Tables

Table 1: SOAP defined roles
Table 2: Overhead estimation of RewritingHealer (without optimization)
Table 3: Overhead estimation of RewritingHealer (with optimization)

Chapter 1: Introduction

1.1 Background

Web service is a distributed systems technology where the network endpoints exchange specific form of XML document called SOAP [1] envelope which contains a mandatory Body element containing a request, response, or fault element, together with an optional Header element containing routing or security information. SOAP allows the existence of network intermediaries. An intermediary can be a routers or a firewall. SOAP allows these intermediaries to process an envelope, by adding or modifying headers. Examples of web services include Internet-based services for ordering goods or invoking search engines, and intranet-based services for linking enterprise [35]

Because of its simplicity, standardization and platform independent nature a lot of business organizations have embraced web services technology for the integration of data, system and application inside and outside their organization boundary. Due to this growing adoption by different organization, security of web services became as a vital issue.

One way for securing SOAP exchanges can be relying on the traditional transport level security like TLS/SSL. This might work well in many situations, however is not suitable for every situation due to its point-to-point security nature. TLS/SSL creates a security tunnel between the two communication end points. The integrity and confidentiality of a message is ensured as long the message exists inside this tunnel but not subsequently in files or databases, and they may not match the security requirements of the application. For instance, Client authentication is often performed by the application rather than by TLS/SSL. Besides, SSL does not fit SOAP's message-based architecture: intermediaries cannot see the contents of the tunnel, and so cannot route or filter messages. [35]

To overcome the above limitation of traditional transport level security, Microsoft, along with IBM and VeriSign came up with a new security framework that can be used to achieve end-to-end security in web services communication. This security framework is named as WS-Securty [6]. WS-Security specifies how security elements like encrypted data, signed data and security tokens like username, X.509 certificate etc. can be embedded in a SOAP message which is the vocabulary for web services communication.

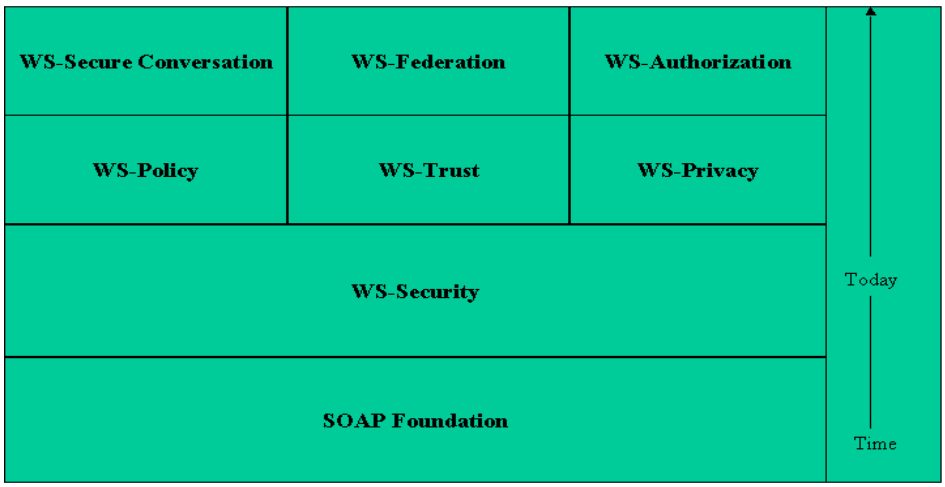

Figure 1.1: The evolution of web services security specifications [36]

After the emergence of WS-Security a number of standards on top of this have been proposed to take care of different security aspects of web services communication. WS-Security Policy [7] along with WS Policy [42] attachment provide a standard way to specify the security requirements of a web service, WS-Secure Conversation [29] specifies ways for securing not only a single SOAP message but also a whole session between the client and the server, WS-Trust [28] specifies how security tokens can be requested, issued and verified. Figure 1.1 shows the different security standards that have evolved with the passage of time.

1.2 Motivation

Despite all of the proposed web services security specifications, web services are still vulnerable to a class of attacks, first described by Needham and Schroeder [38] and first formalized by Dolev and Yao [37], where an attacker may intercept, compute, and inject messages, but without compromising the underlying cryptographic algorithms. In the terminology of web services security, this sort of attack is called *XML rewriting attacks*, as opposed to attacks on web service implementations, such as buffer overruns or SQL injection [21]. WS-Security adopted XML Digital Signature [2] and XML Encryption mechanism, designed for general XML Document, to provide integrity and confidentiality in SOAP communication. XML Digital Signature refers to a signed object of an XML Document in a way that does not take care of the location of that object. This is a weakness of XML Digital Signature. SOAP extensibility model allows a SOAP message to contain a SOAP header element that is not recognized by the receiver. WS-Security allows multiple security headers to exist in the same SOAP message. All of these features along with the weakness of XML Digital Signature work as the weapon for performing XML Rewriting attack on SOAP messages.

Some works have been done for the detection of XML Rewriting Attack previously, but none of them can detect XML Rewriting Attack properly. Therefore we have carried out our present work to formulate a solution for the detection of XML Rewriting Attack.

1.3 Thesis Objective

WS-Security Policy [7] if used correctly can be used to guard against XML Rewriting Attack. However, most of the policy configuration files are hand written. Therefore they are very much error prone. Moreover, due to the flexibility provided by SOAP extensibility model it is hardly possible to prevent attack where the attacker moves a signed element under another unknown element.

A lot of research works have been done and are going on for the detection of XML Rewriting attack like [9], [11], [12], [13]. All of these works proposed solutions that can detect XML Rewriting attack properly in some cases, however they fail to handle some other cases. The goal of our works are as follows:

1. Analyze different web services security specification and find their weakness
2. Elicit the different form of XML Rewriting attack that can take place in a web service communication
3. Investigate all the previous works that have been done for the detection of XML rewriting attack and find their vulnerability
4. Investigate the reasons of these sorts of attacks.
5. Find exiting features of XML, SOAP and WS-Security that can be used in a systematic way to prevent XML Rewriting attack
6. Propose a method for the detection of XML Rewriting attack that can detect attacks in situation where all of the previous solutions fail.

1.4 Thesis Structure

The rest of this report is organized as follows:

In Chapter 2 we will review some literature related to our work. We will see different encryption and authentication mechanisms and their pros and cons. Then we will see what is SOAP and how is it used for web service communication. We will also discuss the motivation for web services security.

Chapter 3 describes different existing frameworks that have emerged to secure web service communication and their limitations. Here we will talk in brief regarding the necessity of formal methods in the field of web services security. We will also specify different works that have been done for introducing formal methods in formalizing web services security standards.

Chapter 4 will demonstrate with scenarios different kinds of XML Rewriting Attack that can take place.

Different proposed solutions for eliminating XML Rewriting Attack will be described in Chapter 5. We will analyze all of the solutions and will try to identify their weakness and vulnerability.

In Chapter 6 we have proposed our approach for the detection of XML Rewriting Attack. We have analyze the problem of XML Rewriting Attack in depth and then we have tried to identify what sort of information can we get after a rewriting attack takes place to guard against this attack. At last we have proposed a method for exchanging the identified information in a structured and standard way.

In Chapter 7, we will describe the implementation of our proposed method. First we will discuss about the environment for our implementation. Then we will discuss our implementation and at last we will use a simple application to demonstrate how our approach can successfully detect XML Rewriting Attack. Finally, in this chapter we will evaluate our proposed method based on different criteria. We will try to formally specify the time complexity of our proposed method. Here we will also see some limitations of our proposed method.

At last in Chapter 8, we will talk about the future works that will be carried out to extend and optimize our proposed method and then we will conclude.

Chapter 9 will provide the references that we have used for our work.

Chapter 2: Background

2.1 Security Protocols:

2.1.1 Cryptography

The word cryptography means the art of secret writing. It is a procedure of transforming information from its original format to a format that is not easily understandable. To get back the original information from the transformed one a secret method has to be used. Cryptography is used for communication between participants in a way that prevents other from reading it.

In the terminology of cryptography the message that is to be transformed is called the plaintext and the resulting message generated after transformation is called the ciphertext. The mechanism of transforming a plaintext into ciphertext is called encryption and the mechanism of getting back the plaintext from the ciphertext is called the decryption.

Two types of cryptographic procedure are commonly used now a day:

i) Symmetric key cryptography
ii) Public key cryptography

2.1.2 Symmetric key cryptography

To secure a communication using Symmetric Key cryptography, parties involved in the communication must share a secret key. This shared key is used both for encryption and decryption of the message exchanged among the communicating parties. As the same key is used for both encryption and decryption this cryptographic procedure is called symmetric key cryptography. It is also known as shared key cryptography.

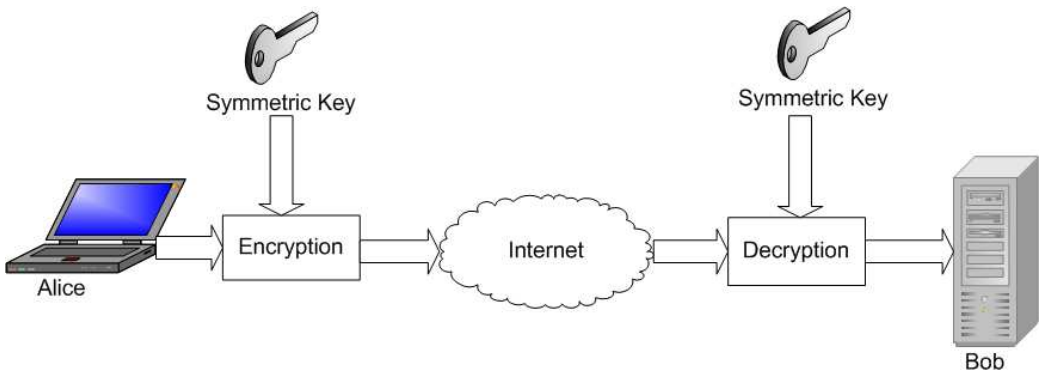

Figure 2.1: Symmetric Key Cryptography

Figure 2.1 shows the concept of symmetric key cryptography in digital communication. Suppose, Alice wants to send a message to Bob over the public network which is not secured. However he wants only Bob to read the message. That means he does want any eavesdropper to grab the message and read it. How Alice can achieve this goal? Symmetric key cryptography is one of the solutions of this problem. Lets assume that Alice and Bob know a common secret key. Furthermore, no one except Alice and Bob have this secret key. Now, what Alice can do is that, before sending the message he will encrypt the message using the secret key to get the ciphertext and will send this ciphertext to Bob. When Bob will receive the ciphertext form Alice he will decrypt it using the same secret key to get the original message back.

Symmetric key cryptography is very simple and efficient mechanism for securing digital communication. However it has some drawbacks. The most critical problem of symmetric key cryptography is the key distribution problem. In the scenario above we have assumed that both Alice and bob know a secret key. We further assumed that no one else except them has any knowledge of that key. But, how can Alice and Bob establish the secret key. We can consider the following solutions:

i) Alice can go to Bob personally and tell him the secret that he wants to use to communicate with Bob. If Alice lives in the same region as Bob then this solution can be considered. However, if Alice and Bob live in two different countries then obviously the above solution is out of question. Moreover, Bob might not the only person with whom bob wants a secure communication. Therefore, to communicate with each person securely, Alice will have to go to them personally. Which is not feasible.

ii) Alice can send Bob a mail and in that mail he can specify the secret. However, here comes the question how Alice will secure the mail? An eavesdropper can intercept the mail and can easily get the secret.

In fact there is no effective solution for the key distribution problem of Symmetric key cryptography. For this reason despite of its simplicity and efficiency, Symmetric key cryptography is not widely used.

2.1.3 Public key cryptography

In last section we saw that the major problem of Symmetric key cryptography is key distribution. To overcome this key distribution problem the idea of public key cryptography emerged. The main idea of public key cryptography is that a key will be composed of two parts. If a message is encrypted using one part of a key then it can be decrypted using only the other part. In the terminology of public key cryptography the two parts of a key are called public key and private key. The public key, as its name implies, will be published in public. That means anyone can know it. However, the private key, as its name implies, will be totally private. That is only the owner of the key will know it.

Suppose Alice wants to send a message to Bob securely. Figure 2.2 how he can do this. He will first get the public key of Bob. Then he will encrypt the message using this key and will send it to Bob. When Bob will receive the ciphertext sent by Alice, he will use his private key to decrypt it.

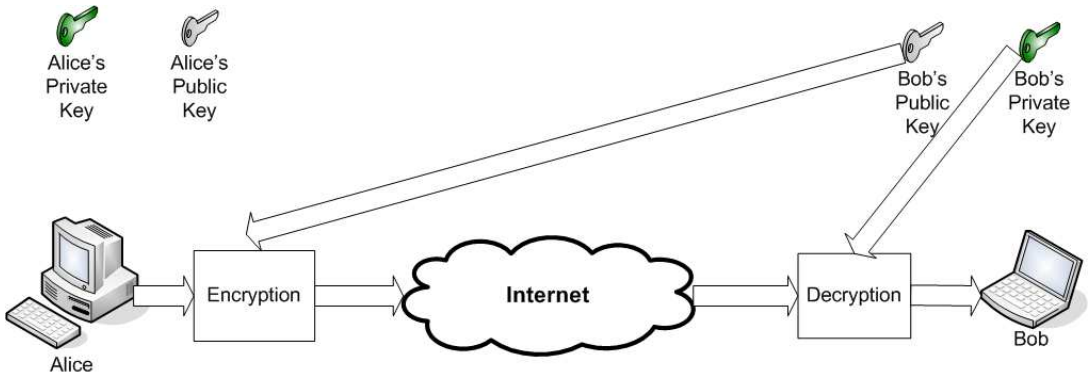

Figure 2.2: Public Key Cryptography

As the message is encrypted using Bob's public key, it can only be decrypted using Bob's private key. However, Bob is the only one who knows his private key. Therefore, only Bob will be able to decrypt the message. The idea of public key removes the problem of key distribution. With this mechanism no secure key distribution is necessary. The owner of a private key does not have to tell anyone about his private key. Everyone can get his public key. However, that public key will be of no use without its associated private key.

Although we saw that public key cryptography eliminates the problem of symmetric key cryptography, it has some drawbacks. As we said before, to send a secure message one has to get the receiver's public key. But how can a sender authenticate the receiver's public key. For instance in the above example, another person Trudy can publish a public key and can say that its Bob's public key. If Alice does not authenticate the public key, he will encrypt the message using Trudy's public key, which will in turn be decrypted by Trudy.

To overcome the above problem a trusted certificate authority is defined whose job is to certify the public key of a user. Despite of the above stated problem, public key cryptography is a widely accepted and used cryptographic digital communication technique.

2.1.4 Digital signature

Digital Signature is the mechanism for signing a document electronically. It is used by the recipient of a document to authenticate the identity of the sender of the document. It is also used to check the integrity of the received document.

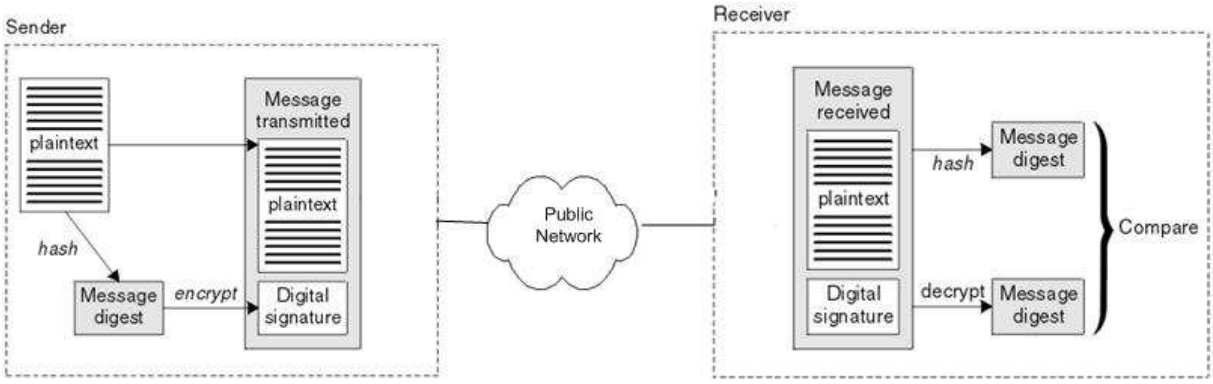

Figure 2.3: Digital Signature Process

By verifying the integrity of the received document the receiver can be assured that the document has not been forged by an adversary on its way from the sender to the receiver.

Figure 2.3 shows how a digital signature is created by the sender and verified by the receiver.

When a sender wants to send a digitally signed message m to a receiver, it goes through the following steps:

i) It generates a message digest MD using an appropriate hash method H of the plaintext message m. We will represent the message digest MD of a plaintext message x as H (x). So the sender first generates MD = H (m) for the plaintext message m.

ii) It then encrypts the message digest MD using its own private key and generates MD^{-1}, where MD^{-1} represents the encrypted message digest.

iii) It appends the encrypted message digest MD^{-1} with the plaintext message m and generates the message $m|MD^{-1}$.

iv) It sends $m|MD^{-1}$ to the receiver.

When a receiver receives a message M = m|MD^{-1} from a sender , it goes through the following steps

i) It extracts the whole message M = m|MD^{-1} into its sub-components. That is it divides the whole message M into m and MD^{-1} .

ii) It generates a message digest MD_1 of the received message m.

iii) It decrypts the received encrypted message digest MD^{-1} and generates MD_2.

iv) It compares the two message digest MD_1 and MD_2. If the two message digests are the same then the message has not been forged on its way and the sender is the one who it supposed to be. Otherwise either the message has been forged or the identity of the sender is not valid.

2.1.5 Digital envelope

The main problem in using secret key cryptography is the distribution of the secret key among the communicating parties. If Bob wants to send a message to Alice using symmetric key cryptography then both Bob and Alice need to share a common secret key. However this secret key has to be exchanged between them in a secure way .If this secret key is intercepted by a middle-man while exchanging, the communication between Bob and Alice with this secret key will not remain secret anymore. A middleman might intercept all the messages exchanged between them and would be able to decrypt and read it.

To overcome this key management problem the idea of public key cryptography emerged. In public key cryptography a key consists of two parts. A public part and a private part. The public part of the key is known to all. However, the private part is only known to the owner. Furthermore, if a message is encrypted using a public or private key then it can only be decrypted using the respective private or public key. This idea has immensely simplified the key management problem. Now the participants don't have to exchange any any secret key . However all they have to know is the public key of the party to whom he/she wants to send a secret message. And as this public key is known to all the sender can just grab it and start the communication.

Every cryptographic idea comes with its own limitation. Public key cryptography is not an exception. One of the limitations of the existing public key cryptographic procedure is that they are computationally expensive. Digital Envelope is a framework, which tries to combine the advantages of the above mentioned cryptographic service.

A Digital envelope consists of a message encrypted using a secret key cryptography and an encrypted secret key . Normally Digital Envelope use public key cryptography to encrypt the secret key.

Suppose a sender wants to send a message M to a receiver using Digital Envelope. Then he will go through the following steps:

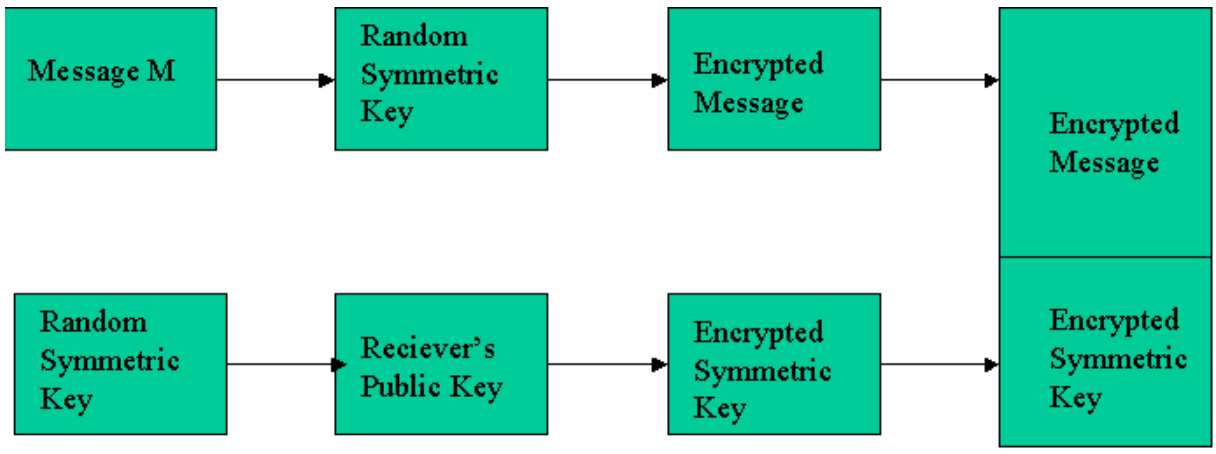

Figure 2.4: Digital Envelope Creation Process

i) The sender will choose a random secret key to encrypt the message M
ii) The sender will encrypt the message using the secret key chosen.
iii) The sender will take the receiver's public key to encrypt the secret key
iv) The sender will encrypt the secret key with the receiver's public key
v) The sender will append the encrypted secret key with the encrypted message and will send it to the receiver.

When a Receiver will receive the message it will go through the following steps:

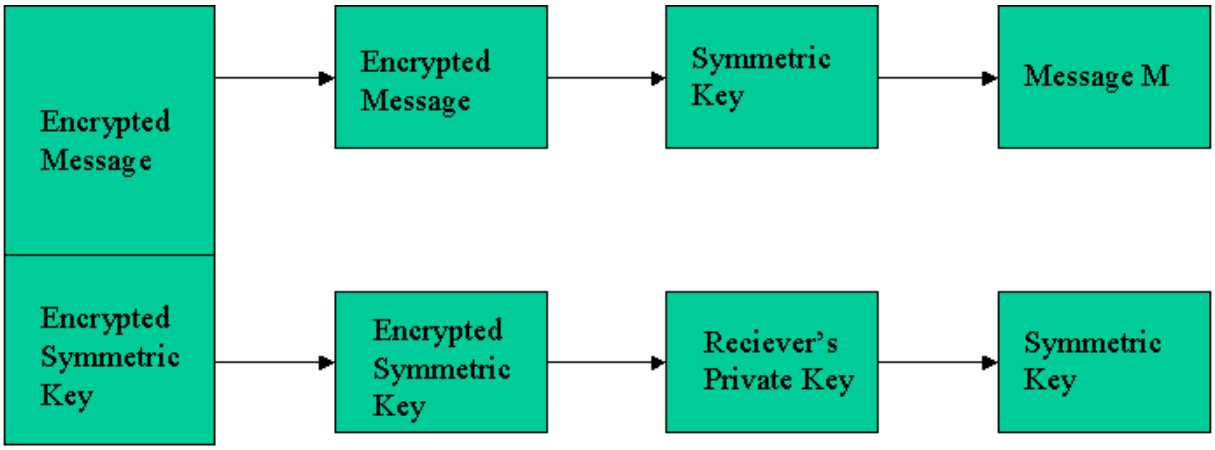

Figure 2.5: Digital Envelope Verification Process

i) The receiver will extract the message into its sub parts. That is, it will retrieve the encrypted message and the encrypted secret key.
ii) The receiver will decrypt the encrypted secret key using its private key to get the secret (symmetric) key.
iii) The receiver will use the decrypted symmetric key to decrypt the encrypted message to get the message M.

The Digital envelope has the following advantages over the Symmetric and Public key cryptographic procedure:

i) It simplifies the key management procedure, which is the main limitation in using symmetric key cryptography.
ii) It increases the performance. Now the whole message, which can be of variable length, need not be encrypted using Public key cryptography. However the message itself is encrypted using secret key and the secret key, which is usually much smaller in size than the message, is encrypted using public key cryptography.

2.1.6 Hashing

A Hash is also known as message digest. It is a one-way function that takes as input a variable length message and generates a fixed length hash value. We call this function one-way as it is possible to generate output y for input x with this function however it is not practically possible to get back input x from output y.

Let us denote the hash of a message m as h(m). Then h(m) should contain the following properties :

i) h(m) should be relatively easy to compute for any given message m . That means it should not consume a lot of processing time to compute the hash h(m) for any message m .

ii) Given h(m) , there is no way to find an m that hashes to h(m) in a way that is substantially easier than going through all possible values of m and computing h(m) for each one .[18]

iii) From the definition of hash it is obvious that more than one message m will map to the same hash value h(m) . However , it should be computationally infeasible to find two messages that map to the same hash value .

Hashing has an important role in security world. It can be used to check the integrity of a transmitted message. It can also be used to verify the authentication of the sender.

There are a number of Hashing algorithms that are used now a days. Like SHA-1,SHA-224,SHA-256,SHA-512 etc. These hashing algorithms are called secure hash algorithm. The numbers associated with the name of the hash algorithms indicate the length of the output in bits. For instance SHA-256 means, this algorithm will take a variable length message as input and will produce a 256 bits message digest.

2.2 Overview of Web service

2.2.1 what is Web service?

Web service is a service that has the following properties:

1. The service can be accessed over the Internet
2. The service uses a standard XML messaging system. SOAP (Simple Object Access Protocol) is a communication protocol used for XML messaging.
3. The service is not dependent on any particular operating system or programming language
4. Although not absolutely necessary, web service can have the following two properties as well
5. A web service will have a description written in XML. WSDL (Web service Description Language) is used for this.
6. A web service can be located using a find mechanism. UDDI is a standard to locate web service.

Web service provides a mechanism for machine-to-machine interaction. It's a way for programmatic access to web sites. Unlike traditional client/server models, such as a Web Server/Web page system, Web service do not provide the user with a GUI. Web service instead access business logic, data and processes through a programmatic interface across a network.

2.2.2 Web service Architecture

Web service architecture can be analyzed in two different ways. The first is to examine the individual roles of each web service actor, the second is to examine the web service protocol stack.

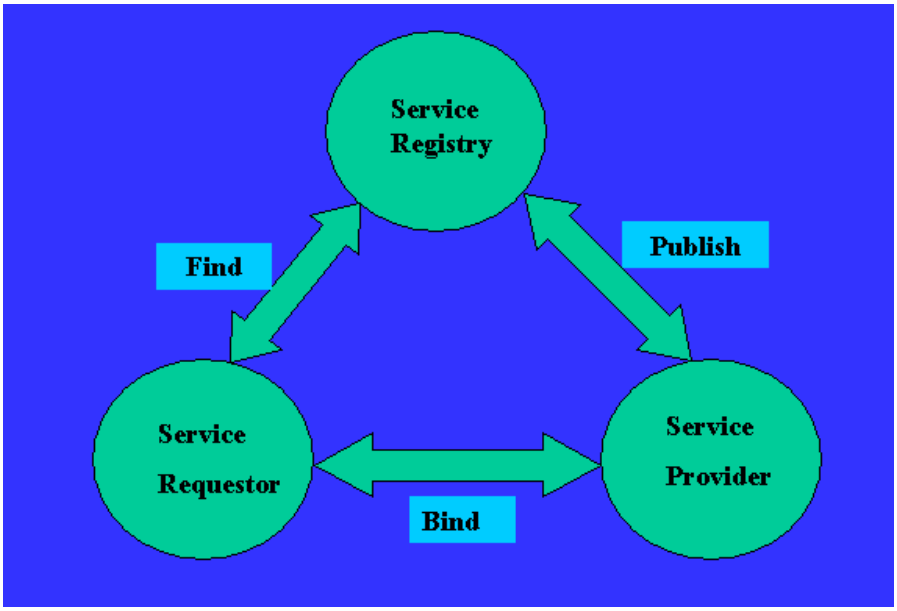

Figure 2.6: Web service Roles [5]

Web service Roles:

Figure 2.6 shows the different roles that exist in web service architecture. There are three major roles within the web service architecture:

1. Service provider
In web service architecture the responsibility of the role Service Provider is to provide the service. The service provider is the one who develops the service and publish it on the Internet.

