NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

WEB SERVICES SECURITY

Graduation Project
COM-400

Student: Özgür Selver

Supervisor: Mr. Jamal Abuhasna

Nicosia-2006

ACKNOWLEDGMENT

First, I wish to thank my adviser, Mr. Jamal Abuhasna, for intellectual support, encouragement, enthusiasm which made this project possible, and for his patience in correcting both my stylistic and scientific errors.

Second, I thank Prof. Fakhreddin Memedov for stimulating discussions and for providing the fast Fourier transform routine used in classical calculations.

Third, I also wish to thank those who helped me to handle various computer problems, especially Mr. Okan Donangil and Mr. Kaan Uyar.

Fourth, I thank my family, for their constant encouragement and support during preparation of this project

Finally, I would also like to thank my friend Ömer Faruk Tosun for his advice and support.

ABSTRACT

WS-Security describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. These mechanisms can be used to accommodate a wide variety of security models and encryption technologies. Specifically, the specification describes how to encode X.509 certificates as well as how to include opaque encrypted keys. It also includes extensibility mechanisms that can be used to further describe the characteristics of the credentials that are included with a message.

WS-Security is a building block that is used in conjunction with other Web service and application-specific protocols to accommodate a wide variety of security models and encryption technologies.

A tool has been developed that implements some security specification and secure message exchanges to exchange data securely with respect to performance issues.

TABLE OF CONTENTS

LIST OF SYMBOLS

LIST OF TABLES

INTRODUCTION

XML Web Services is poised to revolutionize Information Technology, much the same way that client-server and Web-based applications did in the past decade. Through the use of protocols such as XML, SOAP, WSDL and UDDI, applications can more easily communicate with each other over Internet protocols, enabling faster and cheaper enterprise application integration, supply chain integration, distributed development and Internet-based service distribution models.

XML Web Services has had unprecedented support from many of the major vendors including SAP, Siebel, PeopleSoft, Oracle, Sun, IBM and Microsoft with their .NET framework. In addition, Web Services is simple and easy to implement, drastically with each other using XML Reducing cost and development time but at the same time dramatically increasing the security risk.

A1 XML WEB SERVICES

A1.1 XML Web Service Basics

XML Web services are the fundamental building blocks in the move to distributed computing on the Internet. Open standards and the focus on communication and collaboration among people and applications have created an environment where XML Web services are becoming the platform for application integration.

1.1.1 XML Web Service Definition

Applications are constructed using multiple XML Web services from various sources that work together regardless of where they reside or how they were implemented. There are probably as many definitions of XML Web Service as there are companies building them, but almost all definitions have these things in common:

* XML Web Services expose useful functionality to Web users through a standard Web protocol. In most cases, the protocol used is Simple Object Access Protocol (SOAP).

* XML Web services provide a way to describe their interfaces in enough detail to allow a user to build a client application to talk to them. This description is usually provided in an XML document called a Web Services Description Language (WSDL) document.

* XML Web services are registered so that potential users can find them easily. This is done with Universal Discovery Description and Integration (UDDI).

I'll cover all three of these technologies in this chapter but first I want to explain why you should care about XML Web services. One of the primary advantages of the XML Web services architecture is that it allows:

1. Programs written in different languages on different platforms to communicate with each other in a standards-based way.
2. The other significant advantage that XML Web services have over previous efforts is that they work with standard Web protocols—XML, HTTP and TCP/IP.
3.Those of you who have been around the industry a while are now saying, "Wait a minute! Didn't I hear those same promises from CORBA and before that DCE? How is this any different?" The first difference is that SOAP is significantly less complex than earlier approaches, so the barrier to entry for a standards-compliant SOAP implementation is significantly lower.

You'll find SOAP implementations from most of the big software companies, as you would expect, but you will also find many implementations that are built and maintained by a single developer. A significant number of companies already have a Web infrastructure, and people with knowledge and experience in maintaining it, so again, the cost of entry for XML Web services is significantly less than previous technologies. We've defined an XML Web service as a software service exposed on the Web through SOAP, described with a WSDL file and registered in UDDI.

1.1.2 XML Web Service Benefits

The first XML Web services tended to be information sources that you could easily incorporate into applications—stock quotes, weather forecasts, sports scores etc. It's easy to imagine a whole class of applications that could be built to analyze and aggregate the information you care about and present it to you in a variety of ways; for example, you might have a Microsoft® Excel spreadsheet that summarizes your whole financial picture—stocks, 401K, bank accounts, loans, etc. If this information is available through XML Web services, Excel can update it continuously. Some of this information will be free and some might require a subscription to the service. Most of this information is available now on the Web, but XML Web services will make programmatic access to it easier and more reliable.

Exposing existing applications as XML Web services will allow users to build new, more powerful applications that use XML Web services as building blocks. For example, a user might develop a purchasing application to automatically obtain price information from a variety of vendors, allow the user to select a vendor, submit the order and then track the shipment until it is received. The vendor application, in addition to exposing its services on the Web, might in turn use XML Web services to check the customer's credit, charge the customer's account and set up the shipment with a shipping company.

In the future, some of the most interesting XML Web services will support applications that use the Web to do things that can't be done today. For example, one of the services that XML Web Services would make possible is a calendar service. If your dentist and mechanic exposed their calendars through this XML Web service, you could schedule appointments with them on line or they could schedule appointments for cleaning and routine maintenance directly in your calendar if you like. With a little imagination, you can envision hundreds of applications that can be built once you have the ability to program the Web.

1.1.3 Simple Object Access Protocol

Soap is the communications protocol for XML Web services. When SOAP is described as a communications protocol, most people think of DCOM or CORBA and start asking things like, "How does SOAP do object activation?" or "What naming service does SOAP use?" While a SOAP implementation will probably include these things, the SOAP standard doesn't specify them. SOAP is a specification that defines the XML format for messages—and that's about it for the required parts of the spec. If you have a well-formed XML fragment enclosed in a couple of SOAP elements, you have a SOAP message. Simple isn't it?

There are other parts of the SOAP specification that describe how to represent program data as XML and how to use SOAP to do Remote Procedure Calls. These optional parts of the specification are used to implement RPC-style applications where a SOAP message containing a callable function, and the parameters to pass to the function, is sent from the client, and the server returns a message with the results of the executed function.

Most current implementations of SOAP support RPC applications because programmers who are used to doing COM or CORBA applications understand the RPC style. SOAP also supports document style applications where the SOAP message is just a

wrapper around an XML document. Document-style SOAP applications are very flexible and many new XML Web services take advantage of this flexibility to build services that would be difficult to implement using RPC.

The last optional part of the SOAP specification defines what an HTTP message that contains a SOAP message looks like. This HTTP binding is important because HTTP is supported by almost all current OS's. The HTTP binding is optional, but almost all SOAP implementations support it because it's the only standardized protocol for SOAP. For this reason, there's a common misconception that SOAP requires HTTP. Some implementations support MSMQ, MQ Series, SMTP, or TCP/IP transports, but almost all current XML Web services use HTTP because it is ubiquitous. Since HTTP is a core protocol of the Web, most organizations have a network infrastructure that supports HTTP and people who understand how to manage it already. The security, monitoring, and load-balancing infrastructure for HTTP are readily available today.
A major source of confusion when getting started with SOAP is the difference between the SOAP specification and the many implementations of the SOAP specification. Most people who use SOAP don't write SOAP messages directly but use a SOAP toolkit to create and parse the SOAP messages. These toolkits generally translate function calls from some kind of language to a SOAP message. For example, the Microsoft SOAP Toolkit 2.0 translates COM function calls to SOAP and the Apache Toolkit translates JAVA function calls to SOAP. The types of function calls and the data types of the parameters supported vary with each SOAP implementation so a function that works with one toolkit may not work with another. This isn't a limitation of SOAP but rather of the particular implementation you are using.

By far the most compelling feature of SOAP is that it has been implemented on many different hardware and software platforms. This means that SOAP can be used to link disparate systems within and without your organization. Many attempts have been made in the past to come up with a common communications protocol that could be used for systems integration, but none of them have had the widespread adoption that SOAP has.

DCE and CORBA for example took years to implement, so only a few implementations were ever released. SOAP, however, can use existing XML Parsers and HTTP libraries to do most of the hard work, so a SOAP implementation can be completed in a matter of months. There are more than 70 SOAP implementations available [5].SOAP obviously doesn't do everything that DCE or CORBA do, but the lack of complexity in exchange for features is what makes SOAP so readily available.

1.1.4 Web Service Description Language

WSDL (often pronounced whiz-dull) stands for Web Services Description Language. For our purposes, we can say that a WSDL file is an XML document that describes a set of SOAP messages and how the messages are exchanged. In other words, WSDL is to SOAP what IDL is to CORBA or COM. Since WSDL is XML, it is readable and editable but in most cases, it is generated and consumed by software.

To see the value of WSDL, imagine you want to start calling a SOAP method provided by one of your business partners. You could ask him for some sample SOAP

messages and write your application to produce and consume messages that look like the samples, but this can be error-prone. For example, you might see a customer ID of 2837 and assume it's an integer when in fact it's a string. WSDL specifies what a request message must contain and what the response message will look like in unambiguous notation.

The notation that a WSDL file uses to describe message formats is based on the XML Schema standard which means it is both programming-language neutral and standards-based which makes it suitable for describing XML Web services interfaces that are accessible from a wide variety of platforms and programming languages. In addition to describing message contents, WSDL defines where the service is available and what communications protocol is used to talk to the service. This means that the WSDL file defines everything required to write a program to work with an XML Web service. There are several tools available to read a WSDL file and generate the code required to communicate with an XML Web service. Some of the most capable of these tools are in Microsoft Visual Studio® .NET.

Many current SOAP toolkits include tools to generate WSDL files from existing program interfaces, but there are few tools for writing WSDL directly, and tool support for WSDL isn't as complete as it should be. It shouldn't be long before tools to author WSDL files, and then generate proxies and stubs much like COM IDL tools, will be part of most SOAP implementations. At that point, WSDL will become the preferred way to author SOAP interfaces for XML Web services.

1.1.5 Universal Description Discovery and Integration

Universal Discovery Description and Integration is the yellow pages of Web services. As with traditional yellow pages, you can search for a company that offers the services you need, read about the service offered and contact someone for more information. You can, of course, offer a Web service without registering it in UDDI, just as you can open a business in your basement and rely on word-of-mouth advertising but if you want to reach a significant market, you need UDDI so your customers can find you.

A UDDI directory entry is an XML file that describes a business and the services it offers. There are three parts to an entry in the UDDI directory. The "white pages" describe the company offering the service: name, address, contacts, etc. The "yellow pages" include industrial categories based on standard taxonomies such as the North American Industry Classification System and the Standard Industrial Classification. The "green pages" describe the interface to the service in enough detail for someone to write an application to use the Web service. The way services are defined is through a UDDI document called a Type Model or tModel. In many cases, the tModel contains a WSDL file that describes a SOAP interface to an XML Web service. tModel is flexible enough to describe almost any kind of service.

The UDDI directory also includes several ways to search for the services you need to build your applications. For example, you can search for providers of a service in a specified geographic location or for business of a specified type. The UDDI directory will then supply information, contacts, links, and technical data to allow you to evaluate which services meet your requirements.

UDDI allows you to find businesses you might want to obtain Web services from. What if you already know whom you want to do business with but you don't know what services are offered? The WS-Inspection specification allows you to browse through a collection of XML Web services offered on a specific server to find which ones might meet your needs.

1.1.6 What About Security?

One of the first questions newcomers to SOAP ask is how does SOAP deal with security. Early in its development, SOAP was seen as an HTTP-based protocol so the assumption was made that HTTP security would be adequate for SOAP. After all, there are thousands of Web applications running today using HTTP security so surely this is adequate for SOAP. For this reason, the current SOAP standard assumes security is a transport issue and is silent on security issues. When SOAP expanded to become a more general-purpose protocol running on top of a number of transports, security became a bigger issue.

For example, HTTP provides several ways to authenticate which user is making a SOAP call, but how does that identity get propagated when the message is routed from HTTP to an SMTP transport? SOAP was designed as a building-block protocol; so fortunately, there are already specifications in the works to build on SOAP to provide additional security features for Web services. The WS-Security specification defines a complete encryption system.

1.1.7 What's Left?

So far we've talked about how to talk to XML Web services (SOAP), how XML Web services are described (WSDL) and how to find XML Web services (UDDI). These constitute a set of baseline specifications that provide the foundation for application integration and aggregation. From these baseline specifications, companies are building real solutions and getting real value from them.

While much work has been done to make XML Web services a reality, more is needed. Today, people are having success with XML Web services, but there are still things that are left as an exercise for the developer®e.g. security, operational management, transactions, reliable messaging[1]. The Global XML Web Services Architecture will help take XML Web services to the next level by providing a coherent, general purpose model for adding new advanced capabilities to XML Web services which is modular and extensible.

WS-Security[10] is one of the specifications in the Global Web Services Architecture. Operational management needs such as routing messages among many servers and configuring those servers dynamically for processing are also part of the Global Web Services Architecture, and are met by the WS-Routing specification[1] and the WS-Referral[1] specification. As the Global Web Services Architecture grows, specifications for these and other needs will be introduced.

A1.2 XML Web Service Security Overview

Web Services Security Overview There are many things that can be done to create secure XML Web Services today.

Addressing XML Web Services security, we need to consider:

* What are we trying to accomplish?—Restricting access to a XML Web Services to authorized users, protecting messages from being viewed by unauthorized parties, etc.
* How are we going to achieve that desired effect? Network, transport layer, OS, Service, or application.

What level of interoperability do we want and need in our solution? Local or global.

1.2.1 Are XML Web Service Secure?

Given the many aspects of security, authentication and authorization, data privacy and integrity, etc. and the fact that the SOAP specification does not even mention security, it is easy to understand where you might get the impression that the answer is no. How then to secure today's XML Web Services? By answering the questions above and then applying the same techniques we would use to secure any other Web application, namely:

* Securing the connection
* Authenticating and authorizing the interaction

As you will see below, these techniques offer rich alternatives that can be combined to get additional benefits. A firewall, for example, can work with a XML Web Services to limit access to certain functionality (methods) based on who the client is and the policies established for them. Let's begin by reviewing each of the alternatives for securing the infrastructure that are available today to understand what they can do.

1.2.2 Securing the Infrastructure

At the heart of a secure XML Web Services is a secure infrastructure. Microsoft offers a wide range of technologies that, when integrated into an overall security plan, will allow an enterprise to effectively secure its IT infrastructure. The planning process for its proper implementation involves:
*Gaining a detailed understanding of the potential environmental risks (for example, viruses, hackers, and natural disasters).
*Making a proactive analysis of the consequences of and countermeasures to security breaches in relation to risks.
*Creating a carefully planned implementation strategy for integrating security measures into all aspects of an enterprise network, based on this understanding and analysis.

1.2.3 Securing the Connection

One of the easiest methods for securing XML Web Services is to ensure that the connection between the XML Web Services client and the server is secure. This can be done with a variety of techniques depending on the scope of the network and activity profile of the interactions. Three of the most popular and widely available techniques are: firewall-based rules, Secure Sockets Layer (SSL) and Virtual Private Networks (VPN).

If you know exactly which computers need to access your XML Web Services, you can use firewall rules to restrict access to computers of known IP addresses. This technique is particularly useful when you want to restrict access to computers in a private network—a corporate LAN/WAN for example, and you are not worried about keeping the content of messages a secret (encrypted). Firewalls such as the Microsoft Internet Security and Acceleration (ISA) Server can offer advanced policy-based rules that offer different restrictions to different clients based on origin or identity. This is useful when for example different clients may have access to different functionality (methods) of the same XML Web Services.

Secure Sockets Layer can be used to establish secure connections on un trusted networks (such as the Internet). SSL encrypts and decrypts messages sent between a client and server. By encrypting the data, you protect messages from being read while they are transferred. SSL encrypts a message from the client and then sends it to the server. When the message is received by the server, SSL decrypts it and verifies that it came from the correct sender (a process known as authentication). The server, or the client and server, may have certificates that are used as part of the authentication process providing authentication capabilities on top of the connection encryption. While it is a very effective method for creating secure communications, SSL has a performance cost that should be taken into account. Microsoft XML Web Services support integrated SSL in both the client and the server.

A Virtual Private Network is an extension of a private network that connects across shared or public networks like the Internet. A VPN enables you to send data between two computers in a secure connection. While similar to SSL, a VPN is a long-term point-to-point connection. This makes it efficient and conversant with XML Web Services, but requires that a long-term connection be established and left running to gain that efficiency.

1.2.4 Authentication and Authorizing

Authentication: Authentication is the process of verifying identity that someone (or something) is who they claim to be. The "someone" or "something" is known as a principal. Authentication requires evidence, known as credentials. For example, a client application could present a password as its credentials. If the client application presents the correct credentials, it is assumed to be who it claims to be.

Authorization: Once a principal's identity is authenticated, authorization decisions can be made. Access is determined by checking information about the principal against some access control information, such as an Access Control List (ACL). It's possible for clients to have different degrees of access. For example, some clients may have full access to your XML Web Services; others may only be allowed to access certain operations. Some clients

may be allowed full access to all data, others may only be allowed to access a subset of the data, others may have read-only access.

A straightforward approach to implementing authentication in XML Web Services is to leverage the authentication features of the protocol used to exchange messages. For most XML Web Services, this means leveraging the authentication features available for HTTP. Microsoft Internet Information Server (IIS) and ISA Server work together with Windows 2000 Server to offer integrated support of several authentication mechanisms for HTTP:

Basic—Use for non-secure or semi-secure identification of clients, as username and password are sent as base64-encoded text, which is easily decoded. IIS will authorize access to the XML Web Services if the credentials match a valid user account.

Basic over SSL—Same as Basic authentication except that the communication channel is encrypted, thus protecting the username and password. A good option for Internet scenarios; however, using SSL has a significant impact on performance.

Digest—Uses hashing to transmit client credentials in a secure way. However, may not be widely supported by developer tools for building XML Web Services clients. IIS will authorize access to the XML Web Services if the credentials match a valid user account.

Integrated Windows authentication Useful primarily in Intranet scenarios. Uses NTLM or Kerberos. Client must belong to the same domain as the server, or belong to a trusted domain of the server's domain. IIS will authorize access to the XML Web Services if the credentials match a valid user account. Client certificates over SSL requires each client to obtain a certificate. Certificates are mapped to user accounts, which are used by IIS for authorizing access to the XML Web Services. A viable option for Internet scenarios, although use of digital certificates is not widespread at this time. May not be widely supported by developer tools for building XML Web Services clients. Only available over SSL connections, thus performance may be a concern.

From a XML Web Services implementer's perspective, one nice benefit of using any of these authentication mechanisms is that no code changes are required in the XML Web Services—IIS/ISA Server performs all of the authentication and ACL authorization checks before your XML Web Services are called. When implementing the client, however, some additional work will be involved. The client application will need to respond to server requests for authentication credentials.

Other approaches to implementing authentication in XML Web Services include using third-party services such as those found in Microsoft® .NET Passport, using the session features of Microsoft ASP.NET, or creating a custom authentication method

1.2.5 What's Next: Interoperability?

As you can see, standard techniques for Web application security can be applied alone or in combination to create secure XML Web Services today. These techniques build on a rich base of experience and are quite effective. They do not however, offer an integrated solution within the XML Web Services architecture. As the complexity of a XML Web Services scenario increases—crossing trust boundaries, spanning multiple systems or

enterprises, XML Web Services implementers are left building custom solutions that while effective, do not offer ubiquitous interoperability.

To address these needs and enhance interoperability across XML Web Services, Microsoft and partners are working on a set of security specifications that build on the extensibility mechanism of the SOAP specification to offer enhanced security capabilities integrated into the fabric of the XML Web Services. At the heart of these is the XML Web Services Security Language (WS-Security) that provides enhancements to SOAP messaging consisting of three capabilities: credential transfer, message integrity, and message confidentiality. These capabilities by themselves do not provide a complete security solution; WS-Security is a building block that can be used in conjunction with infrastructure and other XML Web Services protocols to address a wide variety of application security requirements. Microsoft Global XML Web Services architecture is the home of WS-Security and related specifications that provide a framework for the evolution of the XML Web Services infrastructure.

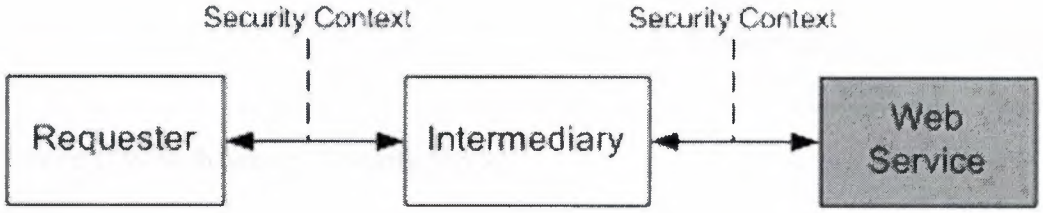
1.2.6 Web Services Security Model Principles

Web services can be accessed by sending SOAP messages to service endpoints identified by URIs, requesting specific actions, and receiving SOAP message responses (including fault indications). Within this context, the broad goal of securing Web services breaks into the subsidiary goals of providing facilities for securing the integrity and confidentiality of the messages and for ensuring that the service acts only on requests in messages that express the claims required by policies.

Today the Secure Socket Layer (SSL) along with the Transport Layer Security (TLS) is used to provide transport level security for web services applications. SSL/TLS offers several security features including authentication, data integrity and data confidentiality. SSL/TLS enables point-to-point secure sessions. IPSec is another network layer standard for transport security that may become important for Web services. Like SSL/TLS, IPSec also provides secure sessions with host authentication, data integrity and data confidentiality.

What is needed in a comprehensive Web service security architecture is a mechanism that provides end-to-end security. Successful Web service security solutions will be able to leverage both transport and application layer security mechanisms to provide a comprehensive suite of security capabilities.

Point-to-point configuration



End-to-End configuration

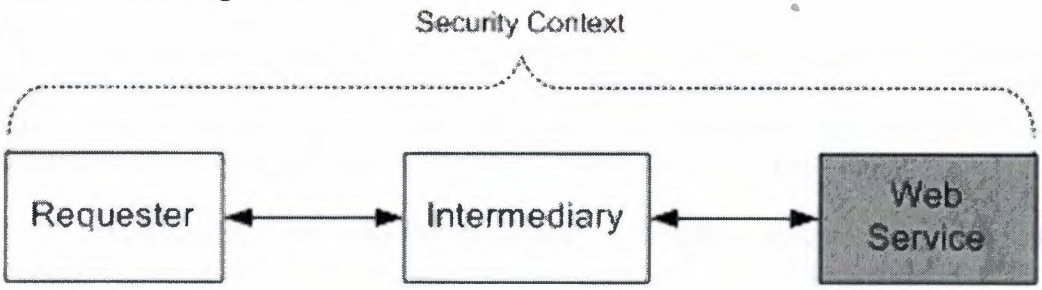


Figure 1.1: Different Security Context

The Web service security model described here in enables us to achieve these goals by a process in which:

A Web service can require that an incoming message prove a set of claims (e.g., name, key, permission, capability, etc.). If a message arrives without having the required claims, the service may ignore or reject the message. We refer to the set of required claims and related information as policy. A requester can send messages with proof of the required claims by associating security tokens with the messages. Thus, messages both demand a specific action and prove that their sender has the claim to demand the action. When a requester does not have the required claims, the requester or someone on its behalf can try to obtain the necessary claims by contacting other Web services.

These other Web services, which we refer to as security token services, May in turn require their own set of claims.Security token services broker trust between different trust domains by issuing security tokens.

This model is illustrated in the figure below, showing that any requester may also be a service, and that the Security Token Service may also fully be a Web service, including expressing policy and requiring security tokens.

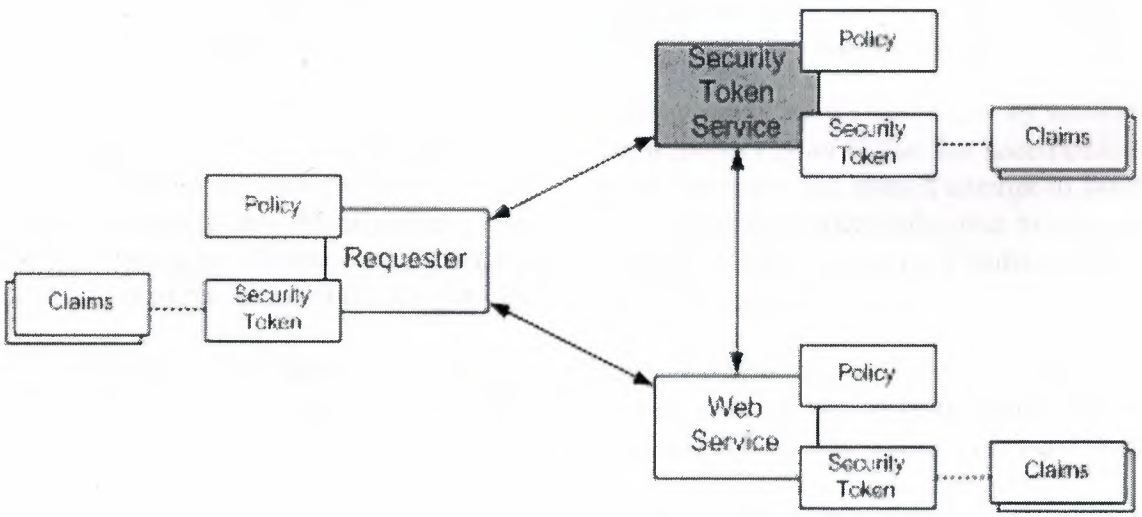


Figure 1.2: Web Service Security Model.

