

**The Florida State University**

**College of Arts and Sciences**

**SSL and SSL Proxy Support  
for SOAP/XML Web Services**

*By*

*Yi Huang*

*November 11, 2002*

A project submitted to the  
Department of Computer Science  
In partial fulfillment of requirements for the  
Degree of Master of Science

## Master Project Committee

---

Prof. Robert van Engelen  
Major Professor

---

Prof. Xiuwen Liu  
Committee Member

---

Prof. Daniel Schwartz  
Committee Member

## *Acknowledgements*

I'd like to thank Dr. Robert van Engelen for giving me the opportunity to work on this project and for his constant guidance and support during my research and project development.

I'd also like to thank my committee members, Dr. Xiuwen Liu and Dr. Daniel Schwartz for their support.

## Table of Contents

<b>1</b>	<b>PROJECT GOALS AND ACCOMPLISHMENTS</b>	<b>1</b>
<b>2</b>	<b>INTRODUCTION</b>	<b>2</b>
<b>2.1</b>	<b>Secure Socket Layer (SSL)</b>	<b>2</b>
2.1.1	Threat on Internet	2
2.1.2	Security primitives	3
2.1.3	Encryption	3
2.1.4	Message digest	4
2.1.5	Public key encryption	4
2.1.6	Digital Signature	7
2.1.7	CipherSuite	8
2.1.8	SSL versions	8
2.1.9	SSL handshake	10
<b>2.2</b>	<b>URI (Uniform Resource Identifier)</b>	<b>12</b>
<b>2.3</b>	<b>HTTP protocol</b>	<b>12</b>
2.3.1	How HTTP protocol works?	13
2.3.2	Request	13
2.3.3	Response	15
<b>2.4</b>	<b>HTTPS</b>	<b>16</b>
<b>2.5</b>	<b>HTTP proxy.</b>	<b>16</b>
<b>2.6</b>	<b>XML</b>	<b>17</b>
2.6.1	Introduction	17
2.6.2	XML schema Definition (XSD)	19

2.6.3	XML namespace	20
<b>2.7</b>	<b>SOAP</b>	<b>21</b>
<b>2.8</b>	<b>Web Services</b>	<b>23</b>
2.8.1	Introduction	24
2.8.2	WSDL (Web Services Definition Language)	25
2.8.3	UDDI (Universal Description, Discovery and Integration Service)	25
<b>3</b>	<b>DESIGN OF THE PROJECT</b>	<b>27</b>
3.1	Survey of available SSL implementations	27
3.2	Requirement analysis	27
3.3	Project outline	28
3.3.1	Use openssl implementation of SSL as toolkit.	28
3.3.2	Use compiler macro to add SSL functionality.	29
<b>4</b>	<b>IMPLEMENTATION</b>	<b>31</b>
4.1	Introduction to gSOAP	31
4.2	General description of openssl programming	32
4.3	Openssl API used in the project.	34
4.4	Proxy server connection	39
4.5	How to get a certificate and related openssl commands	39
<b>5</b>	<b>EXAMPLES</b>	<b>43</b>

<b>5.1</b>	<b>Example of SSL server (sslserverRSA.c/sslserverDH.c)</b>	<b>43</b>
5.1.1	Initialization of SOAP object	43
5.1.2	Accept connections from clients	44
<b>5.2</b>	<b>Example of SSL Client side. (sslclient.c)</b>	<b>45</b>
<b>5.3</b>	<b>Compilation</b>	<b>46</b>
<b>6</b>	<b>RESULTS</b>	<b>47</b>
<b>6.1</b>	<b>Test environment</b>	<b>47</b>
<b>6.2</b>	<b>Monitoring communication with “ssldump”</b>	<b>47</b>
<b>6.3</b>	<b>Test results</b>	<b>51</b>
<b>7</b>	<b>CONCLUSIONS</b>	<b>51</b>
<b>8</b>	<b>REFERENCES</b>	<b>53</b>

## **Abstract**

In this project, I investigated security issues in Internet communication, studied Secure Socket Layer (SSL), implemented SSL support and SSL proxy server in a SOAP/XML web services toolkit and tested these supports using a testing application. Test result is verified by monitoring network traffic.

# 1 Project Goals and Accomplishments

In this project, I investigated security issues in Internet communication, studied Secure Socket Layer (SSL), implemented SSL support and SSL proxy server in a SOAP/XML web services toolkit and tested these supports using a testing application. Test result is verified by monitoring network traffic.

I learned a lot by working on this project. By reading books and documentations related to SSL protocols and Public Key Infrastructure (PKI), I learned the security problems of Internet communication and how to apply SSL protocol, proxy server and PKI to solve these problems. I also learned web service programming and how to achieve platform-independent communication with SOAP, XML and Web services.

The report is organized as follows:

Section 2, introducing to SSL and web service;

Section 3, describing of the design of this project;

Section 4, describing some implementation detail of this project;

Section 5, providing examples of both server and clients;

Section 6, describing testing of this project;

Section 7 concludes this report.

References I used in this project are provided in section 8.

## 2 Introduction

### 2.1 Secure Socket Layer (SSL)

SSL is the de facto Internet standard for security. The SSL protocol secures a wide range of wire-line and wireless applications, including Web commerce, Internet communications and financial management.

#### 2.1.1 Threat on Internet

There are two kinds of possible attacks on the Internet:

**Active attack** is an attack that depends on the attacker writing data to the network.

**Passive attack** is an attack that merely involves reading data off the network.

There are three major goals in security: confidentiality, message integrity and endpoint authentication.

**Confidentiality** means the data is kept secret during transmission. No attacker can get the content of the communication.

**Message integrity** means the content of the communication cannot be added or removed any words.

**Endpoint authentication** means that we need to know that the other end of communication is indeed the one we intended.

### **2.1.2 Security primitives**

Nearly every piece of communications security technology is based on one of four simple pieces: encryption, digest, public key encryption and digital signature. These pieces are called security primitives. I am going to explain each one in the following section.

### **2.1.3 Encryption**

An encryption algorithm takes some data (plaintext) and converts it to cipher text under the control of a key.

There are two kinds of encryption: stream cipher and block cipher.

The stream cipher encryption generates a stream of data one byte at a time. The data is called key stream. The only stream cipher that has received widespread attention and use is RC4.

The block cipher can be considered as a huge lookup table. It processes the data to be encrypted in blocks of bytes (typically 8 or 16). Each possible plaintext block corresponds to a row in the table. The key is used to select a column in the table. In practice, this lookup is achieved by using a function. Widely used block cipher encryptions include DES, 3DES, RC4 and AES. These algorithms differ in their security level and computation complexity.

## 2.1.4 Message digest

A message digest is a function that takes as an arbitrary length message and outputs a fixed-length string that is characteristic of the message. It helps us to check message integrity and create digital signature. The message digest has two important properties: The first is irreversibility which means it should be extremely difficult to compute a message given its digest. The second is collision-resistance which means it should be difficult to produce two messages  $M$  and  $M'$  such that they have the same digest.

The most widely used message digest algorithms are Message Digest 5 (MD5) and Secure Hash Algorithm (SHA-1)

**Message Authentication Codes (MAC)** are usually constructed from message digest algorithms, but it also incorporates a key into the computation. So the MAC depends on both the key being used and the body of the message being authenticated.

## 2.1.5 Public key encryption

Public key encryption is the method we use in SSL communication. The most used algorithms are RSA and DH. I will explain public key encryption in the next section.