2. Service requestor
The responsibility of the role Service Requestor is to request the service and to consume it. It establishes a connection and sends a service request in XML format to the provider.

3. Service registry
The responsibility of the role Service Registry in web service is to provide a centralized directory of services. The provider of a service publishes their service using this registry. Consumer discovers existing services from this registry and then binds to the appropriate service provider to utilize that service.

A second option for viewing the web service architecture is to examine the emerging web service protocol stack. The stack is still evolving, but currently has four main layers. Following is a brief description of each layer.

Web service Protocol Stack:

Figure 2.7 shows the different layers of the Web service Protocol stack. The Web service protocol stack consists of 4 layers:

1. Service transport:

This layer is responsible for transporting messages between applications. Currently, this layer includes hypertext transfer protocol (HTTP), Simple Mail Transfer Protocol (SMTP), file transfer protocol (FTP), and newer protocols, such as Blocks Extensible Exchange Protocol (BEEP).[5]

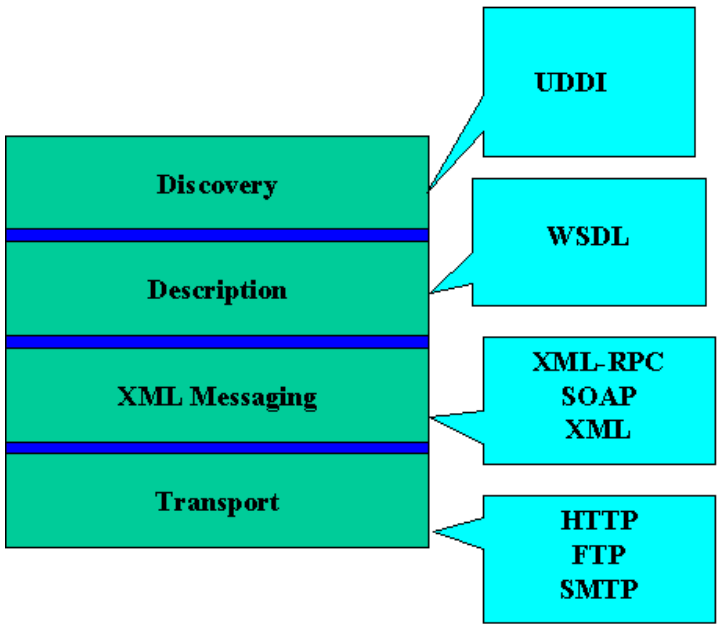

Figure 2.7: Protocol Stack for Web service

2. XML messaging
This layer is responsible for encoding messages in a common XML format so that messages can be understood at either end. Currently, this layer includes XML-RPC and SOAP.[5]

3. Service description
This layer is responsible for describing the public interface to a specific web service. Currently, service description is handled via the Web service Description Language (WSDL).[5]

4. Service discovery
This layer is responsible for centralizing services into a common registry, and providing easy publish/find functionality. Currently, service Discovery is handled via Universal Description, Discovery, and Integration (UDDI).[5]

2.4 SOAP

2.4.1 Introduction

SOAP (Simple Object Access Protocol) is a lightweight protocol for exchanging structured data in decentralized and distributed environment. [1] SOAP provides an extensible messaging framework using XML technologies. It defines a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics. [1]

The SOAP messaging Framework consists of the following, which will be described subsequently in this section.

i) SOAP Message Construct
ii) SOAP Processing Model
iii) SOAP Extensibility Model

iv) SOAP Protocol Binding

2.4.2 SOAP Message Construct

In this section we will discuss the structure of a SOAP message. Figure 2.8 is a pictorial representation of a SOAP Message.

A SOAP message is encoded as an XML document. Each SOAP message will consist of one root element, which is called <Envelope> element. The <Envelope> element will contain the following sub elements as its children.

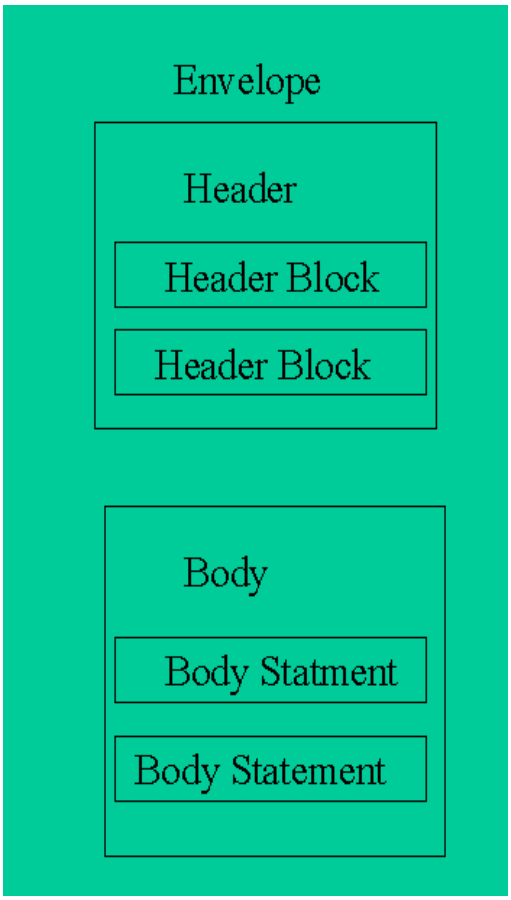

Figure 2.8: SOAP Message Structure [1]

i) An optional <Header> Element
ii) A mandatory <Body> Element

A <Header> element will contain data that is not the application payload. This element is intended to be processed by zero or more intermediaries along the path of the SOAP message from sender to the receiver. <Header> element will contain zero or more <HeaderBlock> as its child element. Each HeaderBlock within the <Header> element may realize zero or more features. For instance, to realize the security feature, which is not specified by the core SOAP messaging framework, a <Security> header block will be used as the sub element of the <Header>.

The <Body> element is mandatory and contains the application payload. The <Body> element is always intended to be processed by the Ultimate Receiver of the SOAP message.

2.4.3 SOAP Processing Model

The SOAP [1] processing model specifies how a SOAP receiver processes a SOAP message. [1] SOAP specifies a distributed processing model. The Initial Sender generates a SOAP message, which is reached to the ultimate SOAP receiver via zero or more SOAP nodes. The distributed processing model of SOAP can be used to support a number of different MEP (Message Exchange Pattern) like One-Way Message, Peer-to-Peer conversation or Request/Response interaction.

When a SOAP message passes on its way from the initial Sender to the Ultimate receiver, all the SOAP nodes between the sender and the receiver along with the receiver node process this SOAP message conforming to the SOAP Processing model Specification.

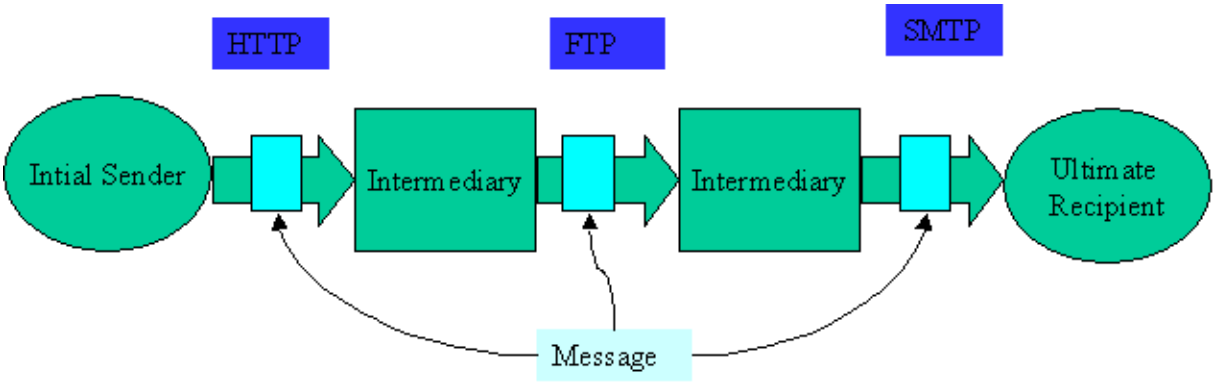

Figure 2.9: Soap Processing Model

After the reception of a SOAP Message a SOAP node tries to determine the parts of that message that is targeted to it. This determination is done by the use of "role" attribute of SOAP Header block.
If a SOAP node finds that the value of role attribute of a header block is the role that it assumes then this block is targeted to it. A role is specified in the header block using URI. SOAP defines 3 standard roles that can be assumed by a SOAP node. These standard roles are shown in Table 1.

Role Name	Role URI
Next	http://www.w3.org/2003/05/soap-envelope/role/next
Ultimate Receiver	http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver
None	http://www.w3.org/2003/05/soap-envelope/role/none

Table 1: SOAP defined roles

Beside these standard SOAP defined roles, an application can also define its own application specific role.

Once the SOAP node finds out all the parts of a SOAP message targeted to it, it will try to process them. SOAP nodes are not obliged to process or understand all the message parts targeted to it. SOAP specifies another attribute "misunderstand" for the header block. If this attribute's value is true in a header block, the targeted SOAP node is obliged to understand and process that block. This sort of header block is called mandatory header block. If a SOAP node finds a mandatory header block targeted to it, however it cannot process it according to the specification of the header block it will generate a soap fault and will cease the relaying of this message further.

After the successful processing of a header block, colloquially the SOAP node removes the block from outgoing message. However, there are circumstances where the node might have to retain the block in the outgoing message. One of the reasons for retaining a header block in the outgoing message is the semantics of the header block. That is if the specification of the header block mandates the presence of this header block in the forwarded message. Another reason is, if the header block contains the "relay" attribute with a value of true. As it is said before, a SOAP node is not obliged to process all the header blocks targeted to it. However, even if it does not process a non-mandatory header block targeted to it, it will remove the block from the outgoing message. If we want a non-mandatory header block to be processed by the first soap node that assumes a particular role and understands this block, we have to use this relay attribute in that header block with a value of true. In this case a soap node that assumes the role specified in the header block, even though cannot process the block, will retain it in the outgoing message.

2.4.4 SOAP Extensibility Model

SOAP Extensibility Model introduces a way of extending the core functionality of SOAP messaging Framework through SOAP Feature. In a word SOAP Feature is an extension of core SOAP messaging Framework. SOAP does not specify any constraint on the potential scope of SOAP Feature. Using SOAP Feature a number of different features like, security, reliability, correlation, and routing, can be introduced into the core SOAP Messaging Framework.

SOAP Features are expressed by two mechanisms of SOAP Extensibility Model.

i) SOAP Processing Model, which describes how a single SOAP node should behave regarding the processing of an individual message. SOAP features are expressed as SOAP header block within a SOAP Envelope. The syntax and semantics of one or more SOAP header block is referred to as SOAP module. That means, a SOAP module describes the syntax and semantics of zero or more SOAP features.

ii) SOAP Protocol Binding Framework, which describes how a SOAP node should send and receive SOAP Message using the underlying protocol. We will describe the SOAP Protocol Binding Framework in next section.

2.4.5 SOAP Protocol Binding

Soap message can be exchanged between two SOAP nodes using a number of different underlying protocols. For instance, it can be exchanged using the well-known HTTP, or using SMTP or even using TCP or UDP. SOAP protocol binding specifies how a SOAP message can be exchanged between two adjacent SOAP nodes using underlying protocol and how the requirements of a web service can be mapped to the capabilities of the underlying protocol. [45]

When a SOAP sender needs to send a SOAP message to another SOAP node, it first creates an abstract representation of the message using SOAP message elements and attributes. To send this abstract SOAP message over the wire to another SOAP node, the message has to be serialized in a specified way so that the recipient can deserialize it. SOAP protocol binding describes how a SOAP message will be serialized and deserialized for sending over the wire using an underlying protocol.

Besides providing a concrete realization of a SOAP message for sending over the wire from one SOAP node to another, SOAP protocol binding provides a mechanism for supporting features that might be needed by an application. A feature specifies certain functionality provided by protocol binding. For instance, HTTP supports request/response message exchange pattern. Therefore if an application requires RPC (request/response) style of SOAP message exchange and SOAP is

transported using HTTP it does not need any additional mechanism for performing this RPC style (request/response) of operation. However, if SOAP is transported using TCP protocol, which supports only one-way message exchange pattern, special mechanism must be provided by the SOAP protocol binding framework to correlate a request message with respective response message for a RPC style of operation. A feature can be referenced by the application using URI.

2.4 Motivation of Web services Security

How can we secure web service communication? There are a lot of existing security technologies that are used to secure Internet resources. One of the most widely used and proven security technologies is SSL. Most of the online shopping sites use SSL to secure their resources. Then can't we use SSL to secure web service? As most of the web service now a days use HTTP for communication, it seems perfectly possible to use SSL for securing web service. Unfortunately In the Web service world however, SSL does have some limitations [15]:

1) SSL/TLS provides point-to-point security [15]. SSL establishes a secure pipe between two adjacent communicating nodes and then transfers data using this pipe. While the data is in the pipe it is secured. However, once data came out of that pipe, it is in clear. That means the security context only exist between two adjacent nodes. Figure 2.10 shows the concept of Point-to-Point security. Hence if a message needs to pass one or more hops before reaching the ultimate destination, the message will be in clear at some point of time in each of the hop. Therefore, if an attacker can get access in one of these hops he can easily read or modify the message.

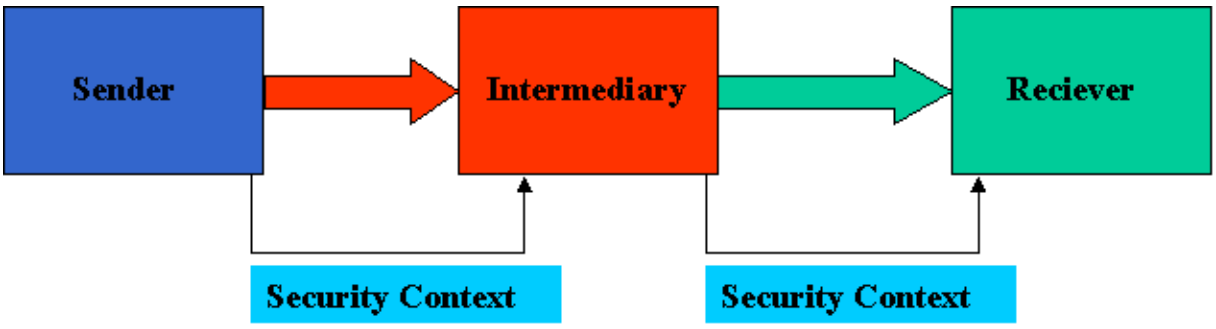

Figure 2.10: Point to Point Security

According to SOAP specification, a message can traverse one or more intermediaries before it is reached to its ultimate destination. Therefore, SSL is not suitable for securing this communication. To secure web service we need to secure the message not the link. That means, the security context for web service must exist end-to-end. End-to-end security is also called message level security as the security information for a message is embedded in the message itself. Figure 2.11 shows the concept of end-to-end security.

WS-Security is a specification that specifies how message level security can be achieved. We will describe this specification in a later chapter.

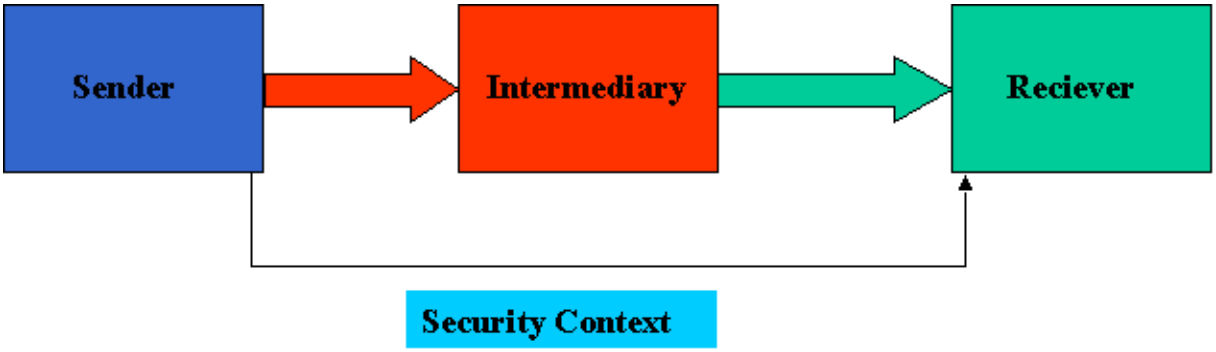

Figure 2.11: End to End Security

2) SSL is bound to HTTP [15]. As we saw previously that SOAP messages can be exchanged using a wide variety of protocols like FTP, SMTP, TCP, and HTTP etc. However SSL can only be used to secure HTTP communication. Therefore, SSL is not suitable for securing web service communication.

3) SSL does not support partial visibility [15]. Partial visibility means that part of a message will be encrypted and the rest will be in clear. SSL encrypts the whole message. However, partial visibility is a vital necessity in Web service communication. According to SOAP specification a sender can direct different parts of a message to different SOAP node. And its on the discretion of the sender which part of the message will be encrypted, which part will be signed and which part will be in clear. SSL does not provide this provision therefore is not suitable for web service communication security.

Due to the above mentioned limitations of traditional transport level security, there was a need for a different type of security mechanism for web service communication. Therefore, in April 2002 Microsoft, IBM and VeriSign proposed a security specification for web service communication. In April 2004 that standard was established as an approved OASIS open standard [14] for securing SOAP exchange. This standard is known as WS-Security now a day.

Chapter 3: State of the Art

3.1: XML Digital Signature

3.1.1 Definition

XML Digital Signature [2] provides a mechanism for signing partial or chosen element of an XML Document. This signature can be used by the recipient of the message for verifying the integrity of the message and the authenticity of the sender.
A number of different types of resources can be signed using XML Digital Signature. For Instance we can sign character-encoded data like HTML, binary-encoded data like Image, XML-encoded data using a single XML Signature.
At the time of XML Signature validation, the data object that has been signed needs to be accessible. XML Signature indicates the location of the signed data object in one of the following ways:

i) The signed data object is referenced using a reference URI within the XML Signature element
ii) The signed data object is a child of the XML Signature element. That means the Signed Object is inside the XML Signature Element.
iii) The Signed Data object contains the XML Signature Element , that contains its signature , within it . That is the Signed data object is the parent of its Signature element .

3.1.2 Structure

Figure 3.1 is the pictorial representation of the general structure of XML Digital Signature element

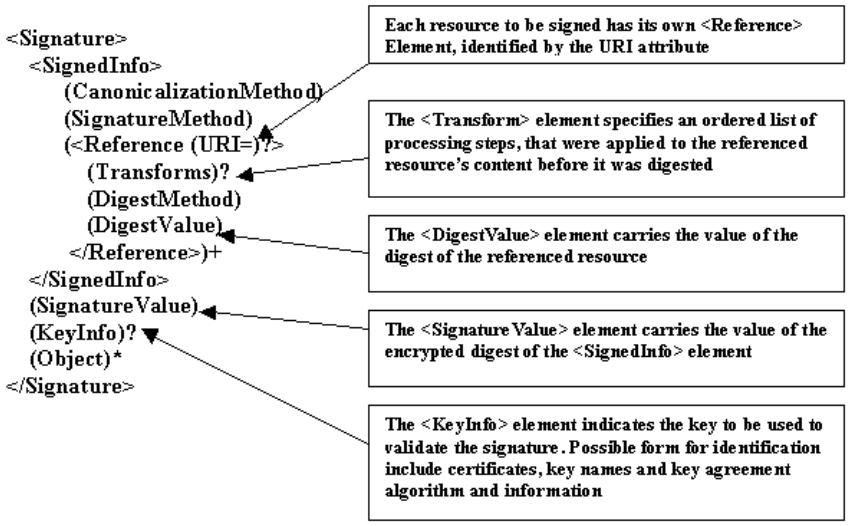

Figure 3.1: Structure of XML Digital Signature [41]

3.1.3 Processing

3.1.3.1 Signature Generation

The Signature generation process for XML Digital Signature can be broadly divided into two major steps.

i) Reference Generation
ii) Signature Generation

1) Reference Generation:

In this step, reference element is created for each data object that has to be signed. First necessary transformation is performed on the data object, then the digest value of the transformed data object is calculated and finally a reference element is created which references the data objects to be signed and includes the following elements:

i) Transformation elements to specify the applied transformation on the data object
ii) Algorithm that is used to generate the digest value of the data object
iii) The digest value of the data object

2) Signature Generation

In this step a <SignedInfo> element is created which includes the list of <ReferenceEelement> (from the Reference Generation step), a <SignatureMethod> element and a <CanonicalizationMethod> Element. The whole <SignedInfo> element is canonicalized* using the method specified within the <CanonicalizationMethod> Element. After that a signature value is generated over these canonicalized <SignedInfo> element using the algorithm specified under the <SignatureMethod> element. Finally a <Signature> element is created which includes the <SignedInfo> element(s) , a <KeyInfo> element and a <SignatureValue> element . The <KeyInfo> element represents the key used to generate the signature value, which is placed under the <Signature> element.

3.1.3.2 Signature Verification

We can categorize the signature validation procedure in two steps:

i) Reference validation
ii) Signature validation

1) Reference Validation:

In this step the digest value for each referenced data object is checked for validity. First the <SignedInfo> element is canonicalized using the Canonicalization method specified under the <SignedInfo> element. Then for each reference element the referenced data object is retrieved and a digest value is calculated on that data object using the digest method specified under the <Reference> Element. The resulting digest value is compared with the digest value specified under the <Reference> element . If these two values are same then the verification proceeds for the second reference element . Otherwise it generates an error message.

2) Signature Validation:

If the Reference Validation step passed successfully then comes the Signature Validation step. In this step the keying information, specified in the <KeyInfo> element of the <Signature> element is retrieved possibly from an external source. Then the Signature method is determined from the <SignatureMethod> element of the <SignedInfo> element. These two information that is , Keying Information and Signature Method , are used to validate the Signature value specified under the <SignatureValue> element of the <Signature> Element .

3.1.4 Limitations

XML represents information using a tree structure. XML Digital Signature allows non-contiguous objects of an XML dataset to be signed separately. The signed object may be referenced using an indirection (URI) by the Reference element of the Signature. This indirect referencing does not give any information regarding the actual location of the signed object. Therefore, the signed object can easily be relocated and the Signature value will still remain valid. In cases where the location of the data object is important in the interpretation of the semantics associated with the data, this can be exploited by an adversary to gain unauthorized access to protected resources [9]. This is the main limitation of XML Digital Signature.

*The term canonicalization refers to the process of transforming something to a form that conforms to a standard. Therefore,XML canonicalization specifies the process of transforming an XML document in a standard format. [43]

3.2 WS-Security

WS-Security [6] is an extension of SOAP [1] Messaging framework. WS-Security specification does not provide any new security protocol itself; instead it specifies how the prevalent security technologies can be used to secure SOAP messages. Authentication, Integrity and confidentiality are three vital features of any secure communication protocol. WS-Security specifies a standard way of achieving these features for SOAP messages. It provides a mechanism for attaching security tokens with SOAP messages. Security Tokens along with other mechanisms can be used to authenticate a client. WS-Security does not restrict applications to use a particular form of security token. However a wide variety of security token formats including binary security tokens like X.509 certificate or Koreros Ticket can be attached with SOAP messages using WS-Security standard. WS-Security provides a mechanism for encoding binary security tokens for attaching to a SOAP message.

Integrity of SOAP messages is provided using XML Digital Signature technology along with the Security token. Security token represents the key using which a portion of the SOAP message is signed. This key is referenced from the signature. As the SOAP message has to go through a number of intermediaries before it is reached to the recipient, it is perfectly valid for the intermediary to produce its own signature. WS-Security provides a way to attach multiple signatures in the same SOAP message. This Signature provides a way for the recipient and/or the intermediaries to ensure that a SOAP message has not been tampered on its travel path. WS-Security does not specify any particular signature technology to be used for signing a SOAP message. Instead a variety of signature technology can be used to sign different portion of the same SOAP message.

In the same way confidentiality of SOAP message is provided using XML Encryption mechanism in conjunction with Security token. The security token represents the key by which a portion of the SOAP message has been encrypted. The security token is referenced from the encryption element. As with XML Signature, WS-Security provides a way to encrypt different portions of a SOAP message by different SOAP actor (recipient/intermediary) and they can use different encryption technology to pursue this.

Limitations:

1) WS-Security uses XML Digital signature for signing non-contiguous parts of a SOAP message. Therefore, all the limitations of XML Digital signature are also applicable to WS-Security.
2) WS-Security allows multiple Security header (with the same name) to exist in the same SOAP message. This creates a pit fall and can be exploited by the attacker. We will see later how this feature can be exploited.
3) WS-Security does not propose any new security technology. However, it specifies how the existing security technology can be used to secure a SOAP message exchange.

4) WS-Security encompasses many other standards like XML Digital Signature, XML Encryption, X.509 certificates Kerberos ticket etc [14]. For this reason, the specification became quite complex.

3.3 WS-Policy

WS-Policy [42] is a generalized grammar for describing the capabilities, requirements, and characteristics of a Web service. [14] WS-Policy is not specific to any particular domain like Security, Reliable Messaging, Privacy or Quality of service. It can be used to express the requirements of a web service for a broad range of domain. However, WS-Policy does not itself specify how the policy will be associated with a particular web service. For this association a separate specification WS-Policy attachment has emerged.

Before using a Web service, the client needs to know the requirements of that service. For Instance, a particular Web service might require that all the request coming to it must satisfy at least one of the following requirements

i) Each request must contain either a User Name token or a X.509 token or both
ii) User Name token if present must be signed by AES algorithm
iii) X.509 token if present must be signed by Triple DES algorithm

The following figure shows how service requirements can be expressed using WS-Policy

```
<Wsp: Policy Id=”Policy1” Name=”MyPolicy”>
  <Wsp:OneOrMore>
    <wsp:All wsp:Preference="100">
      <wsse:SecurityToken TokenType="wsse:UsernameToken" />
      <wsse:Algorithm Type="wsse:AlgSignature"
URI=" http://www.w3.org/2000/09/xmlenc#aes"/>
    </wsp:All>
    <wsp:All wsp:Preference="1">
      <wsse:SecurityToken TokenType="wsse:X509v3" />
      <wsse:Algorithm Type="wsse:AlgEncryption"
            URI="http://www.w3.org/2001/04/xmlenc#3des-cbc"/>
    </wsp:All>
  </wsp:OneOrMore >
</wsp:Policy>
```

Figure 3.2: A Simple WS Policy

A policy is actually a collection of policy assertions. A policy assertion asserts some requirements of a service that a client request must meet in order to use that service. The assertions are wrapped into a policy element. As it can be seen from Figure 3.2, the policy element contains one policy operator <wsp:OneOrMore>. The policy operator can be used to create quite complex policy, for instance nested policy. In Figure 3.2 only two policy operators is shown. There are other policy operators specified in [42]. The operator <wsp:OneOrMore> means that at least one of its child assertions must be applicable. This operator contains two <wsp: All> operator. This operator means that all of its assertions must be met by the request to be identified as valid. It is not mandatory to use policy operator. We can specify all our assertions directly as the child element of the root policy element. In this case all of the child assertions must be met to get a service. In the figure above, each of the <wsp: All> operator contains two assertions within it. The first group of assertions specify that a request must contain a Username token and this token has to be signed by AES Signature algorithm.