1.2.7 Web Services Security Model Terminology

Because terminology varies between technologies, this section defines several terms that may be applied consistently across the different security formats and mechanisms. Consequently, the terminology used here may be different from other specifications and is defined so that the reader can map the terms to their preferred vocabulary.

Web service—the term "Web service" is broadly applicable to a wide variety of network based application topologies. In this document, we use the term "Web service" to describe application components whose functionality and interfaces are exposed to potential users through the application of existing and emerging Web technology standards including XML, SOAP, WSDL, and HTTP. In contrast to Web sites, browser-based interactions or platform-dependent technologies, Web services are services offered computer-to-computer, via defined formats and protocols, in a platform-independent and language-neutral manner.
Security Token—we define a security token as a representation of security-related information (e.g. X.509 certificate, Kerberos tickets and authenticators, mobile device security tokens from SIM cards, username, etc.).

The following diagram shows some of the different kinds of security tokens.

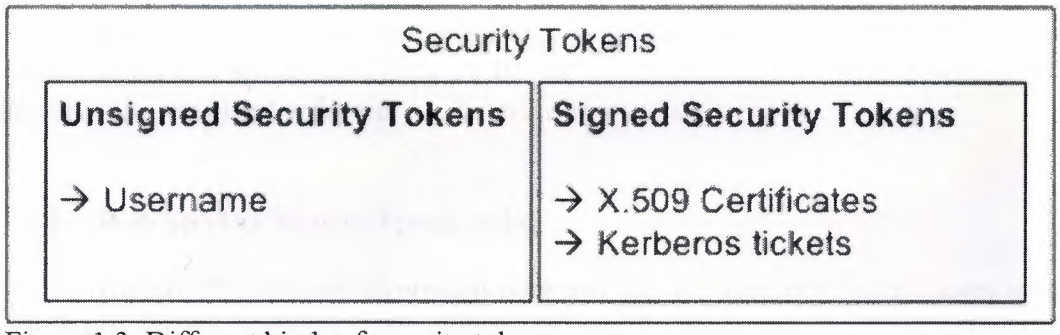


Figure 1.3: Different kinds of security tokens.

Signed Security Token—we define a signed security token as a security token that contains a set of related claims (assertions) cryptographically endorsed by an issuer. Examples of signed security tokens include X.509 certificates and Kerberos tickets.

Claims—a claim is a statement about a subject either by the subject or by another party that associates the subject with the claim. An important point is that this specification does not attempt to limit the types of claims that can be made, nor does it attempt to limit how these claims may be expressed. Claims can be about keys potentially used to sign or encrypt messages. Claims can be statements the security token conveys. Claims may be used, for example, to assert the senders identity or an authorized role.

Subject—the subject of the security token is a principal (e.g. a person, an application or a business entity) about which the claims expressed in the security token apply. Specifically, the subject, as the owner of the security token possesses information necessary to prove ownership of the security token.

Proof-of-Possession—we define proof-of-possession to be information used in the process of proving ownership of a security token or set of claims. For example, proof-of-possession might be the private key associated with a security token that contains a public key.

Web Service Endpoint Policy—Web services have complete flexibility in specifying the claims they require in order to process messages. Collectively we refer to these required claims and related information as the "Web Service Endpoint Policy". Endpoint policies may be expressed in XML and can be used to indicate requirements related to authentication (e.g. proof of user or group identity), authorization (e.g. proof of certain execution capabilities), or other custom requirements.

Claim Requirements—Claim requirements can be tied to whole messages or elements of messages, to all actions of a given type or to actions only under certain circumstances. For example, a service may require a requestor to prove authority for purchase amounts greater
than a stated limit.

Intermediaries—As SOAP messages are sent from an initial requester to a service, they may be operated on by intermediaries that perform actions such as routing the message or even modifying the message. For example, an intermediary may add headers, encrypt or decrypt pieces of the message, or add additional security tokens. In such situations, care should be taken so that alterations to the message do not invalidate message integrity, violate the trust model, or destroy accountability.

Actor—an actor is an intermediary or endpoint (as defined in the SOAP specification) which is identified by a URI and which processes a SOAP message. Note that neither users nor client software (e.g. browsers) are actors.

1.2.8 Web Services Security Specification

The security strategy expressed here and the WS-Security specification introduced below provide the strategic goals and cornerstone for this proposed Web services security model. Moving forward, we are continuing the process of working with customers, partners and standards organizations to develop an initial set of Web service security specifications.

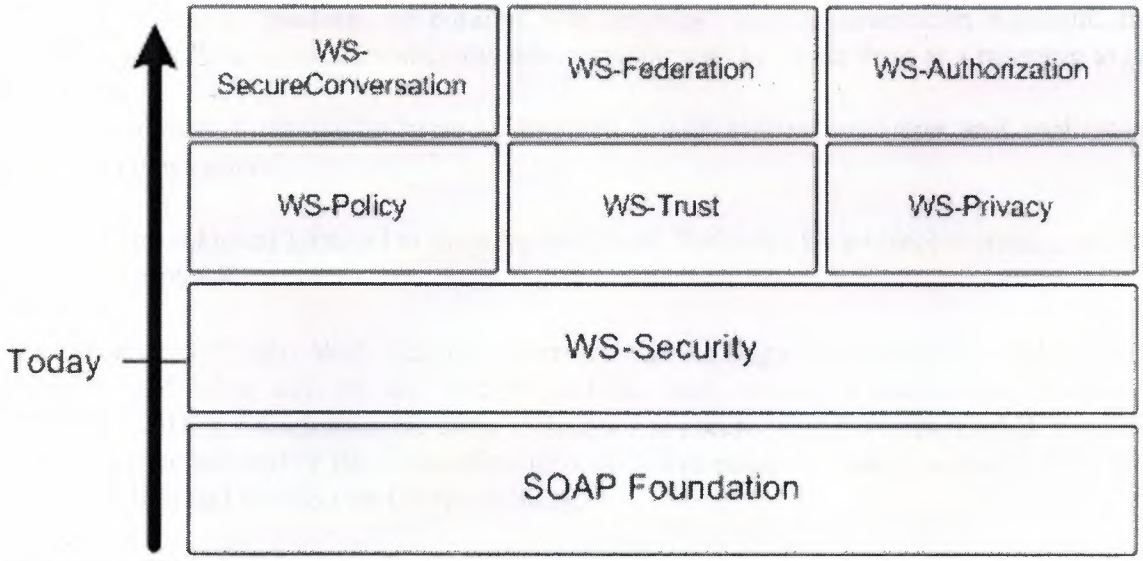


Figure 1.4: Web Services Strategy Today.

This set will include a message security model (WS-Security) that provides the basis for the other security specifications. Layered on this, we have a policy layer which includes a

Web service endpoint policy (WS-Policy), a trust model (WS-Trust), and a privacy model (WS-Privacy). Together these initial specifications provide the foundation upon which we can work to establish secure interoperable Web services across trust domains.

Building on these initial specifications we will continue to work with customers, partners and standards organizations to provide follow-on specifications for federated security which include secure conversations (WS-Secure Conversation), federated trust (WS-Federation), and authorization (WS-Authorization).

The combination of these security specifications enable many scenarios (some of which are described later in this document) that are difficult to implement with today's more basic security mechanisms.

In parallel, we will propose and move forward with related activities that enhance the Web services security framework with specifications related to auditing, management, and privacy.
Additionally, IBM and Microsoft are committed to working with organizations like WS-I on interoperability profiles.

WS-Security describes how to attach signature and encryption headers to SOAP messages. In addition, it describes how to attach security tokens, including binary security tokens such as X.509 certificates and Kerberos tickets, to messages.

The combination of security specifications, related activities, and interoperability profiles will enable customers to easily build interoperable secure Web services, for further Information about WS-Specifications.

A1.3 Project Outline

Web services is a newly evolving technology that has hyped rigidly, Microsoft even describe .net as a "platform for building web services", this documentation highlights the following, Web services are a step towards semantic web in which there is a meaning to all web data.
SOAP provides a messaging protocol between a web service consumer and application "(C2A) environment".

The achieved protocol is there by explained, Followed by an implementation of the achieved protocol.

In Chapter 1 "XML Web Services Overview" , We begin by describing XML Web service, and what can we do with it ,and the web service s technologies (SOAP ,WSDL,UDDI), concentrates on some techniques to secure Web Services through securing the infrastructure and/or the connection through some stages including authentication and authorization and its effect on interoperability.

Chapter 2 "XML Web Services Architecture " , we present the service oriented architecture (SOA) ,the SOA systems , reaching the web services architecture ,we discuss it basic elements and implementations, and describes the extended basic web services architecture

which explains the proposed "Global XML Web Services Architecture " (GXA) ,along with its principles and specifications .

Chapter 3 "WS-Security Approaches" describes the main Web service security approaches that where a basic guide in our developed specification. How you can define your Web Service by using Microsoft IIS check list and configuring the IIS.

Chapter 4 "Our Proposed Architecture and Roadmap" describe our motivation and the architecture with the analysis and design phases with implementation feed.

A2 XML WEB SERVICES ARCHITECTURE

Web services appear to be the latest "new thing". But what are Web services? If you ask five people to define Web services, you'll probably get at least six answers. Even so, most people will agree that a Web service represents an information resource or business process that can be accessed over the Web by another application, and that it communicates using standard Internet protocols.

What distinguishes Web services from other types of Web-based applications is that Web services are designed to support application-to-application communication. Other Web applications support human-to-human communication (email and instant messaging) or human-to-application communication [12]. Web services are designed to allow applications to communicate without human assistance or intervention.

Even though Web services are new, from an architectural perspective, they are based on established middleware design principles for application-to-application communication. These design principles are known as the service-oriented architecture (SOA). Previous SOA systems include RPC, RMI, DCOM, and CORBA. The Web services Architecture (WSA) represents the convergence of SOA and "the Web". The Web makes WSA completely platform- and language-independent. Web services can be developed using any language, and they can be deployed on any platform: from the tiniest device to the largest super computer.

Another way to think about Web services is as Internet middleware. Today most Web services are implemented using a core set of Internet middleware technologies including:

- XML, which provides a platform-neutral mechanism to represent data.
- SOAP, which defines the data communication protocol for Web services.
- WSDL, which describes a Web Service.
- UDDI, which provides a means to advertise and discover Web services.

A Web services platform is a set of products that implement these Internet middleware technologies.

A2.1 Service Oriented Architecture Systems

2.1.1 Service Oriented Architecture

The concept of a service is key to understanding Web services. A Web services environment conforms to a Service Oriented Architecture (SOA). Figure 2.1 depicts the conceptual roles and operations of an SOA. The three basic roles are the service Provider, the service consumer, and a service broker. A service provider makes the service available and publishes the contract that describes its interface. It then registers the service with a service broker. A service consumer queries the service broker and finds a compatible service. The service broker gives the service consumer directions on where to find the service and its service contract. The service consumer uses the contract to bind the client to the service.

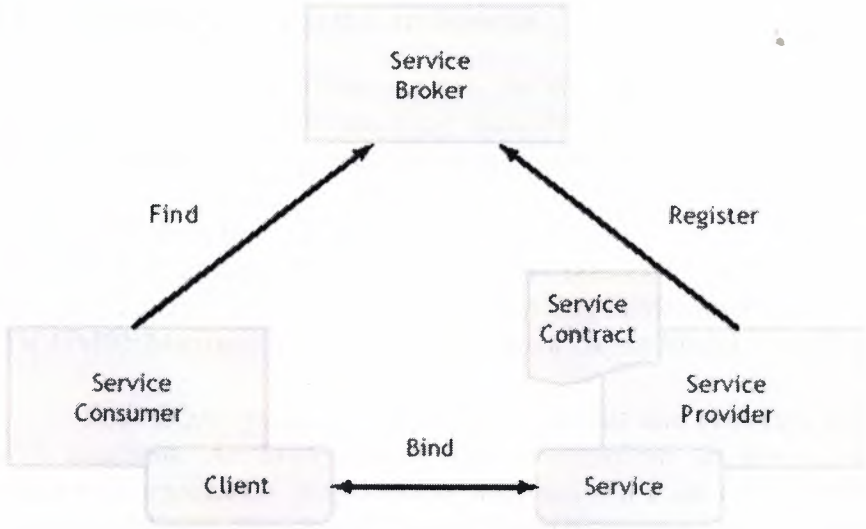


Figure 2.1: The three conceptual roles and operations of a service oriented architecture.

2.1.2 Service Oriented Architecture Functional Components

In order for the three conceptual roles to accomplish the three conceptual operations, an SOA system must supply three core functional architecture components:

Transport:

The transport component represents the formats and protocols used to communicate with a service. The data format specifies the data types and byte stream formats used to encode data within messages. The wire protocol specifies the mechanism used to package the encoded data into messages. The transfer protocol specifies the application semantics that control a message transfer. The transport protocol performs the actual message transfer.

Description:

The description component represents the languages used to describe a service. The description provides the information needed to bind to a service. At a minimum, a description language provides the means to specify the service contract, including the operations that it performs and the parameters or message formats that it exchanges. A description language is a machine-readable format that can be processed by a compiler to produce communication code, such as client proxies, server skeletons, stubs, and ties. These generated code fragments automate the connection between the application code and the communications process, insulating the application from the complexities of the underlying middleware.

Discovery:

The discovery component represents the mechanisms used to register or advertise a service and to find a service and its description. Discovery mechanisms may be used at compile time or at runtime. They may support static or dynamic binding.

2.1.3 Service Oriented Architecture Systems

Although Web services are new, the concepts behind service-oriented systems have been around for quite a while. Most standard distributed computing middleware systems implement a SOA. Examples of earlier SOA systems are:

- Java RMI7: Java Remote Method Invocation
- CORBA8: The Object Management Group Common Object Request Broker Architecture
- DCE9: The Open Group Distributed Computing Environment
- DCOM10: Microsoft Distributed Component Object Model

Each SOA system defines a set of formats and protocols that implement the core SOA functions. An SOA system often also defines an invocation mechanism, which includes an application programming interface and a set of language bindings. Table 2.1 shows the different formats and protocols used by various SOA systems.

A2.2 Internet Middleware

You can think of Web services as a new form of middleware – let's call it Internet middleware. But unlike previous SOA systems, Internet middleware does not require an entirely new set of protocols. The most basic Web services protocol is the industry standard Extensible Markup Language (XML), which is used as the message data format, and is also used as the foundation for all other Web services protocols. Today most Internet middleware systems are implemented using a core set of technologies, Including SOAP, WSDL, and UDDI. These technologies define the transport, description, and discovery mechanisms, respectively. We'll delve deeper into these technologies in the next section of this paper. Each of these technologies are defined and implemented in XML. One important ramification of the use of XML is that any application, written in any language, running on any platform, can interpret Web services messages, descriptions, and discovery mechanisms. No specific middleware technology needs to be available to converse using Web services. Any application can interpret SOAP message using standard XML processing tools.

	Java RMI	CORBA	DCE	Web Services
Invocation Mechanism	Java RMI	CORBA RMI	RPC	JAX-RPC, .NET, ...
Data Format	Serialized Java	CDR	NDR	XML
Wire Format	Stream	GIOP	PDU	SOAP
Transfer Protocol	JRMP	IIOP	RPC CO	HTTP, SMTP, ...
Interface Description	Java Interface	CORBA IDL	DCE IDL	WSDL
Discovery Mechanism	Java Registry	COS naming	CDS	UDDI

Table 2.1: A comparison of SOAP formats and protocols.

JRMP = Java Remote Method Protocol, PDU =
Protocol Data Units, ORB = Object Request Broker,
IIOP=Internet Inter-ORB Protocol, CDR = Common
Data Representation, IDL = Interface Definition Language.
GIOP = General Inter-ORB Protocol, COS = CORBA
Object Services, RPC = Remote Procedure Call,
NDR = Network Data Representation, RPC CO = RPC
Connect-Oriented protocol, CDS = Cell Directory Service.

A2.3 XML Web Service Architecture

In most Internet middleware configurations, the three core functional components (transport, description, and discovery) in the Web Services Architecture (WSA) are implemented using SOAP, WSDL, and UDDI, respectively. Figure 2.2 shows the conceptual SOA architecture using these technologies[12]. A UDDI registry plays the role of service Broker. The register and find operations are implemented using the UDDI Inquiry and UDDI Publish APIs. A WSDL document describes the service contract, and is used to bind the client to the service. All transport functions are performed using SOAP. Let's take a closer look at the WSA transport, description, and discovery functions.

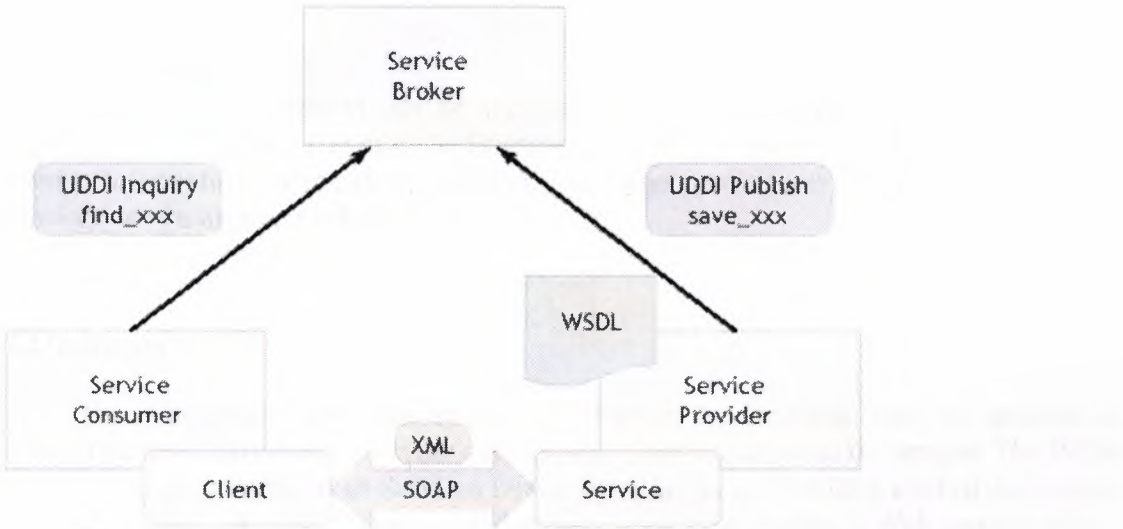


Figure 2.2: The SOA conceptual architecture with SOAP, WSDL, and UDDI.

2.3.1 Transport

The transport functional component defines the formats and protocols used to communicate between clients and services. The WSA formats and protocols are defined by SOAP, a lightweight, extensible XML protocol. SOAP provides a simple messaging framework that allows one application to send an XML message to another application.

Data format:

The SOAP data format is XML. The mechanism by which the data are encoded is totally extensible, and in fact can be specified within each SOAP message. The data format

can represent an RPC invocation, in which case the message body is composed as a structure containing the RPC input parameters or return value. The name of the structure indicates the method to be invoked. When using the RPC representation, the data are normally encoded using an XML-based encoding style. Alternately, the data format can be in the form of an XML document, in which case the data are normally encoded using a specific XML Schema.

Wire format:

The SOAP wire format is an XML document called a SOAP envelope. The envelope contains an optional SOAP header and a mandatory SOAP body. The SOAP body contains the message payload, encoded in XML.

Transfer protocol:

SOAP defines an abstract binding framework that allows SOAP messages to be transferred using a variety of underlying transfer protocols. The SOAP specification defines a protocol binding for HTTP. Bindings have also been defined for HTTPS, SMTP, POP3, IMAP, JMS, and other protocols.

Extensions:

Additional information can be included with a SOAP message within a SOAP header. The SOAP header can provide directive or control information to the service, such as security information, transaction context, message correlation information, session indicators, or management information.

2.3.2 Description

The description functional component defines the language used to describe a service. The service consumer uses the description to bind the client to the service. The WSA description language is the Web Services Description Language (WSDL), a set of definitions expressed in XML. A WSDL document describes what functionality a Web service offers, how it communicates, and where to find it. The various parts of a Web service description can be separated into multiple documents to provide more flexibility and to increase reusability. Figure 3.3 maps the three parts of a WSDL description to the specific WSDL definition elements [12]. A WSDL implementation document can be compiled to generate a client proxy that can call the Web service using SOAP.

Abstract interface:

The what part of a WSDL document describes the abstract interface of the Web service. It essentially describes a service type. Any number of service providers can implement the same service type. The WSDL what part defines a logical interface consisting of the set of operations that the service performs? For each operation it defines the input and/or output messages that are exchanged, the format of each message, and the data type of each element in the message.

Concrete binding:

The how part of a WSDL document describes a binding of the abstract interface to a concrete set of protocols. The binding indicates whether the message is structured as an RPC or as a document; it specifies which encoding style or XML Schema should be used to encode the data; it specifies which XML protocol should be used to construct the envelope; it indicates what header blocks should be included in the message; and it indicates which transfer protocol should be used. The how part includes or imports the associated WSDL what part.

Implementation:

The where part of a WSDL document describes a service implementation. A service implementation is a collection of one or more related ports. Each port implements a specific concrete binding of an abstract interface. The port specifies the access point of the service endpoint. A business might offer multiple access points to a particular service, each implementing a different binding. The where part includes or imports the associated WSDL how part. A service producer should always publish the where WSDL part with the Web service.

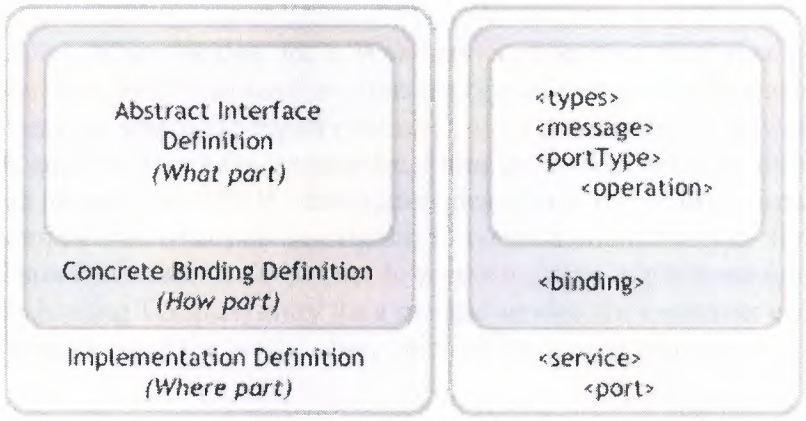


Figure 2.3 The three different parts of WSDL description

2.3.3 Discovery

The discovery functional component provides a mechanism to register and find services. Some discovery functions are used at development time, while others are used at runtime. The WSA discovery mechanism is implemented using a Universal Description, Discovery and Integration (UDDI) registry service. For the most part, UDDI is used at development time, although it can also be used at runtime. UDDI is itself a Web service, and users communicate with UDDI using SOAP messages. UDDI manages information about service types and service providers, and it provides mechanisms to categorize, find, and bind to services.

Service types:

A service type, defined by a construct called a tModel, defines an abstract service. Multiple businesses can offer the same type of service, all supporting the same service

interface. The tModel provides a pointer to the WSDL document that describes the abstract interface (the what part).

Service providers:

A service provider registers its business and all the services it offers. For each service offered, the service provider supplies the binding information needed to allow a consumer to bind to the service. The binding Template construct provides a pointer to the WSDL document that describes the service binding (the how part). It also specifies the access point of the service implementation. The WSDL where part is usually co-resident with the access point.

Categorization:

When a service provider registers a service or service type, he can categorize the entity (business, service, or tModel) using a variety of taxonomies. The UDDI specification defines a core set of taxonomies, such as geographic location, product codes, and industry codes. Additional taxonomies can be added to the registry to support more focused or customized categorization and search.

Find:

When looking for a Web service, a service consumer queries the UDDI registry, searching for a business that offers the type of service that he wants.
Users can search the registry for services by service type or service provider, and queries can be qualified using the taxonomies. From the tModel entry for the service type, the consumer can obtain the WSDL description describing the abstract interface. The consumer can compile this what part description to create a client interface for the abstract service. This abstract interface could be used to access multiple implementations of the service type. From the binding Template entry for a specific service, the consumer can obtain the access point of the service and the WSDL description of the service binding.

Static binding:

Developers can bind clients to services either at compile time or at runtime. Using the WSDL how part, a developer can compile a concrete SOAP client interface or stub that implements the binding required to communicate with a specific Web service implementation. This pre-compiled stub can be included in a client application. The access point can be specified at runtime.

Dynamic binding:

Because a WSDL document is machine-readable, WSA also supports dynamic binding. Using just the WSDL what part at compile time, a developer can generate an abstract client interface that can work with any implementation of a specific service type. At runtime the client application can dynamically compile the WSDL where part (containing the how part) and generate a dynamic proxy that implements the binding.

Dynamic discovery:

Since UDDI is itself a Web service, an application can query the registry at runtime, dynamically discover a service, locate its access point, retrieve its WSDL, and bind to it, all at runtime. The client interprets the WSDL to dynamically construct the SOAP calls.

A2.4 Extending the Basic Web Services Architecture

As mentioned earlier, SOAP provides a built-in extension mechanism via SOAP headers. A SOAP header can be used to pass directive or control information between client and service to implement extended middleware functions, such as routing, intermediate caching, reliable delivery, asynchronous communications, stateful conversations, transactions, security, and management.