### 2.1.5.1 Key exchange problem

If two people can meet each other, they can exchange a secret key for their communication during the meeting and use secret key cryptography. However, this

method is not convenient for Internet communication since we cannot meet everyone we want to communicate with on the Internet to exchange the secret key. Sending the key in plain text is dangerous since an attacker can intercept the key.

What we actually used on the Internet is public key cryptography. It was invented in 1976. The basic idea is to have a function that uses different keys to encrypt and decrypt. You publish your encryption key (the public key) but keep your decryption key (the private key) secret. Other people use your public key to encrypt the message and you use your private key to decrypt the message. This means that everyone can send you a secret message without ever meeting you.

### **2.1.5.2 Certification**

However, we still have the problem of how to publish the public key. If two parties exchange the public key electronically, the attacker can intercept their keys and instead send his own key to each party. This is called a “*man-in-the-middle*” attack.

The solution is to have a trusted third party, called Certificate Authority (CA). What the CA does is sign individual messages that contain the information about the key owner, such as his name, and his public key. This message is known as “**certificate**”.

X.509 is the primary standard for certificate.

### 2.1.5.3 Public key algorithm

There are two flavors of key establishment. The first is key exchange. One side generates a key and encrypts it using the public key of the other side. RSA is a key exchange algorithm. The second way is key agreement. Both sides cooperate to generate a shared key. DH is a key agreement algorithm.

#### *A RSA algorithm*

Ron Rivest, Adi Shamir, and Len Adelman (hence RSA) invented RSA algorithm in 1977. The public keys of RSA consist of two numbers, the modulus ( $n$ ) and the public exponent ( $e$ ). The modulus is the product of two very large prime numbers,  $p$  and  $q$ . The security of RSA is based on the difficulty of factoring  $n$  to get  $p$  and  $q$ .

The public key operation is very expensive compared with secret key operation. It is desirable to combine these two techniques. The solution is to use a session key. A **session key** is a random secret key. Alice creates this secret key, then encrypts it using Bob's public key. Bob can then get the key using his private key. After that, they will send information using this session key.

To use RSA for key transport, the sender generates a random session key and encrypts that under the recipient's public key. The recipient then decrypts the message with his private key. The two parties now share a session key.

## ***B*** ***DH algorithm***

Diffie-Hellman is the first public key algorithm ever published. The sender and receiver collectively generate a key that is private to them. They each have key pairs. To compute the agreed key, the sender combines his private key with the receiver's public key. The receiver combines his private key with the sender's public key.

The DH algorithm also uses modular exponentiation, but the modulus is a larger prime ( $p$ ). They also share another number called generator ( $g$ ). The security of this algorithm is based on the difficulty to calculate discrete logarithm.

### **2.1.6 Digital Signature**

To authenticate a message, we need to use digital signature. It is used to make sure that the message is really sent by the one we intended. Similar to public key establishment, there are two major ways to create digital signatures: RSA and DSS.

Using RSA for digital signature is almost exactly the same as using it for key transport, except the roles of the public and private keys are reversed. To sign one, one computes a message digest and *encrypts* it using one's *private* key. To verify, the receiver *decrypts*

the digest with the sender's *public* key and compares it to the message digest he has independently computed on the message. If they match, the signature is valid.

DSS is based on the same crypto math as DH: modular exponentiation in a prime field. It requires you to perform a computation based on the message digest and the sender's signature. The computation returns a yes or no answer.

### 2.1.7 CipherSuite

All cryptographic selections for a connection are bundled together into cipherSuite. It specifies the server authentication algorithm, the key exchange algorithm, the bulk encryption algorithm, and the digest algorithm. The suites are listed in the order of descending client preference.

Below are some examples of the cipher suite:

Cipher Suite	Auth	Key Exchange	Encryption	Digest
TLS_RSA_WITH_3DES_EDE_CBC_SHA	DSA	RSA	3DES_EDE_CBC	SHA
TLS_DH_DSS_WITH_DES_CBC_SHA	DSS	DH	DES_CBC	SHA
TLS_RSA_WITH_RC4_128_MD5	RSA	RSA	RC4_128	MD5

### 2.1.8 SSL versions

SSL has a relative short history. SSLv1 was created by Netscape but it was not publicly released. So we will start with SSLv2.

#### **2.1.8.1 SSL v2**

Netscape released SSLv2 in 1994. It is the first release version of SSL. However, this version contains a number of security flaws. First, it is missing several features, such as certificate chain. Second, it has a number of security flaws. For example, it is possible for an attacker to forge a TCP connection closure and make it appear that less data was transmitted than in fact was.

#### **2.1.8.2 SSL v3**

SSLv3 is not based on SSLv2. Instead, Netscape decided to burn everything and start over again. SSLv3 invented a complete new specification and new record type and data encoding. It added a number of new features, like DH, DSS, Rehandshake, certificate chains, etc. It is currently the most widely used protocol on the Internet.

#### **2.1.8.3 TLS**

TLS stands for Transport Layer Security. It is finished by IETF (Internet Engineering Task Force) in 1999. It is an updated version of SSLv3. It changed little on SSLv3 except for some bugs. It added more features, such as new MAC algorithm. It also requires implementation of DH, DSS and Triple-DES (3DES).

Deployment of TLS on the Internet at large is still in its infancy. Security application must be able to use SSL to access many existing services.

### **2.1.9 SSL handshake**

The technologies we discussed above need to be combined. For example, public key methods are much slower than secret key methods, so it's convenient to combine the two techniques by using public keys to exchange secret keys. To encrypt a message, Alice generates a random secret key (session key) and encrypts it under Bob's public key, which she gets from Bob's certificate. This combination of public key encryption and secret key encryption provides fast message encryption with the benefits of certificate-based key management.

Similarly, digital signature algorithms are very slow and can only be used with small messages. But combined with message digests, they can be used to efficiently sign large messages. To sign a message, Alice computes the message digest of the message and signs that digest with her private key. The combination of message digests and digital signatures provides message integrity and sender authentication without shared keys.

The basic stages of SSL protocol include:

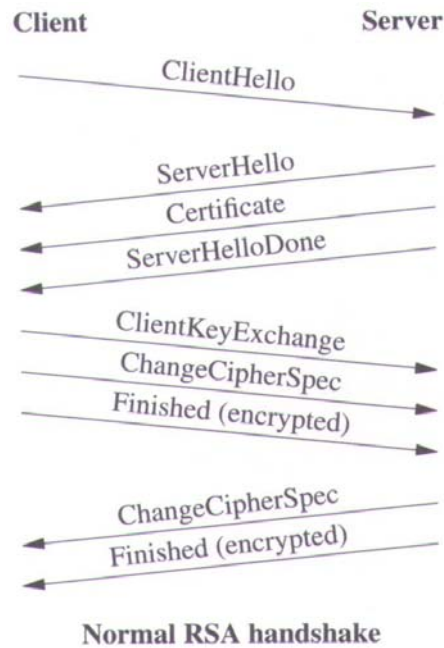
*Handshake:* Alice and Bob use their certificates and private keys to authenticate each other and exchange a shared secret.

*Key derivation.* Alice and bob use the agreed upon shared secret to derive a set of cryptographic keys which can be used to protect the traffic.

*Data transfer.* The data to be transmitted is broken up into a serial of records, each of which is individually protected. This allows data to be transmitted as soon as it is ready and processed as soon as it is received.

*Connection closure.* Special protected closure messages are used to securely close the connection. This prevents an attacker from forging closes and truncating the data being transferred.

Following is time sequence of a normal RSA handshake:



The “ClientHello” is always the first handshake message. It contains three negotiable parameters: version, cryptographic algorithms and compression algorithm. The version field contains the highest SSL version number that the client is prepared to speak. For SSLv3, the number is 3.0. This means major=3 and minor=0. For TLS, this number is 3.1. This means major=3 and minor=1.