The second group of assertions specifies that a request must contain a X.509 token and it has to be encrypted using Triple DES Encryption algorithm. As these groups of assertions are embedded within an <wsp:OneOrMore> operator, this means at least one of those groups of assertions must be met to get a service protected by this policy file.

Limitations:

WS Policy standard lacks semantics. It provides a mechanism for describing the syntactic aspects of service properties. This introduces a limitation on the policy specification and policy intersection. For example, a provider may specify that its service supports a particular algorithm for the adjustment of data retransmission timeout value and a consumer may define a policy requiring a different algorithm. It might be possible to substitute the required algorithm by the provided algorithm, if they are compatible. However, the current standard does not support this kind of relationship identification. Thus, although it is possible, the interaction between the provider and the consumer will not occur. [16]

3.4 WS-Security Policy

As we mentioned before WS-Policy [42] provides a general framework for representing web service constraints and requirements. However it does not specify any security assertion for any particular problem domain like reliable messaging or security. Each domain has its own specific assertion profile. WS-Security Policy [7] specifies policy assertions to represent the security requirements of a web service.

WS-Security policy specifies a number of security assertions like Security Token assertion, Integrity assertion, Confidentiality assertion, Visibility assertion, Message Age assertion and a lot more.

Security Token Assertion: Security Token Assertion specifies what sort of security token is required and accepted by a web service or Web service client. This assertion is applicable to both the request and response messages. [14] Security token assertion can be of much type like UserName Token assertion, X.509 Token assertion etc.

Integrity Assertion: Integrity Assertion specifies whether a SOAP message needs to be signed or not. It also specifies which parts of SOAP message is to be signed and also which algorithm should be used to sign it. This assertion is applicable to both the request and response SOAP messages.

Confidentiality Assertion: This assertion can be used by a Web service or its client to specify that parts of a request or response SOAP message must be encrypted. This assertion can also specify how the encryption should be done, that means which encryption algorithm should be used for the encryption.

Visibility Assertion: The Visibility assertion allows an intermediary to require that a certain portion of the SOAP message be visible to it. Visible means either in the clear or encrypted in a way that the intermediary is able to decrypt it. [14]

Message Age Assertion: The Message Age assertion is used to specify the time period, after which a message will be considered stale.

Limitations:

Securing a web service using WS-Security Policy is no panacea. It is essentially a domain specific language, which selects cryptographic communications protocols, uses low-level mechanisms that

build and check individual security headers. It gives freedom to invent new cryptographic protocols, which are hard to get right, in whatever guise. [17]

3.5 Formal Methods and Web services Security

The specifications for web services security are evolving day by day with an immense speed. Although none of these specifications offer any new security protocol, they provide a framework that integrates existing security protocols to secure web service communication. Therefore, it is necessary to identify whether these web services security specifications have achieved their security goals or not. This can be done by simulating different attack scenarios and verifying whether a security framework can guard against this attack. However, most of the time this brute force approach cannot reason about the security characteristics of a particular security framework. Because, the absence of a security flaw in a particular attack scenario does not specify that the security framework is not vulnerable in any other attack scenario. This type of reasoning regarding the security characteristics of a security framework can be achieved using format method. Therefore, formal methods are now applied to verify the security goals of web services security specifications. One of the earliest project that applied formal methods to verify the security goals of a web services security specification is Microsoft's SAMOA [20] project.

In one of the work of SAMOA project they have proposed TulaFale [22]. TulaFale is a scripting language that formally specifies web services security protocols and analyze their security vulnerability. TulaFale uses pi calculus to specify the interaction among concurrent processes. TulaFale extended pi calculus to include XML syntax and symbolic cryptographic operations for specifying the SOAP message exchange. For specifying the construction and verification of SOAP message, TulaFale uses Prolog-style predicate. The different security goals of a SOAP security specification are specified using assertions. Therefore, TulaFale can be summarized as follows:

TulaFale = Pi Calculus + XML Syntax + predicate + assertion [22]

In another work of SAMOA project they have proposed a language and two new tools [21]. The language they have proposed is a high level link specification language for specifying intended security goals for SOAP message exchange among SOAP processors. Then one of their tools compiles this link specification to generate WS-Security specifications. Then their other tool analyzes the generated WS-Security specifications using a theorem prover to verify whether the intended security goals can be achieved by the generated WS-Security specification. This analyzer uses TulaFale [22] script to specify a formal model for a set of SOAP processors and their security checks and to verify the security goals. According to them the policy-driven web service implementations are susceptible to the usual subtle vulnerabilities of traditional cryptographic protocols; their tools can help preventing such vulnerabilities by verifying the policy when it is being compiled from link specifications, and re-verifying the policy at the time of deployment against their original goals after any modifications [].

Another tool proposed by Microsoft's SAMOA project is WS Policy Advisor [9]. In this work they have identified the gap that exist between the formal model and its implementation. According to them, formal models are most of the time hand written. Therefore they might lack critical details. Consequently, proofs showing the absence of attack in a model do not directly reflect to its implementation. [9] For this reason they have proposed WS Policy Advisor, which will automatically extract TulaFale, model from WSE [32] configuration and policy files and will try to find security vulnerability in this model. We will discuss about this tool later in more detail.

Two recent specification of web services security are WS-Trust [28] and WS-Secure Conversation [29]. WS-Security provides essentially a standard for securing a single SOAP message. While in practice, a series of SOAP messages are exchanged between a client and a server. Therefore, using

WS-Security to secure the whole series of SOAP messages is not an efficient way. As for each SOAP message a security token needs to be generated and also verified. WS-Secure Conversation and WS-Trust specifies a way for securing a session between a client and a server instead of each single SOAP message. They have defined a security context token to establish trust between a client and a server. Another entity, WS Token Service, is introduced between the client and the server for the generation of security context token and to establish trust. This token service is essentially a web service. WS-Trust specifies how security context tokens are requested by client and generated by WS Token Service and WS-Secure Conversation specifies how one of this security context token can be used with SOAP message to secure a conversation. In another work of SAMOA project they have formally specified WS-Secure Conversations and WS-Trust specifications and verified their security characteristics using TulaFale scripting language. [22]

All of the above formalizations indeed make some assumptions. One of the assumptions, which is indeed a limitation of these formalizations, is that a message can be read, written or modified by an attacker if the attacker knows the right key. Otherwise the attacker cannot perform the attack. Later we will see that this is not true for all sort of attack. Moreover, a major limitation of the above formalizations is that they do not model insider attacks.

Chapter 4. Attack Scenarios in Web service

Web service is nothing but programmatic access to web sites. Therefore it is vulnerable to the same kind of attack applicable to traditional web sites such as, DOS (Denial of Service), SQL Injection attack etc. Moreover as web service uses XML data for exchanging messages, it is vulnerable to another class of attack known as XML Rewriting attack, which occurs due to the structural weakness of XML data. XML Rewriting attack is a common name for a range of attacks such as, replay, man-in-the-middle, redirection, dictionary attack. The attacker exploits the flexibility of the SOAP security extensibility to capture, manipulate and replay SOAP messages without violating the integrity of the message. In [9], [11], [12] and [13] different types of attack scenarios have been demonstrated. In this section we will present some attack scenarios in the context of SOAP message exchange to better understand XML Rewriting Attack. All of our example SOAP messages will be represented using a style adopted from [8]. Only the beginning tag of a SOAP element will be represented using the tag's full name. The full name of a element will be omitted from the end tag. For instance, an element A will be represented as <A>…</> instead of <A>…..</A>.
We will consider a simple SOAP-based server that responds to requests for a list of available airline tickets. The server charges a subscriber's account for each request and does not wish to respond to non-subscribers. Therefore, it requires that a message signature generated using an X.509 certificate belonging to one of its subscriber authenticate each request. Moreover, it requires that each request include a unique message identifier to be cached to detect message replay. The scenarios that we have demonstrated in this section are adapted from [9] and [11]

4.1 First Attack Scenario (Replay Attack):

```
<Envelope>
  <Header>
    <MessageID Id="Id-1">123</MessageID>
   <Security mustUnderstand="1">
      <BinarySecurityToken Id="Id-2">abcdefg….</ >
     <Signature>
         <SignedInfo>
           <CanonicalizationMethod Algorithm="….."/>
           <SignatureMethod Algorithm="….. " />
           <Reference URI="#Id-1">
             <DigestMethod Algorithm="#sha1" />
             <DigestValue>4AFDE67…</ >
           </ >
           <Reference URI="#Id-3">
             <DigestMethod Algorithm="…." />
             <DigestValue>EF346A….</ ></ ></ >
       <SignatureValue>34EADB98…</ >
       <KeyInfo>
         <SecurityTokenReference>
             <Reference URI="#Id-2" />
         </ ></ ></ ></ ></ >
  <Body Id="Id-3">
    <AirlineTicketRequest>...</ ></ >
</ >
```

Figure 4.1: SOAP message before Replay Attack

```
<Envelope>
  <Header>
    <Irrelevant>
    <MessageID Id="Id-1">123</ ></ >
    <MessageID>324</ >
   <Security mustUnderstand="1">
      <BinarySecurityToken   Id="Id-2"> abcdefg….</ >
     <Signature>
         <SignedInfo>
           <CanonicalizationMethod Algorithm="….. "/>
           <SignatureMethod Algorithm="...#rsa-sha1" />
           <Reference URI="#Id-1">
             <DigestMethod Algorithm="#sha1" />
             <DigestValue>4AFDE67…</ ></ >
           <Reference URI="#Id-3">
             <DigestMethod Algorithm="...#sha1" />
             <DigestValue>EF346A….</ ></ ></ >
       <SignatureValue>34EADB98…</ >
       <KeyInfo>
         <SecurityTokenReference>
             <Reference URI="#Id-2" />
         </ ></ ></ ></ ></ >
  <Body Id="Id-3">
    <AirlineTicketRequest>...</ >
  </ ></ >
```

Figure 4.2: SOAP message after Replay Attack

Suppose a client wants to see the list of airline tickets. As the client is charged for each request it makes, the request contains a Message ID, which is used by the server to keep track of client requests. Furthermore, MessageID of the request along with some other parts of the message are signed by the client using a X.509 certificate belonging to the client, which will help the server to authenticate the client and to check for the message integrity.

Figure 4.1 shows the SOAP message that is sent by the client to the server. An attacker, sitting in between the client and the server intercepts the message. He then put the <MessageID> header element into another header element <Irrelevant> and creates its own <MessageID> header element with a new value. He then sends the message to the server. Figure 4.2 shows the message the attacker sends to the server after interception and modification.
The Server on the other side, after receiving the message will try to validate the integrity of the message. It can be seen from Figure 4.2, although the attacker has tampered on the message, he did it in a way so that the integrity of the message remains valid.

4.2. Second Attack Scenario (Redirection Attack):

This scenario depicts how XML Rewriting attack can be used to redirect a SOAP request. Suppose a client wants to see the list of airline tickets. WS Addressing [44] specifies a way for embedding the URI of the ultimate recipient through the use of <To> header element. Figure 4.3 shows a request that the client sends to the airline ticket server with the URI of the server embedded into the request under the <To> element. The X.509 certificate of the client signs this <To> header element. The request also uses a <From> element, Specified in WS-Addressing [44] , to indicate the sender of the request. However, this element is not signed.

```
<Envelope>
  <Header>
    <From>http://www.client.com</ >
    <To id=”id4”>http://www.airlineticket.com</ >
   <Security mustUnderstand="1">
       <BinarySecurityToken  Id="Id-2">abcdefg….</ >
      <Signature>
           <SignedInfo>
             <Reference URI="#Id-4">
                <DigestMethod Algorithm="#sha1" />
                <DigestValue>4AFDE67…</ ></ >
             <Reference URI="#Id-3">
                <DigestMethod Algorithm="#sha1" />
                <DigestValue>4AFDE67…</ ></ >
          </ >
          <SignatureValue>34EADB98…</ >
        <KeyInfo>
            <SecurityTokenReference>
                 <Reference URI="#Id-2" /> </ >
        </ >
      </ >
   </ >
  </ >
  <Body Id="Id-3">
     <AirlineTicketRequest>...</ >
  </ >
</ >
```

Figure 4.3: SOAP message before Redirection Attack

```
<Envelope>
  <Header>
    <From>http://www.client.com</ >
    <To>http://www.stockquote.com</ >
    <Attack>
    <To id=”id4”>http://www.airlineticket.com</ >
    </ >
   <Security mustUnderstand="1">
    <BinarySecurityToken   Id="Id-2">abcdefg….</ >
      <Signature>
           <SignedInfo>
             <Reference URI="#Id-4">
                <DigestMethod Algorithm="#sha1" />
                <DigestValue>4AFDE67…</ ></ >
             <Reference URI="#Id-3">
                <DigestMethod Algorithm="#sha1" />
                <DigestValue>4AFDE67…</ ></ >
          </ >
          <SignatureValue>34EADB98…</ >
         <KeyInfo>
            <SecurityTokenReference>
                 <Reference URI="#Id-2" /></ >
        </ >
      </ >
   </ >
  </ >
  <Body Id="Id-3">
     <AirlineTicketRequest>...</ >
  </ >
</ >
```

Figure 4.4: SOAP message after Redirection Attack

Figure 4.4 shows the request modified and relayed by the attacker. The attacker simply puts the signed <To> element under an <Attack> element and introduced his own <To> element with a different URI. As we saw in the previous example, with this modification the integrity of the message will remain unchanged but the application logic will consider the <To> element introduced by the attacker instead of the <To> element that was wrapped by the <Attack> element and the message will be redirected to a different location instead of the location it was sent for.

4.3 Third Attack Scenario (Multiple Security Header Exploitation Attack):

According to WS-Security specification a SOAP Message can include more that one Security header. However, according to this specification no two Security headers can have the same role attribute value. As we saw before, client includes a signed <MessageID> in the request message to help the server to keep track of the client request. The server caches the <MessageID> value. Whenever the server receives a request from the client, it tries to find the <MessageID> value in the cache. If it finds one then it infers that someone, other than the intended client has replayed the message and will generate a SOAP fault. However, it is obvious that cache has a limited capacity. The server has to delete cache entries after a certain amount of time. WS-Security also specifies a <Timestamp> header to indicate the server the time limit after which the server can delete the cache entries associated with the current request. Figure 4.5 shows a client request with a signed <Timestamp> header.

```
<Envelope>
  <Header>
   <Security >
    <BinarySecurityToken   Id="Id-2">abcdefg….</ >
     <Signature>
          <SignedInfo>
            ……..
            <Reference URI="#Id-3">……</ >
            <Reference URI="#Id-4">……</ >
        </ >
        <SignatureValue>34EADB98…</ >
       <KeyInfo>…..</ >
    </ >
    <TimeStamp id = “id-4”>
         <Created>T1</ >
        <Expires>T2</ >
    </ >
   </ >
  </ >
  <Body Id="Id-3">
    <AirlineTicketRequest>...</ >
  </ >
</ >
```

Figure 4.5: SOAP message before Multiple Security Header Exploitation Attack

```
<Envelope>
  <Header>
   <Security >
    <BinarySecurityToken   Id="Id-2">abcdefg….</ >
     <Signature>
          <SignedInfo>
            ……..
            <Reference URI="#Id-3">……</ >
            <Reference URI="#Id-4">……</ >
        </ >
        <SignatureValue>34EADB98…</ >
       <KeyInfo>……</ >
    </ >
    <TimeStamp>
         <Created>T1</ >
        <Expires>T2</ >
    </ >
   </ >
   <Security role=”none” mustUnderstand=”0”>
    <TimeStamp id = “id-4”>
         <Created>T1</ >
        <Expires>T2</ >
    </ >
   </ >
  </ >
  <Body Id="Id-3">
    <AirlineTicketRequest>...</ >
  </ >
</ >
```

Figure 4.6: SOAP message after Multiple Security Header Exploitation Attack

Figure 4.6 depicts how an attacker can exploit the flexibility of WS Security specification. When the client sends the message to the server, the attacker intercepts the message and wait until the <TimeStamp> value is expired. Then he creates a new Security header with role attribute's value "None" and "mustUnderstand" attribute's value "False". Then he cut the <TimeStamp> element from the original security header and put it in the newly created security header. He then creates his own <Timestamp> header element and put it under the original security header. Then he sends this message to the server. The Soap processor on the server side will try to process the message and will ignore the Security header created by the adversary silently. Signature validation will also pass successfully as the Signature element references the <Timestamp> header element using XPointer, which does not take care of the location of the referenced element. Therefore, even though the attacker moved the <Timestamp> header element under a different <Security> header element it will resolve it as if it was not moved. Consequently The SOAP Processor will take the <Timestamp> element created by the attacker into account. As the server has deleted the cache of the client request due to the Timestamp expiration, it will not be able to find the <MessageID> of the attacker's message in its cache and will consider the request as a fresh request from the client. Therefore the server will process the request as if it was sent from the intended client.

Chapter 5: Previous Solutions of XML Rewriting Attack

The attack scenarios presented in the last section does not cover all of the vulnerabilities that can arise in web service communication. A lot of work is going on to detect and remove this sort of attack. Although simple at a first glance, XML Rewriting attack is not something that can be easily detected and removed. In this section we will describe some works that have been done previously in order to detect XML Rewriting Attack. We will discuss our approach for the detection of this attack in next chapter.

5.1 XPath referencing with WS-Security policy:

If used correctly WS-Security policy can act as a countermeasure against XML-Rewriting attack. We have seen previously what is WS-Security Policy [7]. As we saw before, web service operations are published using a WSDL file. This file contains the name of the operations the server offers and the input and output parameters required and produced by those operations. Security policy can be associated with these operations and/or with the input and output messages. These policies specify the security requirements of a web service. However, it is quite difficult to specify all possible security requirements in WS-Security policy file. The author and the implementer of the security policy need to be very careful while writing and implementing the policy. In [11] the author have shown different sort of rewriting attacks and the associated policy files for the detection of these attacks. They have also shown how the attacker can take advantages of security policy hole in order to get unauthorized access to system resources. According to [11] in practice the semantics of XML elements depends on its location. However, XML Signature provides for the referencing of an element independent of its location. They further showed that the flexibility that SOAP header provides can be exploited by an attacker in a naïve way. Lets consider the following example, which is taken from [11].

```
< Envelope>
 < Header>
  < Security>
   < Signature>
        < SignedInfo>
        < Reference URI="#theBody"></ >
        < Reference URI="#theReplyTo">
        </ ></ ></ ></ >
  <ReplyTo Id =”theReplyTo>
       < Address>http://good.com/</ >
  </ ></ >
 <Body  Id = "theBody”>
   <getQuote Symbol=”IBM”/></ ></ >
```

Figure 5.1: SOAP message with a ReplyTo header block signed and referenced using URI

a) The element specified by /Evelope/ Body must be referenced from a signature “S” using WSS with XML Signature, and

b) if present, any element matching / Envelope/ Header/ReplyTo must be referenced from a signature “S” using WSS with XML Signature, and

c) the signature “S” verification key must be provided by an X.509v3 certificate issued by one of a set of trusted Certificate Authorities (CAs).

Figure 5.2: Security Policy assertions for SOAP message of Figure 5.1

Figure 5.1 shows a client request to a stock quote application. This application takes a client request and returns the quote for the symbol specified as <getQuote@Symbol>. The request also includes a ReplyTo header element, which is specified in WS-Addressing. This element specifies where the

response should be sent. Moreover, this ReplyTo element is optional. If it is not specified in the request the response will be sent to location from where the request has come.
Figure 5.2 shows a sample security policy requirements of the stock quote application. It says that the request must contain a body element. This body element has to be referenced from the signature element that means the body element must be signed. The request may contain a Reply To element and if it contains a Reply To element then this Reply To element must be signed. Finally, it says that the signature must be verified using a X.509 certificate generated by a Certificate Authority (CA).

```
<Envelope>
   < Header>
    < Security>
     < Signature>
        < SignedInfo>
            < Reference URI="#theBody"></ >
           < Reference URI="#theReplyTo"></ >
       </ ></ ></ >
    <wrapper>
       < ReplyTo Id = ”theReplyTo>
         <Address>http://good.com/</ >
       </ ></ ></ >
  <Body  Id = "theBody”>
    <getQuote Symbol=”IBM”/>
  </ ></ >
```

Figure 5.3: SOAP message where the attacker wrapped up the ReplyTo header block

At first sight, it might seem that the security policy in Figure 5.2 is sufficient for the stock quote application. But ironically it is not. Figure 5.3 shows how the attacker can bypass the security requirements specified in Figure 5.2 to redirect the response.
As we said before the <ReplyTo> header element is optional. Therefore the attacker can intercept the client request shown in Figure 5.1 and wrap up the <ReplyTo> header element by a <wrapper> element. This modification does not violate the security requirements of the application. The modified message of Figure 5.3 contains a <Body> element, which is signed. It does not contain any <ReplyTo> element as it is now wrapped under a <wrapper> element and according to SOAP specification if a SOAP processor does not understand a header element it will ignore it silently. Therefore, when the stock quote application will receive the request of Figure 5.3 it will discard the <wrapper> element. Moreover, a X.509 certificate generated by a CA can verify the signature. As all the security requirements are fulfilled the application will process the message as if it was not tampered by an attacker and will send the response from where it got the request, in the above case to the attacker.

A deep investigation of the above scenario reveals the following reasons for this sort of vulnerabilities:

1) SOAP Extensibility model and its processing rule allows optional header elements to be added to the header of a SOAP message. This feature provides for the better interoperability. Application can use their own header element without any negotiation. On the other hand this flexibility can be exploited by the attacker.
2) The reference from the signature element is using XPointer to reference the signed element, which does not take care of the location of the signed element.

In [11] they have proposed the use of Xpath expression for the removal of this attack. This Xpath uses absolute path of an element to reference it. Figure 5.4 and 5.5 shows the request presented in 5.1 using Xpath for the reference and the new security requirements.

```
<Envelope>
    < Header>
      < Security>
       < Signature>
         < SignedInfo>
          < Reference URI="#theBody"></ >
          < Reference URI="">
          < Transforms>
             < Transform Algorithm="...">
              <XPath>
                 /Envelope/ Header/ReplyTo
             </ ></ ></ ></ ></ ></ ></ >
    <ReplyTo Id = ”theReplyTo>
        <Address>http://good.com/</ >
    </ ></ >
< Body  Id = "theBody”>
<getQuote Symbol=”IBM”/>
</ ></ >
```

Figure 5.4: SOAP message with a ReplyTo header block signed and referenced using XPath

a) the element specified by /Evelope/Body must be referenced from a signature “S” using WSS with XML Signature, and

b) if present, any element matching /Envelope/Header/ ReplyTo must be referenced via an absolute path XPath expression from a signature “S” using WSS with XML Signature, and

c) the signature “S” verification key must be provided by an X.509v3 certificate issued by one of a set of trusted Certificate Authorities (CAs).

Figure 5.5: Security Policy assertions for the SOAP message of Figure 5.4

In [11] they have named the problem specified above as Optional Element context problem as it is exploiting the optional header element feature of SOAP. Although using Xpath expression for referencing signed element can detect the above sort of problem it is not a panacea. SOAP supports multiple security headers in a SOAP message. An attacker can exploit this SOAP feature to violate the above solution. Lets consider the following examples:

```
<Envelope >
  <Header>
    <Security>
      <Signature>
      <SignedInfo>
        <Reference URI="#theBody"></>
        <Reference URI="">
          < Transforms>
            < Transform Algorithm="...">
              < XPath >
                /Envelope/Header/ Security/Timestamp
             </ ></ ></ ></ ></ ></ >
      < Timestamp Id="theTimestamp">
       < Created>2005-05-29T08:45:00Z</ >
       < Expires>2005-05-29T09:00:00Z</ >
      </ ></ ></ >
  <Body  Id="theBody”>
    <getQuote Symbol=”IBM”/></ ></ >
```

Figure 5.6: SOAP message with a Timestamp element signed and referenced using XPath

a) the element specified by /Evelope/Body must be referenced from a signature “S” using WSS with XML Signature, and

b) if present, any element matching /Envelope/Header/ReplyTo must be referenced via an absolute path XPath expression from a signature “S” using WSS with XML Signature, and

c) if present, any element matching /Envelope/Header/ Security/Timestamp must be referenced via an absolute path Xpath expression from a signature “S” using WSS with XMLSignature, and

d) the signature “S” verification key must be provided by an X.509v3 certificate issued by one of a set of trustedCertificate Authorities (CAs).

Figure 5.7: Security Policy assertions for SOAP message of Figure 5.6

Figure 5.6 shows a sample SOAP message that includes a Security header. This Security header among other information contains a Signed Timestamp element. Figure 5.7 shows a WS Security policy for the application server. In this policy it is specified that the Timestamp element is optional . However if it is present then it must be signed. Moreover, it also specifies that the Timestamp element if present must be located under /Envelope/Header/Security. Now if an attacker wrap up the Timestamp element under another Fake element , the policy of the application server will easily detect it. Nevertheless, it is still possible for an attacker to fool these policies. Figure 5.8 shows how this can be accomplished.

In Figure 5.8 the attacker created a new Security header, which conforms to the WS Security specification. However, according to WS Security specification although a SOAP Message may contain multiple security header, no two security header can have the same role attribute value. Therefore, the new Security header created by the attacker has a role attribute's value "none". It means no SOAP node should process this header. Then the attacker cut and pastes the Timestamp element from the existing Security header to the newly created Security header. After that he/she created a new Timestamp element under the existing security header.

```
<Envelope >
  <Header>
    <Security role=”none” mustUnderstand=”false”>
     <Timestamp Id="theTimestamp">
      <Created>2005-05-29T08:45:00Z</ >
      <Expires>2005-05-29T09:00:00Z</ >
     </ >
    </>
    <Security>
      <Signature>
      <SignedInfo>
        <Reference URI="#theBody"></>
       <Reference URI="">
          <Transforms>
            <Transform Algorithm="">
              < XPath>
                  /Envelope/ Header/ Security/Timestamp
             </ ></ ></ ></ ></ ></ >
     <Timestamp >
      <Created>2006-05-29T08:45:00Z</ >
     <Expires>2006-05-29T09:00:00Z</ >
    </ ></ ></ >
< Body Id="theBody”>
<getQuote Symbol=”IBM”/></ ></ >
```

Figure 5.8: SOAP message where the attacker created a Security Header block with role attribute's value none

When the SOAP Processor on the receiver side will receive the message it does not have any reason to make any complain as the request fully conforms to the security policy. The Signature validator as well will find everything valid as the Xpath expression
"/Envelope/Header/Security/Timestamp"
will resolve to a set that will contain the signed Timestamp element. In [11] they have proposed to include not only the name of the parent element in the Xpath expression but also some attribute values that will uniquely identify the parent of the signed element. For instance, instead of the XPath expression specified above, they proposed to use the following expression:

“/Envelope/ Header/ Security[@role=”.../ultimateReceiver”]/Timestamp”

However, the above solution is not enough to detect all rewriting attack that can take place using the above weakness of XML Digital Signature. They left this problem as a future work.

Besides they have mentioned other two sorts of context problems namely, Simple ancestory context and Sibling value context problem. The former type of problem occurs when the attacker changes the parent of a signed element. This problem is also easily detectable with Xpath referencing and WS Security policy. The sibling value context problem occurs if the application logic is dependent on the order of the signed element. Then the attacker can simply change the order of signed elements and this tampering will not get detected, as Xpath does not provide any information for the sibling of a signed element.

5.2 WSE Policy Advisor

Microsoft’s WSE [32] or Web service Enhancement is an implementation of many WS-* specification. It provides a library to create and process SOAP messages and headers. Most of the web service and their client are written and compiled using strongly typed language.[9] For the ease of parameter adjustment after the deployment of a web service , configuration files are used. These configuration files are loaded by the web service and their clients for the enforcement of the newly created or changed parameters without the recompilation of their code. In the case of WSE, WS-Security policy is part of this configuration file. We already showed that WS-Security policies are vulnerable to XML Rewriting attack if not written correctly.