To illustrate this extension mechanism, let's take a look at how you can use SOAP headers to support security. Security is a rather expansive topic. There are four different functions that fall under this topic:

Authentication and Proof of Identity:

Authentication is the process used to verify an entity's identity. There are a number of mechanisms that can be used to authenticate an entity, such as HTTP Basic and HTTP Digest authentication, a PKI certificate authority, and a Kerberos login. Once an authentication authority has verified your identity, you may receive an authentication token that you can use in future interactions as proof of identity. Such an authentication token could take the form of an X.509 certificate, a Kerberos ticket, or a SAML19 authentication assertion.

Authorization and Access Control:

Authorization is the process used to determine if an authenticated entity has permission to perform a particular action or function. You may want to define access control policies for all services at a given location, for individual services, or for specific operations in a service.

Confidentiality and Integrity:

Encryption protects the confidentiality and integrity of message communication. Confidentiality prevents unauthorized access to the contents of the message. Integrity prevents unauthorized modification of the message.

Proof of Origin:

A digital signature provides proof that the signed data was sent from a specific authenticated identity. All or part of the message may be signed.

The WSA architecture supports security by allowing you to specify and exchange security information in a SOAP header. Figure 2.4 conceptually shows how an authentication token and a digital signature could be specified in a SOAP header. This

diagram is based on the WS-Security specification, which defines a set of SOAP header constructs that can be used to pass security information in SOAP messages. The WS-Security specification supports the following features:

• A way to pass authentication tokens.
• A mechanism to sign message content using XML Signature.
• A way to pass signature and key information to allow the receiver to interpret the signed data.

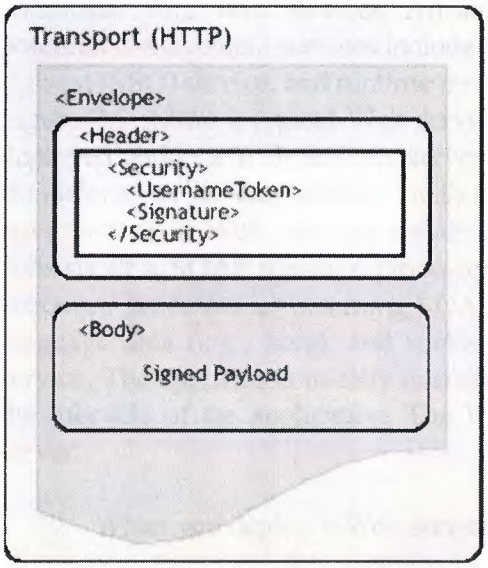


Figure 2.4: Using a SOAP Header to pass security information.

A2.5 W eb Service Environment

Since WSA is based on standard XML, you can implement Web services using the pervasive XML processing technologies that come with most platforms. You can build applications that query UDDI, parse WSDL documents, and construct SOAP requests—all using standard XML parsers. Or you can save yourself a lot of time and effort by using a set of Internet middleware products that implement these technologies.

Internet middleware products, sometimes referred to as a Web services platform, provide a ready-made foundation for building and deploying Web services. The advantage of using a Web services platform is that developers don't need to be concerned with constructing or interpreting SOAP messages. A developer simply writes the application code that implements the service, and the Internet middleware does the rest. A Web services platform generally consists of development tools, a runtime server, and a set of management services:

• Development tools are used to create Web services, to generate WSDL descriptions that describe those services, and to generate client proxies that can be used to send messages to the service. Development tools may also provide wizards to register or discover services in a UDDI registry.

• A runtime server processes SOAP messages and provides a runtime container for Web services. The runtime server often runs within a Web or application server. The runtime server listens for SOAP requests. For each request, the runtime server processes the SOAP message, translates the XML data into the service's native language, and invokes the service. When the service completes its work, the runtime server translates the return value into XML, packages it into a SOAP response message, and sends the message back to the calling application.

• Management tools provide mechanisms to deploy, undeploy, start, stop, configure, and administer your Web services. An administrative console is obviously useful, and other potential management services include a private UDDI registry, a WSDL repository, a single sign-on (SSO) service, and runtime monitoring facilities.
Figure 2.5 shows a typical Web services platform runtime environment. Web services are deployed within a Web services server (the runtime server). A Web services server can run standalone, or it can execute within a Web server or application server. In a Java environment, a Web services server normally runs as a servlet. A Web services server consists of a SOAP message processor and a Web service container. The SOAP message processor processes an incoming SOAP message, converts it from XML into programming language data (e.g., Java), and routes the request to the application that implements the service. The application usually executes within the Web service container, which manages the lifecycle of the application. The Web services server acts as a lightweight application server.

When you deploy a Web service, the Web services server usually generates a WSDL where part document that describes the Web service. (In some cases you may have to generate the WSDL document using the Web services development tools, or you may have to create one manually.) You'll want to register the Web service and its WSDL Description in a UDDI registry to help users find the service. The client application uses UDDI to find the service and its description, and then uses the WSDL file to generate a client proxy. At runtime the client uses the client proxy to construct and send SOAP messages to the Web service. This illustration also shows the client and the service using a single sign-on service for authentication.

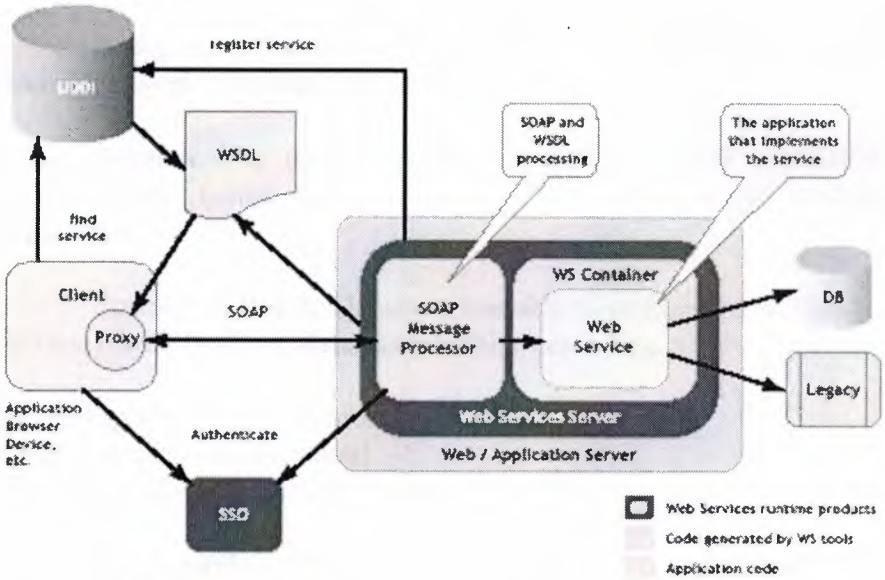


Figure 2.5: Internet middleware provides a ready-made Web Service environment.

A2.6 Global XML Web Services Architecture

Over the past five years, Microsoft has worked with other computing industry leaders to create and standardize a set of specifications for a new model of distributed computing called XML Web services. Already there are hundreds of customers deploying and running XML Web services and thousands more in the process of development. XML Web services standards, which include SOAP, XML, and WSDL, provide a high level of interoperability across platforms, programming languages and applications, enabling customers to solve integration problems easily.

XML Web services solutions become more global in reach and capacity – and therefore more sophisticated – it becomes increasingly important to provide additional capabilities to ensure global availability, reliability and security. Recognizing the need for standardizing these additional capabilities, Microsoft has released a new set of specifications that are the beginning of the Global XML Web Services Architecture. The aim of this architecture is to provide additional capabilities to baseline XML Web services specifications. The Global XML Web Services Architecture specifications are built on current XML Web services standards. Microsoft intends to work with key industry partners and standards bodies on these and other specifications important to XML Web services.

This chapter outlines the state of XML Web services today. It demonstrates the need for a set of additional capabilities as companies deploy XML Web services to support increasingly sophisticated business processes. Finally, it addresses how Microsoft is beginning to address the need for additional capabilities with the Global XML Web Services Architecture and its supporting specifications.

Companies are implementing XML Web services to integrate applications within the enterprise and to extend the reach of their businesses to partners and customers. This creates business efficiencies and exposes companies to new sources of revenue. Consider what the following companies are doing with XML Web services today:

Department Store Chain – Enterprise Application Integration

Background: The chain discovered that different credit approval applications had been developed in various parts of the company.

Solution: The chain exposed one credit approval application as an XML Web service. They linked this, in turn, to their point-of-sale, warehouse, and financial applications.

Business Benefits: The chain was able to expose the new credit approval application and use it with the three distinct applications operating around the company. As a result:

- Credit approvals became more consistent
- Maintenance costs decreased
- Billing back to departments became more efficient

Car Rental Company – Interoperability with Key Partner

Background: A major airline approached the car rental company about putting a link to the car reservation system on the airline's Web site. Linking the two proprietary reservation systems presented an extreme challenge.

Solution: The car rental company created a translation engine for sending data between the two systems.

Business Benefits:
• Car rental company developed another large sales channel
• Solution got to market quickly

Some early-adopter companies are implementing next-generation XML Web services solutions that support more sophisticated business processes. The next snapshot is representative of such a service.

Insurance Company – Interoperability across Several Companies

Background: A large insurer needed to generate quotes for dental coverage and make them available on the intranet of one of their large corporate customers. The insurer had already used XML Web services to integrate its rating and quotes engine. But it had outsourced the maintenance of the dental providers' directory and the credit rating service. These outsourced services were two vital elements for the generation of dental quotes and made the problem of their availability to clients non-trivial.

Solution: The insurance company, credit rating service, and dental provider orchestrated these applications to generate a quote that was requested by the customer on a corporate intranet.

Business Benefits: The insurance company considered this a transformational competitive advantage for the following reasons:

• It generated quotes in half the time of its competitors and provided them via a corporate intranet to one of its major customers.
• It automated existing business relationships at the level of multiple, interoperating applications. As a result, outsourcing became much more valuable, cutting the cost of quote generation by one third.
• It increased profitability in a thin-margin business.
• It provided a more seamless relationship with one of its biggest customers.

Because of application re-use and shortened time-to-market, companies can realize significant cost savings from taking an XML Web services approach to a solution. Through the integration of backend applications and key partner applications, companies are able to realize new efficiencies in the way they do business. Finally, by extending core functionality to partners, businesses are able to create new sales channels.

The following figure show in what areas will XML Web services be most effective for your company?

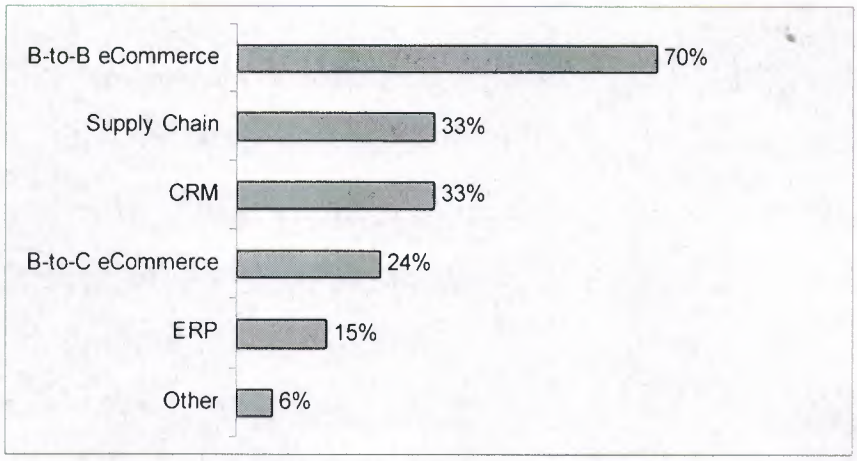


Figure 2.6: XML Web Services Business Areas [13].

As the business requirements that drive XML Web services become more complex, developers require additional capabilities not addressed by current XML Web services standards.

These capabilities include the following:

• Security: This is the most common concern for companies implementing XML Web services solutions today. In business ecosystems, developers need an end-to-end security architecture that is straightforward to implement across companies and trust boundaries.
• Routing: Companies building XML Web services solutions are concerned about the scalability and fault-tolerance of the business ecosystems they are building. Developers need a way of specifying messaging paths and the ability to configure those message paths dynamically.
• Reliable Messaging: This is a key requirement for mission-critical applications. Developers need an end-to-end guarantee of message delivery across a range of semantics such as: at-least-once, at-most-once, and exactly once.
• Transactions: Business ecosystems require the ability to transact across companies. As a result, developers need flexible process and compensation-based transaction schemes.

Because there are no broadly-adopted specifications for security, routing, reliable messaging, and transactions, developers today either have to go without these capabilities or they develop ad-hoc solutions that they must resolve separately with each partner or customer.

Going without these capabilities can expose a company to risks and degrade the value of its XML Web services. Creating ad-hoc solutions is time consuming and expensive. In addition, ad-hoc solutions encroach upon a central value area of XML Web services – cross organizational interoperability. Thus there is a need to provide the additional capabilities required by more sophisticated XML Web services on a broader basis to unleash the full power and promise of XML Web services development.

The Global XML Web Services Architecture builds on today's XML Web services baseline specifications of SOAP, WSDL, and UDDI. The overview of this architecture is shown in Figure 2.7.

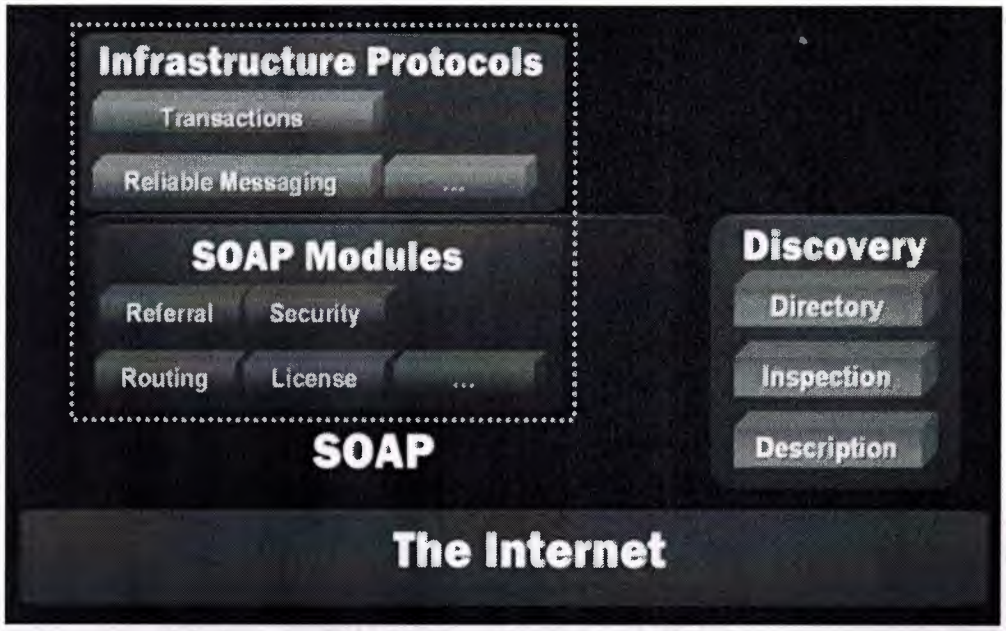


Figure 2.7: Global XML Web Services Architecture (GXA).

The Global XML Web Services Architecture is built on the Internet and the XML family of technologies. The architecture is divided into three conceptual layers: SOAP, SOAP modules, and infrastructure protocols.

SOAP is a lightweight, extensible, XML-based protocol for information exchange in a decentralized, distributed environment. Primarily, SOAP defines a framework for message structure and a message processing model.

Microsoft designed SOAP to be extensible. SOAP modules leverage this extensibility to provide compassable building blocks suitable for building the higher-level capabilities. Removed from the burden of specifying unnecessary capabilities, modules tend to be simple and focused. Modules express elements of functionality that can be used individually or combined to achieve composite and higher-level results. The functions provided by the module are consistently available and consistently expressed. This generality, breadth and uniformity allow a wide range of services to take advantage of XML Web services-enabled network infrastructure such as routers, switches, proxies, caches, and firewalls. Infrastructure protocols build on SOAP modules to provide end-to-end functionality. Protocols at this layer tend to have semantically-rich finite state machines as part of their definition. They maintain state across a sequence of messages and may aggregate the effect of many messages to achieve a higher-level result. Example infrastructure protocols include reliable messaging and transactions.

2.6.1 Design Principles

The Global XML Web Services Architecture rests on four key design principles:

Modular:

Rather than define large, monolithic specifications that offer end-to-end functionality, this architecture is built on modular components. These can be composed into

solutions to offer the exact set of features required by the problem at hand, neither omitting features nor burdening implementations with unnecessary ones. This building-block approach is a distinct advantage in creating an agile architecture. As new or extended features or capabilities are required, modular protocols can be created.

General-Purpose:

The Global XML Web Services Architecture is designed for a wide range of XML Web service scenarios, from B2B and EAI solutions to peer-to-peer applications and B2C services. Each module is uniformly expressed whether used individually or in combination with others, and is independent of any limited application domain.

Federated:

The Global XML Web Services Architecture does not require central servers or centralized administrative functions. The architecture enables communication across trust boundaries and autonomous entities. Neither the modules nor the protocols make any assumption about the implementation technology at the message endpoints. Any technology can be used; technologies can be transparently upgraded over time; services can be delegated or brought inside over time.

Standards-Based:

The Global XML Web Services Architecture builds entirely on the baseline XML Web services specifications SOAP, WSDL, and UDDI. Further, Microsoft is committed to working with industry partners to standardize future specifications critical for XML Web services interoperability.

2.6.2 New Specification

The following describes a set of new Global XML Web Services specifications made available in October 2001. These specifications represent a significant step toward a comprehensive Global XML Web Services Architecture. The four specifications — WS-Security, WS-License, WS-Routing and WS-Referral — build on the SOAP family of Extensible Markup Language (XML) interoperability technologies. For the specifications themselves and more extensive technical background.

Security

Organizations building and managing secure XML Web services need to ensure that only authorized parties are allowed to use the XML Web services and that the SOAP messages sent and received by the XML Web services can only be modified or viewed by appropriate parties. As discussed above, these secure XML Web services typically operate in heterogeneous environments that span multiple authentication technologies and trust domains; consequently the underlying XML Web service security protocols must be very flexible.

WS-Security describes how to use the existing W3C security specifications, XML Signature and XML Encryption, to ensure the integrity and confidentiality of SOAP messages. And together with WS-License, it describes how existing digital credentials and their associated trust semantics can be securely associated with SOAP messages. Together, these specifications form the bottom layer of comprehensive modular security architecture for XML Web services. Future security specifications will build on these basic capabilities to provide mechanisms for credential exchange, trust management, revocation, and other higher-level capabilities. The two initial security specifications provide the following capabilities:

WS-Security is a simple, stateless, SOAP extension that describes how digital credentials should be placed within SOAP messages, and how these credentials should be associated with a message to ensure message integrity and confidentiality. WS-Security describes how message integrity is maintained even for SOAP messages that use the WS-Routing specifications described below. Using WS-Security, XML Web services can examine incoming SOAP messages and, based on an evaluation of the credentials, determine whether or not to process the request. WS-Security supports a wide range of digital credentials and technologies including both public key and symmetric key cryptography.

WS-License describes how several common license formats, including X.509 certificates and Kerberos tickets, can be used as WS-Security credentials. WS-License includes extensibility mechanisms that enable new license formats to be easily incorporated into the specification.

Routing

As SOAP messaging evolves into a general-purpose Global XML Web Services Architecture there must be a means of addressing and transmitting SOAP messages over various types of communications systems. This enables a wide range of communication patterns such as peer-to-peer or store-and-forward networking. It also allows messages to be efficiently vectored to distribute processing nodes.

These features are supported by the following specifications:

WS-Routing is a simple, stateless SOAP extension for sending SOAP messages in an asynchronous manner over a variety of communication transports such as TCP, UDP, HTTP. WS-Routing provides addressing mechanisms that enable specification of a complete message path for the message (including its return path). This enables one-way messaging, two-way messaging such as request/response, peer-to-peer conversations, and long running dialogs. WS-Referral is a simple SOAP extension that enables the routing between SOAP nodes on a message path to be dynamically configured. This configuration protocol enables SOAP nodes to efficiently delegate part or all of their processing responsibility to other SOAP nodes. Because WS-Security, WS-License, WS-Routing, and WS-Referral are modular, they can be used together. For example, WS-Security describes how to digitally sign SOAP messages that use a WS-Routing header. Each of these specifications provides extension and composition mechanisms that enable future Global XML Web Services Architecture specifications to be incorporated into a complete solution.

2.6.3 Roadmap

Interactions across organizations have many opportunities for failure ranging from transmission errors to incompatible or unavailable business processes. Reliable Messaging and Transactions allow the builders of XML Web services to manage the scope and effect of failures.

Reliable Messaging

XML Web services need to operate reliably over intranets and the public Internet, over transport protocols that are not completely reliable. Reliable messaging addresses issues arising from transmission errors. This SOAP-level reliable messaging protocol provides delivery guarantees isolating application processes from the detailed handling of transmission failure and its recovery, allowing a developer to concentrate on automating a process with a much-simplified error handling model. In the exchange of messages, individually or as part of a long-running process, communicating parties will be able to obtain end-to-end delivery guarantees so that messages will not be lost, duplicated or delivered in the wrong order.

Transactions

Transactions address the possibility of business-level inability to complete a process. Transactions allow multiple parties involved in a process to arrive at a consistent final outcome (or discover that this is not possible). Existing "two-phase" commit protocols are appropriate in some circumstances. Also needed are more loosely-coupled techniques, such as exceptions and compensation, which enable a broader range of transactions to be automated across trust boundaries. Developers will have powerful process-modeling languages to express the patterns of messages exchanged between XML Web services, the interactions of those messages, and the business processes they reflect, including both normal and exceptional conditions. Reliable messaging and transactions represent important challenges that companies face today when developing advanced XML Web services across trust boundaries. The Global XML Web Services Architecture provides the flexibility to develop such additional capabilities and the structure to implement and deploy them.

Microsoft's commitment to the standards is supported by a history of participation in the standards process. In 1996, Microsoft co-authored the first draft of XML. In 1999, Microsoft and its partners presented SOAP to the IETF. In March 2001, WSDL was submitted to the W3C by IBM and Microsoft. Microsoft is a member of the industry initiative developing UDDI. This strong commitment carries through the specifications underlying the Global XML Web Services Architecture and is similar to other efforts we have taken with specifications behind XML Web services.

As the business requirements that drive XML Web services become more complex, the architecture for XML Web services requires additional capabilities to handle the increasingly complex nature of XML Web services solutions. Recognizing this emerging need for additional capabilities, Microsoft has released a new set of specifications that are the beginning of the Global XML Web Services Architecture. The aim of this architecture is to provide additional capabilities to current XML Web services standards. Microsoft intends to work with key industry partners and standards bodies on these and other specifications important to XML Web services.

A3 WEB SERVICES SECURITY APPROACHES

Opening up Web services to the world requires a robust security implementation to stop unauthorized access. Learn about two basic Web services security approaches: channel security and package security.

If you ask development architects from either the Java or the .NET camp about the future of software development, they can finally agree that the future of software development is in applications written for virtual machines—such as the Java Virtual Machine (JVM) or the .NET Common Language Runtime (CLR)—where applications on different platforms are tied together using XML Web services. The days of COM and CORBA interoperability problems will be behind us very soon.

There are two basic ways to implement Web services security. The first is to manage security at the channel level. The second is to modify the package to support security.

A3.1 Channel Security

When using channel security, the developer doesn't need to know anything about the specific implementation. Infrastructure engineers simply use standards-based mechanisms to create a secure conversation between two systems consuming each other's Web services.

3.1.1 Channel Security Terminology

Channel security has the advantage of being standards-based so that any two platforms can use the same security mechanism regardless of the operating system. The other major advantage is that channel security involves configuration rather than a change to the way developers write code. There are three disadvantages to using channel security. First, there will be a performance negative impact when the path between two systems is being manipulated by hardware or software to ensure security. For example, using Secure Sockets Layer (SSL) security imposes a 30 to 40 percent performance penalty. The second disadvantage to channel security is that it's protocol dependent. Both sides of the connection need to be able to communicate using the selected protocol. And lastly, many organizations will have to invest in additional infrastructure to support efficient implementation of channel security.

3.1.2 Implementing Channel Security

There are three basic ways to implement channel security. Using credentials and SSL, you can configure IIS to only allow certain user IDs and passwords to access the URL pointing to the Web service, and then protect it with an SSL certificate. You can gain an extra measure of security by decorating your methods with declarative security attributes that only allow those particular user IDs and passwords to execute the Web services methods. The second method involves configuring a local certificate server (an optionally installable, but included, service in Windows 2000) and then sending out certificates to the companies that need to access your Web services. You can then use IIS to configure the Web services'

virtual directory to accept only requests from other systems possessing these certificates. Finally, you can configure your system to only allow Web services calls over a secure PPTP or IPSec connection. You can use a combination of these methods to increase security, but be aware that the more layers you place on the channel, the more your performance will suffer.

A3.2 Package Security

Package security involves making changes to the SOAP package itself in code. For two systems to talk, they must both understand how to process the modified package.

3.2.1 Package Security Terminology

Channel security involves configuration rather than coding, package security focuses on coding instead of configuration. Implementing package security will have less of an effect on performance, but interoperability will suffer.

3.2.2 Implementing Package Security

There are three ways to implement package security. First, you can implement custom SOAP headers. Implementing custom SOAP headers allows you to embed security information into the header and then reject the SOAP package if the information isn't found. The .NET Framework makes it simple to implement customer headers. You need only to inherit from the SoapHeader class and use the SoapHeaderAttribute to describe your new header. The .NET Framework will automatically generate the required proxy class and WSDL to support your new header. Rather than changing the contents of the SOAP message, you can simply encrypt the entire message. Configuring encryption at the package level requires that the server and client have access to the public and private keys necessary to encrypt and decrypt the secured packages and that they use standard encryption algorithms.