The “ServerHello” is used by the server to choose from the various options offered to it by the client.

The “ChangeCipherSpec” message indicates that all messages send afterwards will be encrypted using the just-negotiated cipher.

The “Finished message” is the first message encrypted with the new cryptographic parameters. It allows an implementation to verify that none of the handshake messages have been tampered with by an attacker.

After the handshake, the client and the server can exchange data encrypted under session key.

## **2.2 URI (Uniform Resource Identifier)**

A URI is the way to identify any points of content in the Internet space. The content may be a page of text, a video or sound clip, a still or animated image, or a program. The most common form of URI is the Web page address, which is a subset of URI called a Uniform Resource Locator (**URL**). Another kind of URI is the Uniform Resource Name (**URN**). The exact location of a URN may change from time to time, but some agency will be able to find it.

## **2.3 HTTP protocol**

HTTP (Hypertext Transfer Protocol) is the protocol SOAP uses to communicate with each other. It is the most widely used protocol on the Internet.

### 2.3.1 How HTTP protocol works?

HTTP operates over TCP connections, usually at port 80. After a successful connection, the client transmits a request message to the server, which sends a reply message back. HTTP messages are human-readable. An HTTP server can be manually operated with a command such as “telnet www.cs.fsu.edu 80”. This telnet command is used to establish a TCP connection on a given port of a given server and exchange data with that server.

### 2.3.2 Request

To send a request to the server, the client sends an initial request line and some header lines. The initial request line specifies the HTTP method name, the request URI, (local path of the requested resource) and the version of HTTP being used. The header lines specify the host name, range, etc.

The following is a simple example of an HTTP request.

```
GET /index.html HTTP/1.1<enter>  
Host: www.cs.fsu.edu<enter><enter>
```

There are several methods in HTTP. The most commonly used methods are GET, POST, PUT and DELETE.

**GET method:**

This is the simplest method. It retrieves whatever information is identified by the Request URI. The one used in the example above is “GET” method.

**POST method:**

Another most commonly used method is POST. The POST method is used to request that the destination server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. It is often used to submit the data of a form to a program to process, or to post data in mail lists, news groups, etc.

Below is an example of a POST request.

```
POST /eplan/main.asp HTTP/1.1<enter>
Host: www.lpg.fsu.edu<enter>
Content-Type: application/x-www-form-urlencoded<enter>
Content-Length: 22<enter>

HTTP/1.1 100 Continue
Server: Microsoft-IIS/5.0
Date: Sat, 21 Sep 2002 22:34:57 GMT

ID=visitor&pwd=visitor <enter><enter>
```

This request posts the user’s ID and password to the server. It also includes extra headers to describe this message body, like **Content-Type** and **Content-Length**.

**PUT method:**

The PUT method requests that the enclosed entity be stored under the supplied Request-URI. It is used by CGI scripts and some Web editors to upload files using HTTP.

**DELETE method:**

The DELETE method requests that the origin server delete the resource identified by the Request-URI.

### 2.3.3 Response

The initial response line, called the *status line*, also has three parts separated by spaces: the HTTP version, a *response status code* that gives the result of the request, and an English *reason phrase* describing the status code. Following is an example of status lines:

HTTP/1.0 200 OK

#### Status Code:

The status code is a three-digit integer, and the first digit identifies the general category of response:

**1xx** indicates an informational message only

**2xx** indicates success of some kind

**3xx** redirects the client to another URL

**4xx** indicates an error on the client's part

**5xx** indicates an error on the server's part

The most common status codes are:

Code	Description	Meaning
<b>200</b>	<b>OK</b>	The request succeeded, and the resulting resource (e.g. file or script output) is returned in the message body.
<b>404</b>	<b>Not Found</b>	The requested resource doesn't exist.
<b>301</b>	<b>Moved Permanently</b>	The requested resource has been assigned a new permanent URL and any future references to this resource should be done using that URL.
<b>302</b>	<b>Moved Temporarily</b>	The requested resource resides temporarily under a different URL. Since the redirection may be altered on occasion, the client should continue to use the Request-URI for future requests.

<b>500</b>	<b>Server Error</b>	An unexpected server error. The most common cause is a server-side script that has bad syntax, fails, or otherwise can't run correctly.
------------	---------------------	---

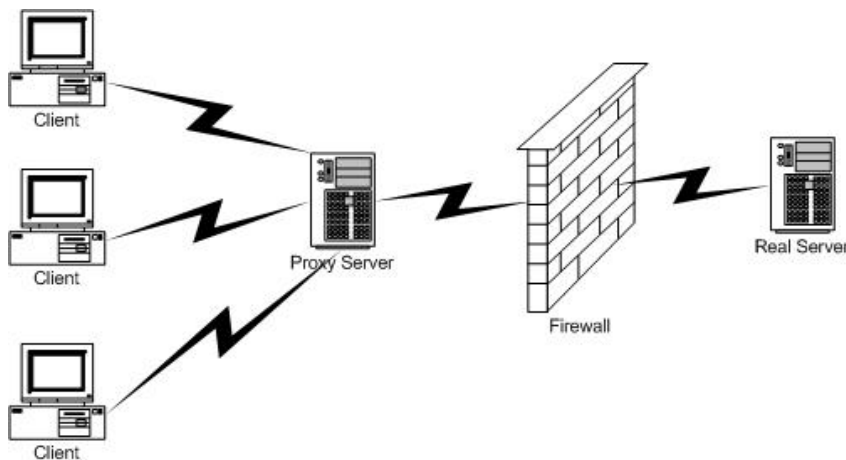
## 2.4 HTTPS

The primary use of SSL is to protect web traffic using HTTP. The combination of HTTP running over SSL is often referred to as HTTPS.

HTTPS usually runs at port 443. In a server that supports HTTPS, it listens to both port 80 and port 443. It should be prepared to accept both secure and non-secured versions of application protocol.

## 2.5 HTTP proxy.

A firewall is a structure intended to keep a fire from spreading. Internet firewalls are intended to keep the flames of Internet out of your private LAN.



In a firewall environment, proxy is the only method available for passing certain firewall.

In HTTP semantics, the proxy examines the client's request in order to determine which server to connect to.

However, HTTP proxy does not work with HTTPS since SSL requires end-to-end connectivity to provide authentication and prevent man-in-the-middle attacks, HTTPS requires that the client pass its request over the encrypted channel. We need some special support to achieve this in proxy server. The special support is a new proxy method--CONNECT.

The **CONNECT** method can establish end-to-end tunnels across HTTP proxies. Here is how it works: The client establishes a TCP connection with proxy and sends the destination host and ports in the **CONNECT** command. The proxy will initiate a TCP connection to the remote host. If the connection is successful, the proxy will send "Connection established" to the client. Then the client transmits the SSL data to the proxy as if the proxy were the server. The proxy server will just pass data between client and server without examining it or changing it.

## **2.6 XML**

XML stands for Extensible Markup Language. It is designed specifically for delivering information over the World Wide Web in a consistent way.

### **2.6.1 Introduction**

The World Wide Web Consortium (W3C) introduced XML in 1996, and the standard's obvious assets -- structure with flexibility, extensibility, adaptability, simplicity and platform-independence -- soon endeared it to Web developers.

Following is an example of a XML file to describe a book.

```
<?xml version="1.0"?>
<book>
```

```
<bookname>Introduction to XML</bookname>  
<author>Yi Huang</author>  
<publishDate>8/8/2002</publishDate>  
<pages>108</pages>  
<price>33.99</price>  
</book>
```

The first line is required in XML documentations. It specifies the use of the XML 1.0 recommendation. The contents inside <book> tag show the properties of the book.