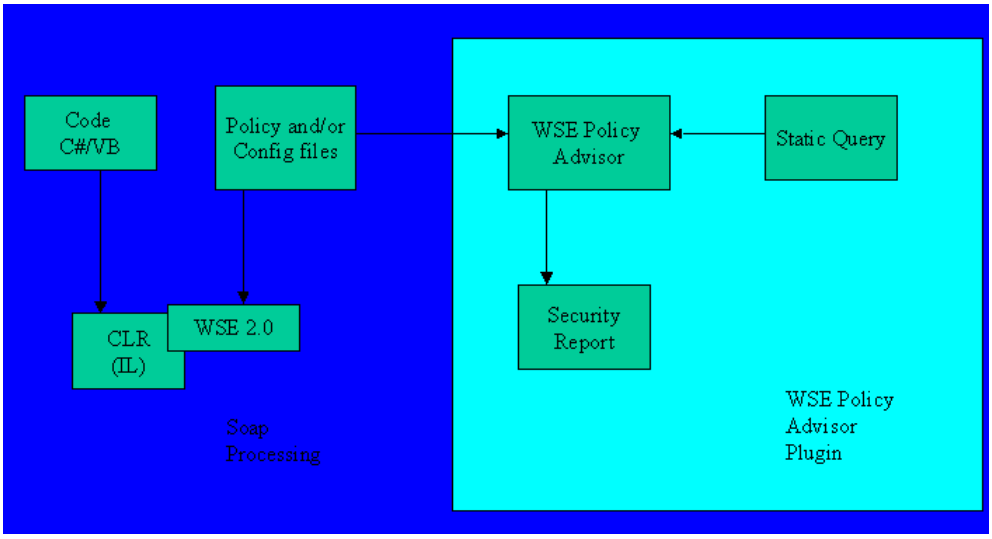

Figure 5.9: WSE Policy Advisor [9]

WSE Policy Advisor is a rule-based tool for detecting typical errors in WSE configuration and policy files. Figure 5.9 shows how this tool works. It takes the policy and configuration files of WSE, runs some static queries on them and generates security reports and remedial actions for security flaws. This tool has more than 30 queries. These queries check for some syntactic conditions of those policy files.

These syntactic conditions are determined by security reviews of the policy and configuration files of WSE. If these security conditions are not met by the policy files the tool generate a report stating the threat that might occur due to this missing syntactic conditions. It also generates a remedial action that can be used by the author of the policy files to fix the flaw. For instance, if the policy configuration file specifies that the MessageID element is optional for a request message or it is not necessary for the MessageID element to be signed, the policy advisor will throw a warning (after running its query on this policy configuration file) stating the possible existence of replay attack and will suggest to make MessageID element mandatory and signed for a request message.

Although WSE Policy Advisor can detect errors that otherwise might be overlooked, it has the following drawbacks:

i) WSE Policy Advisor does not provide any formal guarantees. It only provides a suggestion regarding possible flaws in policy configuration files found by running some queries.
ii) WSE Policy advisor shows very poor performance if the policy configuration file becomes complex.
iii) The queries that are run by WSE Policy advisor cannot detect possible existence of signed element reordering attack.

5.3 SOAP Account

In [12] and [13] the author has proposed an approach for the detection of XML Rewriting attack. They have introduced a new header element SOAP Account. This header element will keep different information of the SOAP message. The structure of their proposed SOAP Account is given in Figure 5.10.

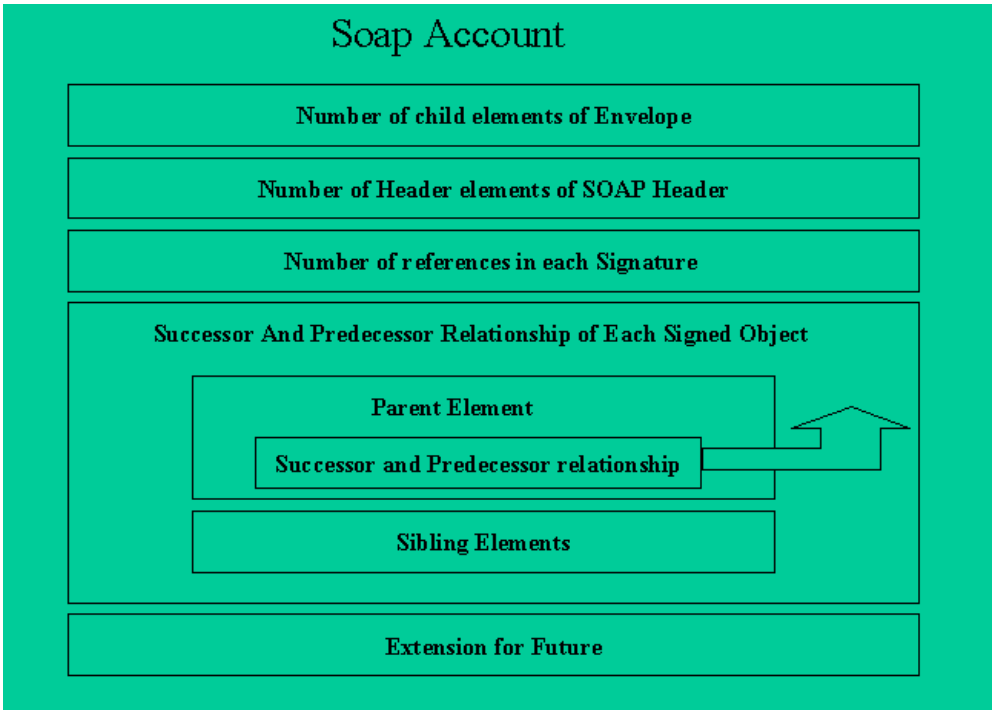

Figure 5.10: Structure of SOAP Account [13]

The SOAP Account contains the following information of the SOAP message:

i) How many elements does the envelope of the SOAP message contain
ii) How many Header elements are there in the Header of the SOAP message.
iii) How many Signed elements are their in the SOAP message
iv) The immediate parent of each Signed element
v) The successor of each signed element
vi) The siblings of each Signed element

In conjunction with these elements they have kept a field for future extensions in the SOAP account. They have created a module that before sending the SOAP message calculates the information necessary for the SOAP account and appends a SOAP account header element in the Header or in the Body of the SOAP message. As the SOAP message can pass through one or more intermediaries on its way to the ultimate recipient, the intermediaries are allowed to append its own SOAP account after the processing of the SOAP message. The author has imposed a restriction

that the SOAP account must be signed by the creator using its X.509 certificate or by some other method. Each successive SOAP node must sign its own SOAP account concatenated with the Signature of the previous node.

```
<Envelope>
  <Header>
    <MessageID Id="Id-1">123</MessageID>
   <Security mustUnderstand="1">
      <BinarySecurityToken    Id="Id-2">
                  abcdefg….</ >
      <Signature>
          <SignedInfo>
           …….
            <Reference URI="#Id-1">….</ >
            <Reference URI="#Id-3">….</ >
             <Reference URI="#Id-4">….</ >
        <SignatureValue>34EADB98…</ >
       <KeyInfo>
          <SecurityTokenReference>
              <Reference URI="#Id-2" />
          </ ></ ></ ></ >
    <SoapAccount>
    <NoChildOfEnvelope>2</>
    <NoChildOfHeader>3</>
    <NoOfSigned>3</>
    <ParentOfId1>Header</>
    <ParentOfId3>Envelope</>
    <ParentOfId4>Header</>
    </>
   </ >
  <Body Id="Id-3">
     <AirlineTicketRequest>...</ ></ >
</ >
```

Figure 5.11: SOAP message with SOAP Account header block

```
<Envelope>
  <Header>
   <Attack>
    <MessageID Id="Id-1">123</MessageID>
   </Attack>
   <MessageID Id="Id-1">234</MessageID>
   <Security mustUnderstand="1">
      <BinarySecurityToken    Id="Id-2">
                  abcdefg….</ >
      <Signature>
          <SignedInfo>
           …….
            <Reference URI="#Id-1">….</ >
            <Reference URI="#Id-3">….</ >
             <Reference URI="#Id-4">….</ >
        <SignatureValue>34EADB98…</ >
       <KeyInfo>
          <SecurityTokenReference>
              <Reference URI="#Id-2" />
          </ ></ ></ ></ >
    <SoapAccount>
    <NoChildOfEnvelope>2</>
    <NoChildOfHeader>3</>
    <NoOfSigned>3</>
    <ParentOfId1>Header</>
    <ParentOfId3>Envelope</>
    <ParentOfId4>Header</>
    </>
   </ >
  <Body Id="Id-3">
     <AirlineTicketRequest>...</ ></ >
</ >
```

Figure 5.12: SOAP Account attack detection

Figure 5.11 and 5.12 demonstrates how this SOAP account can be used to detect XML rewriting attack on the SOAP message of Figure 4.1. As it can be seen from figure 5.11, whenever the attacker relocates the MessageID element of the SOAP message into an Attack element, the structural information of the changed SOAP message does not match the information present in the SOAP account header element. Moreover, It is required that the SOAP account must be signed. Therefore the attacker cannot modify this SOAP account. Nevertheless, there is a possibility that the attacker can delete the SOAP Account completely. To prevent this possibility the WS-Security policy can be used. This policy will check for the presence of SOAP Account header element. If it does not find one it will generate error. That means, on the receiver side, a security policy must be present, which will check for the following:

i) SOAP account element is present under either the Body element or the Header element of the SOAP message
ii) SOAP account element is signed which can be verified by a certificate issued by a trusted authority

The approach presented here can successfully detect a wide range of XML rewriting attack. However, it is not the cure for all types of rewriting attacks that can take place. We can summarize our analysis on this SOAP account approach by the following points.

i) It does not include any mechanism to detect the replay attack. Although Message ID or Timestamp proposed by the WS Security can be used for this detection, we should consider the fact that these elements are optional. It is perfectly valid for a SOAP message to not include a Message ID or Timestamp. In that case, even though the SOAP message contains a SOAP Account element, it is prone to XML Rewriting attack.
ii) The approach does not include any mechanism that can uniquely identify the parent of the Signed element. Therefore, the attacker to gain unauthorized access to protected resources can use this unawareness of SOAP account. We will demonstrate this problem later.
iii) The SOAP account itself is prone to XML Rewriting attack. It is specified that the receiver should check for the presence of the SOAP account element after receiving the SOAP message. However, as we said before, the intermediaries can append its own SOAP Account element in the SOAP message. Therefore the number of SOAP account element in a SOAP message is not fixed. For this reason it is not possible to specify in security policy, how many SOAP account element must be present in a SOAP message. The attacker can exploit this problem. He can just cut one of the several SOAP account elements of the SOAP message and paste it into a header element that is not signed and make the role attribute of the header element none and mustUnderstand attribute to false. Then this header element will not be processed by the ultimate recipient or by any of the intermediaries. However during the signature validation the reference of the relocated will be found as it is not removed but relocated.
iv) In SOAP account one of the field is used to keep track of the siblings of the Signed element. However, according to SOAP specification, an intermediary can append its own element in any place of a SOAP message. Therefore this sibling information might change from node to node. It is not specified how this change can be detected by the ultimate receiver at the time of validation of the message.
v) In SOAP account, there is a field that keeps track of the successor of a signed element. According to us this information does not have any role in the process of validation of a SOAP message. XML Digital signature actually signs the Digest value of an XML element. The digest value is calculated on the sub tree rooted at the element that is to be signed. Therefore, if an element is signed all of its children are signed implicitly.

Figure 5.13 and 5.14 are showing how SOAP account approach can become vulnerable to XML Rewriting attack. Figure 5.13 shows a request SOAP message. This request contains the following:

i) A Timestamp element under the Security header. This element is signed
ii) An Option element which is an Optional header element specific to some application
iii) A Body element which is signed
iv) A SOAP Account header element. This header element contains information regarding the SOAP message. Moreover, this header element is signed.

```
< Envelope>
< Header>
        < Security>
          < BinarySecurityToken  Id="4"></ >
          <Signature>
            < SignedInfo>
              < Reference URI="#1">…</ >
              < Reference URI="#2"></ >
             <Reference URI="# 3 "></ >
            </ >
           <SignatureValue>abcfd….</ >
           < KeyInfo>…</ >
         </ >
         < Timestamp Id="3">
           < Created>t1</ >
           < Expires>t2</ >
        </ ></ >
        <Option>abdbdbd</ >
        <SOAPAccount ID = “2”>
            <NoChildOfEnvelope>2</>
            <NoOfHeader>3</>
            <NoOfReferences>3</>
            <ParentOfID1>Envelope</>
            <ParentOfID2>Header</>
            <ParentOfID3>Security</>
            <SiblingOfID1>Header</>
            < SiblingOfID2> Security,Option</>
            < SiblingOfID3>Signature,BST</>
        </ ></ >
< Body  Id="1">..<getQuote Symbol=”IBM”/>
</ ></ >
```

Figure 5.13: SOAP message with a SOAP Account header block and before modification by attacker

```
< Envelope>
< Header>
        < Security>
          < BinarySecurityToken  Id="4"></ >
         < Signature>
           < SignedInfo>
             < Reference URI="#1">…</ >
             < Reference URI="#2"></ >
             < Reference URI="# 3 "></ >
            </ >
            < SignatureValue> abcfd….</ >
            < KeyInfo></ >
          </ >
          < Timestamp>
             < Created>t3</wsu:Created>
             < Expires>t4</wsu:Expires>
           </ ></ >
        <Option role=”none” mustUnderstand = “0”>
        < Security >
              < BinarySecurityToken/>
              < Signature/>
              < Timestamp Id="3">
                < Created>t1</ >
                <wsu:Expires>t2</ >
              </ ></ >
        </>
          <SOAPAccount ID = “2”>
            <NoChildOfEnvelope>2</>
            <NoOfHeader>3</>
            <NoOfReferences>3</>
            <ParentOfID1>Envelope</>
            <ParentOfID2>Header</>
            <ParentOfID3>Security</>
            <SiblingOfID1>Header</>
           < SiblingOfID2> Security,Option</>
           < SiblingOfID3>Signature,BST</>
        </ ></ >
  < Body Id="1">..<getQuote Symbol=”IBM”/>
</ ></ >
```

Figure 5.14: SOAP Account Vulnerability

Figure 5.14 shows how the attacker can modify the SOAP message and still keep the SOAP account information unchanged. He does this in the following way:

i) He creates a role attribute for the Option header element and sets its value as none. He also creates a “mustUnderstand” attribute for the same header element and sets its value to false.
ii) Under this Option header element he creates a Security Element. He cut and pastes the Timestamp element from the original Security header element to the newly created Security element.
iii) He then creates a BinarySecurityToken and Signature element under the newly created Security header.

iv) He creates his own Timestamp element under the original Security header element.

When the above SOAP message will be received by the service provider, the SOAP Account validation module will try to verify the message. It will not be able to detect any Tampering as none of the information that it contains has been changed. So it will pass the message to the next step for Signature validation. This step will also pass successfully as the signed Timestamp element has not been removed instead it has been relocated. The SOAP Processor will not process the Option header element as it has a role attribute value of none and the processor is not obliged to understand this header element as this element has a "mustUnderstand" attribute value of false. Therefore the request will be processed normally and the attacker's target will be fulfilled.

We should note here that the above attack is not the only one that can be done to jeopardize the SOAP account security. As we said before the SOAP account itself is vulnerable to XML Rewriting attack. How it happens can easily be understood from Figure 5.13 and 5.14. However, it should be noted that the SOAP account itself could be vulnerable to XML Rewriting attack whenever a SOAP message contains multiple SOAP Account header element.

Chapter 6: Proposed Method

In the last chapter we saw different solutions proposed for the detection of XML Rewriting attack. We saw how they work. However, we also saw that none of the solutions could properly remove the XML Rewriting attack. While they are handling one sort of XML Rewriting attack properly at the same time they fail to take care of some other type of XML Rewriting attack. The first solution, using XPath expression with WS Security policy, can remove some XML Rewriting attack, but it fails when there are multiple header elements with the same name under the Header of the SOAP message. It also could not remove the sibling value-reordering problem, when the order of the signed elements is changed to perform an attack. The second solution, WS Policy advisor is quite efficient in dealing with XML rewriting attack. However, they also have shortcomings. The third solution SOAP account approach can deal with a wide range of XML Rewriting attack, but as we saw it is also vulnerable in some cases.
Therefore, taking the shortcomings of all the discussed solutions into consideration in this section we will propose a solution for the detection and removal of XML Rewriting attack. We will show that our proposed solution can detect XML rewriting attacks in cases where all the discussed solutions are failed to do so.

Each SOAP message header element can contain a role attribute according to the SOAP specification. There are three roles defined in SOAP specification, UltimateReciever, Next and None. In our discussion of SOAP before we saw what do these roles mean. We also saw that each SOAP node assumes a role while processing a SOAP message. Beside these three roles each application can define its application specific roles. However, in discussing our method we will only consider the roles, specified in SOAP.

6.1 Problem Analysis

In XML Rewriting attack it is implicitly assumed that the attacker does not have the capability of signing an element. Therefore, he cannot edit, create or delete a signed element. However, he modifies the SOAP message in a way that does not violate the underlying signature. The question then arises, why don't we just sign the whole header and body of the SOAP message. It is obvious that if we could have done this, it would have solved the XML Rewriting attack problem. The attacker would not be able to modify any header element or body element of the SOAP message, as he cannot sign them. However, signing the whole header element is not possible due to the existence of intermediaries in SOAP message path. According to SOAP specification the intermediaries must be allowed to add its header block under the SOAP header. However, if we sign the whole header, the intermediaries will not be able to add its data in the SOAP message. We have tried to summarize the type modifications that can be done by the attacker to perform a rewriting attack in the following points:

i) Wrapping an element into another new Fake element. Here the attacker creates a new Fake element taking the flexibility of the SOAP extensibility model, which allows for any header elements to be added in the SOAP Header. Then he cut a signed element and pastes it under this newly created Fake element
ii) Moving an element under another existing Header element. Here the attacker takes the benefit of SOAP extensibility model which specifies that multiple header element might have the same name
iii) Reorder the signed elements among them. As no information is available regarding the relative position of a SOAP element the attacker can very easily do this. This change might create problem for application that depends on the order of the Signed elements.
iv) The attacker can copy a message and then relay it later. This is the replay attack.

v) The attacker can create his/her own header element under the SOAP Header. If the application does not have proper security protection this can give him/her unauthorized access to system resources.

vi) The attacker can delete or edit an unsigned header element

Hence, we see that all of the attacks except the fourth one, are some kind of modification. Now we need to find out, after an attack has been done what sort of information we can get from the SOAP message to detect the attack. As we already said XML represents its data in a tree like structure. SOAP is nothing but an XML document. Therefore we can also represent SOAP messages using a tree like structure.

Figure 6.1 shows the tree representation of the SOAP message previously presented in Figure 5.1. The tree representation of a SOAP message will always have an envelope as its root. The root can have one or two children. A Body and a Header. In this figure the Header has two children a Security and a ReplyTo and the Body has one child getQuote. The Security has one child; the ReplyTo also has one child and so on.

Now lets try to analyze what happens to this tree when the attacker performs the attack. Lets define the following term:

Depth: The depth of an element in a tree is the length of the path from the root of the tree to the element.

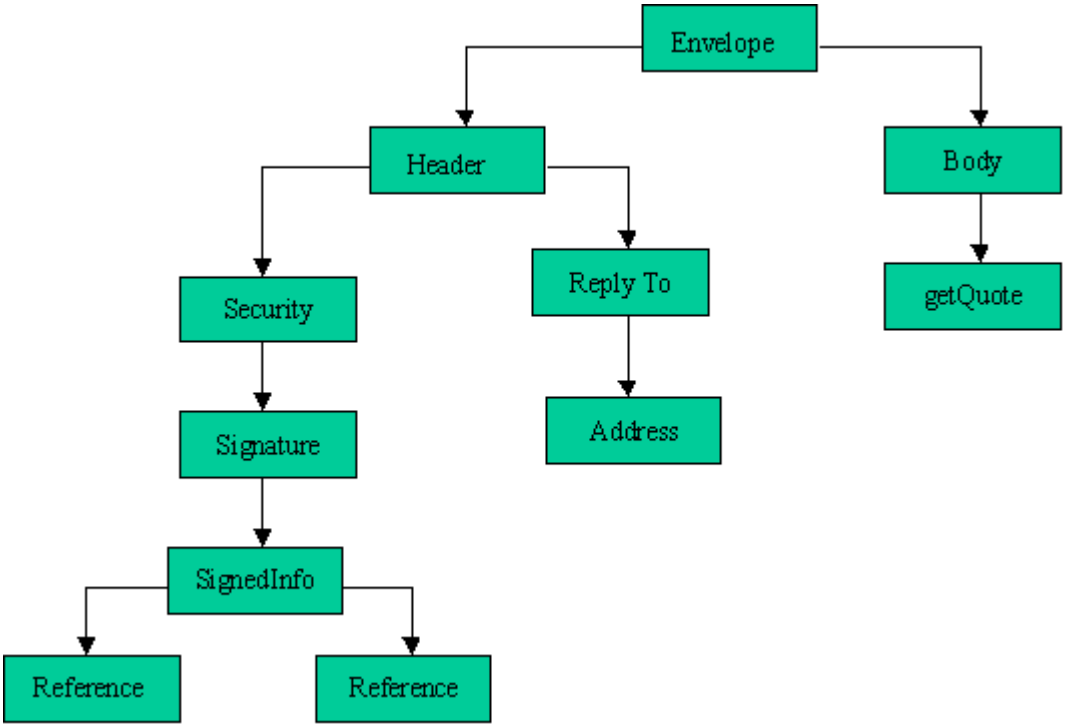

Figure 6.1: Tree representation of the SOAP message of Figure 5.1

For instance the depth of the Reference element in Figure 6.1 is 5. The depth of the Envelope element is 0 etc.

Now it is obvious that, if an attacker wraps up a signed element using a Fake element then the depth of the signed element will be changed. For instance , in Figure 6.1 , if the attacker moves the element

ReplyTo under a Fake header then the depth of the ReplyTo element will be changed from 2 to 3. Therefore, this sort of attack can be detected by just keeping track of the depth information of all of the signed elements. Now does every attack change the depth of an element? Unfortunately not.

Figure 6.2 shows how an element can be relocated without changing its depth. As we discussed before a SOAP message can have multiple header elements with the same name. An example of this sort of header element is Security header, which is specified in WS-Security [6]. However, WS-Security imposes a restriction regarding the role of Security header elements. According to this Specification no two Security header elements can have the same value for their role attribute. The default role attribute's value for a header element is "UltimateReciever". Moreover, one can explicitly specify the role attribute value for a header element as "UltimateReciever". Therefore, there can have two Security header elements targeted for the ultimate recipient. Beside this Security header, an application

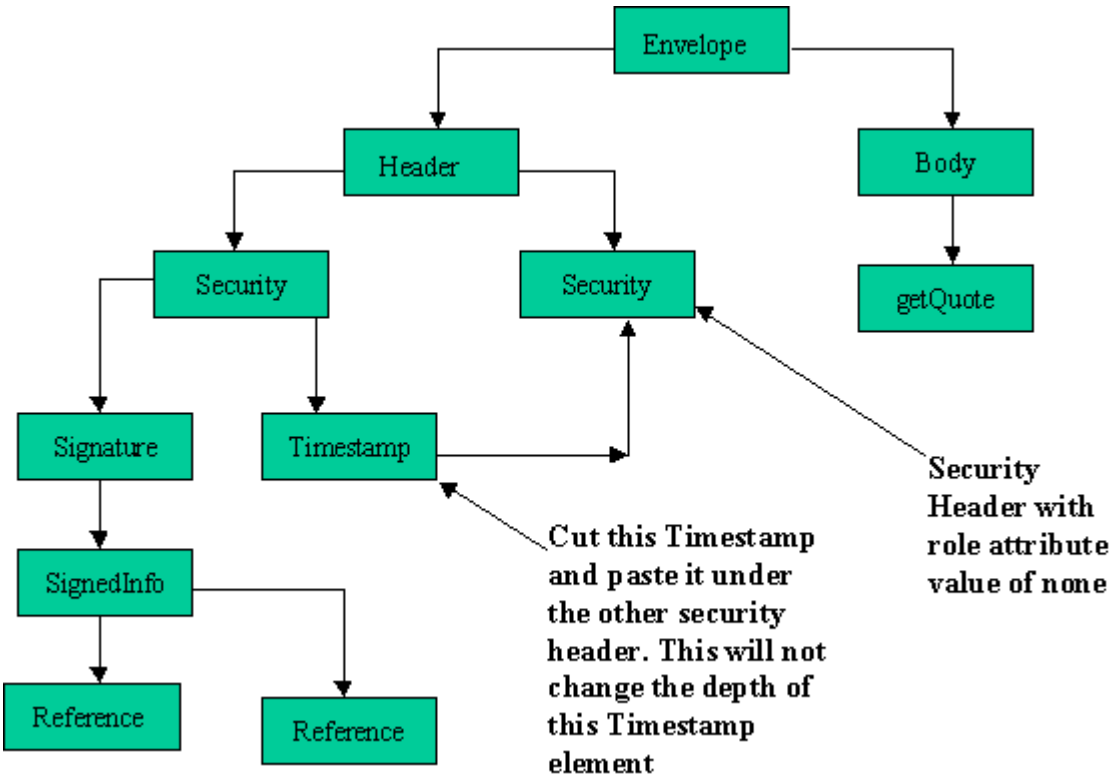

Figure 6.2: Rewriting attack without changing the depth of an element

can define its own header element and allow multiple instance of this header element to exist in one SOAP message. Now lets get back to Figure 6.2. The SOAP message contains two Security header elements. One of the Security header elements contains a Timestamp element, which is signed. The depth of this signed Timestamp element is 3. Now if the attacker cut this element and pastes it under other Security header element, the depth of this Timestamp element will still remain 3. We saw before how this modification can break the security protection of the receiver.

So, how can we detect this sort of modification? One solution that quickly comes into our mind is by keeping information regarding the parent of a signed element. But as we saw before there can have multiple header elements with the same name. So, we cannot just keep the name of the parent. What if we also keep track of the attribute values? However, two application specific header elements might have same attribute values as well. Then how can we uniquely identify the parent? Is there any way at all for identifying an element uniquely in SOAP or XML message? The answer is yes. We know that

XML Digital Signature makes an indirect reference to the element that is to be signed. In fact, this is the weakness of the XML Digital signature that is exploited by the attacker. For this indirect reference to work properly it has to be able to uniquely identify an element. Otherwise, it would sign a wrong element and definitely it is not desirable. For this unique referencing XML Digital Signature makes use of the Id attribute of an element. WS-Security defines an optional Id attribute for uniquely identifying an element. This Id attribute is used as follows:

<anyElement wsu:Id="...">...</anyElement>

wsu:Id is of type xsd:ID. WS-Security also specifies that two elements within a document cannot have the same wsu:Id value. Therefore, we can use this attribute for uniquely referencing the parent of a signed element. In our discussion we will represent wsu:Id as Id . That means we will omit the prefix.