The final method for implementing package security is much more complex, but will make more sense for companies that already have a significant investment in Microsoft's Internet Security and Acceleration Server (ISA). ISA has the ability to implement custom filters that can validate SOAP requests, perform method level authentication, and even cancel method invocation in cases where the filter detects an anomaly. Unfortunately, the current version of ISA requires that the filters be written as ISAPI filters using Microsoft C++, a skill set few developers possess. Still, this is the most secure way to implement package level security; if you have access to the skills and have an existing or planned ISA infrastructure, so it's highly recommended .

A3.3 Defending Your XML Web Service Against Hackers

One of the biggest concerns we hear from developers when we talk about the potential of XML Web Services is the fear of vulnerabilities that might allow malicious users

to attack their services. The bad news is that attacks can result in such atrocities as limiting the availability of your service, private data being compromised, or in the worse case, losing control of your machines to these malicious users. The good news is that there are real protections available to you that can limit the risks involved from these attacks. We are going to take a look at what kind of attacks are out there, and what you can do to protect yourself in the areas of deployment, design and development. This first column on the subject will focus on deployment issues you should consider; in our next column, we will look at design and development issues that you need to be aware of when developing your XML Web Services.

3.3.1 Types of Attacks

The first step to figuring out what the risks are, and what we can do to avoid them, is to understand the types of attacks that might target our services. Once we know the sorts of issues we are vulnerable to, we can mitigate those risks by taking appropriate actions.
Attacks fall under three general categories:
* Spoofing
* Taking advantage of bugs
* Denial of Service
* Spoofing

One of the most common attacks against a system that requires authentication is for a hacker to figure out a user's authentication credentials, log on as that user, and access that user's information. This is bad enough, but it can pose even more of a risk if the compromised credentials belong to a system administrator or some other higher privileged user. In such a case, the attack might not only compromise a single user's data, but could potentially compromise the data of all users.

There are various approaches a hacker might use to determine a user's password. For example: trying words that might be meaningful to the user, such as the user's name, their pet's name, or their birthday. A more persistent hacker may even try every word in the dictionary (a dictionary attack). Other ways of getting at credential information include: sniffing the network packets and reading the information from the data being sent; by DNS spoofing, inserting a malicious machine as an intermediary between a client and the server; posing as a system administrator and requesting the user give their credentials for trouble-shooting purposes; or by recording a logon handshake with a server and replaying the sequence in order to attempt to get authenticated.

Much of the risk of spoofing can be mitigated by taking such measures as enforcing strong passwords, and by using secure authentication mechanisms.

Taking Advantage of Bugs:

One of the key factors determining the vulnerability of a system to attack is the quality of the code running on that system. Bugs on the system can do more than simply cause a particular thread to throw an exception. Hackers can potentially use these vulnerabilities to execute their own code on the system, to access resources with elevated privileges, or to simply take advantage of resource leaks (based on the bug) that can potentially cause your system to slow down or become unavailable. One of the most famous of this sort of attack is the Code Red Worm virus, which took advantage of a bug in the

Index Server ISAPI extension to execute code of its choosing on an infected system, and then went looking for other vulnerable machines.

Another common attack is to take advantage of bugs where assumptions are made on the validity of input data. For instance, consider an XML Web Service that expects a user name to be entered as a parameter. If you assume that the user name just contains an ASCII string, and so place it directly into your SQL query, you may expose your service to serious vulnerability. For instance, say you have a SQL query in your code that is built like this:

sqlQuery = "SELECT * FROM Users WHERE (Username='" & UsernameInput & "')
If the UsernameInput parameter happens to contain something like
Bob') or not (Username='0
then your service may return all records, and not just the one for a specific user.

Denial of Service:

Denials of service attacks are not aimed at breaking into a site, or at changing its data; rather, they are designed to bring a site to its knees, so that it cannot service legitimate requests. The Code Red Worm virus not only infected machines and then looked for more machines to infect, it also caused the infected machines to send a large number of packets to the official White House Web site. Because thousands of machines were infected, the number of requests sent to the White House Web site was extremely high. By sending requests from a large number of machines, the Code Red Worm virus became what is considered a "distributed denial of service attack," which is extremely difficult to limit, since there are so many machines involved. Denial of service requests may come in many forms, because there are so many levels at which bogus requests can be sent to attack your system. For instance your site may allow users to PING your IP address, which causes an ICMP message to be sent to your server and then returned. This is an effective means of troubleshooting connectivity problems. Nevertheless, if hundreds of machines send thousands of packets each to your server at the same time, you will probably find that your machine is so busy handling the PING requests that it can't get the CPU time to handle other, normal requests.

A slightly higher level is a SYN attack, where a low-level network program is written to send what appears to be the first packet (a SYN packet) in a TCP connection handshake. This tends to be more damaging then a PING request, because unlike PING requests, which can be ignored if you want, as long as you have an application listening on a TCP port (like your Web server) you are vulnerable to expending resources whenever you get what appears to be valid connection requests.

On the high end of the spectrum, denial of service attacks can take the form of sending to your XML Web Service multiple, basically valid SOAP requests that cause database lookups to occur. Database lookups may take a long period of time. thus, if thousands of such requests are sent to your server every second, it can cause both the Web server that receives the request, and the database server on the backend, to become excessively busy. Again, this may cause your service to be unable to handle other requests in a timely manner.

If you have code with bugs on your machine, then denial of service attacks may be even easier. For example, if your production Web Service made the mistake of displaying a

message box in the case of a particular type of error, a hacker could use this flaw to send a relatively small number of requests to your machine that cause the message box to be displayed. This can lock up all of the threads handling requests, thereby effectively making your service inaccessible to others.

3.3.2 Limit Who Can Access Your Web Servers

If you are concerned about attacks, particularly if you have information on your XML Web Service that is private, then you should restrict access to your site to legitimate users only. This can be achieved in a number of ways, but here are a few that can keep attackers from accessing your XML Web Service.

Authenticate users by using HTTP authentication; then limit the resources they have access to. Authentication is configured by: right-click on the Web site, virtual directory, or individual file in the Internet Services Manager; select the Properties option from the pop-up menu; go to the Directory Security tab and click the Edit button under Anonymous Access and Authentication Control.

Restrict the IP addresses that can access your Web server. If you have a small list of legitimate users who can use your site, you may want to only allow their particular IP addresses to have access. You can also limit access to certain ranges of IP addresses, or deny access to an IP address or a range of IP addresses. You can even limit access based on domain names, but that requires potentially lengthy domain name lookups on the IP addresses that connect to your machine. IP address restrictions are modified by: go to the Directory Security tab mentioned in step 1, and clicking on the Edit button under IP address and domain name restrictions.

3.3.3 Configure TCP/IP Filtering to Limit Ports

If you don't have a router for your firewall, or if you are unable to administer your router for any reason, you can effectively make your own machine your firewall by limiting the sorts of incoming connections it will receive. In Windows 2000, click the Start button, select Settings, select Network and Dial-Up Connections, right-click the network card that is connected to the Internet, and select Properties. Select Internet Protocol (TCP/IP) and click the Properties button; click the Advanced button and go to the Options tab. Select TCP/IP filtering and click the Properties button. Access is restricted such that only ports 80 and 443 for HTTP and HTTPS connections are allowed.

3.3.4 Use Microsoft IIS Security Checklist

Microsoft created a security checklist for Internet Information Server 4.0 that mentions all that I have mentioned here, as well as much more. Use this checklist to make sure you have at least considered all your options for security. Though you are probably not running Internet Information Server 4.0 (5.0 is the version that comes with Windows 2000), most of the steps still apply, and will continue to apply to future versions of Internet Information Server. The checklist can be found at Microsoft Internet Information Server 4.0 Security Checklist.

A3.4 Making Our Choice

Many companies will opt for the channel-based method because most Web services implementations are with known partners to whom they are willing to issue credentials necessary for validation. Packet-based methods are best for internal implementations where you can control both sides of the development effort and you need more granular control of the security. When planning your Web services based systems, you'll probably find ways to implement both now that you know the options and the pitfalls.

A4 OUR PROPOSED ARCHITECTURE

A4.1 Project Motivation

Providing a comprehensive model of security functions and components for Web services requires the integration of currently available processes and technologies with the evolving security requirements of future applications. It demands unifying concepts; it requires solutions to both technological (secure messaging) and business process (policy, risk, trust) issues; and finally, it requires coordinated efforts by platform vendors, application developers, network and infrastructure providers, and customers. The goal is to enable customers to easily build interoperable solutions using heterogeneous systems.

Our security trust model provides a flexible framework within which the organizations can interconnect when configured with appropriate authorization. At the same time, every customer and every Web service has its own unique security requirements based upon their particular business needs and operational environment. Because these requirements may be combined in many ways and expressed at different levels of specificity, our approach to Web service security has a set of flexible, interoperable security primitives that, through policy and configuration, enable a variety of secure solutions.

Our solution address web services security challenges , by creating secure, interoperable Web services based on a set of security abstractions that unify formerly dissimilar technologies. This enables specialization to particular customer requirements within an overall framework while at the same time permitting technologies to evolve over time and be incrementally deployed. As an example of this evolutionary approach, the secure messaging model can be added to existing transport-level security solutions. A customer can add message-level integrity or persistent confidentiality (encryption of message elements) to an existing Web service whose messages are carried through, for example, Secure Sockets Layer (SSL/TLS). The messages now have integrity (or confidentiality) that persists beyond the transport layer.

We anticipate that the proposed model and specifications that emerge will be broadly available from multiple vendors and will be considered by appropriate standards organizations. Together, the model, specifications, and standards process enable businesses to quickly and cost-effectively increase the security of their existing applications and to confidently develop new interoperable, secure Web services.

The business advantage of such a model is clear. By framing comprehensive security architecture for Web services, organizations and customers can be better ensured that their investments and assets are protected as business processes become increasingly recast as Web services.

A4.2 Moving to Microsoft Visual Studio .Net

The term Web Service will mean the following: software that is exposed to other software over Internet-friendly protocols. Short, sweet, and to the point. I promise to use the term only when it applies to the preceding definition.

Web Services typically use HTTP as their transport, although other Internet-friendly protocols (such as SMTP) can be used. Web Services typically use XML,XML schemas as their data format, although other MIME-friendly formats are possible. Web Services differ from traditional Web pages or applications primarily based on the target audience. Traditional Web pages and applications target carbon-based life forms (humans). Web Services target silicon-based life forms (software agents).

XML Schema provides structure and type over data. Classically, XML's strength has been providing structure, both in terms of its transfer syntax and in terms of its abstract data model. We must mention that the Microsoft© .Net support for XML Schemas . XML Schemas are a critical component for Web Service infrastructure, as SOAP relies on the XML Schema type system for its data model and WSDL (Web Services Description Language) relies on the XML Schema language for defining message formats.

A4.3 Analyze WS-Security

WS-Security is an extension of the SOAP specification. It's designed to provide quality of protection by means of message integrity, message confidentiality, and single message authentication. Perhaps most importantly, WS-Security provides a generic mechanism for associating security tokens with messages. No specific type of security token is required by the specification. It provides a specific description of how to encode X.509 certificates and as well as how to include opaque encrypted keys. The basic premise of WS-Security is to define a means of sending SOAP messages in a variety of formats to a number of different actors, and providing the actors with a method of validation to assert that the entire SOAP request as a whole is valid, as the following figure representing a basic mechanism between actors in a secure web services environment.

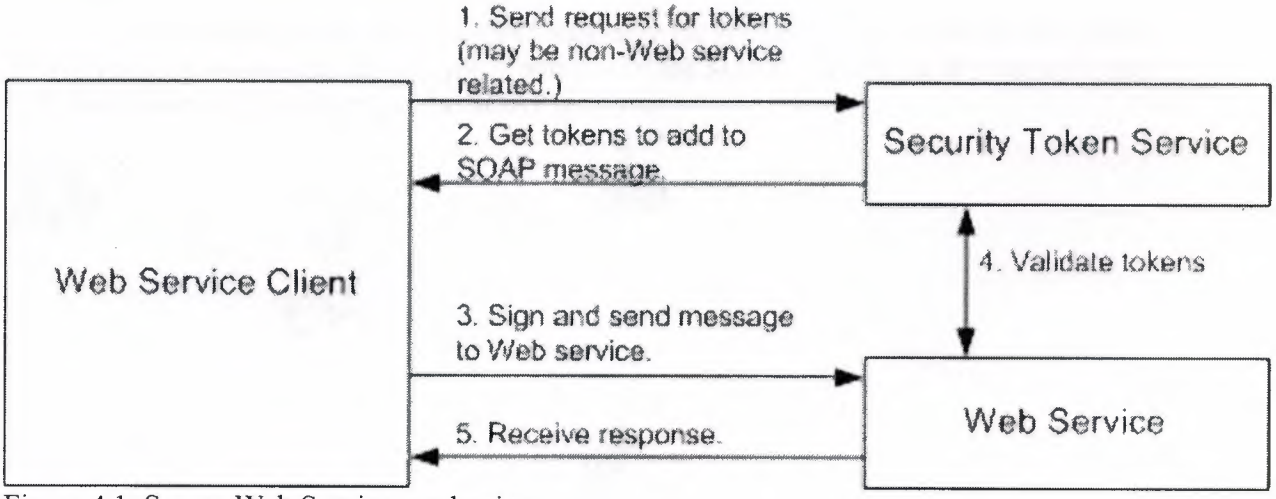


Figure 4.1: Secure Web Service mechanism

There are two key factors in a message that allow an end-point to establish a trust relationship: message integrity and message confidentiality. Message integrity is provided by leveraging the existing XML Signature specification in conjunction with security tokens to ensure that messages are transmitted without modification as Figure 4.2 illustrates the keys

needed by the sender and receiver to both sign a SOAP message and to verify the signature. To sign the SOAP message, the sender needs its private key and for a receiver to verify the signature within a SOAP message it needs the sender's public key.

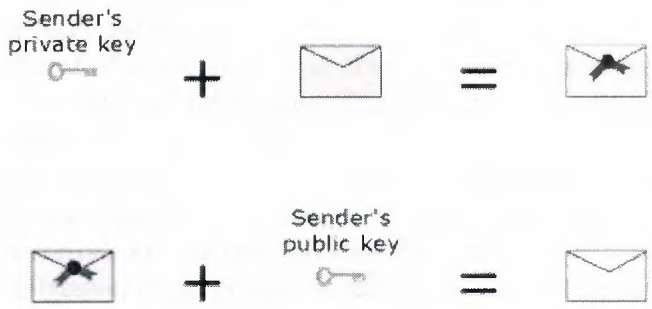


Figure 4.2: Signing SOAP massage.

Message confidentiality leverages the existing XML Encryption in conjunction with security tokens to keep portions of the SOAP message hidden and secure, as following graphic illustrates the keys needed by the sender and receiver to both encrypt and decrypt a SOAP message. To encrypt a SOAP message, the sender needs the receiver's public key and the receiver needs its private key to decrypt the SOAP message.

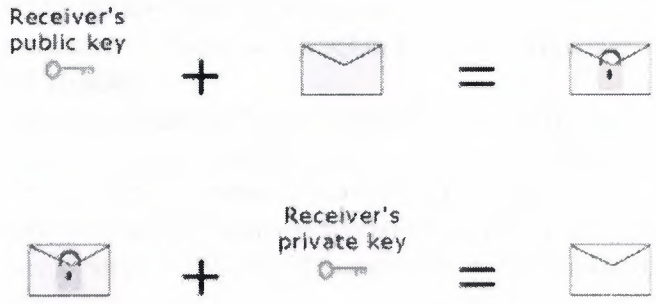


Figure 4.3: Encrypting SOAP message

The following figure 4.4 illustrates that only the intended recipient of an encrypted SOAP message can access the encrypted portions of the SOAP message, as it is the only one with the private key that decrypts the SOAP message.

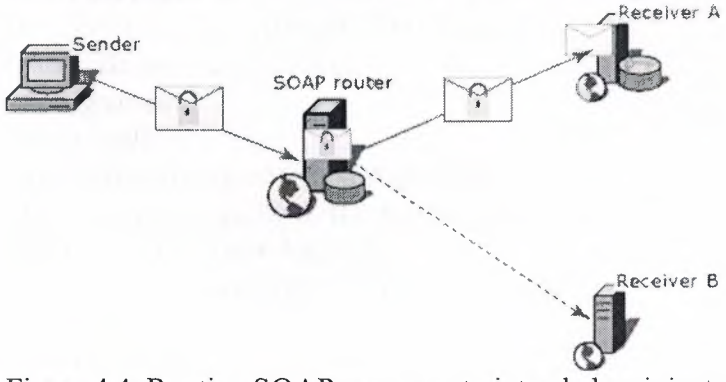


Figure 4.4: Routing SOAP messages to intended recipients

When envisioning SOAP Web service request protected by WS-Security, there are a number of different elements that come into play in the overall request, from the point of transmission to the point of acceptance. The specification defines the following terms:

•Claim--a statement that a client makes (e.g., name, identity, key, group, privilege, capability, etc.)
•Security token--represents a collection of claims
•Signed security token--a security token that is asserted and cryptographically endorsed by a specific authority
•Proof-of-possession--data that is used in a proof process to demonstrate the sender's knowledge of information that should only be known to the claiming sender of a security token
•Integrity--the process by which it is guaranteed that information is not modified in transit
•Confidentiality--the process by which data is protected, so that only authorized actors or security token owners can view the data
•Digest--a cryptographic checksum of an octet stream
•Signature--a cryptographic binding of a proof-of-possession and a digest

The following example is taken from the WS-Security specification. It gives you an idea of what the WS-Security message looks like when transmitted:

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
<S:Header>
<m:path xmlns:m="http://schemas.xmlsoap.org/rp">
<m:action>http://fabrikam123.com/getQuote</m:action>
<m:to>http://fabrikam123.com/stocks</m:to>
<m:from>mailto:johnsmith@fabrikam123.com</m:from>
<m:id>uuid:84b9f5d0-33fb-4a81-b02b-5b760641c1d6</m:id>
</m:path>
<wsse:Security>
<wsse:BinarySecurityToken
ValueType="wsse:X509v3"
EncodingType="wsse:Base64Binary"
Id="X509Token">
MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
</wsse:BinarySecurityToken>
<ds:Signature>
<ds:SignedInfo>
<ds:CanonicalizationMethod Algorithm=
"http://www.w3.org/2001/10/xml-exc-c14n#"/>
<ds:SignatureMethod Algorithm=
"http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
<ds:Reference>
<ds:Transforms>
<ds:Transform Algorithm=
"http://...#RoutingTransform"/>
<ds:Transform Algorithm=
"http://www.w3.org/2001/10/xml-exc-c14n#"/>
</ds:Transforms>
<ds:DigestMethod Algorithm=
```