XML is similar to HTML in the following two facts. First, XML is also a platform-independent industry standard that the World Wide Web Consortium (W3C) manages. Such a standard way of describing data would enable a user to send an intelligent agent (a program) to each computer maker's Web site, gather data, and then make a valid comparison before he buys a computer. The second similarity is that XML is also a tag language. Both XML and HTML contain *markup* symbols to describe the contents of a page or file.

Although XML might look similar to HTML, they have several differences. First, XML is not based on a fixed set of predefined tags. It is an "extensible" meta-language used to create custom markup languages. Users can define their own tags in XML to show the meaning of data. The markup symbols are unlimited and self-defining. All types of information or data can be described in XML, such as documents, binary objects, address book entries, financial transactions, or scripts. Also, XML defines information and data according to purpose rather than presentation. XML tags do not control how a Web browser displays text. By using XML, several applications can use the information and data in ways that promote diverse application reuse and extensibility.

## 2.6.2 XML schema Definition (XSD)

XSD specifies how to formally describe the elements in an XML document. It provides a means for defining the structure, content and semantics of XML documents.

XSD can be used to verify that each item of content in a document adheres to the description of the element in which the content is to be placed. To create a schema for a document, you analyze its structure, define each structural element as you encounter it.

For example, the tags in the above examples can be defined as follows:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="book">
    <xsd:element name="bookname" type="xsd:string"/>
    <xsd:element name="author" type="xsd:string"/>
    <xsd:element name="publishDate" type="xsd:date"/>
    <xsd:element name="pages" type="xsd:integer"/>
    <xsd:element name="price" type="xsd:decimal"/>
  </xsd:complexType>
</xsd:schema>
```

In XML schema definitions, we use complex types and simple types to define the structure. Complex types allow elements in their content and may carry attributes. We used complex type to define the book element. Simple types cannot have element content and cannot carry attributes. All attribute declarations must reference simple types. The *string*, *date*, *integer* and *decimal* are all the simple types built into XML schema.

XSD has several advantages over earlier XML schema languages, such as document type definition (DTD). For example, it's more directly written in XML, which means that it doesn't require intermediary processing by a parser. Other benefits include self-documentation, automatic schema creation, and the ability to be queried through XML Transformations (XSLT).

### 2.6.3 XML namespace

Most variables in SOAP message are defined using XML namespace. It is a way to distinguish between duplicate element types and attribute names.

XML namespaces provide a two-part naming system for element types and attributes.

The first part of the name is the URI used to identify the XML namespace -- the *namespace name*. The second part is the element type or attribute name itself -- the *local part*, also known as the *local name*. Together, they form the *universal name*.

URIs are used to define XML namespace simply because they're a well-known system for creating unique identifiers. We should not try to resolve these URIs when processing XML documents. Actually, the URIs used as XML namespace names are not guaranteed to point to schemas, information about the namespace, or anything else -- they're just identifiers.

The XML namespace is declared using special attributes. The name of the attributes has the form: `xmlns:prefix` or `xmlns`. The `xmlns:prefix` form declares a prefix to be associated with the XML namespace. The `xmlns` form declares that the specified namespace is the

default XML namespace. These attributes are often called *xmlns attributes* and their value is the name of the XML namespace being declared.

Namespaces eliminate this problem in XML by letting you specify every element name as a fully qualified name. As in programming, namespaces can be scoped. This means that if an element name is not fully specified, the default at the immediately outer scope applies, then the next outer scope and so forth. This is similar to the variable scoping in the programming. For example, if you are writing a C program and define a global variable called *x* and then define a variable local to your function that is also called *x*, only one of the variables is accessible at any given time, depending on your scope.

## **2.7 SOAP**

Simple Object Access Protocol (SOAP) is a way programs (objects) communicate with each other. Since it uses HTTP and XML standard as the mechanisms for information exchange, it works on different platforms.

Objects are packages of a collection of related procedures. They are typically platform specific. A COM object in Windows platform cannot talk with a C++ Object in UNIX platform. SOAP can make objects in two different platforms talk with each other. It is a platform-independent way to call remote methods.

Another advantage of SOAP is that it can get through firewalls easily. Distributed object technologies usually require the use of special ports to transmit their data. For example, DCOM uses port 135. However, most corporate firewalls prevent the use of all ports

except the default port for HTTP (port 80) since opening other ports could pose a potential security problem.

SOAP messages are sent in a Request / Response fashion. SOAP specifies exactly how to encode an HTTP header and an XML file for sending request and passing information. It also specifies how the called program can return a response, either the result or error values.

Many implementation details are left to the software developer or a third party software vendor. For example, SOAP protocol doesn't specify how the requestor or the responder send or receive messages and how to create an instance of an object and execute the method once received the message.

The following is an example of SOAP request message. Imbedded within the SOAP Envelope tags, there is a method name tag and several tags for the parameters. Between the opening and closing parameter tags are the actual values that will be sent to the remote method.

This example calls the "add" method in host "pcp660416pcs.prshng01.fl.comcast.net". It sends two values "10" and "20" to the remote server.

```
Host: pcp660416pcs.prshng01.fl.comcast.net
User-Agent: gSOAP/2.1
Content-Type: text/xml; charset=utf-8
Content-Length: 490
SOAPAction: ""
```

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
```

```

xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:ns="urn:ssl"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
  <ns:add>
    <a xsi:type="xsd:double">10</a>
    <b xsi:type="xsd:double">20</b>
  </ns:add>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

After the remote server receives the request message, it will create an object and call related method. Then, it creates a SOAP message to send the result back. Following is an example SOAP response message. It contains the name of the response, and the value of the response.

```

HTTP/1.0 200 OK
Server: gSOAP/2.1
Content-Type: text/xml; charset=utf-8
Content-Length: 485

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:ns="urn:ssl"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
  <ns:addResponse>
    <result xsi:type="xsd:double">30</result>
  </ns:addResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

By using SOAP, we can make systems highly distributed. Software developers will be able to build more reliable systems more quickly and more easily with their expertise and their existing codes.

## 2.8 Web Services

One of the major applications of SOAP is to build web services. Web services help computers to communicate over the Internet.

### **2.8.1 Introduction**

Web services application is a new type of web application. It provides not only information but also services. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the web. The services can be simple requests or complicated business processes. Once a web service is deployed, other applications (and other web services) can discover and invoke the deployed service.

The advantage of Web services is that it allows programs written in different languages on different platforms to communicate with each other based on standards. Compared with previous efforts on distributed computing (e.g. CORBA or DCE), web service is significantly less complex and they work with standard Web protocols—XML, HTTP and TCP/IP.

The web services are remotely invoked on the Web through SOAP, described with a WSDL file and registered in UDDI.

We have described SOAP in details in previous section. Here I will explain WSDL and UDDI.

## **2.8.2 WSDL (Web Services Definition Language)**

WSDL provides a way for service providers to describe the basic format of web service requests over different protocols or encoding. A WSDL file is an XML document that describes a set of SOAP messages and how the messages are exchanged. Besides SOAP message contents, it also describes where the service is available and what communications protocol is used to talk to the service. WSDL usually use SOAP/HTTP/MIME as the remote object invocation mechanism.

## **2.8.3 UDDI (Universal Description, Discovery and Integration Service)**

Universal Discovery Description and Integration is the yellow page of Web services. It provides a mechanism for clients to find other web services. A UDDI registry has two kinds of clients: businesses that want to publish a service (and its usage interfaces), and clients who want to obtain services of a certain kind and bind programmatically to them.