We have managed to get the depth of signed elements and also we have managed to uniquely identify the parent of signed elements. Now if we keep track of the depth and the parent id of a signed element, will it solve our problem? Actually no. Because the parent may have its own parent, that parent may have its own parent and so on. Moreover, as the parent is not signed, the attacker can also change its Id. Figure 6.3 shows how a signed element can be relocated without changing its depth and parent Id. This Figure actually represents the SOAP message of Figure 6.2. However here we have represented the elements by their Ids instead of their name. The Id of the Envelope is “E”, the Header is “H” and the Body is “B”. These three Ids are fixed for all SOAP messages. All other Ids are chosen randomly so that they don’t collide. From now on we will represent all SOAP messages using this mechanism.

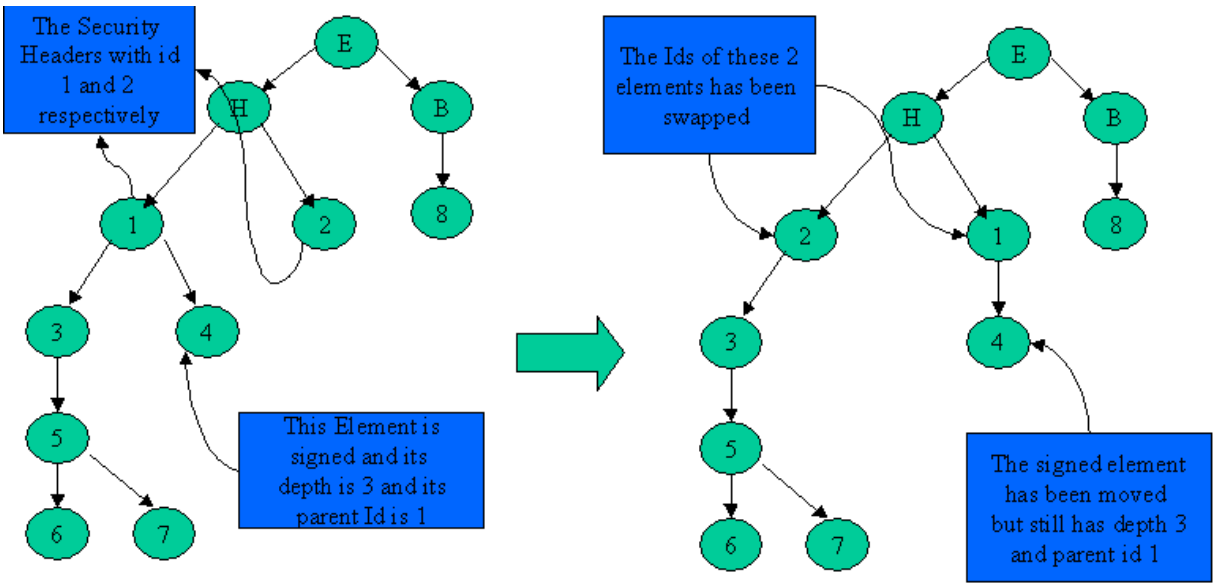

Figure 6.3: Relocation of an element without changing its depth and parent Id

As it can be seen from Figure 6.3 the signed element has Id 4 and its depth is 3. The parent of this element has Id 1. The attacker cut the Signed element and pastes it under the element with Id 2. It does not change the depth of the signed element, however it changes it’s parent’s Id. Therefore the attacker swaps the Id of the Signed element’s parent with the signed element’s parent’s sibling. Now the Signed element’s depth and parent Id became as before. So keeping depth and parent Id information is not helping us a lot. What if we also keep the Ids of the siblings of the Signed elements? It will also not restrain the attacker from relocating the signed element as the attacker can also

manipulate the siblings Ids if they are not signed. It should be noted here that the attacker could also create a new element with a unique Id or delete an unsigned element from the SOAP message.

Till now nothing could help us in restraining the attacker from relocating a signed element. Then how can we protect the message? Well, lets proceed step by step and try to protect the message as strongly as we can. First of all, every element of the message will be given a unique Id. Lets consider there are only two SOAP nodes the initial sender and the ultimate recipient. We will introduce the intermediaries later. As we saw the attacker can create or delete elements from a SOAP message. First we want to restrain him from doing so. Digest value can help us in this regard. Previously in section 2.1.6 we discussed about digest value. Lets consider a string that is created by concatenating all the ids of a SOAP message. We call this string S1.

S1 = "EHB12345678"

Lets consider a digest method D () that takes an string and generates a digest value for that string.

MD = D (S1)

Now if we concatenate an id to the string S1 or delete an id from string S1 the digest value will be changed. Lets assume that another id 9 has been concatenated to the string S1 and the resulting string is S2.

S2 = "EHB123456789"

Now lets calculate the digest value for the string S2, let this digest value is MD'.

MD' = D (S2)

According to the properties of digest method MD' != MD. Therefore we can use this digest method for restraining the attacker from adding or deleting elements in a SOAP message. The sender of the SOAP message will have to ensure that each element has a unique Id. However, Id is an optional attribute. Therefore we will create a module that will ensure that each element gets an Id. After that a digest value will be calculated on the string of Ids of the SOAP message and this digest value will be sent to the sender along with the SOAP message. The sender will check that each element has a unique Id and then will create a digest value on the string of Ids present in the received SOAP message. This digest value will be compared with the digest value sent by the sender. If it does not match then the receiver can infer that the attacker has added or deleted some elements to or from the message. We should note here that we are now considering that there are only two nodes, the initial sender and the ultimate receiver.

So, using message digest we can detect any addition or deletion of elements in a SOAP message. We said before that the message digest method takes as input a string, which is created by concatenating all the Ids of a SOAP message. Now, how the receiver can determine in which order it should concatenate the Ids to create the string? If the receiver cannot determine it then it has to go through the following steps:

i) Concatenate all the Ids of the SOAP message to create a string S
ii) Calculate all the permutation of the string S
iii) For each permutation P do the following
iv) Calculate digest value MD' for P
v) Compare MD' with MD where MD is the received digest value

If the string contains 1000 Ids with an average Id length of 2 then the total number of permutation will be

Factorial (2*1000) = $3.317 * 10^{5735}$

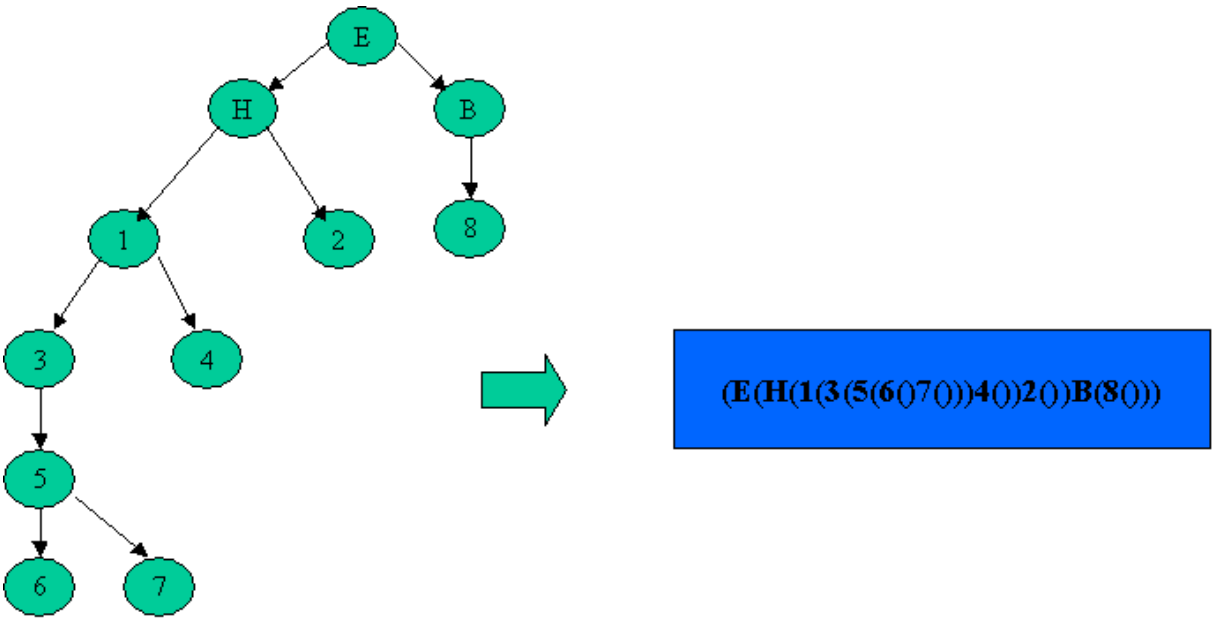

Figure 6.4: Pre-order traversal of a tree and its string representation

This is a huge number. Therefore, certainly it is not an efficient way. One solution of this inefficiency is that the sender gives the receiver a hint regarding the ordering it has used to create the string of Ids along with the digest value. Another solution could be both the sender and the receiver know apriori what ordering to use. We will use the later solution in our present work. Both the sender and the receiver will use the pre-order traversal technique of a tree for this ordering. In a pre-order traversal of a tree the root is traversed first, then the children of the root are traversed from left to right order. We can define this traversal technique recursively as follows:

$$\text{Traverse}(t) = \begin{cases} () & \text{if t is an empty tree} \\ (r(t_1,t_2,t_3,\ldots,t_n)) & \text{if t is a tree with root r and the children of r are } t_1,t_2..tn. \text{ in left to right order} \end{cases}$$

Figure 6.4 shows the pre-order traversal of a list and how this traversal list can be represented as a string. It is obvious from the traversal list that we can build up the original tree if given with this sort of traversal list.

So, the element addition and deletion attack has been removed by using the message digest technique. What about modification? As the traversal list is unique for a tree, if the attacker just cut and pastes elements from its original position to another, the traversal list will be changed which would result in the change of digest value. Moreover, the attacker cannot change the depth of an element.

For instance, Let the absolute path of element m from the root is

AbsolutePath (m) = r, x, y, z, m

The depth of element m is 4. Now if we want to change the depth of element m, we have to introduce a new node or remove some existing node from the absolute path of m. However, if we do this modification, it will change the traversal list of the tree, which will result in the change of the digest value.

However, the attacker can change the name of elements. In this case the traversal list will not change and consequently the digest value will also not change. As a countermeasure of this modification problem we will take not only the Id of an element but also the name of that element while creating the traversal list. We will concat the name of an element with its Id in the following way:

Element Identifier = ElementName:ElementId

Now can the attacker change the Id or name of every element? Not really. The attacker can only change the Id and name of unsigned elements but cannot change the Id or name of signed elements. Therefore, this digest method does not allow the attacker to relocate or modify any signed element of the SOAP message. In our present work we will only protect the signed elements against the rewriting attack. Protection of unsigned elements from rewriting attack is left as future work. For now, we leave it to the discretion of the application to sign all the necessary elements.

So from our discussion above we can conclude that to protect signed elements of a SOAP message against XML Rewriting attack we need the following pieces of information:

i) The name of an element
ii) The id of the signed element
iii) The depth of the signed element
iv) The parent's name and id of the signed element
v) The digest value of the pre-order traversal list of the SOAP message tree.

The above information except the first one will be sent from the sender to the receiver in a header block, which we have named as RewritingHealer. We will describe the structure of this header block later. Here we should take care of the matter that this header block could itself become the subject of XML Rewriting attack. To preclude this attack we have imposed the following restriction regarding the location and the signature of the header block:

i) Rewriting Healer must be signed. Additionally, XPath expression must be used to reference the Rewriting Healer from the Signature element.
ii) The Rewriting Healer must exist as an immediate child of the Header of the SOAP message.
iii) Every message should contain this Rewriting Healer. Even if a message does not have any signed element, it should contain the RewritingHealer with no information regarding signed elements.

If the above conditions are enforced correctly, the attacker will not be able to relocate or modify the Rewriting Healer.

So far we have considered that there are only two SOAP nodes, the sender and the ultimate recipient. However, according to SOAP specification, a SOAP message can traverse zero or more intermediaries before reaching the ultimate receiver. So we have to consider the existence of the intermediaries in our protocol. Now let us introduce intermediaries into the scene.

We saw before in our discussion of SOAP processing model that each SOAP node assumes a role while processing a SOAP message. There are SOAP defined roles and also application defined roles.

However, we also have stated that in our current work we would consider only SOAP defined roles. The role of a SOAP node determines which header blocks it should process. Header blocks contain a role attribute that is used to target it for a SOAP node. This role attribute can assume one of three possible values. These values are Next, UltimateReciever and None. When a SOAP node finds that the role attribute's value of a header block matches the role it has assumed it might process the block. The ultimate recipient of the message can assume the roles UltimateReciever and Next. The intermediaries can only assume the role Next. If the role attribute's value of a header block is None, it means that any SOAP node on its path should not process this header block. In other words no SOAP node can assume the role None. The default role attribute's value for a header block is UltimateReciever. That is, if a header block does not contain any role attribute's value, its role attribute's value will be considered as UltimateReciever. We will assume that all the children of a header block will inherit the role of its parent.

A SOAP intermediary will only process the header blocks targeted for it. It cannot remove or reorder any header blocks that are not targeted to it. The intermediaries can only remove header blocks with role attribute's value Next. Moreover, it can append its own header blocks. But this modification does not change the relative order of the header blocks targeted to the Ultimate Receiver. The Body of the SOAP message is always targeted to the Ultimate Receiver. Lets define a function, which will map a role name to a positive integer value.

$$f(x) = \begin{cases} 1 \text{ if } x \text{ is UltimateReciever} \\ 2 \text{ if } x \text{ is Next} \\ 3 \text{ if } x \text{ is None} \end{cases}$$

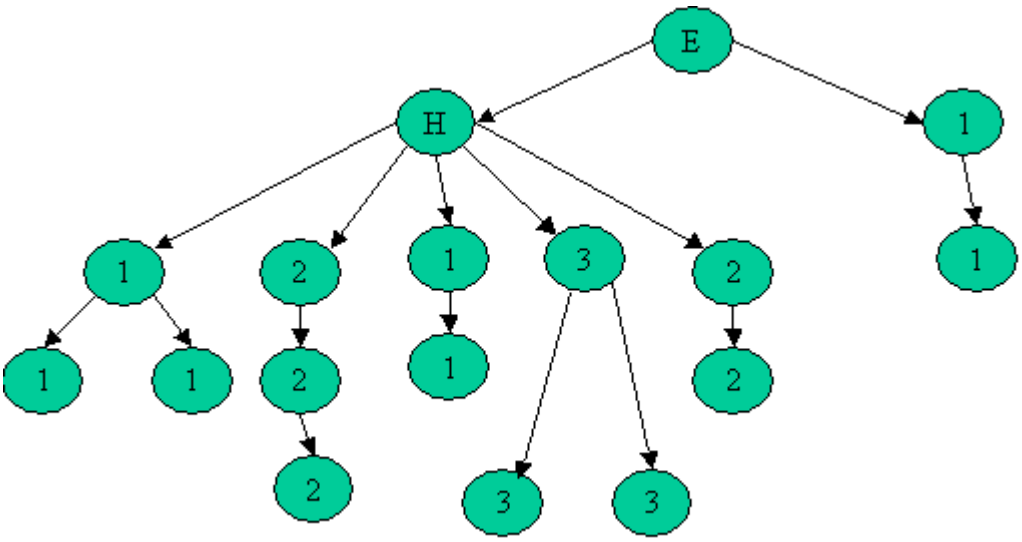

Figure 6.5: Tree representation of a SOAP message, each node is labelled with its role's integer value

Now if we represent a SOAP message using a tree structure and label each node of the tree with the role attribute's integer value of corresponding element, it will look something like Figure 6.5. We should recall that an element inherits the role of its parent.

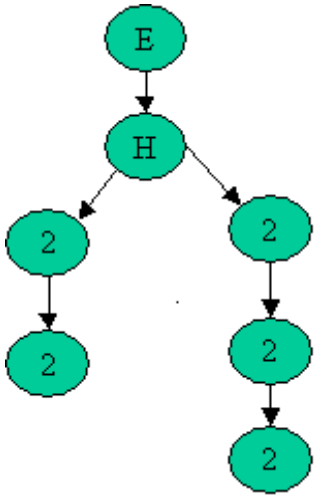

Figure 6.6: Tree only representing nodes corresponding to elements targeted to the intermediary

It should be noted that the envelope and the header of a SOAP message does not have any role therefore they are represented using their Id E and H respectively. All the nodes of the tree in Figure 6.5 (Except the nodes with label E and H) have one of three labels 1, 2 or 3.

Now if we delete all the nodes from the tree in Figure 6.5 except the nodes with label 2 and the nodes with label E and H, we will get a tree which represents all the elements of a SOAP message that are targeted to the intermediary. The resulting tree is shown in Figure 6.6. In the same manner if we delete all the nodes from the tree of figure 6.5, except the nodes with label 1, we will get a tree composed of nodes representing the elements targeted to the Ultimate Receiver. It is also true for the nodes representing elements with role None.

So we saw that from the tree of a SOAP message we can extract three different trees corresponding to the three different roles. Moreover, we will assign each SOAP node a number starting from 0. Each SOAP node will determine this number itself by looking at how many SOAP nodes have processed the SOAP message previously. A SOAP node will get this information from our RewritingHealer header block. Later we will see how. We also have defined a rule for the creation of the Id of an unsigned element. Each SOAP node should create the Id for an unsigned element as follows:

Unsigned_element_id = node_number.element_role.random_integer

That is, the Id will be composed of three parts separated by a dot, the number of the SOAP node who is creating the Id, the role attribute's integer value of the element for which the Id is being created, a random integer value so that the resulting Id is unique within the document.

The above Id format allows the receiver of a SOAP message in identifying, which SOAP node has added which unsigned elements and which element is added for which role. For the signed element we don't need this Id format. Because our header block will contain the Id for each signed element and from this information the receiver can identify, which signed elements have been added by which SOAP node and for which role the element was added. Two of the attributes of RewritingHealer header block are Role and NodeNumber. The Role attribute specifies for which role's tree (as we discussed before) this RewritingHealer header block contains information and the NodeNumber specifies which SOAP node created this RewritingHealer header block.

Therefore each SOAP node will create three RewritingHealer header block corresponding to the three roles. Then the node will extract three trees corresponding to the three roles from the tree representation of the SOAP message as we described before. Let the RewritingHealer header block for

role UltimateReciever is h_1, for role Next is h_2 and for None is h_3. We also assume that the tree for role UltimateReciever is t_1, for role Next is t_2 and for role None is t_3. Now h_1 will be filled up with information from t_1, h_2 will be filled up with information from t_2 and h_3 will be filled up with information from t_3. Then the SOAP node will sign the three RewritingHealer header blocks and will append them to the outbound SOAP message.

When an intermediary will receive a SOAP message, it will retrieve the RewritingHealer header block with role attribute value “Next”. Then it will take the node_number from this RewritingHealer header block. Let this node number is x .It will get all the signed elements added by node x from the RewritingHealer header. Then it will find all the unsigned elements added by node x for role next. The Id format we specified before helps a node in easily determining it. Then it will verify all the signed elements depth, parent’s Id and parent’s name. If it is successful, it will form the tree with the determined signed and unsigned elements added by node x, make the pre-order traversal list of the tree, take the digest value of the traversal list and compare it with the digest value present in the header block. If the two matches, the verification succeeds. Otherwise an error is generated. Moreover, after processing the header block the node must remove it.

The processing for the ultimate recipient is the same as the intermediaries. However, instead of processing only the RewritingHealer header block with role attribute’s value next, the ultimate recipient will process all of the Rewriting Healer header blocks.

6.2 Rewriting Healer

In previous section we saw that in order to protect the SOAP message from XML Rewriting attack the sender needs to send some information regarding the structure of the SOAP message to the receiver. We said that we have introduced a header block to carry these information and we named that header block as Rewriting Healer. But we did not provide the detail structure of this Rewriting Healer. This is the section where we will describe the structure of Rewriting Healer.

Figure 6.7 shows the overall structure of our proposed Rewriting Healer header block. The Rewriting Healer header block contains some attributes and some child elements.

RewritingHealer@ID: This mandatory attribute contains the Id of a RewritingHealer header block

RewritingHealer@Role: This mandatory attribute specifies which SOAP node should process a particular RewritingHealer header block. It also specifies that this RewritingHealer contains information only for the elements whose role attribute’ value is the same as this Rewriting Healer’s role attribute.

RewritingHealer@NodeNumber: This mandatory attribute specifies the node that created a particular RewritingHealer header block with a positive integer value.

RewritingHealer/Time: This child element contains the Time at which the sender created the RewritingHealer header block. Each subsequent time must have monotonically increasing value. This element is used for the protection against the store and replay attack. Moreover, the initial sender will only add this element in its RewritingHealer header block whose role attribute’s value is UlimateReciever. That means with node with number 0 will add this element.

RewritingHealer/MessageID: This element contains an Identifier for the RewritingHealer header block. The client uses this element for the correlation of responses. The server copies the MessageID of the request into the response so that the client can correlate the response with the request. Only the initial sender creates this element in the RewritingHealer header block whose role attribute’s value is UlimateReciever.

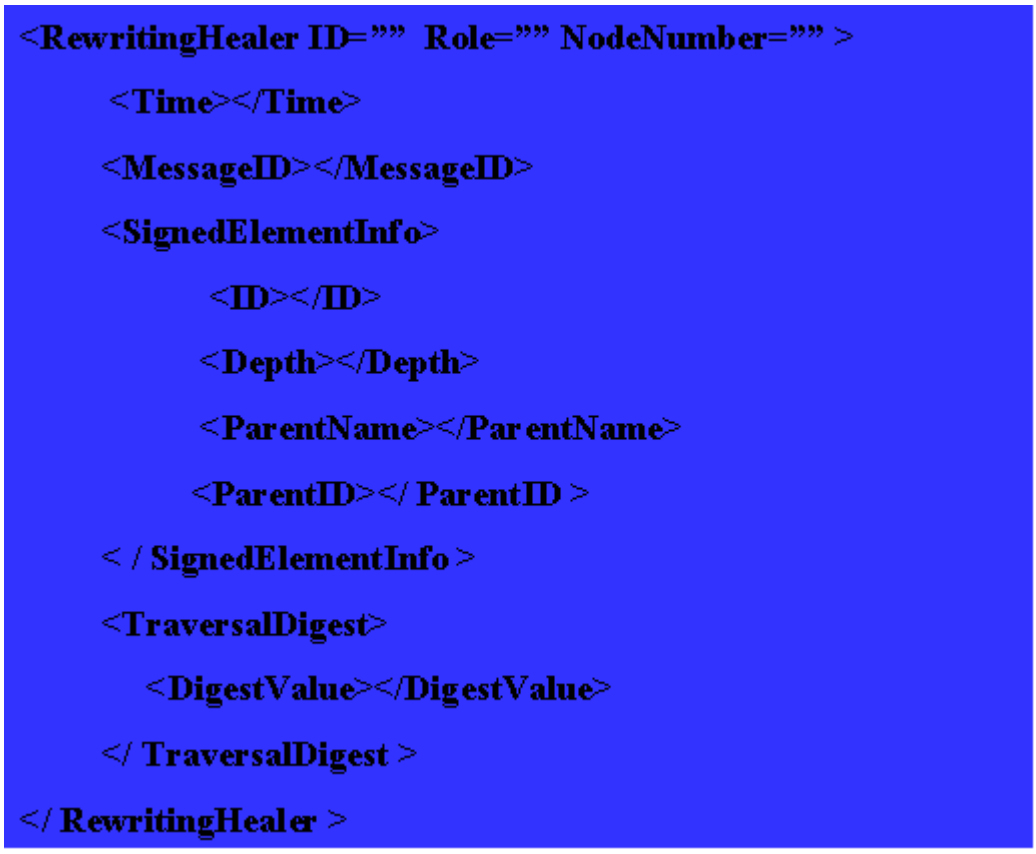

Figure 6.7: Structure of RewritingHealer

RewritingHealer/SignedElementInfo: This element contains information regarding the signed elements of a SOAP message, which has the same role attribute's value as the RewritingHealer's role attribute's value.
RewritingHealer/SignedElementInfo/ID: This element specifies the Id of a signed element

RewritingHealer/SignedElementInfo/Depth: This element specifies the depth of a signed element with the above Id.

RewritingHealer/SignedElementInfo/ParentName: This element specifies the name of the parent of a signed element with the above Id.

RewritingHealer/SignedElementInfo/ParentID: This element specifies the Id of the parent of a signed element with the above Id.

RewiritingHealer/TraversalDigest: This element is a container for the digest value of the pre-order traversal of a tree.

RewritingHealer/TraversalDigest/DigestValue: This element contains the Digest Value. SHA-256 algorithm is used to create this digest value. The digest value is created on the string representation of the pre-order traversal list of a tree as described before. This tree is composed of nodes which represent elements of a SOAP message that have the same role attribute's value as this RewritingHealer's role attribute value.

6.3 RewritingHealer Processing Rule

In the previous section we saw the Structure of the RewritingHealer. We showed what attributes and elements this header block contains and also described what each of those attributes and elements specify. In this section we will see the processing rules for RewritingHealer header block. We will see how this header block is created and verified by different SOAP nodes like intermediaries and ultimate recipient. In the description of the processing rule "the role of an element" means the value of the role attribute of that element. However if the element does not have this attribute it means the role attribute's value is inherited from its parent as we explained before.

Creation Rule for RewritingHealer header block:

1. Determine the node number of this SOAP node. Let the number is NN
2. For each unsigned element UE do step 3 to 7
3. If UE contains an Id then do 4 to 5
4. If the Id is not referenced from anywhere of this document then remove it and go to step6
5. Change the Id of UE to NN.R.I, where NN is the current node number, R is the role of the unsigned element, I is some random positive integer value unique within the current SOAP message. Adapt the reference element and Go to step 8
6. Create an Id IDUE of the form NN.R.I, where NN is the current node number, R is the role of the unsigned element, I is some random positive integer value unique within the current SOAP message.
7. Set the Id attribute's value of UE to IDUE
8. Create an empty set of Ids . Let the set is PreviouslySignedElement
9. For each existing RewritingHealer header of this SOAP message do step 10
10. Retrieve the Id from all SignedElementInfo elements of the RewritingHealer header block and insert the Ids in the set PreviouslySignedElement
11. For each role R in the set (UltimateReciever, Next, None) do step 12 to 28
12. Set IR = f(R), where f maps a Role R to a positive Integer
13. Get a unique ID value Id
14. Create a RewritingHealer header block with ID = Id , Role = IR , NodeNumber = NN and prepend it under the Header of the SOAP message. If the Message does not contain a header then create it before prepending.
15. If NN is 0 and IR is 1 then do step 16 and 17
16. Create a Time element and insert the current time under this element and append this Time element under the current RewritingHealer header block
17. Create a MessageID element and insert a random integer number under this element. After that append this element under the current RewritingHealer header block. Put this MessageID in a buffer.
18. For each Signed element SE with Identifier Id in the SOAP message do step 19 to 22
19. If Id is in the set PreviouslySignedElement go to step 18
20. Determine the role R' of SE and compute f(R') = IR'. Also determine the depth D of SE , Parent's Name PN of SE and Parent's ID PID of SE.
21. If IR'! = IR go to step 18
22. Create a SignedElementInfo element with ID = Id, Depth = D, ParentName = PN and ParentID = PID. After that append this element under the current RewritingHealer header block
23. Create a tree with all the elements of the SOAP message that have role R'' and f(R'') = IR and that have been added by node with number NN. The tree will be rooted at the envelope element and the label of the root will be E. Moreover, the label of the node which will specify the Header of the SOAP message will be H

24. Compute a string S by making a pre-order traversal on the above tree
25. Compute a digest value MD using SHA-256 on string S
26. Create a TraversalDigest element an append it under the current RewritingHealer header block
27. Create a DigestValue element and set its value with MD. Then append this element under the TraversalDigest element.
28. Sign the created RewritingHealer header using XPath referencing

RewritingHealer verification rule for Intermediaries

1. Retrieve the RewritingHealer header block HDR that has role attributes value Next
2. Check if HDR is signed or not. If it is signed go to next step else generate an error and terminate.
3. Create an empty set SEID.
4. Retrieve all the SignedElementInfo from HDR and put them in a set SE.
5. For each SignedElementInfo in SE do step 6 to 12
6. Set D = Depth element value of SE
7. Set PN = ParentName element value of SE
8. Set PID = ParentID element value of SE
9. Set MID = ID element value of SE
10. Retrieve the element Elem with Id MID. Let the depth of this element is D', Name of the parent of this element is PN' and Id of the parent of this element is PID'
11. Compare D with D', PID with PID' and PN with PN' for equality. If all of the comparisons succeed go to next step. Else generate an error message and terminate
12. Put MID in the set SEID
13. Create a set USEID with all the Ids of the unsigned elements in the SOAP message
14. For each Id I in USEID do step 15 to 16
15. Retrieve the role part IR from I.
16. If IR != f(Next) , where f maps a Role name to a positive integer value , then remove this Id from USEID
17. Create a tree T with all the elements with Id in SEID or USEID
18. Create a String S with the pre-order traversal list of the above tree.
19. Compute digest value MD' using SHA-256 on the String S
20. Retrieve the Digest Value MD present in HDR
21. Compare MD with MD'. If they are equal go to next step else generate error and terminate.
22. Remove HDR from the SOAP message

RewritingHealer verification rule for ultimate recipient

1. Retrieve the Time T and MessageID MID from the RewritingHealer header block which has NodeNumber attribute's value 0 and Role attributes value UltimateRecepient
2. Fetch the last Time LT value from the cache, which was sent from the same location as the current request.
3. If LT>=T then generate an error and terminate. Else go to next step.
4. Store T in the cache as LT
5. Put MID in a buffer to be sent with the response
6. Determine total number of distinct NodeNumber N of RewritingHealer header blocks
7. For NN = 0 to N do step 8 to 30
8. For each role R in the set {UltimateReciever, Next, None} do step 9 to 30
9. Retrieve the RewritingHealer header block HDR that has role attributes value R and NodeNumber value NN
10. Check if HDR is signed or not. If it is signed go to next step else generate an error and terminate
11. Create an empty set SEID.