```
"http://www.w3.org/2000/09/xmldsig#sha1"/>
<ds:DigestValue>EULddytSo1...</ds:DigestValue>
</ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>
BL8jdfToEb1l/vXcMZNNjPOV...
</ds:SignatureValue>
<ds:KeyInfo>
<wsse:SecurityTokenReference>
<wsse:Reference URI="#X509Token"/>
</wsse:SecurityTokenReference>
</ds:KeyInfo>
</ds:Signature>
</wsse:Security>
</S:Header>
<S:Body>
<tru:StockSymbol xmlns:tru="http://fabrikam123.com/payloads">
QQQ
</tru:StockSymbol>
</S:Body>
</S:Envelope>
```

A4.4 Design WS-Security

Web Service Security Solution provides message integrity through digital signing SOAP message by using username token and X509 certificate in the case of digitally signing the SOAP message by the user name token which will be the security token in the SOAP message. In the above mentioned case the chosen certificate must support digital signature and have a non-NULL private key and the expiration of the certificate must be valid.

The enhanced SOAP message constructed at the consumer must be complied with the same set of Security rules that have been agreed on in the protocol to insure the integrity and consistency of the data. If the provided tokens are not authorized then the Web Service will reject the incoming message.

Figure 4.5 shows the enhanced structure of the SOAP message. It provides a description of how the client constructs an enhanced SOAP message.

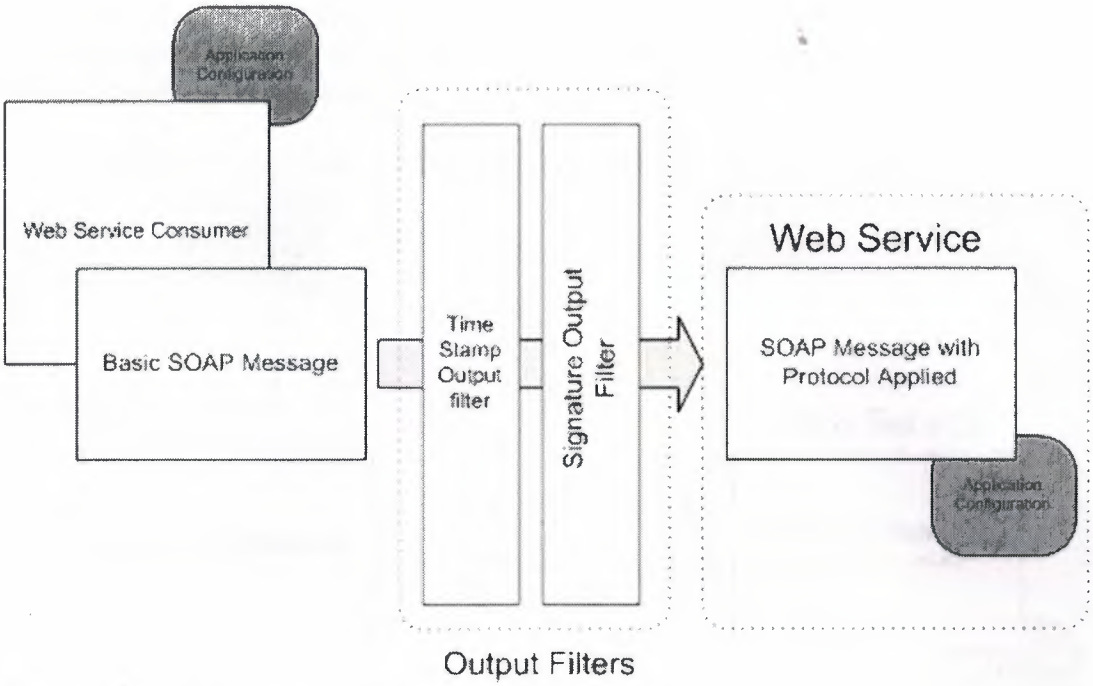


Figure 4.5: Client Signing SOAP message construction.

The following figure 4.6 shows how the Web Service processes the enhanced SOAP message.

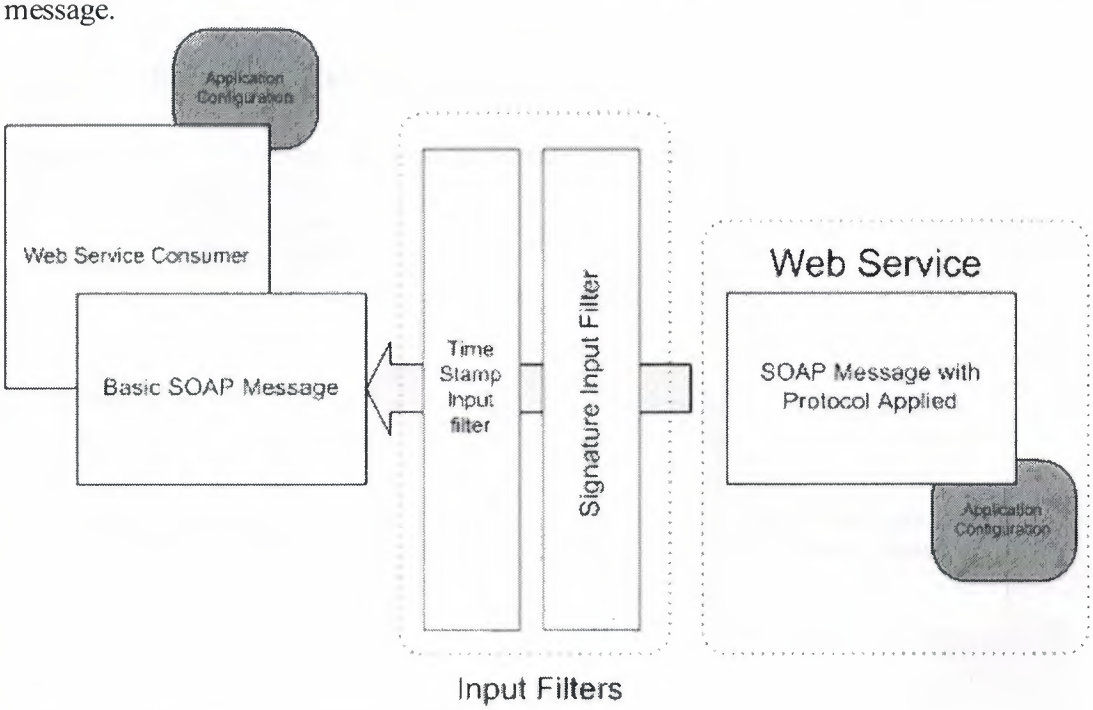


Figure 4.6: Processing Signed SOAP message.

Web Service Security Solution also provides message confidentiality through encrypting the SOAP message by shared symmetric key between the consumer and the web service and encrypting the SOAP message by X509 certificate.

In case of encrypting the SOAP message we took into consideration some security issues such as constructing the shared symmetric key at the beginning of communication between the consumer and the Web Service.

In the case of encrypting the SOAP message by X509 certificate the chosen certificate for encrypting the SOAP message4 must have a public key for encryption and the expiration of the certificate must be valid.

The following figure 4.7 shows encrypting the SOAP message by the client.

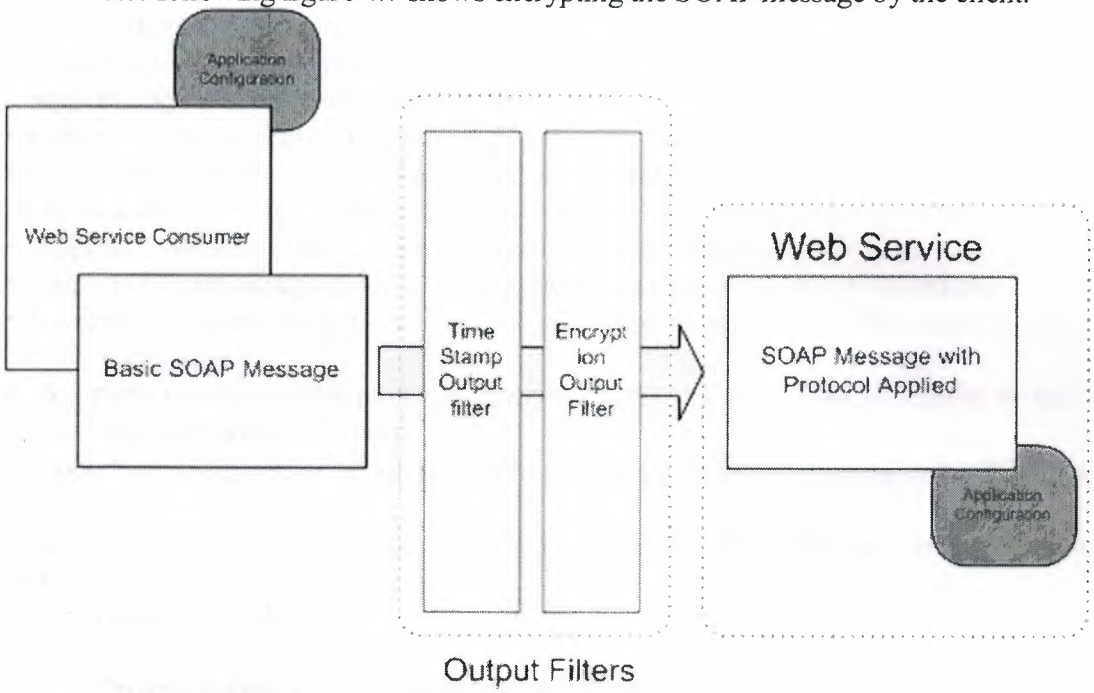


Figure 4.7: Encrypting SOAP message by the client

Figure 4.8 shows how the secure Web Service processes the encrypted message.

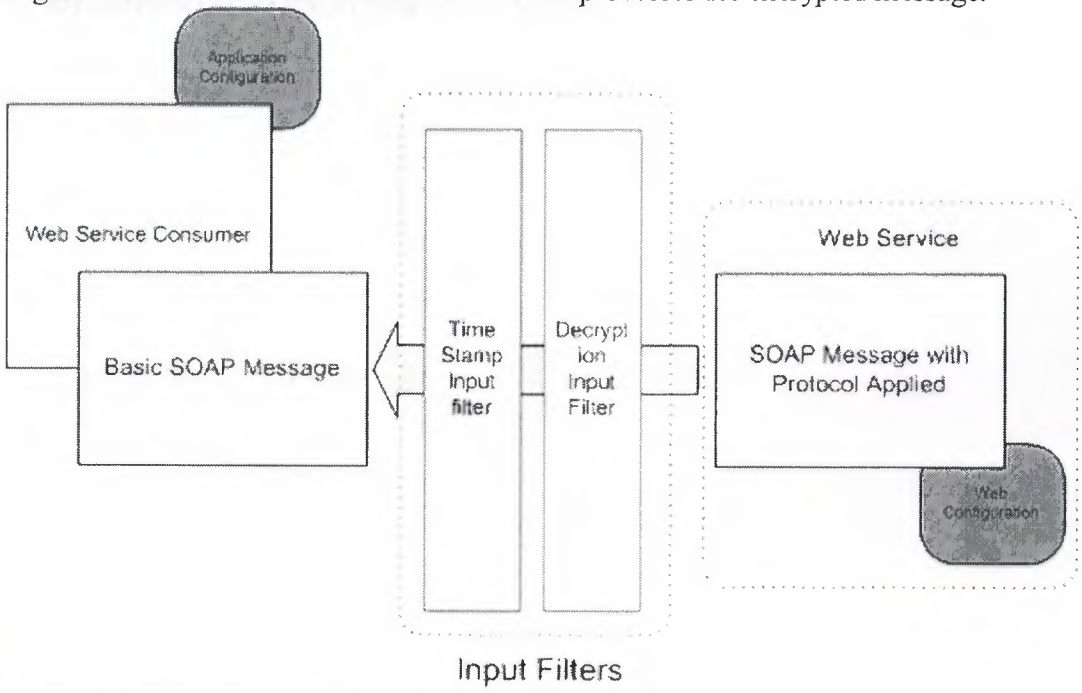


Figure 4.8 processing the encrypted SOAP message

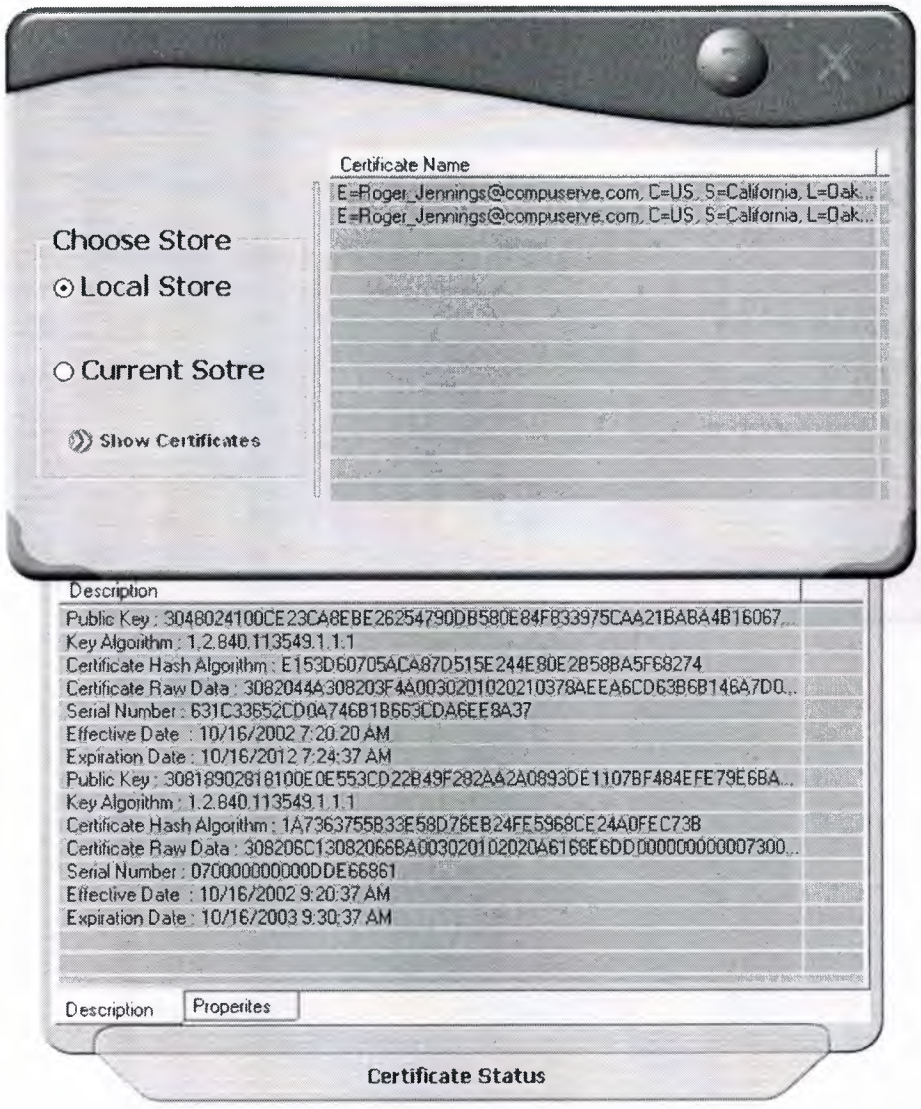
4.5 WS-Security Implementation

The solution achieved has the following characteristics:

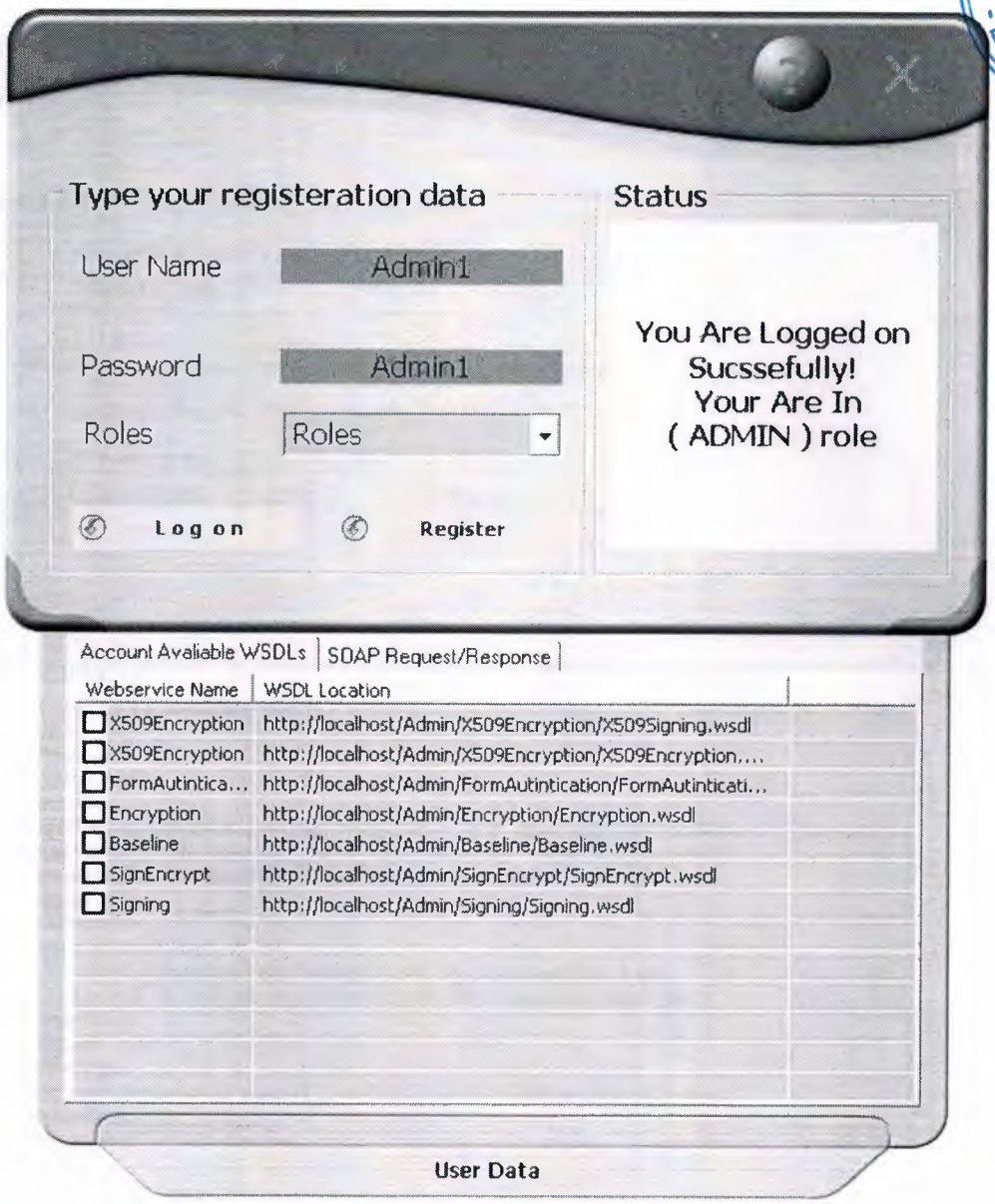
- Is complied with the World wide Consortium (W3C) standards Ver. 1.2 accommodated for the WS-Security.
- Extend SOAP security context from End to End.
- Supports synchronous and asynchronous calls.
- Supports secure socket layer (SSL) Ver. 3.0.
- Platform independent (.NET Framework implementation).
- Can be extended to the future web service specifications.
- Supports Symmetric and Asymmetric encryption techniques.
- Supports Digital signing the SOAP message by user name and X.509 certificate.
- Supports one way encryption i.e. encrypting the message from the client to the Web Service (one-way).
- Supports two way encryption i.e. encrypting the message from the client to the Web Service and vice versa (two-way).
- Supports one way signature i.e. encrypting the message from the client to the Web Service (one-way).
- Supports two way signatures i.e. encrypting the message from the client to the Web Service and vice versa (two-way).
- Monitoring capability.

Presented below are some of the achieved results and screen shots that are taken from the application presenting the achieved studies.

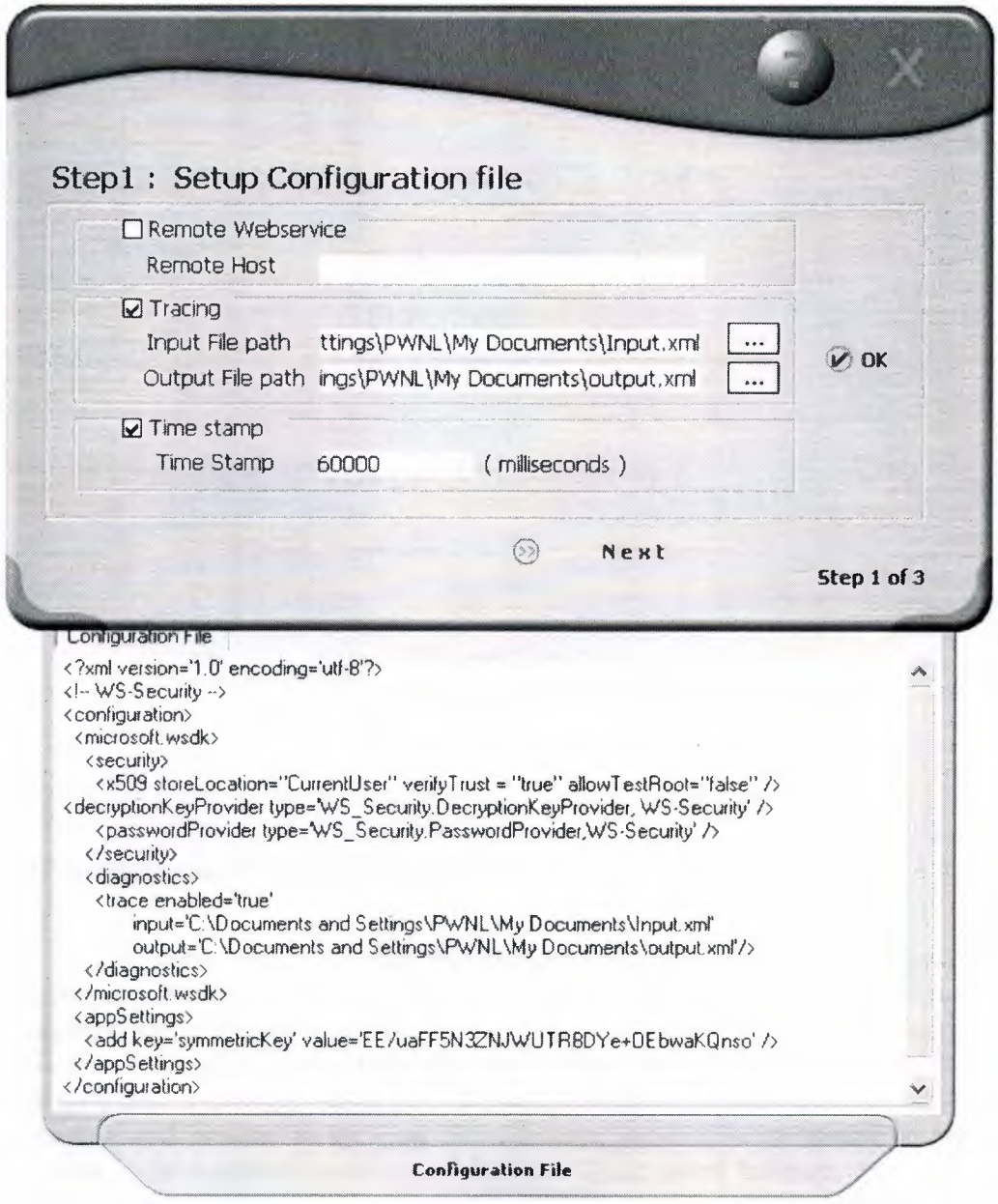
X.509 certificates on local or remote machine store.



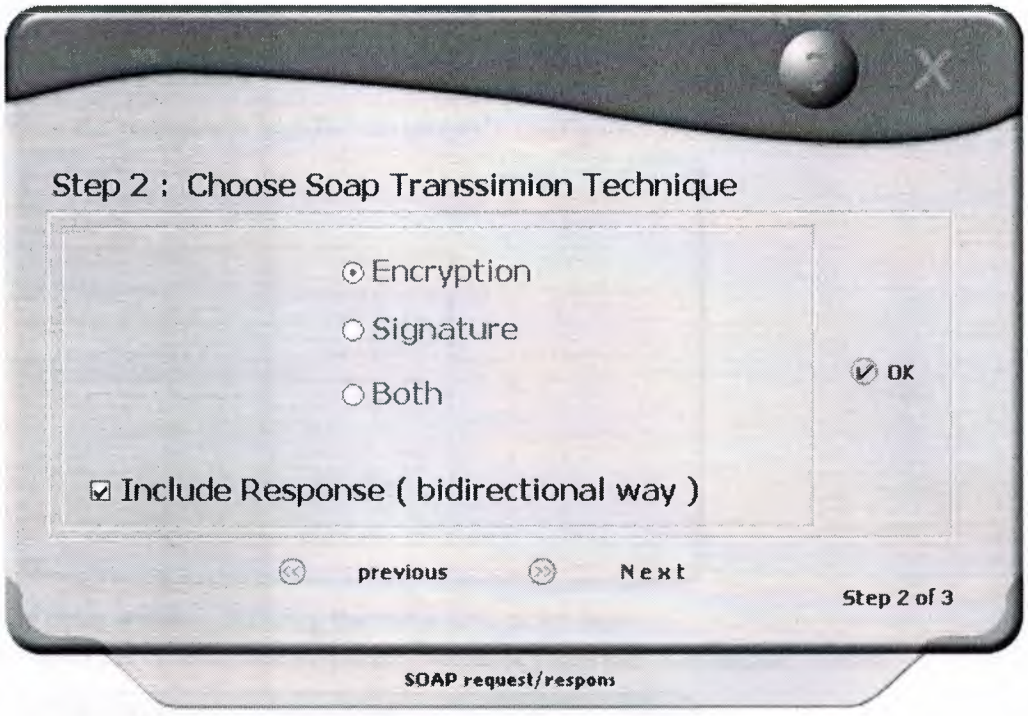
The user must be login according to the Web Server roles.



The client must configure the protocol configuration to communicate with the Web Service.



Capability to monitor the input and the output traced SOAP messages.



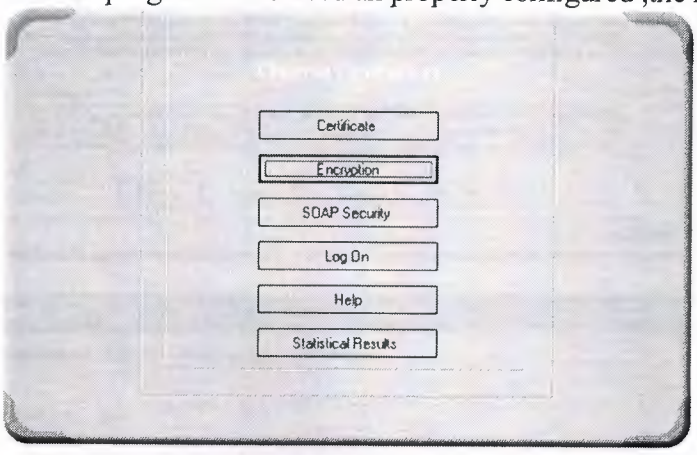
A4.5 Results and Recommendations

The performance for the one way encrypting is better than the signature in comparison with the time required for processing the SOAP message. However using both encryption and signature will result in a through degradation of the efficiency. Providing more than one option gives a flexibility for the business to choose which sort of trade off to give up if time is the major issue then choosing encryption would be the best choice however if secrecy and privacy is the goal then choosing both is what would be preferred. By choosing only signature you can achieve a balance point between time efficiency and secrecy.

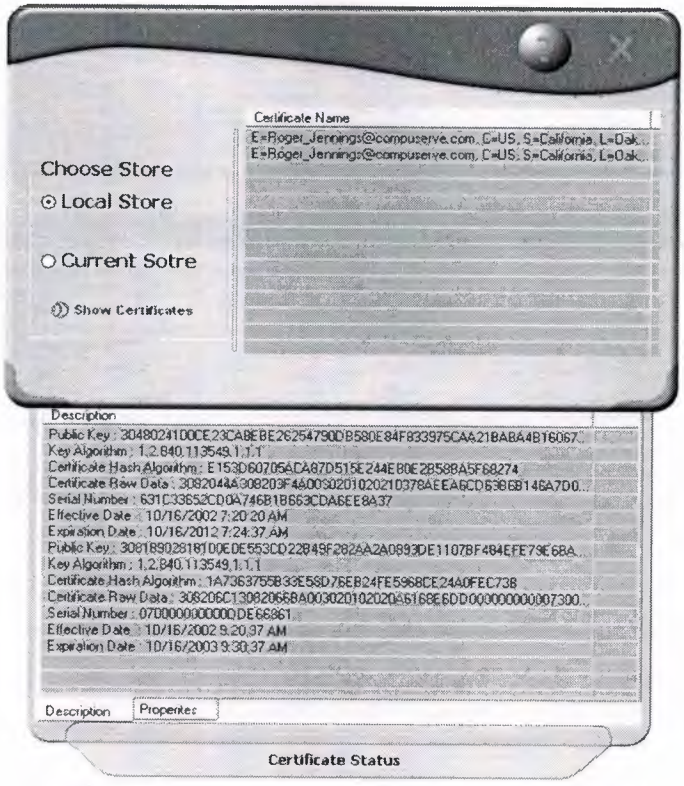
A5 APPENDIX A

A5.1 User Manual

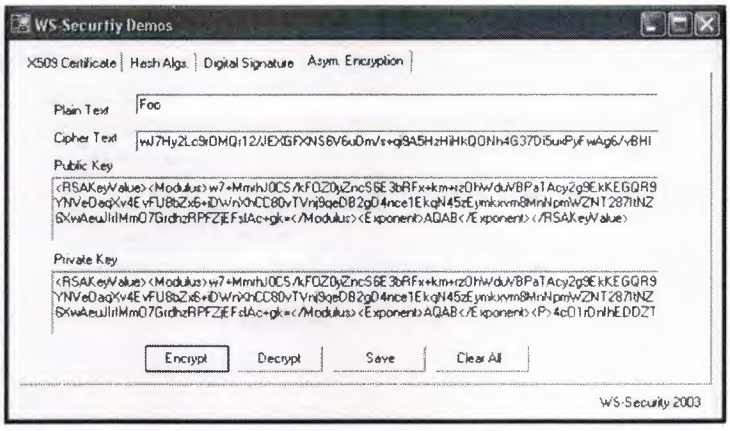
When the program is installed an properly configured ,the main window appears

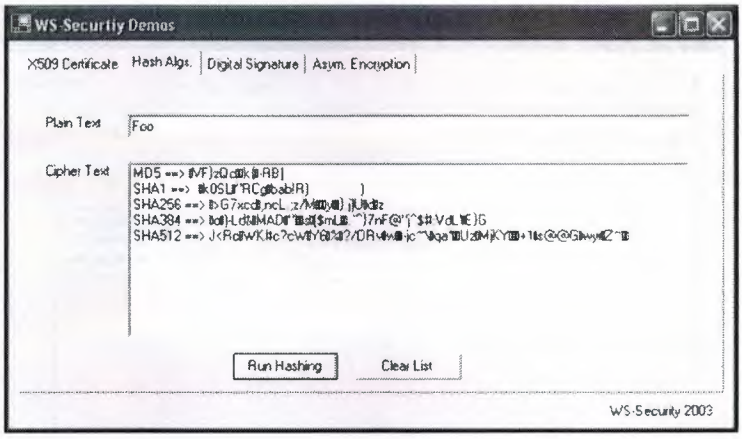


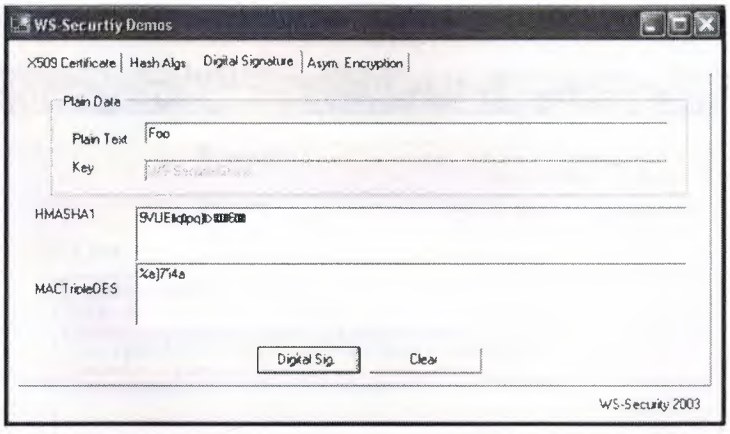
The main window showing the main navigation buttons
To open the "Certificate Inspector" window press the "Certificate" button

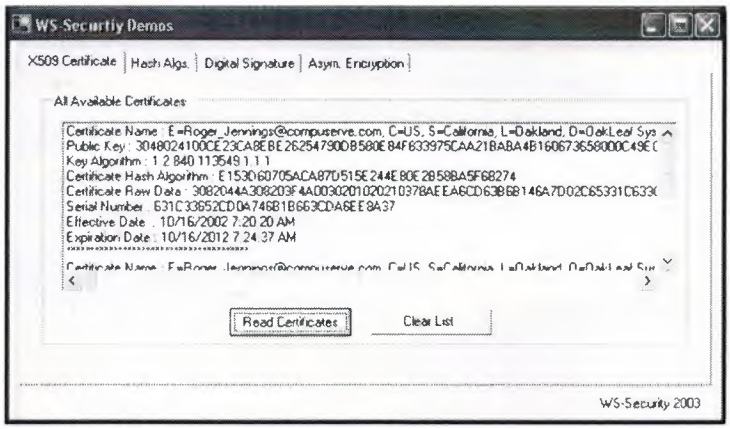


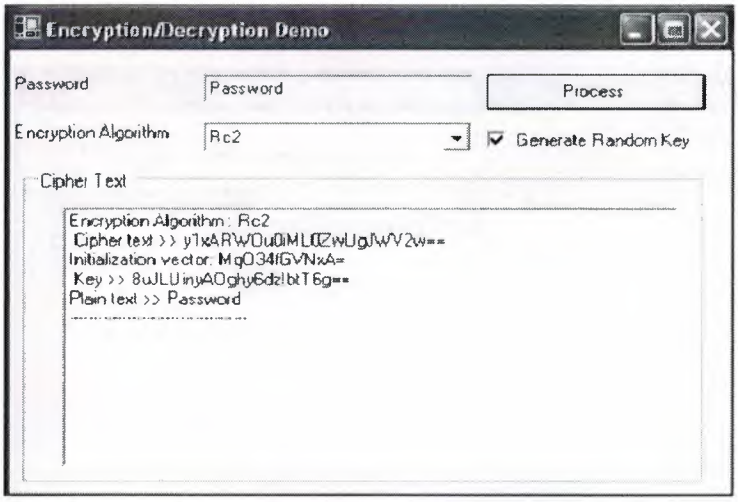
The Certificate window showing the local user store and the certificate status panel
Here you can choose either the local or the current store of the user certificates ,and click on the certificate and selecting the panel showing the properties and description for the selected certificate ,which can be used further for the SOAP encryption window.
Note : press the exit button to return to the main navigation window
To open the "Encryption tester" press the "Encryption" button
Which will show the two programs to test the various encryption hashing and the signature techniques.

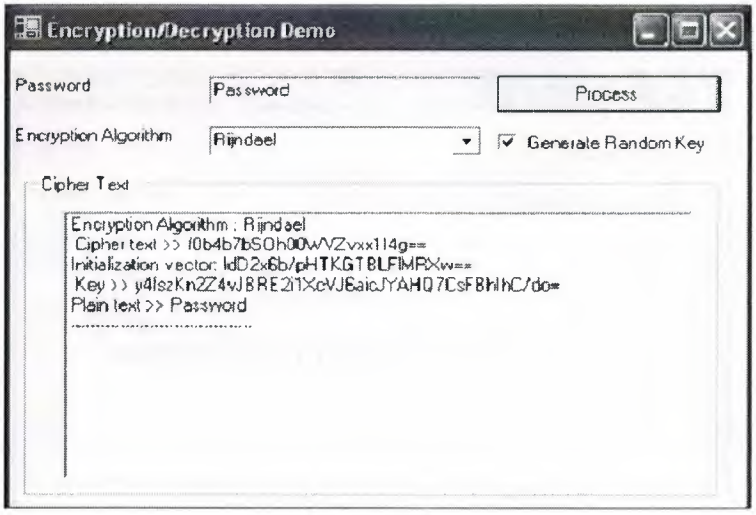

WS-Securtiy Demos
X509 Certificate
Hash Algs.
Digital Signature
Asym. Encryption
Plain Text
Foo
Cipher Text
Public Key
Private Key
Encrypt
Decrypt
Save
Clear All
WS-Security 2003

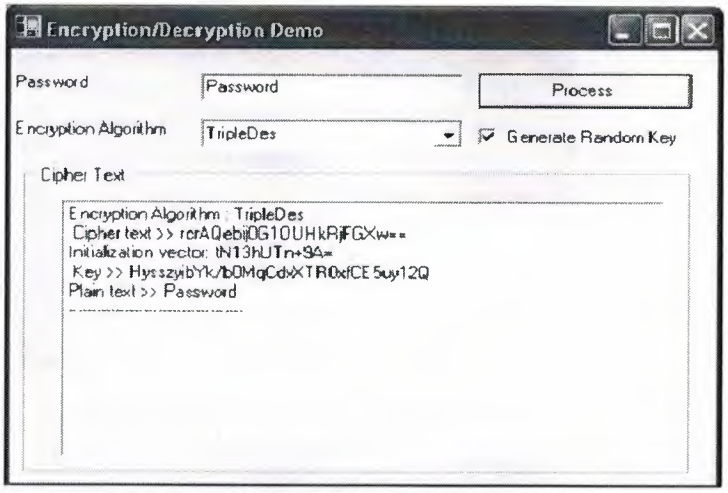

WS-Securtiy Demos
X509 Certificate
Hash Algs.
Digital Signature
Asym. Encryption
Plain Text
Foo
Cipher Text
Run Hashing
Clear List
WS-Security 2003

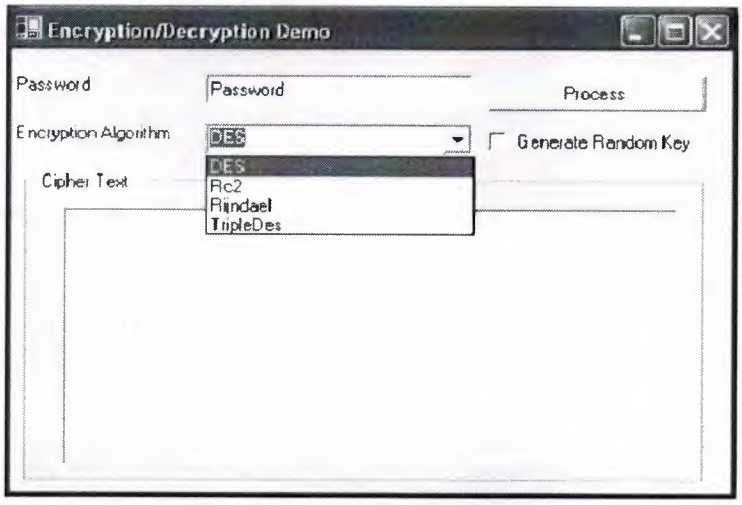

WS-Securtiy Demos
X509 Certificate
Hash Algs.
Digital Signature
Asym. Encryption
Plain Data
Plain Text
Foo
Key
HMASHA1
MACTripleDES
Digital Sig.
Clear
WS-Security 2003

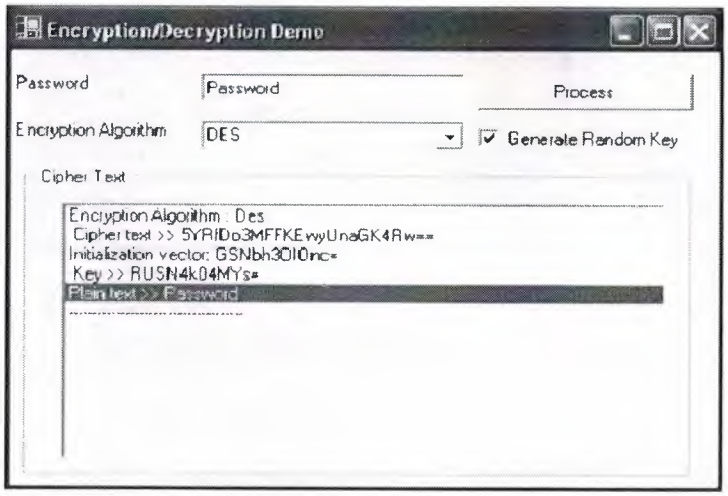

WS-Security Demos
X509 Certificate
Hash Algs.
Digital Signature
Asym. Encryption
All Available Certificates
Certificate Name : E=Roger_Jennings@compuserve.com, C=US, S=California, L=Oakland, O=OakLeaf Sys
Public Key : 30480241 00CE23CA8EBE26254790DB580E84F633975CAA21BABA4B160673658000C49E
Key Algorithm : 1.2.840.113549.1.1.1
Certificate Hash Algorithm : E153D60705ACA87D515E244E80E2658BA5F68274
Certificate Raw Data : 3082044A308203F4A003020102021037BAEEA6CD63BBB146A7D02C65331D633
Serial Number : 631C33652CD0A746B1B663CDA6EE8A37
Effective Date : 10/16/2002 7:20:20 AM
Expiration Date : 10/16/2012 7:24:37 AM
Read Certificates
Clear List
WS-Security 2003


Encryption/Decryption Demo
Password
Password
Process
Encryption Algorithm
Rc2
Generate Random Key
Cipher Text
Encryption Algorithm : Rc2
Cipher text >> y1xARWOu0iMLlQZwUgJWV2w==
Initialization vector: MqO34fGVNxA=
Key >> 8uJLUinyAOghy6dzlbtT6g==
Plain text >> Password

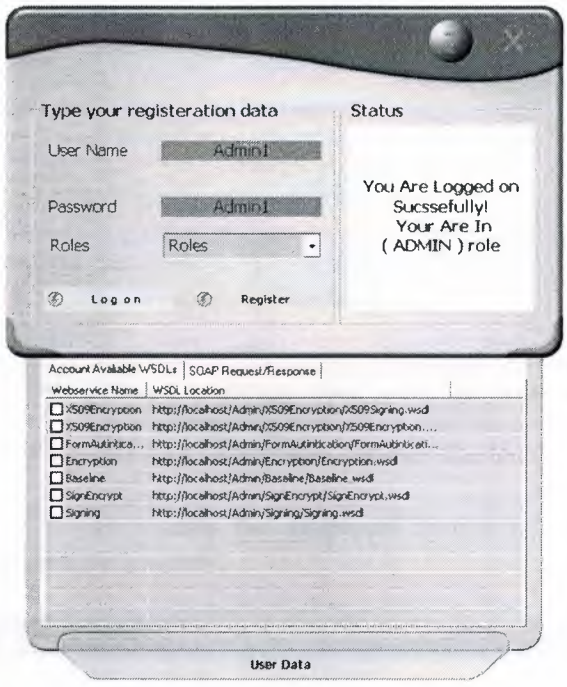

Encryption/Decryption Demo
Password
Password
Process
Encryption Algorithm
Rijndael
Generate Random Key
Cipher Text
Encryption Algorithm : Rijndael
Cipher text >> f0b4b7bSOh00WVZvxx114g==
Initialization vector: ldD2x6b/pHTKGTBLFlMPXw==
Key >> y4fszKn2Z4vJBRE2i1XcVJ6aicJYAHQ7CsFBhlhC/do=
Plain text >> Password



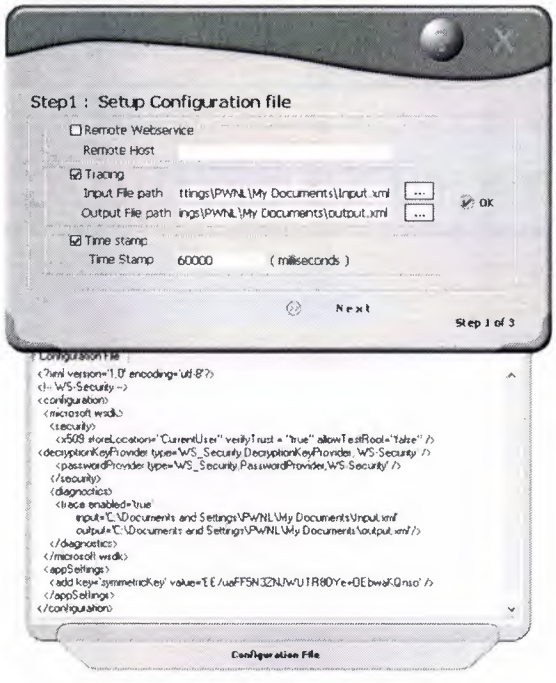




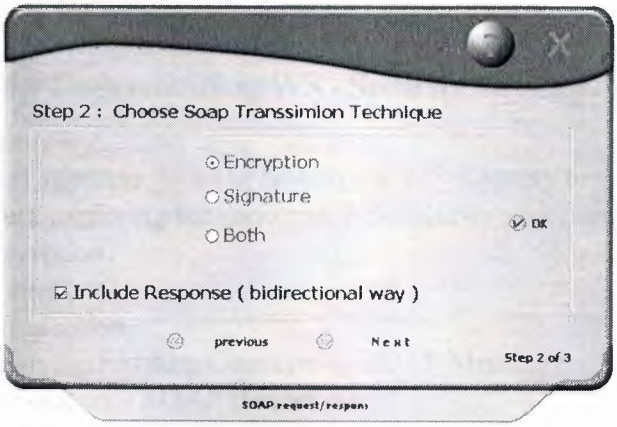
To log on the server that hosts the web services securely, press "log on" button where you will be authenticated and authorized according to your previous registered roles .



the log on widow showing the user logged on as admin and the register and the roles combo-box enabled, where the lower panel shows the available web services and soap request and response for the client calling to access the "SOAP encryption" press the "SOAP Encryption" button to use the user certificates to communicate securely with the web services

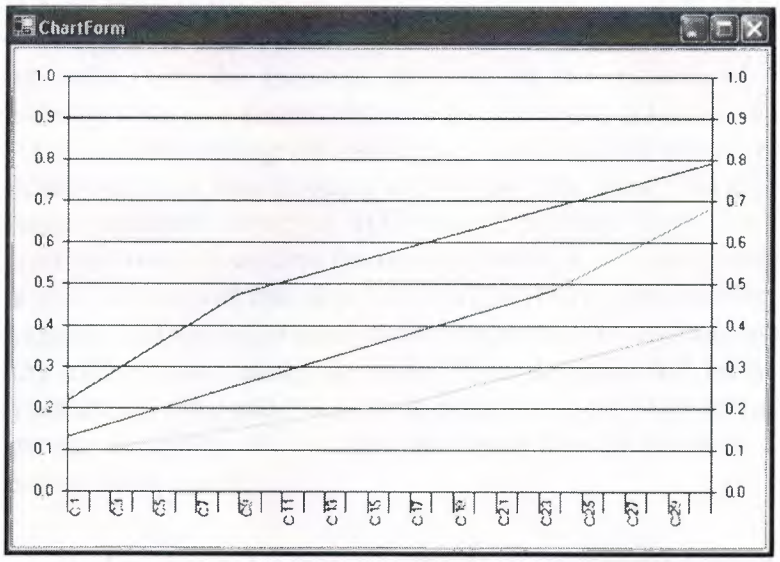


the first step to communicate with a web service ,where the configuration file is written



The second step where user chooses the technique of SOAP transmission

To access the graphical results of calling the web service with the encryption ,signature ,or both



A6 TECHNICAL MANUAL

A6.1 Understanding WS - Security

Summary:
This appendix looks at how to use WS-Security to embed security within the SOAP message itself, exploring the concerns WS-Security addresses: authentication, signatures, and encryption.
Contents

6.1.1 Introduction

Before I explain what WS-Security is, I believe that it is important to understand why WS-Security exists at all. Many people new to Web services see SOAP as a way to exchange messages between two endpoints over HTTP. Over HTTP, one can authenticate the caller, sign the message, and encrypt the contents of the message. This makes the message secure in several dimensions: the caller is known, the receiver of the message can verify that the message did not change in transit, and entities watching the wire traffic cannot figure out what data is being exchanged. For those looking at SOAP messaging to solve bigger problems, however, HTTP-based security simply isn't enough. Many of the bigger problems involve sending the message along a path more complicated than request/response or over a transport that does not involve HTTP. The identity, integrity, and security of the message and the caller need to be preserved over multiple hops. More than one encryption key may be used along the route. Trust domains will be crossed. HTTP and its security mechanisms only address point-to-point security. More complex solutions need end-to-end security baked in. WS-Security addresses how to maintain a secure context over a multi-point message path.

WS-Security addresses security by leveraging existing standards and specifications. This avoids the necessity to define a complete security solution within WS-Security. The industry has solved many of these problems. Kerberos and X.509 address authentication. X.509 also uses existing PKI for key management. XML Encryption and XML Signature describe ways of encrypting and signing the contents of XML messages. XML Canonicalization describes ways of making the XML ready to be signed and encrypted. What WS-Security adds to existing specifications is a framework to embed these mechanisms into a SOAP message. This is done in a transport-neutral fashion.

WS-Security defines a SOAP Header element to carry security-related data. If XML Signature is used, this header can contain the information defined by XML Signature that conveys how the message was signed, the key that was used, and the resulting signature value. Likewise, if an element within the message is encrypted, the encryption information such as that conveyed by XML Encryption can be contained within the WS-Security header. WS-Security does not specify the format of the signature or encryption. Instead, it specifies how one would embed the security information laid out by other specifications within a SOAP message. WS-Security is primarily a specification for an XML-based security metadata container.

What does WS-Security do, beyond leveraging other existing protocols for message authentication, integrity, and privacy? It specifies a mechanism for transferring simple user credentials via the UsernameToken element. Within this header, messages can store information about the caller, how the message was signed, and how the message was encrypted. WS-Security presents an end-to-end solution for Web service security by keeping all security information in the SOAP part of the message. In this appendix, we will take a look at how to use WS-Security and friends to embed security within the SOAP message itself. We will look at the concerns WS-Security addresses:

* Authentication
* Signatures
* Encryption

This triad addresses the main concerns of security and answers the questions:

* Who am I authorizing?
* Was the message modified between hops?
* Did this message come from whom I think it came from?
* How do I hide things I only want certain parties to see?

6.1.2 Applying Existing Concepts to SOAP Messages

WS-Security seeks to move a lot of these concepts about identification and authorization into the world of SOAP messaging. In order to do something meaningful with a SOAP message, that message must contain information that does the following things:

* Identify the entity or entities involved with the message.
* Prove that the entities have the correct group memberships.
* Prove that the entities have the correct set of access rights.
* Prove that the message has not changed.

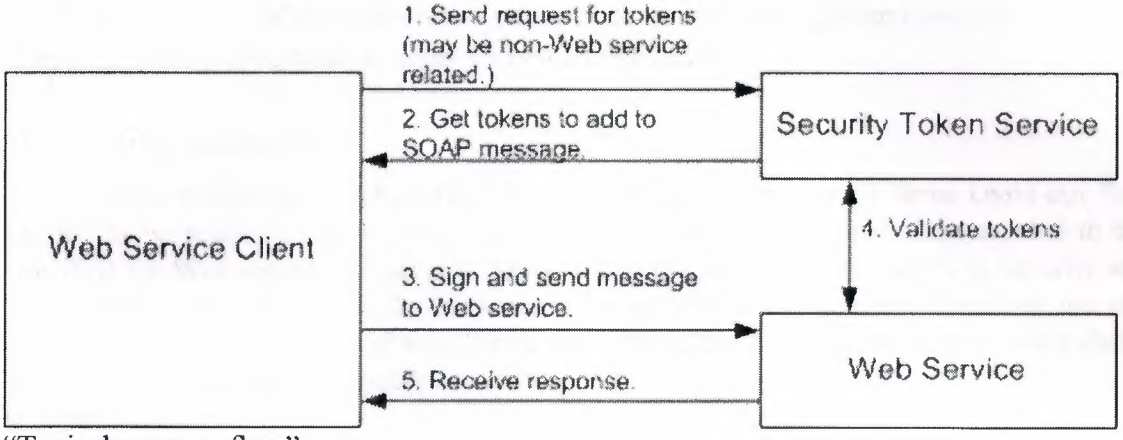
Finally, we also want a mechanism that would hide information from unauthorized parties. In the world of personal identification, I prove who I am with my driver's license or passport. I prove that I have certain rights through membership cards. In my wallet, I have cards that allow me to charge goods and services, check out books from the library, direct medical bills to my insurance provider, and receive discounts at local grocery stores. WS-Security allows me to apply the same concepts to SOAP messages. Using security tokens to identify the caller and assert its rights, a message could convey the following information:

* Caller identity: I am Joe User.
* Group membership: I am a ColdRooster.com developer.
* Rights assertions: Because I am a ColdRooster.com developer, I can create databases and add Web applications to the ColdRooster.com machines.

To create a message that can create a new database on the ColdRooster.com servers using an authentication technology such as Kerberos, the application would have to acquire a number of security tokens. To start with, the application creating the message would need to acquire a security token that identifies it as acting on behalf of Joe User. Joe User provides that token when he logs in via a username/password or by using a smart card. Assuming that the security infrastructure uses Kerberos, the environment Joe is using has a Key Distribution Center that grants Joe a Ticket Granting Ticket (TGT) when he logs in. When Joe decides to create a new database on ColdRooster.com, the environment goes to a Ticket Granting Service and requests a Service Ticket that shows that Joe has the right to create a new database on ColdRooster.com. The environment takes that Service Ticket (ST) and presents

it to the database server at ColdRooster.com. That database server validates the ticket and then allows Joe to create the new process.

WS-Security seeks to encapsulate the security interactions described above within a set of SOAP Headers. WS-Security handles credential management in two ways. It defines a special element, UsernameToken, to pass the username and password if the Web service is using custom authentication. WS-Security also provides a place to provide binary authentication tokens such as Kerberos Tickets and X.509 Certifications: BinarySecurityToken.
Figure 1 depicts what will become a fairly common message flow.



"Typical message flow"

The Security Token service might be Kerberos, PKI, or a username/password validation service. This service may not be Web service-based. Indeed, a Kerberos service ticket granting service might be accessed through the Kerberos protocols using operating system security functions. Once the client gets the tokens it wants to use in the message, the client will embed those tokens within the message. The client should sign the message with a piece of data that only they know. The server will be able to deduce the signature in a number of ways. If the client is using a UsernameToken for authentication, the client should send a hashed password and sign the message using that password. The server will be able to verify that the client sent the message if the signatures it generates for the message match the signatures contained in the message.

When using X.509 certificates, the message can be signed using the private key. The message should contain the certificate in a BinarySecurityToken. When using X.509, anyone who knows the X.509 public key can verify the signature. Finally, when using Kerberos, the message could be signed or encrypted with a session key embedded in the Kerberos ticket. Because the Kerberos ticket will be keyed for the receiver of the token, only the receiver will be able to decrypt the ticket, discover the session key, and verify the signature.

It is critical that SOAP messages be signed or encrypted if authentication is important. Why? It isn't enough that a valid identity token is added to a message. These tokens can be lifted from a valid message and added to messages used by attackers. There needs to be evidence that the identity used in the message created the message. Without using XML Signature and signing the message, you cannot tell that the message has not been changed or that the identity token has not been abused.

At this point, I think you understand what WS-Security does. Let's dig a little deeper and look at how.

WS-Security SOAP Header

Starting in this section and continuing throughout the rest of the article, I will be using XML snippets. So that I don't have to show XML Namespace declarations all over and muddy the snippets, I will use the following XML Namespaces:

Table 1: XML Namespaces

Namespace	Description	Namespace URI
Xs	XML Schema	http://www.w3.org/2001/XMLSchema
Wsse	WS-Security	http://schemas.xmlsoap.org/ws/2002/07/secext
Wsu	Utility elements	http://schemas.xmlsoap.org/ws/2002/07/utility
Soap	SOAP elements	http://schemas.xmlsoap.org/soap/envelope/

The WS-Security specification defines a new SOAP header.

WS-Security Addendum

After evaluating WS-Security for a little while, a number of items came out that needed to be made clearer for security in particular. Also, additional items needed to be specified for Web services in general. The parts of the addendum that apply to security are covered throughout the article. In this section, I want to take a look at two items that are not specific to security: wsu:Id and wsu:Timestamp. The addendum specifies exactly what these two items do and how they should be used.

wsu:Id

The Id attribute uses the XML Schema ID type. This element was added to simplify processing for Web service intermediaries as well as receivers. The value of this attribute must not be duplicated elsewhere in the document. The addendum does not go into more detail about how the element should be used other than as a unique identifier in GXA specifications. The door is left wide open to allow other specifications to restrict the usage of Id.

wsu:Timestamp

A common concern in message-oriented systems relates to the timeliness of data. If the data is too old, it may get thrown out. If two contradicting messages arrive, the related timestamps may be used to decide which message gets executed and which one is ignored. To handle the time-related issues that showed up in WS-Security and that will show up in other GXA specifications, the wsu:Timestamp element, along with a few helper elements, was defined.

The interesting events in a message's life are the time it was created, the time the sender wants the message to expire, and the time that the message was received. By knowing the creation and expiration time, a receiver can decide if the data is new enough for its own use or if the data has become so stale that the message should be discarded. The elements that convey this data are:

* wsu:Created: Contains the time that the message was created.
* wsu:Expires: Set by a sender or intermediary, this identifies when the message expires.
* wsu:Received: Explains when the message was received by a particular intermediary.

All of the above elements may appear independently or as part of a wsu:Timestamp element. Each may contain a wsu:Id attribute to uniquely identify the item. By default, these timestamps express the time as an xs:dateTime type. To allow flexibility for other,

nonstandard time stamps that may be meaningful in other problem domains, each of these items also contains an attribute named ValueType. This attribute does not need to appear if the time is expressed as xs:dateTime.

The wsu:Received element allows for two extra attributes not found on wsu:Created or wsu:Expires. The element can express the URI of the actor it is related to, using the Actor attribute and the amount of delay, in milliseconds, caused by the actor using the Delay attribute.

As I mentioned, you can use the wsu:Received, wsu:Created, and wsu:Expires elements within other structures. For example, it may be common to see the wsu:Created element to indicate when a particular element was added to the message. When indicating more information about a message and using more than one of these elements at a time, the elements can be wrapped inside of a wsu:Timestamp element. Each of three elements may only appear once within a timestamp. The timestamp may used on the message as a whole, in which case it appears as a child of the soap:Header node. For example, a message could indicate it was valid for the next five minutes using the following wsu:Timestamp header.