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.



## **3 Design of the project**

In this section, I will explain my design consideration in this project.

### **3.1 Survey of available SSL implementations**

There are many SSL implementations available. I did some research on the available SSL implementations for this project.

Major C/C++ implementations include both free toolkit and commercial toolkit.

OpenSSL is the leading open source SSL implementation. There are also some vendors that sell C/C++ SSL/TLS toolkits, such as certicom ([www.certicom.com](http://www.certicom.com)) , RSA Security ([www.rsasecurity.com](http://www.rsasecurity.com)), etc.

There are also java implementations. For example:

PureTLS: This is a free Java-only implementation of SSLv3 and TLSv1. It is widely used for Java implementation.

JDK 1.4: Sun's JDK 1.4 comes with SSL support (not just https support) built in. The package is javax.net.ssl.

### **3.2 Requirement analysis**

There are several basic requirements for this project:

*First, I need to support all versions of SSL during the communication.*

Currently, the most popular security protocol over the Internet is SSLv3. Definitely I need to support it. TLS is relatively new and not fully accepted. I need to support it for future expansion. I also need to support SSLv2 although it has many security problems and it is obsolete. SSLv2 is not recommended in communication.

*Second, I need to maintain the current functionality of gSOAP.*

Since SSL support is an option to gSOAP users, we cannot force users to use SSL in their communication. The users need to have a way to switch between SSL mode and plain text mode.

*Third, I need to support SSL proxy in this project.*

In real world, there are many computers sitting behind firewall. Many users of gSOAP have requests for implementing proxy support in gSOAP. They need to send SOAP request and receive SOAP reply through proxy servers.

## **3.3 Project outline**

After analyzing all these requirements, I decided the following outline for this project.

### **3.3.1 Use openssl implementation of SSL as toolkit.**

Since there are many SSL implementations available, we will use the API provided by the SSL toolkit instead of implementing SSL from scratch. I used openssl toolkit in this project for the following reasons:

First, it is implemented in C. gSOAP is implemented in C with support for C++. The API in OpenSSL toolkit is compatible with other code in gSOAP.

Second, it is freely available and of high quality. It is the leading open source in SSL. gSOAP is a open source project, using open source for SSL support is convenient for gSOAP users.

Third, it supports multiple platforms, linux, unix, and windows. Without changing code, we can use openssl on all the platforms that gSOAP support.

Fourth, it supports all versions of SSL. You can specify which version of SSL you want to support or you can specify it to support all versions.

Fifth, there are many users using it and there are many tutorials and discussion groups available on the Internet. Help is readily available.

### **3.3.2 Use compiler macro to add SSL functionality.**

I decided to use compiler macro to add SSL functionality. In this way, the user only needs to add a compiler flag “-DWITH\_OPENSSL” to specify that we need SSL related code in the final executable. For example:

```
gcc -DWITH_OPENSSL -o sslclient sslclient.c stdsoap2.c soapC.c soapClient.c -lssl -
lcrypto
```

I will add code related to SSL between “`#ifdef WITH_OPENSSL`” and “`#endif`”. For example:

```
#ifdef WITH_OPENSSL
    if (soap->ssl)
        nwritten = SSL_write(soap->ssl, s, n);
    else
#endif
    ...
    nwritten = send(soap->socket, s, n, 0);
    ...
```

This code is compiled as follows: if `WITH_OPENSSL` is not defined, the compiler will ignore the code between “`#ifdef WITH_OPENSSL`” and “`#endif`”. If the user wants to use SSL security, the code will call `SSL_write`; otherwise, the code will call standard Unix API “`send`” to send the message in plain text.

### **3.3.3 Proxy support is coded in the function that sets up TCP connection.**

After setting up TCP connection to the proxy, the client needs to send request in “`CONNECT`” command to connect real host through the proxy server. This command will be coded in string and should be sent to proxy in TCP connection in plain text.

## 4 Implementation

In this part, I will explain major methods in the `stdsoap.c` that I have modified to add SSL support function and proxy support function.

### 4.1 Introduction to gSOAP

The gSOAP project is led by Professor Rebert Van Engelen at the Florida State University. The project aimed to develop a useful and convenient toolkit that can deploy C and C++ applications as either SOAP/XML web services or client or both in a distributed system. It can automatically map native and user-defined C and C++ data types to semantically equivalent SOAP data types and vice-versa. Full SOAP interoperability is achieved with a simple API. “gSOAP” users do not need to know SOAP details. This enables them to concentrate on the application-essential logic.

The transparent binding between C and C++ data types and SOAP/XML data types is achieved by compiler technology. The user specifies the new interface to remote server/client by creating a new header file and modifies the related function call in source code. Then he can just call “soapcpp2” compiler to generate some intermediate files and then use `gcc/g++` or other compilers to compile his code together with the intermediate files. The generated executable file contains a stub (for client) or a skeleton (for server) that can send messages using SOAP protocol.

Since SOAP is a standard for distributed computing, the generated clients can communicate with any web services using SOAP protocol and in XML format. They can be written using any languages, running anywhere and on any platform. Similarly, the generated servers can accept requests from any clients written by any language, running anywhere and on any platform.

gSOAP supports multiple platforms, including Windows, Windows CE, Linux, Solaris, Mac OS X. The gSOAP toolkit also has a WSDL converter for conversion of WSDL service descriptions into a header file for gSOAP to achieve automated implementation of client stubs.

## 4.2 General description of openssl programming

The API programming of openssl is similar to UNIX socket programming. Many methods have the same parameters as methods in UNIX socket programming, like connect, bind, read, write, etc.

Socket API	openssl API
int socket(int, int, int)	SSL *SSL_new(SSL_CTX *)
int connect (int, const struct sockaddr *, int)	int SSL_connect(SSL *)
ssize_t write(int, const void *, size_t)	int SSL_write(SSL *, char *, int)
ssize_t read(int, void *, size_t)	int SSL_read(SSL *, char *, int)

However, there are several differences in openssl. We need three major objects in SSL programming: SSL \_CTX object, SSL object, and BIO objects.

**SSL\_CTX object** is used to set up the context of the SSL connection, it contains SSL version information and it helps to link only related library functions to the final object. Before we create an SSL connection, first we need to create a context object that will be used by the system. This context object is then used to create a new connection object for each new SSL connection. These connection objects are then used to do SSL handshakes, reads, and writes.

There are two advantages to this approach. First, it saves time since the context object allows many structures to be initialized only once. In most applications, every SSL connection will use the same keying material, root list, etc. After we load the keying material into the context object at initialization, we can reload this material for every ssl connection simply by pointing the connection to the context object.

The second advantage is that it allows multiple SSL connections to share data. For example, when we want to resume a previous SSL session, we can simply create a new connection with the same context object since all the session data are stored in the context object.

**SSL object** is used to represent an SSL connection.

One thing we need to note here is the **BIO object**. It is a layer between SSL object and socket connection. We create a BIO object using socket and then attach the SSL object to the BIO. OpenSSL uses BIO object to provide a layer of abstraction for I/O. This provide more flexibility since as long as your object meets the BIO interface, it doesn't matter what the underlying I/O device is. It can be sockets, or memory buffer, or serial line.

### 4.3 Openssl API used in the project.

To setup SSL connection, we need to do the following steps: Initialization, Server accept or Client connect, read and write and closure of SSL connection.