12. Retrieve all the SignedElementInfo from HDR and put them in a set SE.
13. For each SignedElementInfo in SE do step 14 to 20
14. Set D = Depth element value of SE
15. Set PN = ParentName element value of SE
16. Set PID = ParentID element value of SE
17. Set MID = ID element value of SE
18. Retrieve the element Elem with Id MID. Let the depth of this element is D', Name of the parent of this element is PN' and Id of the parent of this element is PID'
19. Compare D with D', PID with PID' and PN with PN' for equality. If all of the comparisons succeed go to next step. Else generate an error message and terminate
20. Put MID in the set SEID
21. Create a set USEID with all the Ids of the unsigned elements in the SOAP message
22. For each Id I in USEID do step 23 to 24
23. Retrieve the role part IR and Node Number part NN' from I.
24. If IR != f(R), where f maps a Role name to a positive integer value, or NN' != NN then remove this Id from USEID
25. Create a tree T with all the elements with Id in SEID or USEID
26. Create a String S with the pre-order traversal list of the above tree.
27. Compute digest value MD' using SHA-256 on the String S
28. Retrieve the Digest Value MD present in HDR
29. Compare MD with MD'. If they are equal go to next step else generate error and terminate.
30. Remove HDR from the SOAP message

When generating a response for the client the RewritingHealer processing module will go through the following step:

1. Create an empty RewritingHealer header block and append it under the Header of the of the SOAP response message. If no header is present, create a new Header and then append it.
2. Fetch the MessageID MID that was present in the last request message and that was kept in a buffer.
3. Create a MessageID element under the RewritingHealer header block and set its value with MID.
4. Sign the RewritingHealer header block.

After getting the response the RewritingHealer processing module on the client side will verify the Signature of the RewritingHealer header block. Then it will take the MessageID from the header block and will try to correlate this MessageID with the last sent request MessageID. If it fails, it will generate an error.

6.4 Scenario with RewritingHealer:

In this section we will provide some scenarios to demonstrate how RewritingHealer can detect potential rewriting attack in SOAP message. The SOAP messages that we will use for our demonstration are not specific to any particular application. Instead they are general SOAP messages where the name of the Header elements and Body elements are totally arbitrary. Moreover, we will omit a lot details from the example SOAP messages. For instance we will not show the Signature element that contains reference to the RewritingHealer header element using xpath expression.

Figure 6.8 shows a SOAP message that has been processed using our RewritingHealer creation rule. Every unsigned element has been given a unique Id. The message contains a RewritingHealer header element. This header element has NodeNumber attribute's value of 0 as it has been created

by the initial sender and Role attributes value of 1 as it contains information regarding the elements of the SOAP message targeted to the UltimateRecipient

```
<Envelope>
  <Header>
    <A Id="Id-1">123</ >
    <D  id=”Id-4”>…. </ >
    <B  id=”0.1.1”><C id=”0.1.2”>…..</ ></ >
    <Security  id = “0.1.6”>
     <BinarySecurityToken   Id="0.1.4"> abcdefg….</ >
       <Signature>
            <SignedInfo>
              <Reference URI="#Id-1">……</ >
              <Reference URI="#Id-4">……</ >
          </ >
      </ >
   </ >
  </ >
  <RewritingHealer ID=”5” NodeNumber=”0” Role=”1”>
   <MessageID>1</MessageID>
   <Time>tttt</Time>
   <SignedElementInfo>
    <ID>Id-1</ID><Depth>2</Depth><PID>H</PID>
    <Pname>Header</Pname>
   </SignedElementInfo>
   <SignedElementInfo>
    <ID> Id-4</ID><Depth>2</Depth><PID>H</PID>
    <Pname>Header</Pname>
   </SignedElementInfo>
    <TraversalDigest>
   <DigestValue>XYZUSDF</DigestValue>
   </TraversalDigest>
  </RewritingHealer>
  <Body Id= “0.1.6”>
     <G Id= “0.1.7”>...</ >
  </ >
</ >
```

Figure 6.8: SOAP message with RewritingHealer header block

```
<Envelope>
  <Header>
    <D  id=”Id-4”>…. </ >
    <A Id="Id-1">123</ >
    <B  id=”0.1.1”><C id=”0.1.2”>…..</ ></ >
   <Security  id = “0.1.6”>
    <BinarySecurityToken   Id="0.1.4"> abcdefg….</ >
      <Signature>
           <SignedInfo>
             <Reference URI="#Id-1">……</ >
             <Reference URI="#Id-4">……</ >
         </ >
     </ >
   </ >
  </ >
  <RewritingHealer ID=”5” NodeNumber=”0” Role=”1”>
   <MessageID>1</MessageID>
   <Time>tttt</Time>
   <SignedElementInfo>
    <ID>Id-1</ID><Depth>2</Depth><PID>H</PID>
    <Pname>Header</Pname>
   </SignedElementInfo>
   <SignedElementInfo>
    <ID> Id-4</ID><Depth>2</Depth><PID>H</PID>
    <Pname>Header</Pname>
   </SignedElementInfo>
    <TraversalDigest>
       <DigestValue>MNOPQR</DigestValue>
    </TraversalDigest>
  </RewritingHealer>
  <Body Id= “0.1.6”>
     <G Id= “0.1.7”>...</ >
  </ >
</ >
```

Figure 6.9: SOAP message with RewritingHealer header block and signed elements reordered by attacker

The SOAP message of Figure 6.8 contains two signed header elements with Ids id-1 and id-4. Therefore the RewritingHealer header contains two <SignedElementInfo> elements. These two <SignedElementInfo> elements contain information regarding the two signed header elements. <TraversalDigest> element contains a digest value. SHA-256 has been used for digest value computation. The digest value is calculated on the following string, which is the pre-order traversal string for the SOAP message of Figure 6.8:

(Envelope:E (Header:H(**A:Id-1()D:Id-4()**B:0.1.1(C:0.1.2())Security:0.1.6(BST:0.1.4()))
Body:0.1.6(G:0.1.7()))

Now let the attacker has intercepted the SOAP message of Figure 6.8 and he wants to modify it. Figure 6.9 shows how the attacker can modify the message. The attacker neither deleted nor introduced any element in the SOAP message. He just re-ordered the signed header elements. In Figure 6.9 the two signed elements are shown in bold. As we can see from Figure 6.9, none of the

information that is present under the <SignedElementInfo> element of the RewritingHealer header block can detect the modification made by the attacker. This is because the modification will not have any effect on these information. Lets see what happens to the digest value. The receiver will now calculate a digest value on the following string, which is the pre-order traversal string for the SOAP message of Figure 6.9:

(Envelope:E(Header:H(**D:Id-4()A:Id-1()**B:0.1.1(C:0.1.2())Security:0.1.6(BST:0.1.4()))
Body:0.1.6(G:0.1.7()))

As it can be seen that this traversal string is not equal to the traversal string we showed before, according to the feature of the SHA-256 the new calculated digest value will not be equal to the one that is present in the received SOAP message. So , the receiver will detect the tampering.

```
<Envelope>
  <Header>
    <A Id="Id-1" role=”next”>123</ >
   <B  id=”0.1.1”>
         <C id=”0.1.2”>…..</ ><D  id=”Id-4”>…. </ >
   </ >
    <Security  id = “0.1.6”>
     <BinarySecurityToken   Id="0.1.4">abcdefg….</ >
       <Signature>
            <SignedInfo>
              <Reference URI="#Id-1">……</ >
              <Reference URI="#Id-4">……</ >
           </ >
      </ >
    </ >
  </ >
  <RewritingHealer ID=”5” NodeNumber=”0” Role=”1”>
   <MessageID>1</MessageID>
   <Time>tttt</Time>
   <SignedElementInfo>
     <ID> Id-4</ID><Depth>3</Depth><PID>0.1.1</PID>
     <Pname>B</Pname>
   </SignedElementInfo>
    <DigestMethod>SHA-256</DigestMethod>
    <DigestValue>XYZUSDF</DigestValue>
  </RewritingHealer>
  <RewritingHealer ID=”6” NodeNumber=”0” Role=”2”>
  <SignedElementInfo>
    <ID> Id-1</ID><Depth>2</Depth><PID>H</PID>
    <Pname>Header</Pname>
   </SignedElementInfo>
   <TraversalDigest>
     <DigestValue>XYZUSDF</DigestValue>
   </TraversalDigest>
  </RewritingHealer>
  <Body Id= “0.1.6”>
     <G Id= “0.1.7”>...</ >
  </ >
</ >
```

Figure 6.10: SOAP message with two RewritingHealer header blocks

```
<Envelope>
  <Header>
    <A Id="Id-1" role = “next”>123</ >
    <B id=”0.1.2”> <D  id=”Id-4”>…. </ ></ >
   <B  id=”0.1.1”> </ >
    <Security  id = “0.1.6”>
     <BinarySecurityToken   Id="0.1.4">abcdefg….</ >
       <Signature>
            <SignedInfo>
              <Reference URI="#Id-1">……</ >
              <Reference URI="#Id-4">……</ >
           </ >
      </ >
    </ >
  </ >
  <RewritingHealer ID=”5” NodeNumber=”0” Role=”1”>
   <MessageID>1</MessageID>
   <Time>tttt</Time>
   <SignedElementInfo>
     <ID> Id-4</ID><Depth>3</Depth><PID>0.1.1</PID>
     <Pname>B</Pname>
   </SignedElementInfo>
    <DigestMethod>SHA-256</DigestMethod>
    <DigestValue>XYZUSDF</DigestValue>
  </RewritingHealer>
  <RewritingHealer ID=”6” NodeNumber=”0” Role=”2”>
  <SignedElementInfo>
    <ID> Id-1</ID><Depth>2</Depth><PID>H</PID>
    <Pname>Header</Pname>
   </SignedElementInfo>
   <TraversalDigest>
      <DigestValue>XYZUSDF</DigestValue>
   </TraversalDigest>
  </RewritingHealer>
  <Body Id= “0.1.6”>
     <G Id= “0.1.7”>...</ >
  </ >
</ >
```

Figure 6.11 : SOAP message with two RewritingHealer header blocks and element created by attacker

Figure 6.10 shows another example SOAP message. However, this SOAP message contains some data targeted to the ultimate recipient and the rest targeted to the intermediary. Therefore, the message contains two RewritingHealer header blocks. Both of them have the same NodeNumber attribute's value but they have different value for Role attribute. RewritingHealer with Role attribute's value of 1 contains information regarding the element targeted to the ultimate recipient and with Role attribute's value of 2 contains information regarding the element targeted to the intermediary. The digest value for the RewritingHealer with Role attribute's value of 1 is calculated on the following string:

(Envelope:E(Header:H(B:0.1.1(C:0.1.2()D:Id-4())Security:0.1.6(BST:0.1.4()))
Body:0.1.6(G:0.1.7())))

On the other hand the digest value for the RewritingHealer with Role attribute's value of 2 will be calculated on the following string :

(Envelope:E(Header:H(A:Id-1())))

Suppose the attacker intercepted the message of Figure 6.10. Now he wants to modify the message. As it can be seen from the intercepted SOAP message it contains one header element with name B and Id 0.1.1. This header element contains two children with name C and D. The D child is signed. At first the attacker creates a new header element with name B. As we said before that each unsigned element of a SOAP message would be given a unique Id by our RewritingHealer creation process, the attacker has to provide this new element with an Id. However, we showed previously that no new Id could be created; otherwise it will have effect on the DigestValue. The only way left for the attacker is to delete an existing unsigned element and give the newly created element the Id of the deleted element. Therefore, the attacker deleted the unsigned element with name C and Id 0.1.1 and gives the newly created element this Id. Now he cut the signed element with name D and Id id-4 and pastes it under the newly created B element.

When the intermediary will receive the above message he will only verify the information present in the RewritingHealer with Role attribute's value of 2. Therefore the intermediary will not identify this tampering as none of the elements targeted for the intermediary has been forged on. However, when the ultimate receiver will receive the modified message he will be able to identify the tampering in two different ways. First of all he can identify the modification by verifying the information that is present under the <SignedInfoElement> of the RewritingHealer whose Role Attribute value is 1. This is because although the parent's name of the relocated element is still the same, the parent's id has been changed. The second way of identifying this modification is the Digest Value comparison. Because of the modification of the SOAP message, the traversal string of the modified message will not be equal to the one for which the digest value was created. Consequently their digest value will be different as well. Therefore, by computing the digest value and comparing it with the one present in the SOAP message, the receiver can easily identify the tampering.

Chapter 7: Implementation

We will provide a java implementation of our proposed method. We have chosen Axis, a soap processor engine, for the creation and processing of SOAP messages. For signature creation and verification we will use a Apache XML Security [40] library version 1.4. This library implements the security standard for xml. Apache provides both java and c++ implementation of this library. We have used Tomcat 5.0, which is a servlet engine for the deployment of both axis servlet and our produced modules.

7.1 Axis Overview:

Apache Software Foundation has provided an implementation for SOAP. They have named it Axis. Axis is a SOAP engine, which provides a framework for constructing SOAP processors such as clients, servers, gateways, etc.[39] This engine is currently implemented in java. However a c++ implementation of this engine is on its way. Presently axis engine is available in two different forms.

1. Axis engine as a servlet. This form of axis engine needs to be deployed as an web application in a servlet engine like Tomcat
2. Axis engine as a stand-alone server.

Axis does not only provide a SOAP processor. [1] It also provides a tool, Java2WSDL, for creating WSDL file from the java code of a web service, a tool, AdminClient, for deploying web service in Axis a tool, tcpmon, for monitoring SOAP message exchange between the client and the server and a tool, WSDL2Java for creating client and server proxy. Recent version of axis also provides a web service called SOAPMonitor that can be used to monitor SOAP message exchange between client and server without changing the configuration file of the servlet engine.
The core processing logic of an Axis engine can be invoked in two different ways.[39] It can be invoked by an application on the client side, or by a transport listener on the server side. Whoever calls it, once it is called, it creates a message context with the received message. This message context is a package that may contain a request or a response along with some properties. Then the core processing logic passes the message to a series of handlers. Each of these handlers process the message and pass it to the next handler until it is reached to its ultimate destination, which may be a web service or a transport listener. Figure 7.1 shows the message flow in an axis engine on server side.

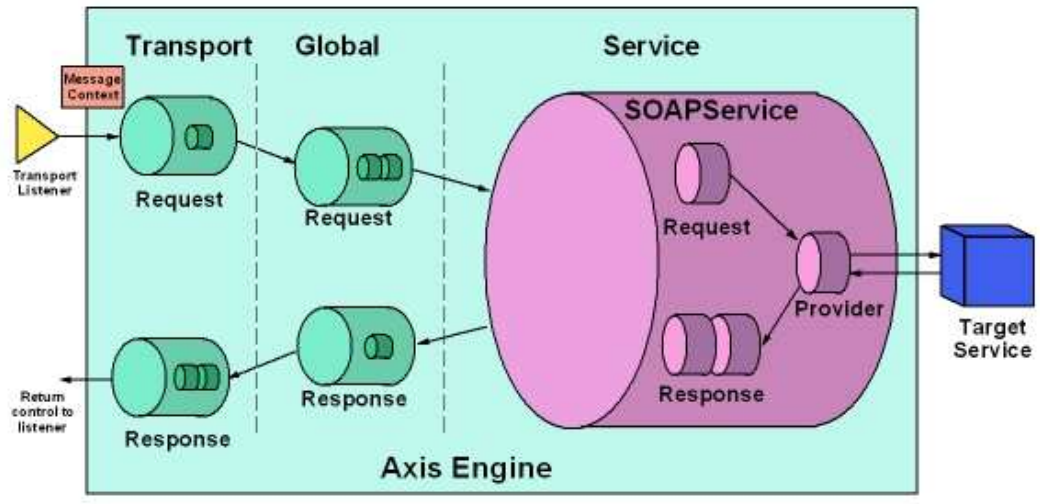

Figure 7.1: Message flow through an Axis engine on server side [39]

In Figure 7.1 the small cylinders represent handlers and the big cylinders represent a chain.[39] A handler can be thought of as a plug-in in Axis engine. Handlers are nothing but web service. They intercept a SOAP message, process it in a predefined way and then passes it to the next entity. There are three different types of handlers.[39] Transport specific handler, Global handler and Service specific.
We know that SOAP message can be exchanged using different transport mechanism like http, SMTP, ftp etc. The transport listener's job is to retrieve protocol specific data from the received message.[39] With this data the transport listener creates a Message object and pack it into a MessageContext object along with some properties. It also put the transport name, for instance http, in the MessageContext object. Once the message context is prepared it is passed to the axis engine. The axis engine first determines the type of transport by transport name. A transport is an object that can contain a request chain or a response chain or both of them.[39] A chain is a sequence of same type of handlers. For instance, all the Transport handlers are aggregated to form a Transport chain and all the Service Handlers are aggregated to form a Service chain. The message passes through all the transport handlers in the chain in turn. Once all transport handlers have processed the message, it is passed to the global chain. Global handlers are applicable for all messages. In this Global chain there is a handler, which determines the service to be called from the URL. For instance from the URL "http://localhost: 8080/axis/services/AirLineTicketService" it determines the service name AirlineTicketService. Once all the global handlers are done with the message, it is passed to the Service chain. A Service chain may contain request handlers and/or response handlers. However at the end of the service chain there is always a Service Provider that implements the back end logic of the web service.[39] Each service might have its own specific handlers. After the processing of the service chain the message is routed to the target service. Although most of the web service generate response, not all of them do. If the target service generates a response, the response passes through the same path as before but in reverse order to the transport listener. The transport listener sends the message to the client. As we saw from our discussion so far, a message passes through a series of handlers in turn once it gets into the axis engine and before reaching the target service and vice versa. Each handler has a specific method, which is called invoke(). This method takes a MessageContext object as its argument. When the transport listener, or a handler wants to pass the message to the next handler it calls the invoke () method of that handler. In which order the handler will be called depends on two factors [39]:

1. The configuration of the deployment descriptor file of axis. We will see later how this file is configured.
2. Whether the axis engine is running on the client or on the server side

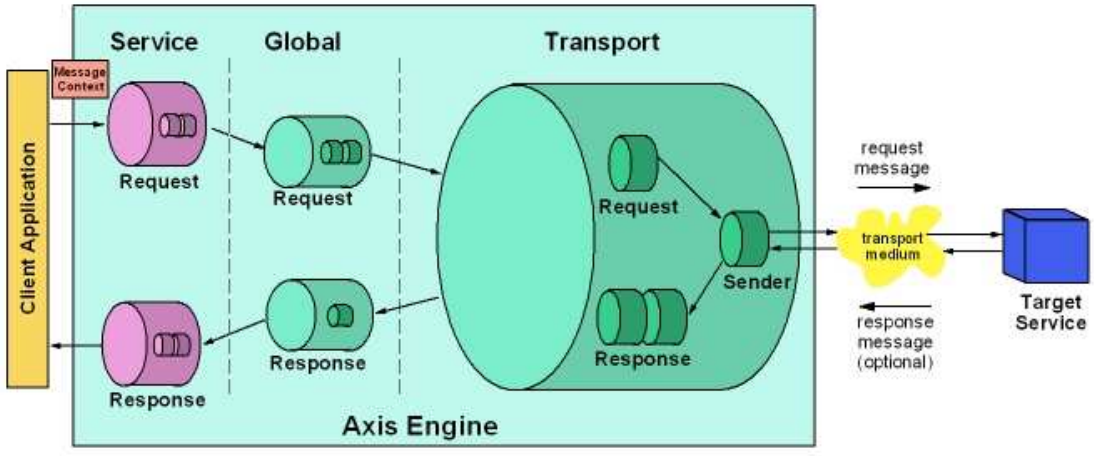

Figure 7.2: Message flow through the axis engine on the client side [39]

Figure 7.2 shows how message flows through the axis engine on the client side. It is almost same as the message flow of server side however the flow path is reversed. [39] In this case the client application invokes the core axis engine processing logic instead of the transport listener. Then the message context is passed to the request service chain, if it is present. In the request service chain the message passes through all the handlers. It should be noted here that the service chain does not contain any service provider as a distant service provider provides the service. After the service chain, if there is any global chain, it gets the message and passes it to the transport chain after processing. The handlers of transport chain process the message and send it to the remote node. When a response arrives it passes through the same sequence of handler chain but in a reverse order.

7.2 Module Description:

Figure 7.3 shows the module structure of our implementation of RewritingHealer.

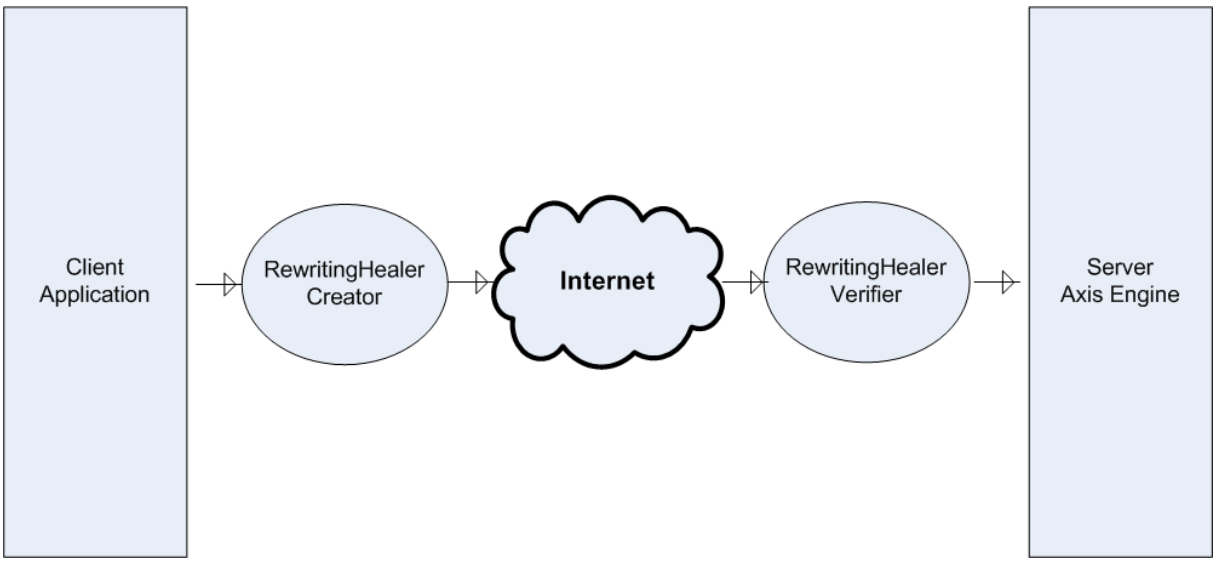

Figure 7.3: Module Structure of RewritingHealer

Presently our implementation of RewritingHealer is composed of two modules, RewrtingHealer Creator and RewritingHealerVerifier. The RewritingHealerCreator module takes a SOAP message, adds Id attributes to appropriate elements, generates RewritingHealer header with necessary information. The RewritingHealerVerifier module takes a SOAP message, extract RewritingHealer headers from the message and verify whether the information is correct or not.

We have implemented the RewritingHealerVerifier module as a handler for Axis engine. Previously we saw what is a handler and how it works in the context of Axis. RewritingHealerCreator module is implemented both as a handler of an axis engine and as a general library.

From Figure 7.3 we can see that when the client application sends a message to the server, the message first goes to the RewritingHealerCreator. If the client is using the RewritingHealerCreator library, it has to call a method of that library with the envelope (Root element of a SOAP message) it has prepared to send to the server. Then this library will append necessary information to this envelope and give it back to the client application. Then the client application can send this message to the server.

```
public void invoke( MessageContext mcnxt ) throws AxisFault{
              ////Boolean variable that indicates whether
              ///a message is tampered or not
            boolean tampered = false ;
            ////Retrieve the SOAPMessage form the message context
            SOAPMessage messg = mcnxt.getRequestMessage();
            ////Get the parts of the message
            SOAPPart part = messg.getSOAPPart();
            try{
              ////Retrieves the envelope
              envelope = part.getEnvelope();
              ////This vector data structure contains all the
              ///determined roles for a node
              my_role_set.clear();
              ///This vector data structure contains all the
              /// RewritingHealer header of a SOAP message
              rhset.clear();
              ////This hashmap data structure contains all the Ids
              /// of the signed elements of this SOAP message

              signedidlist.clear();

              ////This hashmap data structure contains all the Ids present
              ///under the <SignedInfoElement> elements of a particular
              ///RewritingHealer header element

              cursignedidlist.clear();

              ////This method determines the role set for a node
              determineRoleSet();

              ////Display the SOAP message in the GUI
              this.gui.setRecievedMessage(envelope.toString());

              ////Create
              this.createSignedIDList();

              /////Retrieve all rewriting healer from the SOAP message
              this.retrieveAllRewritingHealer();

              ////Verify whether the message is tampered or not
              if( !verify()){
                    msg += "Message Tampered\n";

                    tampered = true ;
              }else{
                    msg += "Message was not Tampered\n";
              }
            }catch( Exception e ){
              this.gui.setRecievedMessage(e.toString());
            }
            ///Display the GUI of the rewriting healer
            gui.setVisible(true);

            ////If the message was tampered then generate an AxisFault
            if( tampered ) throw new AxisFault("Message Was Tampered");
      }
```

Figure 7.4: The invoke method of the RewritingHealerVerifier handler

On the other hand, if the client application wants to use the RewritingHealerCreator module as a handler of axis engine the, it has to configure the deployment descriptor configuration file, of the Axis engine. At the same time, a stand-alone Axis engine, not Axis servlet, has to be used if the client wants to use the RewritingHealerCreator handler.