```
<wsu:Timestamp>
<wsu:Created wsu:Id=
"Id-2af5d5bd-1f0c-4365-b7ff-010629a1256c">
2002-08-19T16:15:31Z
</wsu:Created>
<wsu:Expires wsu:Id=
"Id-4c337c65-8ae7-439f-b4f0-d18d7823e240">
2002-08-19T16:20:31Z
</wsu:Expires>
</wsu:Timestamp>
```

At this point, I believe that you have enough background information to dig into how authentication, signing, and encryption work with WS-Security.

Authentication

WS-Security provides for an infinite number of ways to validate a user. The specification addresses three methods from that infinite number:

* Username/Password
* PKI through X.509 Certificates
* Kerberos

In this section, we will look at how each of these authentication methods works and how that information is encoded into a SOAP message.

Username/Password

One of the most common ways to pass around caller credentials is to use a username and password combination. This is a technique used in HTTP Basic and Digest authentication. As a matter of fact, if you are familiar with how HTTP Digest authentication works, you will feel right at home with this authentication mechanism. To pass user

credentials in this manner, WS-Security has defined the UsernameToken element. Schema for the element is as follows:

```
<xs:element name="UsernameToken">
<xs:complexType>
<xs:sequence>
<xs:element ref="Username"/>
<xs:element ref="Password" minOccurs="0"/>
</xs:sequence>
<xs:attribute name="Id" type="xs:ID"/>
<xs:anyAttribute namespace="##other"/>
</xs:complexType>
</xs:element>
```

This schema fragment references two other types: Username and Password. These two types are essentially strings that may contain extra attributes as needed. Password contains an attribute named Type that indicates how the password is being passed around. A password can be passed as plain text or in digest format. When passing a UsernameToken in a SOAP message, the XML may come across as something like this:

```
<wsse:UsernameToken>
<wsse:Username>scott</wsse:Username>
<wsse:Password Type="wsse:PasswordText">password</wsse:Password>
</wsse:UsernameToken>
```

What you see here is an example of the password being sent as plain text. This particular solution looks pretty easy to break. If you want the password sent in a more secure manner, you can send it is as a digest hash.