#### **Initialization:**

The codes related to ssl initialization are in the method of `ssl_auth_init(struct soap *soap)`. This method calls `ssl_init()`. Major Openssl API used in initialization are as follows:

```
SSL_library_init();
```

This method initializes the OpenSSL library, which is necessary to create an openssl context.

```
SSL_load_error_strings();
```

This method initializes the error list so that the error can be displayed in a meaningful string in stead of error numbers.

```
soap_ssl_ctx = SSL_CTX_new(SSLv23_method());
```

This method sets the SSL version that the context is prepared to negotiate and the functions that the linker should use to link. If you only specify `SSLv3_method()`, the `SSLv2` and `TLS` functions are never even linked into the final project, reducing the binary size. In this project, I used `SSLv23_method()` so that the client or server can work with any SSL version. The return type of `SSLv23_method` is pointer to `SSL_METHOD`.

```
SSL_CTX_use_certificate_chain_file(soap_ssl_ctx, soap->keyfile)
```

The `SSL_CTX_use_certificate_chain_file()` method load the certificate file.

```
SSL_CTX_set_default_passwd_cb(soap_ssl_ctx, fpassword);  
SSL_CTX_use_PrivateKey_file(soap_ssl_ctx, soap->keyfile, SSL_FILETYPE_PEM)
```

These two methods handle the information in the private key file. When loading or storing private keys, a password is needed to retrieve the password-protected private key.

The `SSL_CTX_set_default_passwd_cb()` set passwd callback for encrypted PEM file handling. `fpassword` is the name of the function to hand back the password to be used during decryption of the private key file. The `SSL_CTX_use_PrivateKey_file()` loads the private key file.

```
SSL_CTX_load_verify_locations(soap_ssl_ctx, soap->cafile, 0)
```

`SSL_CTX_load_verify_locations()` specifies the locations for CTX, at which CA certificates for verification purposes are located. The certificates available via `CAfile` and `CApath` are trusted. In this project, we only specified the CA file but did not specify CA path (set as 0). The CA path can be specified in this parameter if necessary.

```
bio = BIO_new_file(soap->dhfile, "r");
r = PEM_read_bio_DHparams(bio, NULL, NULL, NULL);
BIO_free(bio);
SSL_CTX_set_tmp_dh(soap_ssl_ctx, r)
```

This part loads DH parameters from DH file.

```
rsa=RSA_generate_key(512,RSA_F4,NULL,NULL);
SSL_CTX_set_tmp_rsa(soap_ssl_ctx,rsa)
RSA_free(rsa);
```

This part creates a random session key and then sets the RSA key.

```
SSL_CTX_set_verify_depth(soap_ssl_ctx, 1);
```

We set the maximum certificate chain length to be 1.

```
RAND_load_file()
```

If the random number generator used is pseudo-random number generator (PRNG), we need to seed the PRNGs with some random data in order to produce a high quality random stream. `RAND_load_file()` loads a file containing some random data as seed data.

## **Client Connect**

To set up a SSL connection to the server, first the client needs to set up a tcp connection with the server. This is the same as UNIX socket connection. Then the client creates a BIO object from the connecting socket and attaches ssl object to the BIO object. The code about client connection is in the tcp\_connect() method. This method calls ssl\_auth\_init() first then creates client connection.

```
soap->ssl = SSL_new(soap_ssl_ctx);
soap->bio = BIO_new_socket(soap->socket, BIO_NOCLOSE);
SSL_set_bio(soap->ssl, soap->bio, soap->bio);
SSL_connect(soap->ssl)
```

The above code connects the SSL socket.

```
SSL_get_verify_result(soap->ssl)
```

This method verifies server's certificate with CA's root certificate.

```
peer = SSL_get_peer_certificate(soap->ssl);
```

This method returns a pointer to the X509 certificate of the server.

```
X509_NAME_get_text_by_NID(X509_get_subject_name(peer),
                          NID_commonName, soap->msgbuf, 1024);
```

This method retrieves common names from the certificate. Then we can compare the common name with the host name and see if they match. The chain length is automatically checked by openssl.

## Server accept

The code for server accept is conceptually similar to client connect. The server accepts a TCP connection in an infinite loop and gets the socket number. Then the server creates the BIO and SSL object exactly as client accept. The difference is that the server calls `SSL_accept()` instead of `SSL_connect()`.

### **Read and write data**

Writing and reading data is achieved by `fsend()` and `frecv()` callback functions in gSOAP respectively. `fsend()` calls `SSL_write()` to send data to the network and `frecv()` calls `SSL_read()` to receive data from network.

### **Closure**

The code to close the ssl connection is in `tcp_disconnect()` gSOAP function. `SSL_shutdown()` shuts down an active TLS/SSL connection. It sends the "close notify" shutdown alert to the peer. `SSL_free()` frees the SSL object. Then it closes the socket.

### **Error handling:**

If we encounter any error in ssl communication we can call `ssl_error(struct soap *soap, int ret)` to display the error.

In this method, we call `SSL_get_error` to get the error code of the error and convert the error code to a string.

## **4.4 Proxy server connection**

The code that supports proxy connection is in the `tcp_connect()` method.

To connect a server from a proxy server, the first thing the client needs to do is to construct a `CONNECT` request. This request contains the real host and the port that the proxy should connect to. The “`\r\n\r\n`” string at the end of the string signals the end of the HTTP header with a blank line.

Then it reads the response from the proxy and parses the first line of the response (status line). If the connection is successful, the status code is 200. The end of the response is also signaled by a blanked line.

Then the client can send `ssl_client_hello` to the real server. The proxy will just forward the messages between the client and real server.

## **4.5 How to get a certificate and related openssl commands**

In this part, I will explain how to get a free trial certificate from entrust and related openssl commands. This is how I got my certificate in this project. You can also get certificates from other CA. But you need to pay CA for their service.

1, Generate a private key and change the permission of the file to be owner read only.

```
openssl genrsa -des3 -out mykey.pem 1024  
chmod 400 mykey.pem
```

You can see the content of your private key by the following command:

```
openssl rsa -noout -text -in mykey.pem
```

2, Request a free certificate from <http://freecerts.entrust.com/>

During the request, you will be asked to provide your host name. In SSL authentication, this name will be checked and see if it is the same as the host name you are trying to connect to. Then entrust will provide a reference number for your server, such as JLO9-AW6M-PDND.

You can see the content of your request using the following command:

```
openssl req -noout -text -verify -in userreq.pem
```

3, Generate a certificate request (CSR)

```
openssl req -new -days 365 -key mykey.pem -out userreq.pem
```

You will be prompted to provide: Country name, state or province name, locality name, organization name, organization unit name, common name. The common name is the reference number you will get from entrust for your host name. The command above will generate the certificate request for you. Below is an example for generating certificate request for entrust.

```
.  
  
Country Name (2 letter code) [GB]:US  
State or Province Name (full name) [Berkshire]:FL  
Locality Name (eg, city) [Newbury]:Tallahassee  
Organization Name (eg, company) [My Company Ltd]:FSU  
Organizational Unit Name (eg, section) []:CS  
Common Name (eg, your name or your server's hostname) []:JLO9-AW6M-PDND  
Email Address []:yi@cs.fsu.edu
```

4, Copy and paste the content of userreq.pem to the textbox to the entrust webpage and click next. Your certificate will be generated for you. You can save it as cert.pem. You also need to retrieve the certificate of CA (entrust) as the root in the authentication chain. You can save the CA certificate as root.pem.

To see the content of the certificate, you need to use the following command:

```
openssl x509 -noout -text -in cert.pem
```

To verify the certificate, use the following command:

```
openssl verify -CAfile root.pem cert.pem
```

This command will return either “OK” or “FAIL”.