The RewritingHealerVerifier module is only available as a handler of Axis engine. To use this verification module for a service, it has to be configured in the configuration file, server-config.wsdd, of the server side axis engine. When the RewritingHealerVerifier gets the request SOAP message from the client, it determines its role and extract appropriate RewritingHealer header from the received SOAP message. Then it verifies those information. If it could not find any tampering, it lets the message go, otherwise it generates a SOAP fault. Ideally, there should have another two modules in the module structure of RewritingHealer to append the received MessageID from the Request SOAP message in the response SOAP message and to correlate a request MessageID with a response MessageID. We did not implement those modules yet. However, their implementation is trivial. The implementation of those two modules is left as future work.

As we mentioned before, each handler has a special method called invoke(). This method takes one argument namely MessageContext. The MessageContext encapsulates a Message, which in turn

encapsulates a request or a response message along with some properties. This method is the entry point to a handler. When one handler needs to pass the request or response message, it calls the invoke() method of that handler. Figure 7.4 shows a snapshot of the invoke() method of our RewritingHealerVerifier handler. As it can be seen from Figure 7.4 this method throws an AxisFault if any tampering is found on the received message. In this method we are first initializing some data

```
public boolean verify(){

        /////For each role of this node
        for( int i = 0 ; i < my_role_set.size(); i++ ){
              ///Determine the role
             String cur_role = (String)my_role_set.get(i);
             ////Map the role name to an integer value
             int irole = this.mapRole(cur_role);

             ////Retrieve all rewriting healer with this role
             Vector rh = this.retrieveRewritingHealer(irole);

             ///For each rewriting healer with irole

             for( int j = 0 ; j < rh.size(); j++ ){

                    ///Take the RewritingHealer as a SOAP element
                    SOAPElement sel = (SOAPElement)rh.get(j);

                    ///Clear cursignedlist vector
                    cursignedidlist.clear();

                    ////Verify whether the information present in this
                    ///RewritingHealer is correct or not
                    if( !verifyRewritingHealer(sel)) return false;

             }
        }
        return true;
  }
```

Figure 7.5: The verify method of RewritingHealerVerifier

structures and determining the role of the current node. Then we are creating a list containg all the Ids of the signed elements of the current SOAP message. After that we are retrieving all the RewritingHealer header element(s) from the SOAP message and invoking the method verify. The snapshots of this verify method is shown in Figure 7.5. This method returns a Boolean value depending on the status of the verification. If this method returns true then the handler let's the message go further, otherwise it generates an AxisFault. Our RewritingHealerVerifier has a graphical user interface. This GUI shows the received SOAP message and the result of different types of verification like SignedElementInfo verification, Digest value verification etc. Later we will see how this GUI looks like.

```
public SOAPEnvelope invoke( SOAPEnvelope env ){

        try{
                ////This hashmap datastructure will keep all the Ids present
                ///in the current envelope
                idlist.clear();
                ////This hashmap data structure will keep all the Ids of the
                //Signed Elements of the current envelope
                signedidlist.clear();
                ///This hashmap data structure will keep the Ids of signed
                ///Elements that was signed by some previous node
                prevsignedidlist.clear();
                ////This vector data structure will keep the Ids of
                /// all the Ids of RewritingHealer header added by
                ///this node
                added_rewriting_healer.clear();

                ////This variable is used to generate unique Ids
                counter = 0 ;

                this.envelope = env;

                /////Determine the node number
                node_number=findNodeNumber();

                ////This method creates a list of Ids of all the Signed
                ///Elements that is present in the current envelope
                createSignedIDList();

                ////This method creates a list of Ids of all the Signed
                ///that was signed by some previous node
                createPrevSignedIDList();

                ////Retrieve the header of the current envelpe
                javax.xml.soap.SOAPHeader hdr = envelope.getHeader();

                ////Put all the Ids of elements present in this header
                createExistingIDList(hdr);

                ////Retrieve the body of the current envelpe
                javax.xml.soap.SOAPBody body = envelope.getBody();
                ////Put all the Ids of elements present in this body
                createExistingIDList(body);

                ////Provide Id to all unsigned elements of Header
                addIdAttr(hdr,1);
                ////Provide Id to all unsigned elements of Body
                addIdAttr(body,1);

                ////Create RewritingHealer for role UltimateReciepient
                createRewritingHealer("UltimateReciever");
                ////Create RewritingHealer for role Next
                createRewritingHealer("next");
                ////Create RewritingHealer for role none
                createRewritingHealer("none");

                ///Sign all created RewritingHealer using xpath
                this.signRewritingHealer();

                ///Return the envelope to the client application
                return this.envelope;

        }catch( Exception e ){
                System.out.println(e);
                writeToLog(e);
        }
        return null;
    }
```

Figure 7.6: The invoke method of RewritingHealerCreator library

The RewritingHealerCreator module is available as both a general library and as an Axis hanlder. This handler is to be configured on the client side axis engine. However, whether it is a general library or an axis handler, both of them contain an invoke() method. In the case of an axis handler this method takes a MessageContext as its argument and in library this method takes an Envelope as its argument. The rest of the working processes of these two modules are almost identical. Figure 7.6 shows a snap shot of the invoke method of RewritingHealerCreator module. The method takes an Envelope as its argument. In the method, at first some data structures are initialized and the number of the current node is determined. Then two types of Id lists are created. One list contains all the Ids of the signed elements, and the other contains Ids of signed elements that were signed by some previous node. Then another list this method creates another list which will contain the Ids of all the signed and unsigned elements. After that this method gives all the unsigned elements added by this node a unique Id of a specified form and creates RewritingHealerHeader for different roles.

7.3 A Simple Application

We have developed a simple web service application to demonstrate how our proposed method can be used to detect XML Rewriting Attack in real life. The name of our proposed web service is MathService as it provides the service of different arithmetic operations like, addition, subtraction, multiplication, division and exponentiation. Figure 7.7 shows the interface of our MathService. As it can be seen from Figure 7.7, there are five methods in the web service, which perform the five mentioned arithmetic operations. Each of these methods take two arguments a and b. Although for

```
package com.thesis.service;

public class MathService {
     public long add( long a , long b ){
        return a+b;
     }
     public long subtract( long a , long b ){
        return a-b;
     }
     public long multiply( long a , long b ){
        return a*b;
     }
     public long divide( long a , long b ){
        return a/b;
     }
     public long exponent( long a , long b ){
        return (long)Math.pow((double)a, (double)b);
     }
}
```

Figure 7.7: Interface of the Math Service

the addition and multiplication operation the order of the arguments does not matter, for the rest of the operations it does. For instance, for any two integers a and b, (a-b) != (b-a). Moreover, when the client will send a request to this web service, it has to provide a signed subscriber ID as a header element of the request SOAP message. This MathService has a handler, which will validate a client using the subscriber ID.

A web service is published using a web service description language (wsdl), which describes the different interfaces of a web service, its message format and their order etc. The wsdl file for our web service is provided in Appendix A. In axis every service needs to be deployed. There are two ways to deploy a service in Axis.

i) Using the AdminClient tool provided by Axis
ii) By configuring the server-config.wsdd deployment descriptor file of the server side Axis Engine

We will use the second approach to deploy our web service in Axis. We need to add the following lines in this server-config.wsdd file to deploy our web service. Line 001 specifies the name of our

```
001.  <service name="MathService" provider="java:RPC">
002.  <parameter name="allowedMethods" value="add subtract multiply divide exponent"/>
003.  <parameter name="className" value="com.thesis.service.MathService"/>
004.  <parameter name="scope" value="Session"/>
005.  </service>
```

Listing 1

service and the type of message exchange pattern used by our service. We are using RPC style of message exchange. That means a request/response style of message exchange. Line 002 to 003

specifies the methods provided by the service, the name of the main class of that service along with its package name and the scope.
Once we have deployed our web service any client can call this service and use it. Now we want to use our proposed method with our MathService for the protection against XML Rewriting Attack. Our RewritingHealerVerifier is a service specific handler. If a service wants to use it , it has to configure it in the server-config.wsdd. This can be done by adding a couple of more lines with the lines in Listing 1. Listing 2 shows the resulting configuration information.

```
001. <service name="MathService" provider="java:RPC">
002.  <requestFlow>
003.      <handler type="java: com.sid.verifier.VerifySubscriberID"/>
003.      <handler type="java:com.xml.healer.RewritingHealerVerifier"/>
004.  </requestFlow>
005.  <responseFlow>
006.  </responseFlow>
007.   <parameter name="allowedMethods" value="add subtract multiply divide exponent"/>
008.  <parameter name="className" value="com.thesis.service.MathService"/>
009.   <parameter name="scope" value="Session"/>
010. </service>
```

Listing 2

As it can be seen from Listing 2, a new tag <requestFlow> is added with the previous lines. This tag contains the information of all service specific handler in top to bottom order. When a request will come for a service they will pass through all of these handlers in the specified order before reaching the target service. We have specified the name of two handlers in the <requestFlow> tag of Listing 2. The first handler will verify the subscriber ID of the request message and the second one is our RewritingHealerVerifier handler. Therefore, when a request will come for MathService it will first go to VerifySubscriberID handler. Then if the processing of this handler succeeds, it will pass the request to RewritingHealerVerifier, which will verify the request SOAP message against XML rewriting attack. In Listing 2 we can see that another tag <responseFlow> is added. This tag contains the name of all the handlers that should get the response message before being sent to the client.

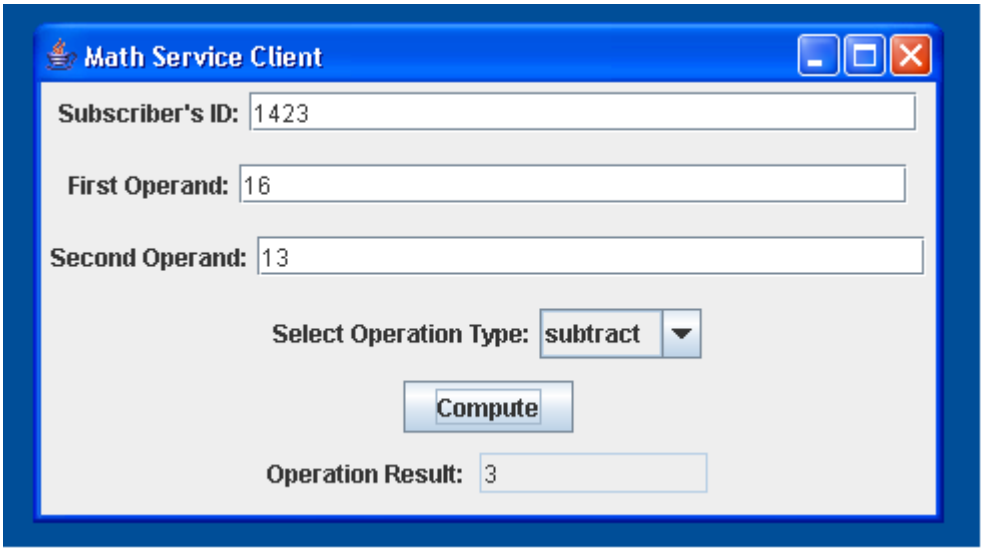

Figure 7.8: MathService Client Interface

We have created a simple client application that will use our MathService. Figure 7.8 shows the graphical user interface of our client. This Figure demonstrates that, the user has to provide a Subscriber's ID , the first and the second operand for the operation. Then the user has to select the

type of opration from a list operation type like add, subtract etc. At last the user will have to press the compute button. The result of the opration is displayed at the bottom of the interface. When the user presses the compute button, the application creates a SOAP message. In this SOAP message there will have three signed elements. The subscriber's ID, the first operand and the second operand. From 7.8 we can see that the user has selected subtraction operation. Therefore, the order of the operands in the SOAP message determines the result of the operation. In Figure 7.8, it is shown that the user has given the first operand's value 16 and the second operands value 13. So the result is 3. However if we change the order of these operands the result will be -3. After the SOAP message is fully created, the client application will give it to our RewritingHealerCreator, which will append necessary information in it and will give it back to the client. After that the client invokes the remote MathService with the SOAP message. The client's code is given in Appendix B. Now lets see what will happen when the RewritingHealerVerifier will receive this message.

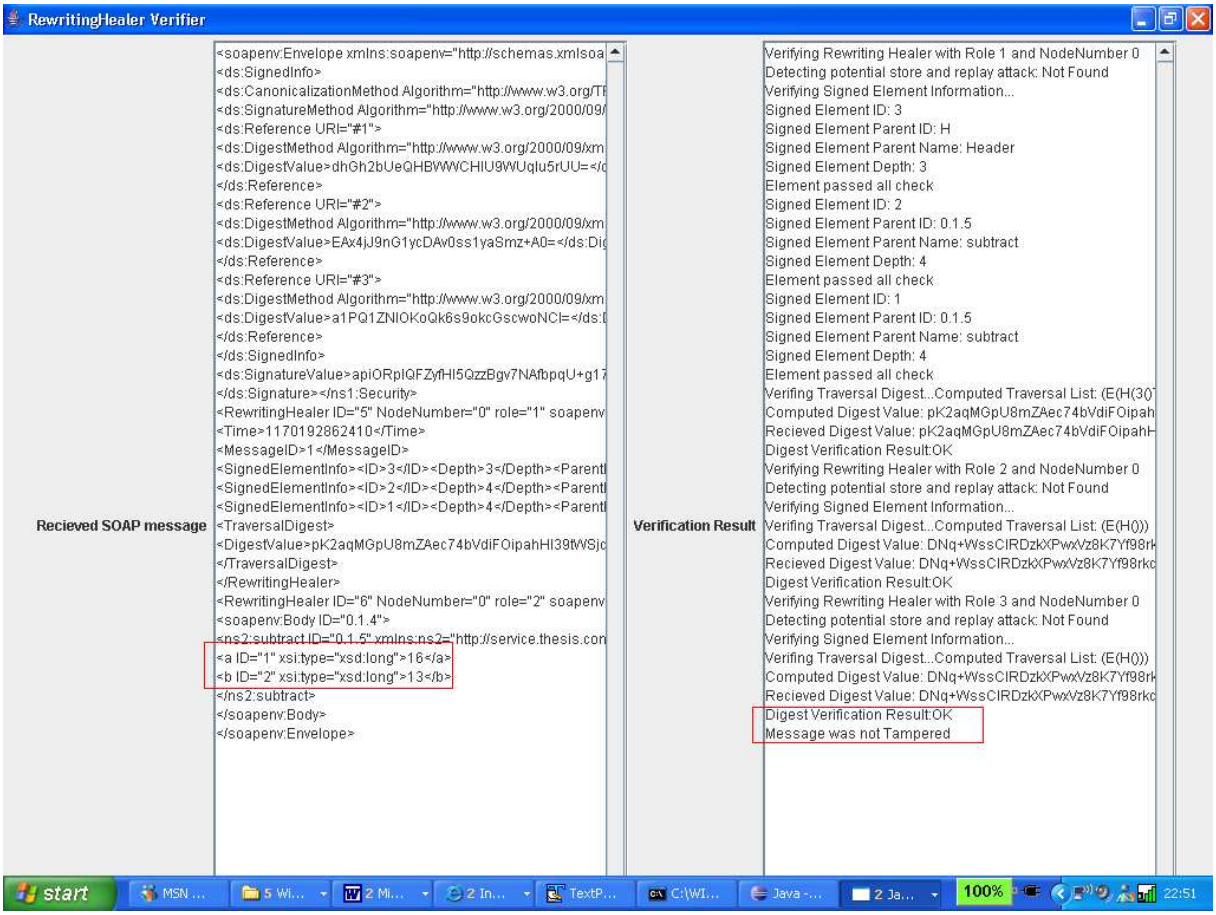

Figure 7.9: The RewritingHealerVerifier interface

The interface of RewritingHealerVerifier has two parts. The left part shows the received SOAP message and the right part shows the result of different types of verification. Figure 7.9 represents the status of the RewritingHealerVerifier when it received the above mentioned SOAP message from the client. The red border rectangle on the Received SOAP message part of Figure 7.9 shows the order of the two signed operands the client sent with the SOAP message. The Received SOAP message part also demonstrates the information added by the RewritingHealerCreator on the client side like the Ids of unsigned elements, the RewritingHealer header elements etc. The Verification Result part shows diferrent verification result. The red border rectangle on the right side of Figure 7.9 shows the final result of the verification. It is saying that no tampering was done on the Received SOAP message. The Verification Result also demonstrates the different steps of the verification.

Now lets assume that an attacker in the middle of the client and the server intercepted the above client SOAP message and changed the order of the signed operands. For the reason we described previously, this modification will not have any effect on the signature value of those signed operands. Moreover, all the previous solutions for XML Rewriting Attack will not be able to detect this modification. Therefore, the attacker will easily be able to get unauthorized access to our service. We have created a simple attacker for this demonstration. The code for the attacker is given in Appendix B.

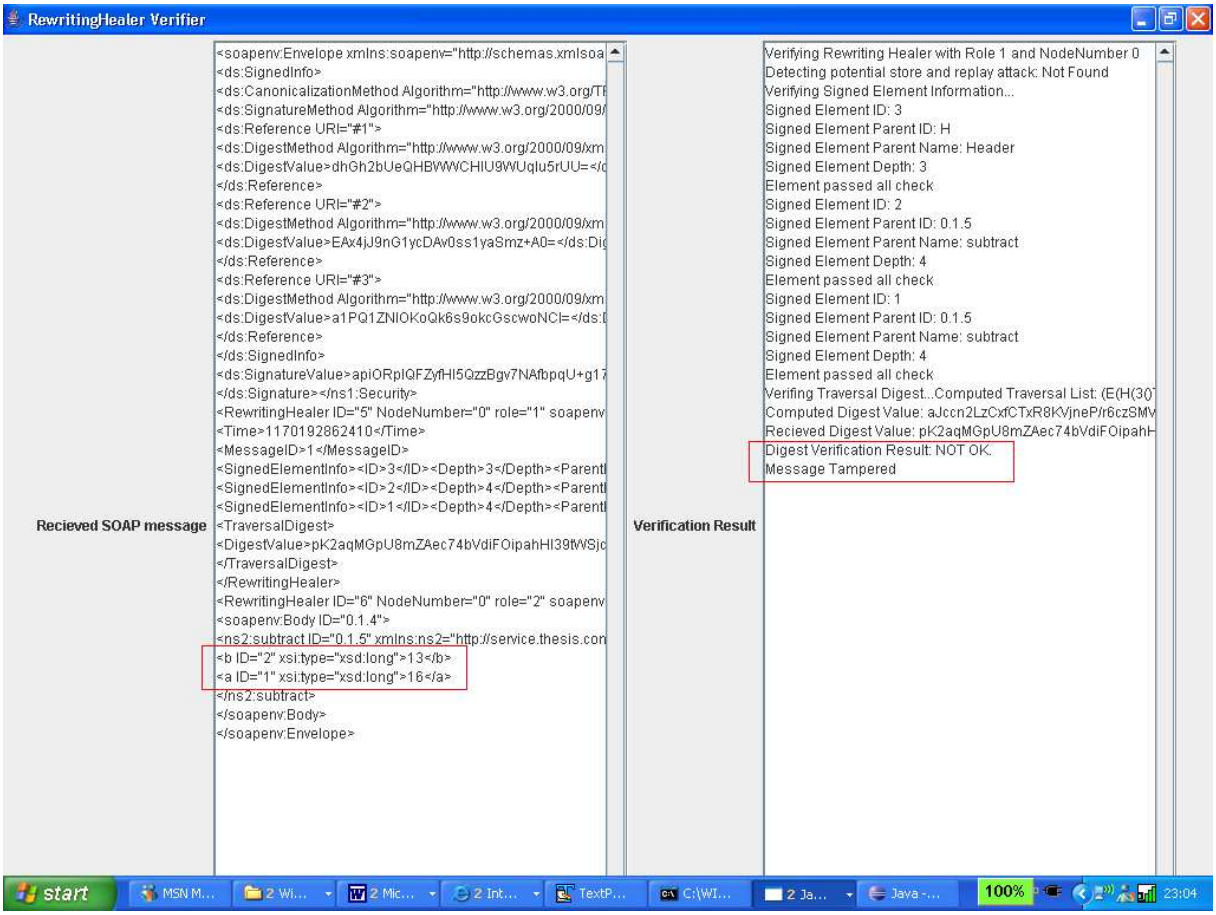

Figure 7.10: RewritingHealerVerifier detecting XML Rewriting Attack

Figure 7.10 shows the status of the RewritingHealerVerifier when it receives the SOAP message from the attacker. The red border rectangle in Received SOAP message part shows that the order of the operands has changed by the attacker. In the Verification Result part we can see that, the verifier found all the information in the SignedElementInfo of the RewritingHealer header correct. However when it compares the computed digest value with the one present in the RewritingHealer header, it found the attack, as the two digest values are not equal. The red border rectangle on the Verification Result part shows the final verification result.

We have shown already that with our proposed method, it is not possible for an attacker to add or delete an element to or from a SOAP message. It is also not possible for the attacker to change the depth of an element. The only modification the attacker can make is the re-ordering of elements. However, if he reorders the signed elements of a SOAP message our demonstration showed that our algorithm detects this modification successfully. Moreover, in our demonstration we only demonstrated the signed element reordering attack and its detection. In chapter 6 we saw how our algorithm can detect other types of attack.

7.4 Evaluation

In this section we will try to provide a simple evaluation of our proposed approach. Our evaluation will be based on the following criteria:

i) Processing Time
ii) Bandwidth Consumption
iii) Attack Detection

Processing Time:

Here we will analyze the time complexity of our proposed approach. First we will analyze the time complexity for RewritingHealer verification process.

Lets assume that the message contains k RewritingHealer header. We should note here that each node could append 3 header blocks for three respective roles. However, each subsequent intermediary will remove the header block for role "Next". Therefore we can assume that each node append 2 header blocks. It is very unlikely that there would be more than 20 intermediaries. So the total number of header blocks will not exceed 41 normally. We assume that we can retrieve all the RewritingHealer header blocks in constant time. It is obvious, as we know that they will be the immediate children of the Header of SOAP envelope.
Now if we map the ids of signed elements to its associated information (depth, parent-name, parent-id, node_number, role) in a hash table, then the following task can be done in constant time C1

iv) Determining whether a element was previously signed or not
v) Determining whether a element was signed by a particular node with a particular role
vi) Determining whether a element is signed at all or not
vii) Retrieving the associated information of a particular signed element

This hash tables can be created in C*k*n1 times, where k is the number of header blocks and n1 is the total number of signed elements ids in all of the header blocks and C is some constant value.

For each header block, traversing the whole tree would take O(n) times. However at the time of this traversal it is possible to perform the following task in constant time using our created hash tables:

i) Determining an element is signed or not
ii) Determining if it is signed by a node with current node number and role
iii) Retrieving the associated information of a particular signed element

Moreover while traversing we can keep track of the depth of a particular element. Whenever it is determined that a particular element is signed and it is signed by the node whose added RewritingHealer header block is being verified, the other verifications like depth is equal or not, parent name is equal or not etc, would take a constant time C2.

Therefore, for a particular RewritingHealer header block the time complexity of the whole verification process is:

$$C1*C2*O(n)$$

Where n is the number of elements of the SOAP message and C1 and C2 are the constants we specified before.
So for k header blocks the time complexity of the verification process would be:

$$C1*C2*k*O(n)$$

Where n is the number of elements of the SOAP message, k is the total number of RewritingHealer header block; C1 and C2 are the constants we specified before.

Therefore our overall time complexity is:

	C*k*n1+C1*C2*k*O(n)	
⇨	C*k*n1+C1*C2*k*C3*n	(we can write O(n) = C3*n for some constant C3)
⇨	C*k*n+C1*C2*C3*k*n	(assuming n1 < n)
⇨	C4*k*n	(assuming C4 = C+C1*C2*C3)
⇨	O (k*n)	

Therefore, if there are no intermediaries or none of the intermediaries appended any information in the SOAP message the verification process complexity would become O (n).

In the same way we can show that the time complexity for the creation of RewritingHealer header block is also O(kn) where k is the total number of header blocks present in the SOAP message and n is the total number of elements in the SOAP message.

It is worth mentioning here that our RewritingHealer approach eliminates the necessity of WS Security policy for the detection of XML Rewriting Attack. In WS Security Policy, each assertion needs to have a module to verify whether a message satisfies the assertion or not. Therefore, if there are assertions in a WS Security policy file associated with the different patterns of XML Rewriting Attack, each of these assertions will have a module and each of these modules will have to process the message. It is certainly a time consuming procedure.

Bandwidth Consumption:

The main limitation of our RewritingHealer approach is that, it is bandwidth consuming. Each node on the processing path of a SOAP message might add one or more RewritingHealer header depending on the destination of their added elements. We should note here that, the SOAP account approach is also adding SOAP Account header element. And if there are intermediaries, more than one SOAP account might be added to the message. In our RewritingHealer header block there are three types of information depending on which node has created the header block.

1. The Signed Element information
2. The Digest value of the pre order traversal list.
3. MessageID and Time Information. This information is only present in the RewritingHealer header block created by the Initial node.

The length of the digest value is always fixed which is only 256 bits that means 32 bytes. However the length of the signed element information is not fixed. It depends on the number of signed elements added by a node. The more signed elements a node will append in a SOAP message, the more overhead our RewritingHealer approach will introduce in the SOAP message. To get an estimation of the overhead of our proposed approach we ran some evaluation test. Table 2 depicts the result of our evaluation test. We have assumed that there are only two nodes, the Initial Sender and the UltimateReceiver. We created a SOAP message with 13 elements. Initially only one of the 13 elements is signed. We recorded the length of the SOAP message before the addition of our RewritingHealer information. Then we appended our RewritingHealer information and again recorded the total length of the SOAP message. The difference between the two recorded information represents the overhead introduced by our RewritingHealer information. We repeated the above process 12 times. Each iteration signs one more element than the previous one. From Table 2, it can be seen that, as new elements are getting signed the overhead of our RewritingHealer approach is increasing quite rapidly.

Number of Unsigned Elements	Number of Signed Elements	Total Size of SOAP message before addition of RewritingHealer header (bytes)	Total Size of SOAP message after addition of RewritingHealer header (bytes)	Overhead (bytes)	Overhead (%)
12	1	4932	5565	633	11.37
11	2	5242	5994	752	12.55
10	3	5552	6442	870	13.55
9	4	5862	6850	988	14.42
8	5	6172	7278	1106	15.20
7	6	6482	7706	1224	15.88
6	7	6792	8134	1342	16.50
5	8	7102	8562	1460	17.05
4	9	7413	8992	1579	17.56
3	10	7724	9422	1698	18.02
2	11	8035	9852	1817	18.44
1	12	8346	10282	1936	18.83

Table 2: Overhead estimation of RewritingHealer (without optimization)

When the message contains 1 signed element the overhead of our approach is 11.37%, however when the message contains 12 signed elements the overhead raised to almost 19%. Some optimizations can be done in our RewritingHealer approach to minimize this overhead. We have found that only the signed elements ID and the Digest value alone can detect all types of XML Rewriting Attack. Therefore, if each RewritingHealer header block contains only the Signed Elements Ids and the digest value, it would be enough to detect all rewriting attacks. We ran the same evaluation test specified above with this optimization. Table 3 depicts the result of this evaluation test with optimization.