```
<wsse:UsernameToken>
<wsse:Username>scott</wsse:Username>
<wsse:Password Type="wsse:PasswordDigest">
KE6QugOpkPyT3Eo0SEgT30W4Keg=</wsse:Password>
<wsse:Nonce>5uW4ABku/m6/S5rnE+L7vg==</wsse:Nonce>
<wsu:Created xmlns:wsu=
"http://schemas.xmlsoap.org/ws/2002/07/utility">
2002-08-19T00:44:02Z
</wsu:Created>
</wsse:UsernameToken>
```

This adds a bit more security because the password is now obscured using a SHA1 hash. The password digest is the concatenation of the nonce plus the creation time plus the password. The nonce is 16 bytes long and is passed along as a base64 encoded value. The way this works is that the client creates the password hash using all of this information plus the password. The receiver verifies the data by getting the plain password and creating the hash again. If the results agree, the password must be correct. This protection does not protect against replay attacks. If you use it, make sure to also include a wsu:Timestamp header with a small enough time window for the created and expired values. Then, sign the wsu:Timestamp elements within the message so that any tampering with the timestamp can be detected. Otherwise, an attacker could use the complete UsernameToken to attack your Web service. To defend against a replay attack, you will also need to include a mechanism

that tracks some unique characteristic of the incoming messages. This mechanism needs to save this characteristic in a cache for at least the timeout period of the message.

X.509 Certificates

Another option to use when authenticating users is to simply send around an X.509 certificate. An X.509 certificate tells you exactly who the user is. Using PKI, you can map the certificate to an existing user in your application. Using the certificate on its own would make for some pretty easy replay attacks. As a result, it's a good idea to force the message sender to also sign the message using their private key. That way, when the message gets decrypted, you'll know it is really the user.

When a message does send along an X.509 certificate, it will pass the public version of the certificate in a WS-Security token named BinarySecurityToken. The certificate itself gets sent along as base64 encoded data. The BinarySecurityToken has the following schema:

```
<xs:element name="BinarySecurityToken">
<xs:complexType>
<xs:simpleContent>
<xs:extension base="xs:string">
<xs:attribute name="Id" type="xs:ID" />
<xs:attribute name="ValueType" type="xs:QName" />
<xs:attribute name="EncodingType" type="xs:QName" />
<xs:anyAttribute namespace="##other"
processContents="strict" />
</xs:extension>
</xs:simpleContent>
</xs:complexType>
</xs:element>
```

At its most basic, this item contains a string, a unique identifier, and some information indicating what type of value is included and how it was encoded. The ValueType may be any of the following values, defined by the ValueTypeEnum in the WS-Security schema document:

* wsse:X509v3: An X.509, version 3 certificate.
* wsse:Kerberosv5TGT: A ticket granting ticket as defined by section 5.3.1 of the Kerberos specification.
*wsse:Kerberosv5ST: A service ticket as defined by section 5.3.1 of the Kerberos specification.

If this information on Kerberos doesn't make any sense to you, I'll explain it a little better in the next section. The EncodingType is another enumeration. If it is set to wsse:Base64Binary or wsse:HexBinary. As you might guess, this value simply indicates which encoding method was used. In a WS-Security header, this element would look something like this when passing an X.509 certificate:

```
<wsse:BinarySecurityToken
ValueType="wsse:X509v3"
EncodingType="wsse:Base64Binary"
Id="SecurityToken-f49bd662-59a0-401a-ab23-1aa12764184f"
```

>MIIHdjCCB...</wsse:BinarySecurityToken>

Remember, when you use an X.509 certificate, you want to do something else as well, such as sign the message. The signature, created using the certificate's private key, proves that the client is the rightful owner of the certificate. Such a message could be replayed. To help mitigate the problems around replays, you would institute a policy that states how old a message can be before it is ignored. The time should travel in a wsu:Timestamp element that is shipped as a SOAP Header within the message.

Kerberos

To use Kerberos, a user presents a set of credentials such as username/password or an X.509 certificate. If everything checks out, the security system grants the user a ticket granting ticket (TGT). The TGT is an opaque piece of data that the user cannot read but must present in order to access other resources. The user will typically present the TGT in order to get a service ticket (ST). The way the system works is as follows:

A client authenticates to a Key Distribution Center (KDC) and is granted a TGT.
The client takes the TGT and uses it to access a Ticket Granting Service (TGS).
The client requests an ST for a particular network resource. The TGS then issues the ST to the client.

The client presents the ST to the network resource and begins accessing the resource with the permissions the ST indicated.

Kerberos is appealing because it contains a mechanism for the client to prove their identity to a service and for the service to prove their identity to the client. The ST is only good for accessing the one network resource and can be used to discover who the caller is. When presenting a Kerberos ticket in a message, the data needs to be blindly copied into the message itself. WS-Security does not explain how a TGT or ST is obtained.

Signing

When a message is signed, it is nearly impossible to tamper with the message. Message signing does not protect the message itself from external parties seeing its contents. Using the signature, the receiver of the SOAP message can know that the signed elements have not changed en route. You should use XML Signature to sign messages whenever possible. Why? XML Signature already handles a number of the tougher items to figure out. WS-Security simply explains how to use signing to prove that the message has not changed. All three of the authentication mechanisms mentioned above provides a way to sign the message so that you can be sure of two things:

The user identified by the X.509 certificate, UsernameToken, or Kerberos ticket signed the message.

The message has not been tampered with since it was signed.
Each of the methods provides a secret that can be used to sign the message. X.509 allows the sender to sign the message using their private key. Kerberos provides a session key that the sender creates and transmits in the ticket. Only the intended receiver of that message can read the ticket, discover the session key, and verify the authenticity of the signature. Finally, the UsernameToken could be signed using the password.

The signature is generated using XML Signature. To sign a simple message such as "Hello World," almost every element in the message needs to be individually signed. wsu:Timestamp presents an interesting problem because an intermediary may add a wsu:Received element to wsu:Timestamp. Every time an element changes, the signature needs to be updated or else things won't look right. Why? If the content changes, the signature should not match. Within a SOAP Message, the signatures and required extra data add quite a bit of extra information.

Encryption

There are times when proving the message sender's identity and showing that the message was not changed is not enough. If you send a credit card number or bank account number in a plain-text but signed manner, an attacker can actually verify that no other attackers have changed the contents of the message. As a result, they have a high confidence that the data is valid. That's no good for you, is it? Instead, you'd like the data encrypted in such a way that only the intended message recipient can read the message. Anyone watching the wire exchange should remain oblivious to the contents of the message. As with signing messages, the WS-Security specification does the right thing and adopts a standard that already exists and does the job of encryption well. That's right, they incorporated XML Encryption.

When you encrypt data, you can choose to use either symmetric or asymmetric encryption. Symmetric encryption requires a shared secret. That is, the key that is used to encrypt the message is the same key that you would use to decrypt the message. Symmetric encryption is good if you control both endpoints and can trust the people and applications that use the keys. Symmetric encryption does have a problem with key distribution. At some point in time, the key needs to be sent to the receiver. How do you do this? Do you ship a disk in the mail or negotiate the key when it is needed? Both options will work.

If you need to send data using easily distributed keys, look to asymmetric encryption. X.509 certificates allow for this. The endpoint receiving the data can publicly post its certificate and allow anyone and everyone to encrypt information using the public key. Only the receiver will know the private key. Because of this, only the receiver can take the encrypted data and turn it back into something readable.

So, what would an encrypted message look like? If you are using Triple-DES, both the sender and receiver would have to exchange the key in some secure manner. The symmetric key can be hidden inside a Kerberos ticket, or exchanged out of band. The WS-Security-based message with embedded XML Encryption information would look something like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
<soap:Header
xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext"
xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">
<wsu:Timestamp>
<wsu:Created
```

```
wsu:Id="Id-3beeb885-16a4-4b65-b14c-0cfe6ad26800"
>2002-08-22T00:26:15Z</wsu:Created>
<wsu:Expires
wsu:Id="Id-10c46143-cb53-4a8e-9e83-ef374e40aa54"
>2002-08-22T00:31:15Z</wsu:Expires>
</wsu:Timestamp>
<wsse:Security soap:mustUnderstand="1" >
<xenc:ReferenceList>
<xenc:DataReference
URI="#EncryptedContent-f6f50b24-3458-41d3-aac4-390f476f2e51" />
</xenc:ReferenceList>
<xenc:ReferenceList>
<xenc:DataReference
URI="#EncryptedContent-666b184a-a388-46cc-a9e3-06583b9d43b6" />
</xenc:ReferenceList>
</wsse:Security>
</soap:Header>
<soap:Body>
<xenc:EncryptedData
Id="EncryptedContent-f6f50b24-3458-41d3-aac4-390f476f2e51"
Type="http://www.w3.org/2001/04/xmlenc#Content">
<xenc:EncryptionMethod Algorithm=
"http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />
<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
<KeyName>Symmetric Key</KeyName>
</KeyInfo>
<xenc:CipherData>
<xenc:CipherValue
>InmSSXQdBV5UiT... Y7RVZQqnPpZYMg==</xenc:CipherValue>
</xenc:CipherData>
</xenc:EncryptedData>
</soap:Body>
</soap:Envelope>
```

The preceding message contains information about what data was encrypted as well as how the encryption was performed. For anyone who does not have access to the key, the cipher text inside the soap:Body cannot be decrypted.

When performing asymmetric encryption, the private key needs to be known to the receiver of the message in order to decrypt that message. Exchanging the public key has to be figured out ahead of time.

Conclusion

WS-Security allows for a SOAP message to identify the caller, sign the message, and encrypt message contents. Whenever possible, existing specifications are reused to reduce the amount of invention required to securely deliver a SOAP message. Because all of the information is delivered within the message itself, the message becomes transport neutral. The message would be secure if it was delivered by HTTP, e-mail.

A7 CODE DOCUMENTATION

A7.1 Securing XML Web Services

XML Web services can be secured today, but limitations exist when it comes to building scalable distributed applications based on XML Web services. Specifically, it is difficult to build scalable applications that cross security domains. Today, you can secure XML Web services by having the message sent over a secure transport, such as Secure Sockets Layer (SSL), but that only works when the communication is point-to-point. That is, if the SOAP message must be routed to one or more intermediaries before reaching the ultimate receiver and the entire route uses SSL, then the ultimate receiver still has to communicate with the sender to authenticate the sender of the SOAP message. That scenario is difficult to scale.

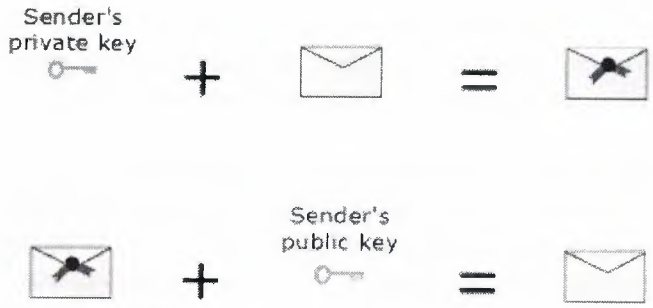
One of the ways the Web Service Enhancement (WSE) helps to build scalable distributed applications is by providing an efficient and scalable mechanism to secure XML Web services. It uses the mechanism defined in the WS-Security specification to place security credentials in the SOAP message itself. This is done by having a client obtain security credentials from a source that is trusted by both the sender and receiver. When a client makes a SOAP request, those security credentials, which are generically known as security tokens, are then placed in the SOAP message. When the Web server receives the SOAP request, it does not need to make another network request back to either the client's computer or a trusted third party to verify the integrity of the security tokens. This is possible if the security credentials are from a trusted source. Instead, the WSE cryptographically verifies that the credentials are authentic before passing execution to the XML Web service. By not having to go back to the source of the credentials, at least one network request is saved, further improving application scalability.

Digitally signing and/or encrypting the SOAP message can further secure XML Web services. Digital signing of a SOAP message secures an XML Web service by enabling a SOAP message recipient, such as an XML Web service, to cryptographically verify that a SOAP message has not been altered since it was signed. Encryption of a SOAP message helps secure an XML Web service by making it highly likely that only the intended XML
The Protocol provides three features that contribute to the secure transmission of SOAP messages:

Security credentials enable the scalable securing of XML Web services over the complete route a SOAP message takes. This is different from a secure transport, such as SSL, which only secures from one point to another. For details on how to use the this protocol to add security credentials, see Adding Security Credentials to a SOAP Message.

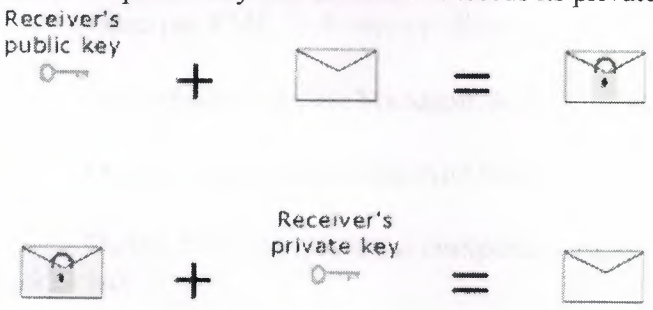
Digital signing allows a SOAP message recipient to cryptographically verify that a SOAP message has not been altered since it was signed. For details on how to digitally sign SOAP messages using the protocol, see Digitally Signing a SOAP Message.

The following graphic illustrates the keys needed by the sender and receiver to both sign a SOAP message and to verify the signature. To sign the SOAP message, the sender needs its private key and for a receiver to verify the signature within a SOAP message it needs the sender's public key.

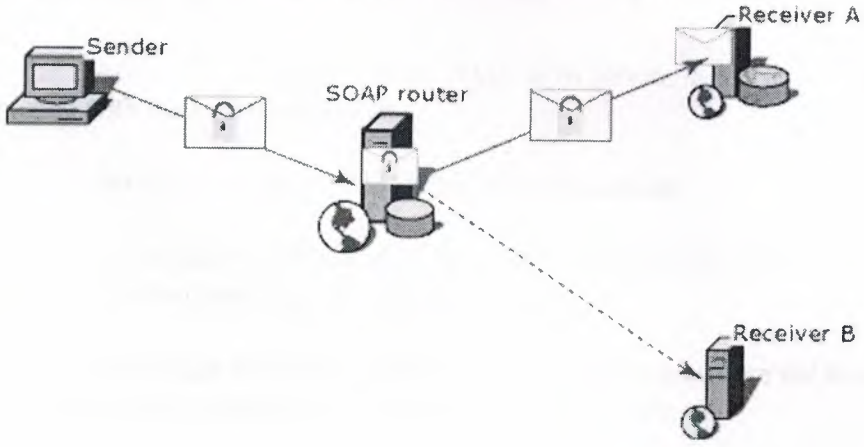


Encryption makes it highly likely that only the intended recipient can read the contents of a message. Encrypting a SOAP message creates a cryptographic secret that is shared with the intended recipient of the message. For details on how to use the protocol to encrypt and decrypt SOAP messages, see Encrypting a SOAP Message.

The following graphic illustrates the keys needed by the sender and receiver to both encrypt and decrypt a SOAP message. To encrypt a SOAP message, the sender needs the receiver's public key and the receiver needs its private key to decrypt the SOAP message..



The following graphic illustrates that only the intended recipient of an encrypted SOAP message can access the encrypted portions of the SOAP message, as it is the only one with the private key that decrypts the SOAP message.



A. Adding Security Credentials to a SOAP Message.

By Using Web Services Enhancements for Microsoft .NET (WSE) provide a mechanism for creating one or more security credentials that can be added to a SOAP message. Without using the WSE, you can set the Credentials property of the proxy class, which enables the ultimate destination to authenticate the client application. However, if the SOAP message is routed through intermediaries before the ultimate destination, the client credentials might not flow to the ultimate destination, and thus the client is not authenticated.

If you add the security credentials to the SOAP message using the WSE, rather than at the transport layer using the proxy's Credentials property, the credentials can be routed through one or more intermediaries and still be validated at the ultimate destination.

The following procedure details how to add one or more security credentials to a SOAP message. For the following procedure to be successful, a computer must be set up to accept SOAP messages embedded with security credentials. The following table outlines the procedures for setting up a computer running our protocol specification, to accept the security credentials.

A7.2 To Add A Security Token To A SOAP Message

The following procedure provides code that adds a UsernameToken to a SOAP message; however, the same principles apply to other types of security tokens.

Open the XML Web service client project in Visual Studio .NET.

Add references to the Microsoft.Web.Services and System.Web.Services assemblies.

On the Project menu, click Add Reference.

On the .NET tab, select the component named Microsoft.Web.Services.dll, and then click Select.

On the .NET tab, select the component named System.Web.Services.dll, and then click Select.

Click OK.

Add a Web Reference to the XML Web service that is to receive the SOAP message signed with the UsernameToken.

On the Project menu, click Add Web Reference.

In the Add Web Reference dialog box, type the URL for the XML Web service in the Address box, and then click the arrow icon.

Verify that the items in the Available references box are the items you want to reference in your project, and then click Add Reference.

Edit the proxy class to derive from WebServicesClientProtocol.
In Solution Explorer, right-click the Reference.cs file for the Web reference just added, and then click View Code.

Note: If the Reference.cs file containing the proxy class is not visible, click the Show All Files icon in the Solution Explorer toolbar, and then expand the Reference.map node.

Change the base class of the proxy class to Microsoft.Web.Services.WebServicesClientProtocol.

This modifies the proxy class to emit WS-Routing and WS-Security SOAP headers when communicating with the XML Web service. The following code example modifies a proxy class named Service1 to derive from
Microsoft.Web.Services.WebServicesClientProtocol.
public class Service1 : Microsoft.Web.Services.WebServicesClientProtocol {

Note: If you select Update Web Reference in Visual Studio .NET, the proxy class is regenerated, the base class is reset to SoapHttpClientProtocol, and you must change the class the proxy class derives from again.

Add the following using directives to the top of the file containing the XML Web service client code:
using Microsoft.Web.Services;
using
Microsoft.Web.Services.Security;

Edit the method that communicates with the XML Web service to add the security credentials to the SOAP message.

Create a new instance of a class deriving from SecurityToken.
The following code example creates an instance of the UsernameToken class.
UsernameToken userToken =
new UsernameToken(userName,password,
PasswordOption.SendNone)
Note It is recommended that a UsernameToken be sent with the
PasswordOption set to SendNone or SendHashed when the underlying transport protocol is not secure.

Get the SoapContext for the SOAP request that is being made to the XML Web service.
Service1 is the name of a proxy class for the XML Web service in the following code example.
Service1 serviceProxy = new Service1();
SoapContext requestContext = serviceProxy.RequestSoapContext;

Add the SecurityToken to the WS-Security SOAP header.
requestContext.security.Tokens.Add(userToken);
Create a new instance of Signature using the custom binary security token just added to the WS-Security SOAP header.

Signature sig = new Signature(customToken);

Add the digital signature to the WS-Security SOAP header.
requestContext.Security.Elements.Add(sig);

Specify the time-to-live (TTL) for the SOAP message to diminish the chance of someone intercepting the message and replaying it. The following code sample sets the TTL to one minute.
requestContext.Timestamp.Ttl = 60000;

Call the XML Web service.

```
serviceProxy.sayHello();
```

The following code example adds a UsernameToken to a SOAP message.

```
UsernameToken userToken;
userToken = new UsernameToken(userName, userName,
PasswordOption.SendNone);
try
{
// Creates an instance of the XML Web service proxy.
AddNumbers serviceProxy = new AddNumbers();
// Gets the SoapContext associated with the SOAP request.
SoapContext requestContext = serviceProxy.RequestSoapContext;
// Sets the TTL to one minute.
requestContext.Timestamp.Ttl = 60000;

// Adds the token to the SOAP header.
requestContext.Security.Tokens.Add(userToken);

// Calls the XML Web service.
int sum = serviceProxy.AddInt(a, b);
}
catch (System.Web.Services.Protocols.SoapException se)
{
MessageBox.Show(se.ToString());
}
catch (Exception ex)
{
Error("Exception caught while invoking an XML Web service.", ex);
return;
}
```

Digitally Signing a SOAP Message

The protocol Specification provides a mechanism for digitally signing SOAP messages using a Username Token and X.509 certificates .When a SOAP message is signed with a Username Token, the client provides a user name and the password or password equivalent.

Although the XML digital signature does offer a mechanism for verifying the message has not been altered since it was signed, it does not encrypt the SOAP message; the message is still plain text in XML format. Once the SOAP message is digitally signed, it can be encrypted, however

Signing a SOAP Message Using an X.509 Certificate

Web Services Enhancements (WSE) supports signing SOAP messages using X.509 certificates that meet the following criteria:

The certificate must not be expired when the SOAP message is received.

The certificate must support digital signatures.

The issuer of the certificate must be a trusted root.

The following steps detail how an XML Web service client signs a SOAP request using an X.509 certificate. An XML Web service can also sign the SOAP response using similar steps.
To sign a SOAP message using an X.509 certificate

Obtain the X.509 certificate.

The client X.509 certificate can be obtained in one of the following ways:

Purchase a certificate from a certificate authority, such as VeriSign, Inc.

Set up your own certificate service and have a certificate authority sign the certificates. Windows 2000 Server, Windows 2000 Advanced Server, and Windows 2000 Datacenter Server all include Certificate Services that support public key infrastructure (PKI).

Set up your own certificate service and not have the certificates signed.

Whichever approach you take, the recipient of the SOAP request containing the X.509 certificate must trust the X.509 certificate. This means that the X.509 certificate or an issuer in the certificate chain is in the Trusted People certificate store and that the X.509 certificate is not in the Untrusted Certificates store.

When the certificate is requested, choose the option to install the certificate into the current user certificate store on the local computer.

For more details, see Managing X.509 Certificates.

Open the XML Web service client project in Visual Studio.

Add references to the Micrsosoft.Web.Services and System.Web.Services assemblies.

In Solution Explorer, right-click References and click Add Reference.

On the .NET tab, select the component named Microsoft.Web.Services.dll.

Click Select.

On the .NET tab, select the component named System.Web.Services.dll.

Click Select.

Click OK.

Add a Web Reference to the XML Web service that is to receive the SOAP message signed with the X.509 certificate.

On the Project menu, choose Add Web Reference.

In the Add Web Reference dialog box, type the URL for the XML Web service in the Address text box, and then click the Arrow Icon.

Verify that the items in the Available References box are the items you want to reference in your project, and then choose Add Reference.

Edit the proxy class to derive from the WebServicesClientProtocol class.