Here is the content of the certificate I got for my linux server.

```
[root@localhost soapcpp-linux-2.1.5]# openssl x509 -noout -text -in server.pem
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 999939974 (0x3b99df86)
    Signature Algorithm: sha1WithRSAEncryption
    Issuer: C=US, O=Entrust, OU=Entrust PKI Demonstration Certificates
    Validity
      Not Before: Jul 17 03:22:16 2002 GMT
      Not After : Jul 17 03:52:16 2003 GMT
    Subject: C=US, O=Entrust, OU=Entrust PKI Demonstration Certificates, OU=Entrust/Web Connector, OU=No
    Liability as per http://freecerts.entrust.com/license.htm, CN=pcp660416pcs.prshng01.fl.comcast.net
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public Key: (1024 bit)
      Modulus (1024 bit):
        00:bc:68:81:66:e6:c8:7c:7a:ef:bb:5f:07:e4:0d:
        38:cc:dc:91:85:9f:5c:37:0b:d6:2e:de:c3:71:ed:
        7b:39:6c:d1:b9:49:c2:36:a0:24:11:78:55:7d:f2:
        4b:ce:d0:de:a0:9b:f5:c8:97:15:49:5c:b9:9a:01:
        88:b9:6a:99:80:a5:98:97:67:9f:f3:39:31:bd:44:
        0f:92:36:ac:f4:27:90:c4:c7:46:08:a9:5b:0e:13:
        a1:8b:82:d8:83:73:38:0c:73:af:ee:65:c4:4e:25:
        44:f7:a8:71:16:01:bd:05:0f:18:7f:f0:07:be:67:
        f8:92:86:75:0f:97:eb:2b:f3
      Exponent: 65537 (0x10001)
    X509v3 extensions:
      X509v3 Key Usage:
        Digital Signature, Key Encipherment
      X509v3 Private Key Usage Period:
        Not Before: Jul 17 03:52:16 2002 GMT, Not After: Mar 29 15:52:16 2003 GMT
      Netscape Cert Type:
        SSL Server
      X509v3 CRL Distribution Points:
        DirName:/C=US/O=Entrust/OU=Entrust PKI Demonstration Certificates/CN=CRL37

      X509v3 Authority Key Identifier:
        keyid:73:52:B2:F2:FC:3D:37:0C:AA:17:DF:68:C0:0E:3A:96:ED:56:25:BA

      X509v3 Subject Key Identifier:
        F8:29:F2:28:1C:B3:58:E6:D2:77:06:E1:49:E5:4C:70:B2:36:04:33
      X509v3 Basic Constraints:
        CA:FALSE
        1.2.840.113533.7.65.0:
          0
    ..V4.0....
    Signature Algorithm: sha1WithRSAEncryption
      71:e6:1e:43:bd:e4:c0:21:ff:3e:53:22:ed:22:ae:1b:b5:0b:
      0a:97:29:1c:42:c5:5a:04:19:f4:8c:fa:06:ad:f7:57:17:ac:
      d9:f4:33:00:5e:be:95:1f:1f:d5:df:3a:84:f7:99:39:72:01:
      13:69:36:03:b2:37:81:f7:55:6f:db:e8:81:77:f0:d1:64:3d:
      1d:f4:08:23:0a:dd:b9:d1:95:be:86:b4:e4:2b:84:92:ed:0b:
      47:cb:e7:c2:95:4e:42:6e:e9:65:c3:5d:5f:11:34:18:52:d1:
      f7:ab:a0:b6:e6:e6:d9:35:71:81:52:56:ea:d5:79:5d:8a:7a:
      24:b3
```

## 5 Examples

This is a simple example that demonstrates how to use gSOAP to create web service and how to use SSL and proxy in communication. The client provides two numbers to the server and the server return the sum of these two numbers.

### 5.1 Example of SSL server (sslserverRSA.c/sslserverDH.c)

#### 5.1.1 Initialization of SOAP object

The gSOAP server can support SSL initialization using either RSA or DH algorithm. To use RSA algorithm, the server initialization part needs to specify a flag to indicate this. To use DH algorithm, the server initialization part needs to specify the name of file that contains DH group information. I created two examples for each algorithm:

sslserverRSA.c and sslserverDH.c.

Below is the code in these two examples to initialize the soap object.

sslserverRSA.c

```

soap_init(&soap);
soap.keyfile = "server.pem"; /* see SSL docs on how to obtain this file */
soap.password = "password"; /* your password */
soap.cacert = "cacert.pem"; /* see SSL docs on how to obtain this file */
soap.randfile = NULL; /* if randfile!=NULL: use a file with random data to seed randomness */
soap.rsa=1;

```

sslserverDH.c

```

soap_init(&soap);
soap.keyfile = "server.pem"; /* see SSL docs on how to obtain this file */
soap.password = "password"; /* your password */
soap.cacert = "cacert.pem"; /* see SSL docs on how to obtain this file */

```

```
soap.randfile = NULL; /* if randfile!=NULL: use a file with random data to seed randomness */
soap.dhfile = "dh512.pem"; /* see SSL docs on how to obtain this file */
```

As we can see, the initialization of the two algorithms is quite similar. We need to specify the following in the initialization of the server: the file that contains the server's certificate and private key; the password for the private key and the file that contains the root certificate. We just need to set the flag (soap.rsa) for RSA algorithm and specify the DH file (soap.dhfile) for DH algorithm.

### 5.1.2 Accept connections from clients

After initialization of SOAP object, we call soap\_bind to bind the program to a socket and call soap\_accept in an infinite loop to get TCP connection. Then we call soap\_ssl\_accept to get SSL connection. Also, we need to create a thread for each connection so that the server can support multiple clients.

```
m = soap_bind(&soap, NULL, 18081, 100);
if (m < 0)
{ soap_print_fault(&soap, stderr);
  exit(-1);
}
fprintf(stderr, "Socket connection successful: master socket = %d\n", m);
for ( ; ; )
{ s = soap_accept(&soap);
  fprintf(stderr, "Socket connection successful: slave socket = %d\n", s);
  if (s < 0)
  { soap_print_fault(&soap, stderr);
    exit(-1);
  }
  if (soap_ssl_accept(&soap))
  { soap_print_fault(&soap, stderr);
    exit(-1);
  }
  tsoap = soap_new();
  if (!tsoap)
  exit(-1);
```

```

tsoap->socket = soap.socket;    /* set by soap_accept */
tsoap->ssl = soap.ssl;          /* set by soap_ssl_accept */
tsoap->bio = soap.bio;          /* set by soap_ssl_accept */
pthread_create(&tid, NULL, &process_request, (void*)tsoap);
}

```

The functionality of this server is very simple. It just adds two numbers. The code is in function `ns_add`.

```

int ns__add(struct soap *soap, double a, double b, double *result)
{
    *result = a + b;
    return SOAP_OK;
}

```

## 5.2 Example of SSL Client side. (`sslclient.c`)

In the client side, we do not need to specify keyfile and certificate if the server doesn't require client authentication. We can simply call `soap_call_ns__add(&soap, "https://192.168.0.102:18081", "", a, b, &result)` to run the soap connection. Here 192.168.0.102 is the IP of the server. You can also specify the server by host name, like "diablo.cs.fsu.edu"; 18081 is the port number that the server is listening; "a" and "b" are the two numbers to be added. The result is passed back by reference in variable "result".

```

if (soap_call_ns__add(&soap, "https://192.168.0.102:18081", "", a, b, &result)
== SOAP_OK)
    fprintf(stdout, "Result = %f\n", result);
else
    soap_print_fault(&soap, stderr);
return 0;