Number of Unsigned Elements	Number of Signed Elements	Total Size of SOAP message before addition of RewritingHealer header (bytes)	Total Size of SOAP message after addition of RewritingHealer header (bytes)	Overhead (bytes)	Overhead (%)
12	1	4932	5495	563	10.25
11	2	5242	5864	612	10.45
10	3	5552	6213	661	10.64
9	4	5862	6572	710	10.80
8	5	6172	6931	759	10.95
7	6	6482	7290	808	11.08
6	7	6792	7649	857	11.20
5	8	7102	8008	906	11.31
4	9	7413	8369	956	11.42
3	10	7713	8719	1006	11.54
2	11	8024	9080	1056	11.63
1	12	9441	1106	1106	11.71

Table 3: Overhead estimation of RewritingHealer (with optimization)

From Table 3 we can see that with optimization the overhead introduced by our approach has been reduced almost by a factor of 2. With one signed element the overhead is 10.25% and with 12 signed elements the overhead is 11.71%, which was 18.83% without optimization. Figure 7.11 represents a

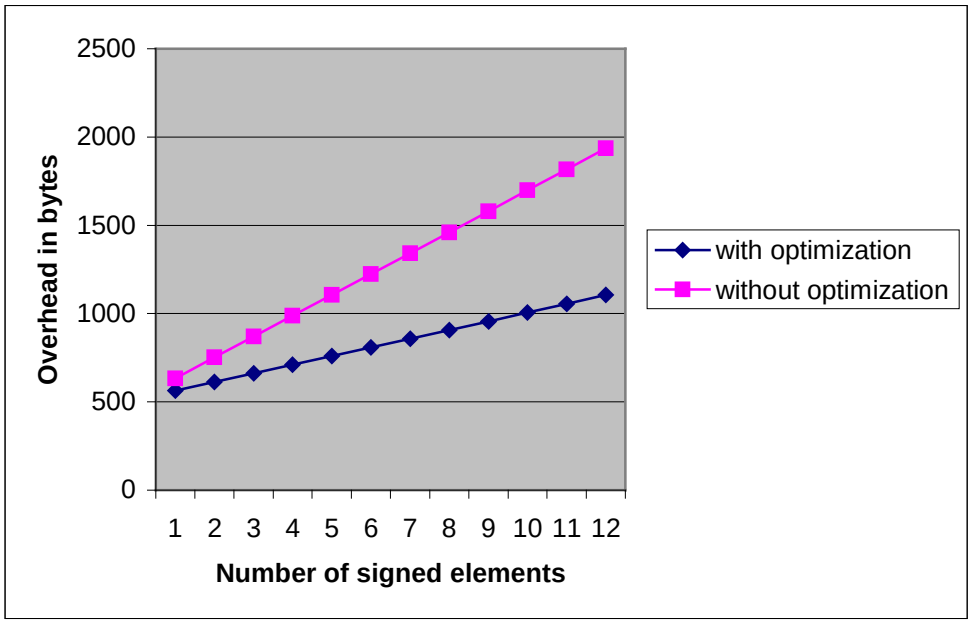

Figure 7.11: Comparison of the overhead introduced by RewritingHealer approach with and without Optimization

chart, which shows how the proposed optimization can reduce the overhead of RewritingHealer approach. We have plotted number of signed elements along the X-axis and the overhead in bytes along the Y-axis of the chart. It can be seen from the chart that RewritingHealer has a linear overhead growth rate both with and without optimization. However, with optimization the growth rate has substantially reduced.
Therefore, in the future it has to be proved formally that only the Signed Elements ID and the digest value can detect any sort of rewriting attacks then the structure of the RewritingHealer can be changed to contain only the specified information.

Attack Detection:

Every XML Rewriting Attack performs some sort of modifications on the SOAP message. As we said previously XML Rewriting Attack takes place as XML Digital Signature permits the use of XPointer to refer a signed element and XPointer does not contain any information regarding the location of the referenced element. However, XPointer is not the only reason for the existence of XML Rewriting Attack. We showed previously that, even XPath is used instead of XPointer, XML Rewriting Attack can still take place. This is due to the fact that SOAP [1] allows the existence of header block that is not recognized by the receiver. Therefore, if the receiver finds a header block that it does not understand, it cannot make complain if the header block is not mandatory. Moreover, WS-Security [6] allows the existence of multiple security headers with the same name in a single SOAP message. Therefore, it is possible for the attacker to redirect the XPath expression to a different element. The concept of XML Rewriting Attack assumes that the attacker does not own the proper key to sign an element. He/She changes the SOAP message in a way that does not compromise the signature of that element. That means the attacker neither can delete a signed element nor can add a signed element in a SOAP message. The attacker also cannot modify any information of a signed element. However, the attacker can change the location of a signed element and this is the only way for him/her to perform an

attack. Therefore, any solution of XML Rewriting Attack should have to provide a way for identifying the location of signed elements.

We discussed previously different solutions for XML Rewriting Attack. Most of the solutions tried to identify the location of signed elements. In the first solution, they have used XPath with WS Security Policy to identify the location of signed elements. But we showed how this method could become vulnerable. The SOAP Account approach is also trying to identify the location of signed elements by associating information regarding the signed element in a header block. However, We showed that the information present in the SOAP account header block is not sufficient enough to prevent XML Rewriting Attack. Most importantly, the attack where the attacker changes the order of signed elements of a SOAP message cannot be detected by any of the previously proposed solutions.

The solution that we have provided here prevents the attacker from performing the following task in a SOAP message:

i) The attacker cannot add a new element in a SOAP message
ii) The attacker cannot delete an existing element from a SOAP message
iii) The attacker cannot change the order of the signed elements of a SOAP message

We are representing the SOAP message, which is in fact an XML document, using a tree structure. Then we are providing each unsigned element a unique identifier. In our discussion so far we have assumed that a signed element will be referenced from the Signature element using XPointer and therefore it already contains an identifier. However, if XPath is used for referencing, the name of a signed element can be used as its identifier. Then we are taking the string representation of the pre-order traversal list of the tree. In this string each element is represented using its name and its identifier separated by a colon. After the string is ready we are taking the digest value using SHA-256 algorithm of that string. We then put this digest value in our proposed RewritingHealer header block and sign this header block. To prevent rewriting attack on our header block we have imposed a restriction that RewritingHealer header block must be referenced using XPath expression from the Signature element. Therefore, we can say that our created digest value is indirectly signed. That means, although the digest value is not signed by itself, the header block that contains this digest value is signed.

Now if an attacker adds a new element in the SOAP message, it would definitely change the string representation of the pre-order traversal list of the SOAP message elements. This will in turn change the digest value.

Lets consider the following general string as an example. We can think of string S as the string representation pre-order traversal list of a SOAP message. :

S = “ABCDEFGHIJ”

Let the digest value of that string is D. That is

MD(S) = D, where MD is a digest value function

Now if the attacker adds a new element with identifier K in the SOAP message the new string representation of the pre-order traversal list of the SOAP message would become something like the following

S1 = “ABCD**K**EFGHIJ”

Let the digest value of this string is D’. That is

MD(S1) = D’, where MD is a digest value function

Now as S1 != S, from the properties of hash function (see section 2.1.6) we can say that:

D' != D.

Moreover, it is not possible for the attacker to generate the digest value and put it in the RewritingHealer header. This is because the RewritingHealer header block is signed and if the attacker wants to do this he/she would have to sign the RewritingHealer header block, which we presumed is out of the attacker's capability. Therefore, we can conclude that the attacker cannot add new elements in the SOAP message. In the same way we can show that the attacker cannot delete an existing element from the SOAP message.

Now lets see what happens when the attacker changes the order of signed elements. Figure 7.11

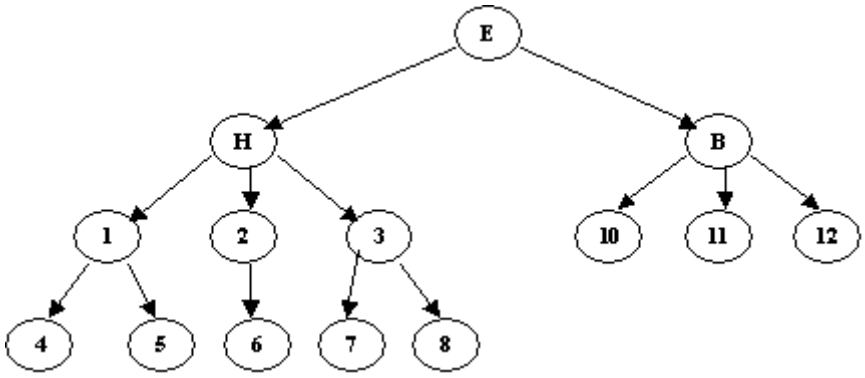

Figure 7.12: A simple tree representing a SOAP message

Depicts the tree representation of a SOAP message. Each node of the tree is representing an element of a SOAP message. Nodes with label E, B and H represent the Envelope element, Body element and the Header element of a SOAP message. The rest of the nodes represent other elements of a SOAP message and they are labelled using the ID value of the respective element. According to the string representation rule of the pre-order traversal list of a SOAP message that we specified in section 6.1 , the tree in 7.11 can be represented as

S2 = (E(H(1(4()5())2(6())3(7()8()))B(10()11()12())))

Now lets assume that the nodes with label 4 and 5 are representing two signed elements. As they are signed it is not possible for an attacker to perform any sort of modification on these elements. However, lets assume that the attacker wants to change the order of these two signed elements. As we specified before that the attacker could relocate a signed element and hence he can reorder the two signed elements with Id 4 and 5. The resulting tree after this modification is represented in Figure 7.12.

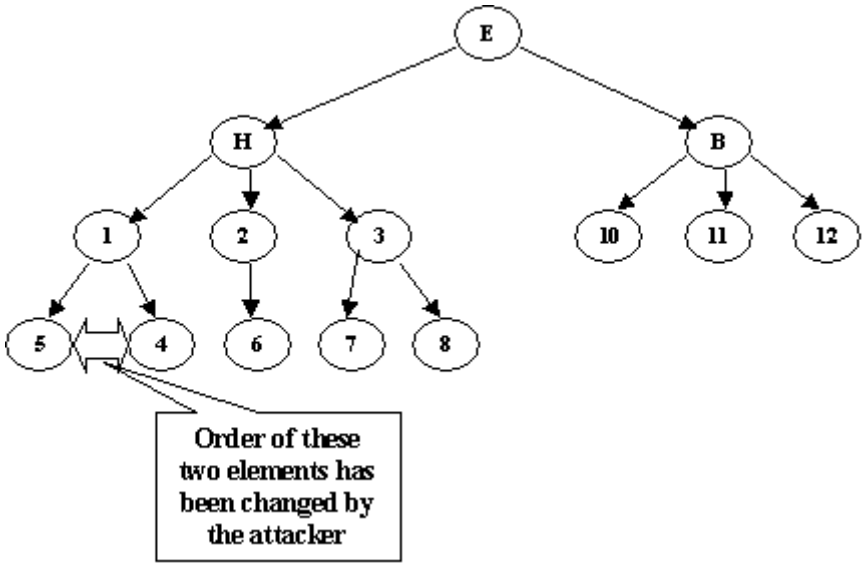

Figure 7.13: Tree of Figure 7.11 after elements order has been changed

Now the string representation of the pre-order traversal list of the tree of Figure 7.12 will become as follows:

S3 = (E(H(1(**5()4()**)2(6())3(7()8()))B(10()11()12())))

If we compare the two strings S2 and S3, we can see that due to the modification, the two strings S2 and S3 are no longer equal. Therefore, their digest value will not be equal as well according to the properties of hash function. Consequently, the above modification of the attacker will be easily detected by our approach. So the attacker cannot change the order of signed elements of a SOAP message.

We have shown that the attacker cannot add any new elements in a SOAP message, cannot delete an existing element from a SOAP message and cannot reorder signed elements of a SOAP message. However, if the attacker is prevented in performing any of these tasks, he/she is in turn prevented from performing XML Rewriting Attack. Therefore, we can conclude that our approach provides a mechanism for the detection of any sort of XML Rewriting Attack.

Chapter 8: Conclusions and Future Work

The main goal of RewritingHealer is to guard SOAP messages against XML Rewriting Attack. Although all of the previous solutions tried to fulfil the same goal as ours, we have shown scenarios where these solutions are vulnerable. Most importantly none of the previous solution can guard against the signed element reordering attack. However, we have already shown that our RewritingHealer can guard against a wide range of XML Rewriting Attack. One of the future works of our proposed method is to prove formally that, using the digest value of pre-order traversal prevents the attacker to add or delete or modify any signed element form the SOAP message. We are now using only the pre-order traversal technique for the generation of an ordered list of SOAP message elements. Another future work might be to find other traversal techniques for the ordered representation of SOAP message elements. One important point here is that, the traversal techniques that will be used must generate a unique ordered representation of the SOAP elements. That means the traversal techniques must satisfy the following proposition where our universe of discourse is the set of all general trees.

$$\forall t1((X = R(t1)) => !(\exists t2((t1 != t2) \Lambda (X = R(t2)) \text{ , Where } R(x) \text{ is the traversal of tree } x$$

Therefore, along with the discovery of new traversal techniques, the future work must also prove that the traversal techniques satisfy the above proposition.

In this report initially we have discussed about different security protocols like symmetric key cryptography, public key cryptography, digital signature, digital envelope and hashing. This discussion gives an idea regarding the security technologies that prevails in current digital communication. Then we have given an overview of web service technology and SOAP. Different web services security standards like XML Digital Signature, WS Security, WS Policy and WS Security Policy and their limitations have been discussed in brief. We have demonstrated with scenarios different types of XML Rewriting Attack that can take place in web service communication. We have specified some previous works that have been done for the detection of XML Rewriting Attack. We have shown with vulnerability scenarios that none of these solutions can fully eliminate XML Rewriting Attack.

Then we have proposed an approach for the detection of XML Rewriting Attack and demonstrated its working process with different scenarios. We also have implemented rudimentary modules to illustrate the attack detection capability of our approach. We have evaluated our approach based on processing time, bandwidth consumption and attack detection capability. Moreover, we have discussed some previous works that have been done for the detection of XML Rewriting Attack and demonstrated their vulnerabilities with scenarios. In our future work we will establish a formal prove showing that our approach prevents any sort of signed element modification in the SOAP message. We will also implement our approach with the optimizations that we specified in section 7.4.

Chapter 9: References

[1] SOAP Version 1.2 Part 0 Primer W3C Recommendation
[2] XML-Signature Syntax and Processing, W3C Recommendation 12 February 2002
[3] WS-Policy specification at
http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnglobspec/html/ws-policy.asp ,2007
[4] Georgiadis C., Mavridis I. and Pangalos G.
http://www.ibm.com/software/solutions/Webservices/pdf/WSCA.pdf , 2007
[5] Web service Essential, Publisher, Orielly, ISBN: 0-596-00224-6
[6] Web services Security: SOAP Message Security 1.0 (WS-Security 2004) OASIS Standard 200401, March 2004
[7] Web services Security Policy Language (WS-Security Policy)
[8] TulaFale: A Security Tool for Web services Karthikeyan Bhargavan, C´edric Fournet, Andrew D. Gordon, and Riccardo Pucella at Microsoft Research
[9] An Advisor for Web services Security Policies by Karthikeyan Bhargavan C´edric Fournet Andrew D. Gordon Greg O'Shea at Microsoft Research
[10] A Multi-party Implementation of WS-SecureConversation Hongbin Liu, Geoffrey Fox, Marlon Pierce, Shrideep Pallickara Community Grids Lab, Indiana University, Bloomington, Indiana 47404
[11] XML Signature Element Wrapping Attacks and Countermeasures by Michael McIntosh , Paula Austel from IBM research
[12] Towards Secure SOAP Message Exchange in a SOA by Mohammad Ashiqur Rahman
[13] An Inline Approach for Secure SOAP Requests and Early Validation by Mohammad Ashiqur Rahman
[14] Securing Web services with WS-Security: Demystifying WS-Security, WS-Policy, SAML, XML Signature, and XML Encryption by Jothy Rosenberg, David Remy
[15] http://msdn2.microsoft.com/en-us/library/ms954606.aspx, 2007
[16] Semantics-enriched QoS Policies for Web service interactions by Diego Zuquim Guimarães Garcia and Maria Beatriz Felgar de Toledo
[17] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. In 11th ACM Conference on Computer and Communications Security (CCS'04), pages 268–277, October 2004.
[18] Network Security: Private communication in a public word. By Charlie Kaufman, Radia Perlman, Mike Spencier
[19] A. D. Birrell. Secure communication using remote procedure calls. ACM Transactions on Computer Systems,3(1):1–14, 1985.
[20] Microsoft Research; http://research.microsoft.com/projects/Samoa/ , 2007
[21] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. In 11th ACM Conference on Computer and Communications Security (CCS'04), pages 268–277, October 2004.

[22] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. TulaFale: A security tool for web services. In International Symposium on Formal Methods for Components and Objects (FMCO'03), LNCS. Springer, 2004
[23] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In Proceedings of the 14th IEEE Computer Security Foundations Workshop, pages 82–96. IEEE Computer Society Press, 2001.
[24] D. Balfanz, D. Dean, and M. Spreitzer. A security infrastructure for distributed Java applications. In Proceedings of the IEEE Symposium on Security and Privacy, pages 15–26. IEEE Computer Society Press, 2000.
[25] L. C. Paulson. Inductive analysis of the internet protocol TLS. ACM Trans. Inf. Syst. Secur., 2(3):332–351,1999.
[26] Isabelle, http://isabelle.in.tum.de/ , 2006
[27] M. Abadi, B. Blanchet, and C. Fournet. Just fast keying in the pi calculus. In Proceedings of the 13th European Symposium on Programming (ESOP'04), volume 2986 of LNCS, pages 340–354. Springer, 2004.
[28] WS-Trust; http://specs.xmlsoap.org/ws/2005/02/trust/WS-Trust.pdf, 2006
[29] WS-SecureConversation;
http://specs.xmlsoap.org/ws/2005/02/sc/WS-SecureConversation.pdf , 2006
[30] K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. In 2004 ACM Workshop on Secure Web services (SWS), pages 11–22, October 2004.
[31] K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. In 31st ACM Symposium on Principles of Programming Languages (POPL'04), pages 198–209, 2004. An extended version appears as Microsoft Research Technical Report MSR–TR–2003–83.
[32] WSE, http://www.microsoft.com/downloads/details.aspx?FamilyId=FC5F06C5-821F-41D3-A4FE- 6C7B56423841&displaylang=en
[33] Mobile Values, New Names, and Secure Communication ;
http://research.microsoft.com/~fournet/papers/mobile-values-new-names-and secure-communication.pdf , 2007
[34] D. Dolev and A.C. Yao. On the security of public key protocols. IEEE Transactions on Information Theory, IT–29(2):198–208, 1983.
[35] A Semantics for Web services Authentication , Karthikeyan Bhargavan , C´edric Fournet, Andrew D. Gordon , February 2004 , Technical Report , MSR–TR–2003–83
[36] http://www-128.ibm.com/developerworks/library/ws-secroad/ , 2007
[37] D. Dolev and A. Yao. On the security of public key protocols. IEEE Transactions on Information Theory, IT–29(2):198–208, 1983.
[38] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. Commun. ACM, 21(12):993–999, 1978.
[39] http://ws.apache.org/axis/java/architecture-guide.html , 2007
[40] http://xml.apache.org/security/ , 2007
[41] http://www.xml.com/pub/a/2001/08/08/xmldsig.html , 2007
[42] http://www.w3.org/Submission/WS-Policy/ , 2006
[43] Exclusive XML Canonicalization,Version 1.0,W3C Recommendation 18 July 2002
[44] WS Addressing; http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/, 2007
[45] Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI, 2nd Edition , By Steve Graham, Doug Davis, Simeon Simeonov, Glen Daniels, Peter Brittenham, Yuichi Nakamura, Paul Fremantle, Dieter Koenig, Claudia Zentner. ISBN-10: 0-672-32641-8; ISBN-13: 978-0-672-32641-7

Appendix A:

WSDL of MathService:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://service.thesis.com"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://service.thesis.com" xmlns:intf="http://service.thesis.com"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->

   <wsdl:message name="addRequest">
      <wsdl:part name="a" type="xsd:long"/>
      <wsdl:part name="b" type="xsd:long"/>
   </wsdl:message>
   <wsdl:message name="exponentRequest">
      <wsdl:part name="a" type="xsd:long"/>
      <wsdl:part name="b" type="xsd:long"/>
   </wsdl:message>
   <wsdl:message name="divideRequest">
      <wsdl:part name="a" type="xsd:long"/>
      <wsdl:part name="b" type="xsd:long"/>
   </wsdl:message>
   <wsdl:message name="divideResponse">
      <wsdl:part name="divideReturn" type="xsd:long"/>
   </wsdl:message>
   <wsdl:message name="addResponse">
      <wsdl:part name="addReturn" type="xsd:long"/>
   </wsdl:message>
   <wsdl:message name="multiplyRequest">
      <wsdl:part name="a" type="xsd:long"/>
      <wsdl:part name="b" type="xsd:long"/>
   </wsdl:message>
   <wsdl:message name="subtractRequest">
      <wsdl:part name="a" type="xsd:long"/>
      <wsdl:part name="b" type="xsd:long"/>
   </wsdl:message>
   <wsdl:message name="exponentResponse">
      <wsdl:part name="exponentReturn" type="xsd:long"/>
   </wsdl:message>
   <wsdl:message name="subtractResponse">
      <wsdl:part name="subtractReturn" type="xsd:long"/>
   </wsdl:message>
   <wsdl:message name="multiplyResponse">
      <wsdl:part name="multiplyReturn" type="xsd:long"/>
   </wsdl:message>
   <wsdl:portType name="MathService">
```

```
      <wsdl:operation name="add" parameterOrder="a b">
         <wsdl:input message="impl:addRequest" name="addRequest"/>
         <wsdl:output message="impl:addResponse" name="addResponse"/>
      </wsdl:operation>
      <wsdl:operation name="divide" parameterOrder="a b">
         <wsdl:input message="impl:divideRequest" name="divideRequest"/>
         <wsdl:output message="impl:divideResponse" name="divideResponse"/>
      </wsdl:operation>
      <wsdl:operation name="multiply" parameterOrder="a b">
         <wsdl:input message="impl:multiplyRequest" name="multiplyRequest"/>
         <wsdl:output message="impl:multiplyResponse" name="multiplyResponse"/>
      </wsdl:operation>
      <wsdl:operation name="subtract" parameterOrder="a b">
         <wsdl:input message="impl:subtractRequest" name="subtractRequest"/>
         <wsdl:output message="impl:subtractResponse" name="subtractResponse"/>
      </wsdl:operation>
      <wsdl:operation name="exponent" parameterOrder="a b">
         <wsdl:input message="impl:exponentRequest" name="exponentRequest"/>

         <wsdl:output message="impl:exponentResponse" name="exponentResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="MathServiceSoapBinding" type="impl:MathService">
      <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="add">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="addRequest">
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://service.thesis.com" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="addResponse">
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://service.thesis.com" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="divide">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="divideRequest">
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://service.thesis.com" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="divideResponse">
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://service.thesis.com" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="multiply">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="multiplyRequest">
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://service.thesis.com" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="multiplyResponse">
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://service.thesis.com" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="subtract">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="subtractRequest">
```

```
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://service.thesis.com" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="subtractResponse">
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://service.thesis.com" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="exponent">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="exponentRequest">
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://service.thesis.com" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="exponentResponse">
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://service.thesis.com" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="MathServiceService">
      <wsdl:port binding="impl:MathServiceSoapBinding" name="MathService">
         <wsdlsoap:address
location="http://localhost:8080/axis/services/MathService"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Web service Deployment Descriptor of Server Axis Engine

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
 <globalConfiguration>
  <parameter name="adminPassword" value="admin"/>
  <parameter name="attachments.Directory" value="D:\Tomcat\apache-tomcat-
4.1.32\webapps\axis\WEB-INF\attachments"/>
  <parameter name="sendMultiRefs" value="true"/>
  <parameter name="sendXsiTypes" value="true"/>
  <parameter name="attachments.implementation"
value="org.apache.axis.attachments.AttachmentsImpl"/>
  <parameter name="sendXMLDeclaration" value="true"/>
  <requestFlow>
   <handler type="java:org.apache.axis.handlers.JWSHandler"/>
  </requestFlow>
 </globalConfiguration>
 <handler name="LocalResponder"
type="java:org.apache.axis.transport.local.LocalResponder"/>
 <handler name="URLMapper" type="java:org.apache.axis.handlers.http.URLMapper"/>
 <handler name="RPCDispatcher"
type="java:org.apache.axis.providers.java.RPCProvider"/>
 <handler name="Authenticate"
type="java:org.apache.axis.handlers.SimpleAuthenticationHandler"/>
 <handler name="MsgDispatcher"
type="java:org.apache.axis.providers.java.MsgProvider"/>
 <handler name="soapmonitor"
            type="java:org.apache.axis.handlers.SOAPMonitorHandler">
     <parameter name="wsdlURL"
                value="/axis/SOAPMonitorService-impl.wsdl"/>
     <parameter name="namespace"
```

```
                value="http://tempuri.org/wsdl/2001/12/SOAPMonitorService-
impl.wsdl"/>
     <parameter name="serviceName" value="SOAPMonitorService"/>
     <parameter name="portName" value="Demo"/>
   </handler>
   <service name="SOAPMonitorService" provider="java:RPC">
     <parameter name="allowedMethods" value="publishMessage"/>
     <parameter name="className"
                value="org.apache.axis.monitor.SOAPMonitorService"/>
     <parameter name="scope" value="Application"/>
   </service>
 <service name="MathService" provider="java:RPC">
   <requestFlow>
      <handler type="java:com.xml.healer.RewritingHealerVerifier"/>
  </requestFlow>
   <parameter name="allowedMethods" value="add subtract multiply divide exponent"/>
   <parameter name="className" value="com.thesis.service.MathService"/>
   <parameter name="scope" value="Session"/>
  </service>

 <service name="AdminService" provider="java:MSG">
  <parameter name="allowedMethods" value="AdminService"/>
  <parameter name="enableRemoteAdmin" value="false"/>
  <parameter name="className" value="org.apache.axis.utils.Admin"/>
  <namespace>http://xml.apache.org/axis/wsdd/</namespace>
 </service>
  <service name="Version" provider="java:RPC">
  <parameter name="allowedMethods" value="getVersion"/>
  <parameter name="className" value="org.apache.axis.Version"/>
 </service>

 <transport name="http">
  <requestFlow>
   <handler type="URLMapper"/>
   <handler type="java:org.apache.axis.handlers.http.HTTPAuthHandler"/>
  </requestFlow>
 </transport>
 <transport name="local">
  <responseFlow>
   <handler type="java:org.apache.axis.transport.local.LocalResponder"/>
  </responseFlow>
 </transport>
</deployment>
```

Appendix B:

MathService Attacker Code:

```
import java.io.FileInputStream;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.message.SOAPEnvelope;

////This class represents an attacker of our MathService
public class Attacker {
      public Attacker(){

      }
      public static void main(String unused[]) throws Exception {
            ////The mofidied SOAP message is in the file attack.xml
            FileInputStream fin = new FileInputStream("attack.xml");
            ///Create an Envelope with the modified SOAP message
            SOAPEnvelope env = new SOAPEnvelope(fin);
            ////The endpoint of MathService
```

```
        String endpoint ="http://localhost:8080/axis/services/MathService";
            ////Create a Service
              Service service = new Service();
              ///Create a Call.
              Call call = (Call)service.createCall();
               try{
                     ///Set the target endpoint in the Call
                     call.setTargetEndpointAddress(new
java.net.URL(endpoint));
                     ///Invoke the service
                     SOAPEnvelope ret= call.invoke(env);
                     ///Print the response
                     System.out.println(ret);
               }catch( Exception e ){
                     e.printStackTrace();
               }


         }

}
```