In Solution Explorer, right-click the Reference.cs file for the Web reference just added and then click View Code.

Note: If the Reference.cs file containing the proxy class is not visible, click Show All Files in the Solution Explorer toolbar, and then expand the Reference.map node.

Change the base class of the proxy class to WebServicesClientProtocol.

This modifies the proxy class to emit WS-Routing and WS-Security SOAP headers when communicating with the XML Web service. The following code example modifies a proxy class named Service1 to derive from Microsoft.Web.Services.WebServicesClientProtocol.

```
public class Service1 : Microsoft.Web.Services.WebServicesClientProtocol {
```

Note: If you select Update Web Reference in Visual Studio .NET, the proxy class is regenerated, the base class is reset to SoapHttpClientProtocol, and you must change the class the proxy class derives from again.

Add the following using directives to the top of the file that communicates with the XML Web service.

In Solution Explorer, right-click the file containing the client code, and then click View Code.

At the top of the file, add the following using directives:

```
using Microsoft.Web.Services;
using Microsoft.Web.Services.Security;
using Microsoft.Web.Services.Security.X509;
using System.Security.Cryptography;
```

Add code to get an X.509 certificate.

Open the certificate store containing the certificate that will be used to sign the SOAP message.

The following code example opens the certificate store for the currently logged-in user.

```
X509CertificateStore store;
store = X509CertificateStore.CurrentUserStore(
X509CertificateStore.MyStore);
bool open = store.OpenRead();
```

Select a certificate from the certificate store.

The certificate can be programmatically chosen by iterating over the X509CertificateStore.Certificates collection or by invoking one of the "Find" methods. X509CertificateStore supports the following methods to find a certificate:

Method	Description
FindCertificateByHash	Finds an X509Certificate in the store using the certificate's (SHA-1) hash value.
FindCertificateByKeyIdentifier	Finds an X509Certificate in the store using the certificate's authority key identifier.
FindCertificateBySubjectName	Finds an X509Certificate in the store using the certificate's name value.
FindCertificateBySubjectString	Finds an X509Certificate in the store using the certificate's name value. This search uses substring matching.

The following code example retrieves the certificate from the certificate store using the hash for the certificate. The hash is the certificate's thumbprint. To obtain a certificate's thumbprint, open the Certificates snap-in using Microsoft Management Console (MMC) and select the Details tab.

```
byte[] certHash = {0x98, 0xec, 0x08, 0x4b, 0xa5 ,0x7a, 0x6c, 0x2f,
0x39 ,0x26 ,0xb3 ,0x0a ,0x58, 0xbf ,0x65 ,0x25, 0x61, 0xc5,
0x64 ,0x59};
X509CertificateCollection certs =
store.FindCertificateByHash(certHash);

Microsoft.Web.Services.Security.X509.X509Certificate cert =
((Microsoft.Web.Services.Security.X509.X509Certificate) certs[0]);
```

Verify that the certificate can be used for signing.

The certificate must support digital signatures and a private key must be available.

The following code example determines whether the certificate supports digital signatures and whether its private key is accessible.

```
if (cert.SupportsDigitalSignature &&
(cert.Key != null))
{
```

The following code example defines a GetSecurityToken method that displays the certificate store to the current user and allows the user to select an X.509 certificate.

```
public X509SecurityToken GetSecurityToken() {
X509SecurityToken securityToken = null;
X509CertificateStore store =
X509CertificateStore.CurrentUserStore(
X509CertificateStore.MyStore);
bool open = store.OpenRead();

try
{
byte[] certHash = {0x98, 0xec, 0x08, 0x4b, 0xa5, 0x7a,
0x6c, 0x2f, 0x39, 0x26, 0xb3, 0x0a,
0x58, 0xbf, 0x65, 0x25, 0x61, 0xc5,
0x64, 0x59};
X509CertificateCollection certs =
store.FindCertificateByHash(certHash);

Microsoft.Web.Services.Security.X509.X509Certificate cert =
((Microsoft.Web.Services.Security.X509.X509Certificate) certs[0]);

if (cert == null)
{
MessageBox.Show(
"You chose not to select an X.509 " +
"certificate for signing your messages.");
securityToken = null;
}
else if (!cert.SupportsDigitalSignature ||
(cert.Key == null))
{
MessageBox.Show(
"The certificate must support digital " +
"signatures and have a private key available.");
securityToken = null;
}
else
{
securityToken = new X509SecurityToken(cert);
}
}
finally
{
if (store != null)
store.Close();
```

```
}
return securityToken;
}
```

Edit the method that communicates with the XML Web service to get the X.509 certificate and specify that the SOAP request must be signed by the X.509 certificate. For details about signing portions of the SOAP message other than the defaults, see Specifying the Parts of a SOAP Message That Are Signed or Encrypted.

Call the method defined in the previous step to get the client's X.509 certificate, which has a private key, to sign the SOAP message.

```
X509SecurityToken signatureToken = GetSecurityToken(true);
```

Get the SoapContext for the SOAP request that is being made to the XML Web service. Service1 is the name of the proxy class to the XML Web service in the following code example.

```
Service1 svc = new Service1();
SoapContext requestContext = svc.RequestSoapContext;
```

Add the client's X.509 certificate to the WS-Security SOAP header.

```
requestContext.Security.Tokens.Add(signatureToken);
```

Create a new instance of the Signature class using the X.509 certificate just added to the WS-Security SOAP header.

```
Signature sig = new Signature(signatureToken);
```

Add the digital signature to the WS-Security SOAP header.

```
RequestContext.Security.Elements.Add(sig);
```

Specify the time-to-live (TTL) for the SOAP message to diminish the chance of someone intercepting the message and replaying it. The following code sample sets the TTL to 1 minute.

```
requestContext.Timestamp.Ttl = 60000;
```

Call the XML Web service.

```
svc.sayHello();
```

The following code example calls the GetSecurityToken method to get the X.509 certificate and then adds that token to the RequestSoapContext associated with the XML Web service proxy.

```
try
{
X509SecurityToken signatureToken = GetSecurityToken();
if (signatureToken == null)
{
return;
}
Proxy.Service1 svc = new Proxy.Service1();
SoapContext requestContext = svc.RequestSoapContext;

// Sets the TTL to one minute.
requestContext.Timestamp.Ttl = 60000;

// Adds the security token.
requestContext.Security.Tokens.Add(signatureToken);
// Specifies the security token to sign the message with.
requestContext.Security.Elements.Add(new
Signature(signatureToken));
```

```
MessageBox.Show( svc.sayHello() );
}
catch (Exception ex)
{
MessageBox.Show( ex.ToString() );
}
```

Sign the soap message by user name

To sign a SOAP message using a UsernameToken

Open the XML Web service client project in Visual Studio.

Add references to the Microsoft.Web.Services and System.Web.Services assemblies.
In Solution Explorer, right-click References and then click Add Reference.
On the .NET tab, select the component named Microsoft.Web.Services.dll.

Click Select.

On the .NET tab, select the component named System.Web.Services.dll.

Click Select.

Click OK.

Add a Web reference to the XML Web service that is to receive the SOAP message signed with a user name and password.
On the Project menu, choose Add Web Reference.
In the Add Web Reference dialog box, type the URL for the XML Web service in the Address text box, and then click the Arrow icon.

Verify that the items in the Available References box are the items you want to reference in your project, and then choose Add Reference.

Edit the proxy class to derive from WebServicesClientProtocol.
In Solution Explorer, right-click the Reference.cs file for the Web reference just added, and then click View Code.

Note: If the Reference.cs file containing the proxy class is not visible, click Show All Files in the Solution Explorer toolbar, and then expand the Reference.map node.

Change the base class of the proxy class to Microsoft.Web.Services.WebServicesClientProtocol.

This modifies the proxy class to emit WS-Routing and WS-Security SOAP headers when communicating with the XML Web service. The following code example modifies a proxy class named Service1 to derive from Microsoft.Web.Services.WebServicesClientProtocol.

```
public class Service1 :

Microsoft.Web.Services.WebServicesClientProtocol {
```

Note: If you select Update Web Reference in Visual Studio .NET, the proxy class is regenerated, the base class is reset to SoapHttpClientProtocol, and you must change the class the proxy class derives from again.

Add the following using directives to the top of the file containing the XML Web service client code:

```
using Microsoft.Web.Services;
using Microsoft.Web.Services.Security;
```

Edit the method that communicates with the XML Web service to sign the SOAP message using the UsernameToken. For details about signing portions of the SOAP message other than the defaults, see Specifying the Parts of a SOAP Message That Are Signed or Encrypted.

Create a new instance of UsernameToken, specifying the user name, password, and how the password is sent in the SOAP message.

The XML Web service provides a class that implements the IPasswordProvider interface, which includes the GetPassword method. The GetPassword method is given the user name in the SOAP message, but no form of the user name's password. The GetPassword method is expected to return the user name's password or password equivalent. The WSE calls the GetPassword method and then compares the password or password equivalent to the one received in the SOAP message. The following table details the options for passing the password in the SOAP message.

PasswordOption member	Description
SendNone	The password is never sent in any form in the SOAP message, but the WSE does use the password to sign the SOAP message. A recipient would then need to provide a password to the WSE during the signature verification stage.
SendHashed	The SHA-1 hash of the password is sent in the SOAP message. This is the best way to help protect the password. When a SOAP message is received with a UserNameToken, the WSE calls the GetPassword method of the class implementing the IPasswordProvider interface that is registered in the configuration file. The GetPassword method returns a password or password equivalent, which the WSE creates a SHA-1 hash from. That SHA-1 hash is compared to the one in the SOAP message and if they are identical, the hashed password is deemed valid.
SendPlainText	The password is always sent in plain text in the SOAP message. This option is only recommended when SSL or a similar protocol is used to connect to the XML Web service; otherwise, the password could be intercepted. The WSE running on the recipient's computer compares the plain text password or password equivalent in the SOAP message to the one returned from the GetPassword method of the class implementing the IPasswordProvider interface. If they are identical, the password is deemed valid.

The following code example does not send the password in the SOAP request.

```
UsernameToken userToken = new UsernameToken(userName,password,
                    PasswordOption.SendNone)
```

Get the SoapContext for the SOAP request that is being made to the XML Web service. Service1 is the name of the proxy class for the XML Web service in the following code example.

```
Service1 serviceProxy = new Service1();
SoapContext requestContext = serviceProxy.RequestSoapContext;
```

Add the UsernameToken to the WS-Security SOAP header.

```
requestContext.Security.Tokens.Add(userToken);
```

Create a new instance of the Signature class using the UsernameToken just added to the WS-Security SOAP header.

```
Signature sig = new Signature(userToken);
```

Add the digital signature to the WS-Security SOAP header.

```
requestContext.Security.Elements.Add(sig);
```

Specify the time-to-live (TTL) for the SOAP message to diminish the chance of someone intercepting the message and replaying it. The following code sample sets the TTL to 1 minute.

```
requestContext.Timestamp.Ttl = 60000;
```

Call the XML Web service.

```
serviceProxy.sayHello();
```

The following code example signs a SOAP message using a UsernameToken.

```
UsernameToken userToken;
userToken = new UsernameToken(userName, userName,
PasswordOption.SendHashed);
try
{
// Creates the Web service proxy.
AddNumbers serviceProxy = new AddNumbers();
// Gets the SoapContext associated with the SOAP request.
SoapContext requestContext = serviceProxy.RequestSoapContext;
// Sets the TTL to 1 minute.
requestContext.Timestamp.Ttl = 60000;

// Adds the token to the SOAP header.
requestContext.Security.Tokens.Add(userToken);
// Signs the SOAP message using the UsernameToken.
RequestContext.Security.Elements.Add(new Signature(userToken));

// Calls the XML Web service.
int sum = serviceProxy.AddInt(a, b);
}
catch (System.Web.Services.Protocols.SoapException se)
{
MessageBox.Show(se.ToString());
}
catch (Exception ex)
{
Error("Exception caught while invoking an XML Web service.", ex);
return;
}
```

Encrypting a SOAP Message.

The protocol Specification using (WSE) enables .NET Framework clients and XML Web services created using ASP.NET to encrypt and decrypt SOAP messages used to communicate with XML Web services. Encrypting and decrypting SOAP messages can be key to securing a Web application, because SOAP messages are by default plain text and thus can be read by any recipient. An encrypted SOAP message is cryptographically encoded, so that only the owner of a private key or a symmetric key can read the contents of the message.

The protocol specification supports both asymmetric and symmetric encryption. Asymmetric encryption allows an XML Web service client to encrypt the message using the public key of an X.509 certificate, such that only the owner of the private key of the X.509 certificate can decrypt the SOAP message. Symmetric encryption requires that an XML Web service and client share a secret key outside the SOAP message communication. Then, a client encrypts SOAP messages using that shared key and an XML Web service decrypts the SOAP messages using the same secret key.

Encrypting the soap message with X509 Certificate

To sign a SOAP message using a UsernameToken

Open the XML Web service client project in Visual Studio.

Add references to the Microsoft.Web.Services and System.Web.Services assemblies.

In Solution Explorer, right-click References and then click Add Reference.

On the .NET tab, select the component named Microsoft.Web.Services.dll.

Click Select.

On the .NET tab, select the component named System.Web.Services.dll.

Click Select.

Click OK.

Add a Web reference to the XML Web service that is to receive the SOAP message signed with a user name and password.

On the Project menu, choose Add Web Reference.

In the Add Web Reference dialog box, type the URL for the XML Web service in the Address text box, and then click the Arrow icon.

Verify that the items in the Available References box are the items you want to reference in your project, and then choose Add Reference.

Edit the proxy class to derive from WebServicesClientProtocol.

In Solution Explorer, right-click the Reference.cs file for the Web reference just added, and then click View Code.

Note: If the Reference.cs file containing the proxy class is not visible, click Show All Files in the Solution Explorer toolbar, and then expand the Reference.map node.

Change the base class of the proxy class to Microsoft.Web.Services.WebServicesClientProtocol.

This modifies the proxy class to emit WS-Routing and WS-Security SOAP headers when communicating with the XML Web service. The following code example modifies a proxy class named Service1 to derive from Microsoft.Web.Services.WebServicesClientProtocol.

```
public class Service1 :

Microsoft.Web.Services.WebServicesClientProtocol {
```

Note If you select Update Web Reference in Visual Studio .NET, the proxy class is regenerated, the base class is reset to SoapHttpClientProtocol, and you must change the class the proxy class derives from again.

Add the following using directives to the top of the file containing the XML Web service client code:

```
using Microsoft.Web.Services;
using Microsoft.Web.Services.Security;
```

Edit the method that communicates with the XML Web service to sign the SOAP message using the UsernameToken.

Create a new instance of UsernameToken, specifying the user name, password, and how the password is sent in the SOAP message.

The XML Web service provides a class that implements the IPasswordProvider interface, which includes the GetPassword method. The GetPassword method is given the user name in the SOAP message, but no form of the user name's password. The GetPassword method is expected to return the user name's password or password equivalent. The WSE calls the GetPassword method and then compares the password or password equivalent to the one received in the SOAP message. The following table details the options for passing the password in the SOAP message.

PasswordOption member	Description
SendNone	The password is never sent in any form in the SOAP message, but the WSE does use the password to sign the SOAP message. A recipient would then need to provide a password to the WSE during the signature verification stage.
SendHashed	The SHA-1 hash of the password is sent in the SOAP message. This is the best way to help protect the password. When a SOAP message is received with a UserNameToken, the WSE calls the GetPassword method of the class implementing the IPasswordProvider interface that is registered in the configuration file. The GetPassword method returns a password or password equivalent, which the WSE creates a SHA-1 hash from. That SHA-1 hash is compared to the one in the SOAP message and if they are identical, the hashed password is deemed valid.
SendPlainText	The password is always sent in plain text in the SOAP message. This option is only recommended when SSL or a similar protocol is used to connect to the XML Web service; otherwise, the password could be intercepted. The WSE running on the recipient's computer compares the plain text password or password equivalent in the SOAP message to the one returned from the GetPassword method of the class implementing the IPasswordProvider interface. If they are identical, the password is deemed valid.

The following code example does not send the password in the SOAP request.

```
UsernameToken userToken = new UsernameToken(userName,password,
                          PasswordOption.SendNone)
```

Get the SoapContext for the SOAP request that is being made to the XML Web service. Service1 is the name of the proxy class for the XML Web service in the following code example.

```
Service1 serviceProxy = new Service1();
SoapContext requestContext = serviceProxy.RequestSoapContext;
```

Add the UsernameToken to the WS-Security SOAP header.

```
requestContext.Security.Tokens.Add(userToken);
```

Create a new instance of the Signature class using the UsernameToken just added to the WS-Security SOAP header.

```
Signature sig = new Signature(userToken);
```

Add the digital signature to the WS-Security SOAP header.

```
requestContext.Security.Elements.Add(sig);
```

Specify the time-to-live (TTL) for the SOAP message to diminish the chance of someone intercepting the message and replaying it. The following code sample sets the TTL to 1 minute.

```
requestContext.Timestamp.Ttl = 60000;
```

Call the XML Web service.

```
serviceProxy.sayHello();
```

The following code example signs a SOAP message using a UsernameToken.

```
UsernameToken userToken;
userToken = new UsernameToken(userName, userName,
PasswordOption.SendHashed);
try
{
// Creates the Web service proxy.
AddNumbers serviceProxy = new AddNumbers();
// Gets the SoapContext associated with the SOAP request.
SoapContext requestContext = serviceProxy.RequestSoapContext;
// Sets the TTL to 1 minute.
requestContext.Timestamp.Ttl = 60000;

// Adds the token to the SOAP header.
requestContext.Security.Tokens.Add(userToken);
// Signs the SOAP message using the UsernameToken.
RequestContext.Security.Elements.Add(new Signature(userToken));

// Calls the XML Web service.
int sum = serviceProxy.AddInt(a, b);
}
catch (System.Web.Services.Protocols.SoapException se)
{
MessageBox.Show(se.ToString());
}
catch (Exception ex)
{
Error("Exception caught while invoking an XML Web service.", ex);
return;}
```

Encryptin the soap mesaaeg with the shared secret key

Open the XML Web service client project in Visual Studio.

Add references to the Microsoft.Web.Services and System.Web.Services assemblies.
On the Project menu, click Add Reference.
On the .NET tab, select the component named Microsoft.Web.Services.dll and then click Select.
On the .NET tab, select the component named System.Web.Services.dll and then click Select.
On the .NET tab, select the component named System.Security and then click Select.

Click OK.

Add a Web reference to the XML Web service that is to receive the SOAP message encrypted with the shared secret.
On the Project menu, click Add Web Reference.
In the Add Web Reference dialog box, type the URL for the XML Web service in the Address box, and then click the arrow icon.

Verify that the items in the Available references box are the items you want to reference in your project, and then click Add Reference.

Edit the proxy class to derive from WebServicesClientProtocol.
In Solution Explorer, right-click the Reference.cs file for the Web reference just added, and then click View Code.

Note: If the Reference.cs file containing the proxy class is not visible, click the Show All Files icon in the Solution Explorer toolbar, and then expand the Reference.map node.

Change the base class of the proxy class to Microsoft.Web.Services.WebServicesClientProtocol.

This modifies the proxy class to emit WS-Routing and WS-Security SOAP headers when communicating with the XML Web service. The following code example modifies a proxy class named Service1 to derive from Microsoft.Web.Services.WebServicesClientProtocol.

```
public class Service1 : Microsoft.Web.Services.WebServicesClientProtocol {
```

Note: If you select Update Web Reference in Visual Studio .NET, the proxy class is regenerated, the base class is reset to SoapHttpClientProtocol, and you must change the class the proxy class derives from again.

Add the following using directives to the top of the file containing the XML Web service client code:

```
using Microsoft.Web.Services;
using Microsoft.Web.Services.Security;
using System.Web.Services.Protocols;
using System.Security.Cryptography;
using System.Security.Cryptography.Xml;
```

Create an instance of EncryptionKey and populate it with a string representing the secret key used to encrypt the SOAP message. This string must be difficult to generate and be shared only by an XML Web service and its clients to ensure the SOAP message cannot be decrypted.

The following code example defines a method named GetEncryptionKey that returns a secret key using the Triple Data Encrypton Standard (DES) algorithm.

```
private EncryptionKey GetEncryptionKey()
{
```

```
// Creates an instance of a class deriving from
// SymmetricAlgorithm. The TripleDESCryptoServiceProvider provides
// an implenentation of the Triple DES algorithm.
SymmetricAlgorithm algo = new TripleDESCryptoServiceProvider();

// Defines the 128-bit secret key shared
// by the XML Web service and its clients.
// NOTE: Modify these bytes to create a unique secret key for
// your application. The numbers added to the keyBytes array
// must be the same as those used in the decryption
// process.
byte[] keyBytes = { 1, 2, 3, 4, 5, 6, 7, 8,
9, 10, 11, 12, 13, 14, 15, 16 };
// Defines the initialization vector for the algorithm.
byte[] ivBytes =  { 1, 2, 3, 4, 5, 6, 7 };

// Sets the 128-bit secret key for the Triple DES algorithm.
algo.Key = keyBytes;

// Sets the initialization vector for the Triple DES algorithm.
algo.IV = ivBytes;

// Creates a key that will be used by the WSE to encrypt the
// SOAP message.
SymmetricEncryptionKey key = new SymmetricEncryptionKey(algo,algo.Key);

// Populate a KeyInfo for the key, which is used by a recipient when
// it retrieves the decryption key.

KeyInfoName keyName = new KeyInfoName();
keyName.Value  = "http://www.contoso.com/SymmetricKeyEncryptionExample";
key.KeyInfo.AddClause(keyName);

return key;
}
```

Edit the method that communicates with the XML Web service to encrypt the SOAP message using the secret key. For details about encrypting portions of the SOAP message beyond the defaults, see Specifying the Parts of a SOAP Message That Are Signed or Encrypted.

Get the SoapContext for the SOAP request that is being made to the XML Web service.

The following code example gets the SoapContext for a proxy class named Service1.

```
Service1 svc = new Service1();
SoapContext requestContext = svc.RequestSoapContext;
```

Create an instance of the Security class, which represents the WS-Security SOAP header that will be added to the SOAP request.

```
Security security = new Security();
```

Get the secret key.

```
EncryptionKey key = GetEncryptionKey();
```

Create a new instance of the EncryptedData class using the key.

```
EncryptedData enc = new EncryptedData(key);
```

Add the EncryptedData to the WS-Security SOAP header.

```
requestContext.Security.Elements.Add(enc);
```

Specify the time-to-live (TTL) for the SOAP message to diminish the chance of someone intercepting the message and replaying it. The following code example sets the TTL to 1 minute.

```
requestContext.Timestamp.Ttl = 60000;
```

Call the XML Web service.

```
sum = serviceProxy.AddInt(a, b);
```

The following code example encrypts a SOAP message using the key returned from the method defined in step 6.

```
// Generates the symmetric key.
EncryptionKey key = GetEncryptionKey();

// Creates a new instance of the proxy class.
AddNumbers serviceProxy = new AddNumbers();

// Gets the SoapContext for the impending SOAP request.
SoapContext requestContext = serviceProxy.RequestSoapContext;

// Specifies that the SOAP message be encrypted
// using the key generated by the procedure knowing
// the shared secret.
EncryptedData enc = new EncryptedData(key);
requestContext.Security.Elements.Add(enc);

// Sets the time-to-live to 1 minute.
requestContext.Timestamp.Ttl = 60000;
// Makes a SOAP request to the XML Web service.
sum = serviceProxy.AddInt(a, b);
```

CONCLUSION

The conclusion of my project work is a set of specifications that describes How to establish a secure communications between different parties that exchange data using web services protocol (SOAP).

The protocol includes specification that assures data integrity and confidentiality and Identity Authentication and authorization which is meant to lead to a secure communications in B2B environment.

The protocol based on SOAP specifications include Description for registering and Consuming a web service in a universal identified format using Microsoft Web Services Development kit WSDK, Microsoft Web Services Enhancement 1.0 and Microsoft Visual Studio.Net Development tools.

The Developed Specification has been interfaced by implementing it using various Encryption and Digital Signature Techniques.

The performance has been measured to reach global standards to ensure that the developed specifications comply with the light weight SOAP protocol. The Future work will be extended to comply with the rapid development of the world wide consortium (W3C) standards of Web Services Future Specification, such as WS-Security, WS- License, WS-Routing and WS-Referral.

REFERENCES

[1] XML Web Services Developer Center.
http://msdn.microsoft.com/webservices/deafult.aspx

[2] W3C.Simple object access protocol.
http://www.w3.org/2000/xp/

[3] UDDI Project, \UDDI Technical White Paper", September 2000, Available at
http://www.uddi.org

[4] W3C. Web services description language. http://www.w3.org/TR/wsdl/

[5]From the W3C WS-Arch discussion list: http://lists.w3.org/Archives/Public/www-ws-arch/2002Mar/0028.html

[6] MSDN SOAP home page.
http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000523

[7] WSDL description.
http://msdn.microsoft.com/library/en-us/dnwebsrv/html/wsdlexplained.asp

[8] WSDL specification:
http://www.w3.org/TR/wsdl

[9]More information about UDDI is available at http://www.uddi.org/about.html.

[10] Defining Web Services.
http://www.stencilgroup.com/ideas_scope_200106wsdefined.html

[11] Web Services FAQ:
http://www.oreillynet.com/pub/a/webservices/2002/02/12/webservicefaqs.html
http://www.westbridgetech.com

[12] Systinet Corporation Copyright 2002.
www.systinet.com

[13] Microsoft Corp.2002.All Rights Reserved.
www.microsoft.com

[14] Microsoft Web Services.
http://msdn.microsoft.com/webservices.