```

If you want to connect the server through a proxy server, you need to specify the host name and port number of the proxy server in `soap.proxy_host` and `soap.proxy_port` variables at the beginning of the code.

```
soap.proxy_host="192.168.0.102";  
soap.proxy_port=3128;
```

## 5.3 Compilation

To compile the program, you need to do the following:

```
./soapcpp2 -c ssl.h
```

The `ssl.h` just contains the signature of the `add` function which is to be called remotely.

```
int ns__add(xsd__double a, xsd__double b, xsd__double *result);
```

This command will compile this header file and generate the following message:

```
Saving soapStub.h  
Saving soapH.h  
Saving soapC.cpp  
Saving soapClient.cpp  
Saving soapServer.cpp  
Using ns service name: ssl  
Using ns service location: https://linprog1.cs.fsu.edu:18081  
Using ns service namespace: urn:ssl  
Saving ssl.wsdl Web Service description  
Saving ssl.nsmapping namespace mapping table
```

Saving ns.xsd XML Schema description

To generate the executable for the client, run the following command:

Linux:

```
gcc -DWITH_OPENSSL -o sslclient sslclient.c stdsoap2.c soapC.c soapClient.c -lssl -lcrypto
```

Solaris:

```
gcc -DWITH_OPENSSL -o sslclient sslclient.c stdsoap2.c soapC.c soapClient.c -lssl -lcrypto -  
lxnet -lsocket -lnsl
```

To generate the executable for the server, run the following command:

Linux:

```
gcc -DWITH_OPENSSL -o sslserver sslserver.c stdsoap2.c soapC.c soapServer.c -lssl -lcrypto -  
lpthread
```

Solaris:

```
gcc -DWITH_OPENSSL -o sslserver sslserver.c stdsoap2.c soapC.c soapServer.c -lssl -lcrypto -  
lpthread -lxnet -lsocket -lnsl
```

## 6 Results

In this part, I will explain how I tested this project and the test result.

### 6.1 Test environment

I have tested the ssl support and proxy support under Linux and Solaris platforms. To be able to run ssldump, I set a private network at home using Comcast cable modem connection. The ssldump program can be downloaded at <http://www.rtfm.com/ssldump/>.

### 6.2 Monitoring communication with “ssldump”

To show the traffic of local loop, use this command:

```
ssldump -Ad -i lo
```

Here is the sample output

```
New TCP connection #1: diablo.cs.fsu.edu(46388) <-> 192.168.0.102(18081)
1 1 0.0642 (0.0642) C>S SSLv2 compatible client hello
  Version 3.1
  cipher suites
  TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
  TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
  TLS_RSA_WITH_3DES_EDE_CBC_SHA
  SSL2_CK_3DES
  TLS_DHE_DSS_WITH_RC4_128_SHA
  TLS_RSA_WITH_IDEA_CBC_SHA
  TLS_RSA_WITH_RC4_128_SHA
  TLS_RSA_WITH_RC4_128_MD5
  SSL2_CK_IDEA
  SSL2_CK_RC2
  SSL2_CK_RC4
  SSL2_CK_RC464
  TLS_DHE_DSS_WITH_RC2_56_CBC_SHA
  TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
  TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
  TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
  TLS_RSA_EXPORT1024_WITH_RC2_CBC_56_MD5
  TLS_RSA_EXPORT1024_WITH_RC4_56_MD5
  TLS_DHE_RSA_WITH_DES_CBC_SHA
  TLS_DHE_DSS_WITH_DES_CBC_SHA
  TLS_RSA_WITH_DES_CBC_SHA
  SSL2_CK_DES
  TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
  TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
  TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
  TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
  TLS_RSA_EXPORT_WITH_RC4_40_MD5
  SSL2_CK_RC2_EXPORT40
  SSL2_CK_RC4_EXPORT40
1 2 0.1479 (0.0837) S>CV3.1(74) Handshake
  ServerHello
  Version 3.1
  random[32]=
    3d 56 c5 5a 3c 2e 85 9f 1b d0 09 38 11 09 40 99
    a5 18 28 8d 0e 4d 3a 29 be 20 50 2f 5a c1 23 b9
  session_id[32]=
    ec 80 93 30 0a 69 c6 76 8c 12 41 c1 8c 6b 33 41
    0f 47 e6 ea 38 36 cd cb 09 93 28 0d e8 4d e2 16
  cipherSuite      TLS_RSA_WITH_3DES_EDE_CBC_SHA
  compressionMethod  NULL
1 3 0.1479 (0.0000) S>CV3.1(1850) Handshake
  Certificate
1 4 0.1479 (0.0000) S>CV3.1(4) Handshake
  ServerHelloDone
1 5 0.2187 (0.0707) C>SV3.1(134) Handshake
  ClientKeyExchange
  EncryptedPreMasterSecret[128]=
```

```

69 64 db 6e 85 f6 9a e9 69 95 48 29 20 ca 4d 32
08 6b 3c 19 8f 72 94 e4 6c dd 57 6f 19 ea ec d1
f9 25 44 b7 1b 52 45 d7 34 9c 6b be 33 e3 5d 40
81 5d f3 12 82 e2 aa f1 af 74 46 69 b9 d7 86 68
1c 74 11 a1 17 3c eb de 74 ef 02 12 52 13 7c 19
60 c1 f4 8a f5 5b 1b 74 1c 0b b1 71 9d 8f ed 9c
7a db 39 16 90 44 3e bd 83 4e 3f fc 9e d4 32 ac
24 71 c5 b0 f9 77 99 a5 64 65 df ef 95 b4 d6 a4
1 6 0.2187 (0.0000) C>SV3.1(1) ChangeCipherSpec
1 7 0.2187 (0.0000) C>SV3.1(40) Handshake
1 8 0.2385 (0.0198) S>CV3.1(1) ChangeCipherSpec
1 9 0.2385 (0.0000) S>CV3.1(40) Handshake
1 10 0.2970 (0.0585) C>SV3.1(680) application_data
1 11 0.2983 (0.0013) S>CV3.1(608) application_data
1 12 0.2984 (0.0000) S>CV3.1(24) Alert
1 0.2985 (0.0000) S>C TCP FIN
1 13 0.3575 (0.0590) C>SV3.1(24) Alert
1 0.3577 (0.0002) C>S TCP FIN

```

If you want to decrypt the traffic and show the application data, you need to provide file name of the private key and the password. The command is:

```
ssldump -Ad -k server.pem -p password
```

In this example, if the file name of the private key is server.pem and the password is “password”.

We can see the application data as this:

```

1 10 0.5210 (0.0547) C>SV3.1(680) application_data
-----
POST / HTTP/1.0
Host: pcp660416pcs.prshng01.fl.comcast.net
User-Agent: gSOAP/2.1
Content-Type: text/xml; charset=utf-8
Content-Length: 490
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:ns="urn:ssl" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><SOAP-
ENV:Body><ns:add><a xsi:type="xsd:double">10</a><b xsi:type="xsd:double">20</b></ns:add></SOAP-
ENV:Body></SOAP-ENV:Envelope>

1 11 0.5227 (0.0016) S>CV3.1(608) application_data
-----
HTTP/1.0 200 OK
Server: gSOAP/2.1
Content-Type: text/xml; charset=utf-8

```

Content-Length: 485

```
<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:ns="urn:ssl" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><SOAP-
ENV:Body><ns:addResponse><result xsi:type="xsd:double">30</result></ns:addResponse></SOAP-
ENV:Body></SOAP-ENV:Envelope>
```

From the messages above, we can see that the data is transported with HTTP header and soap envelope. We also can see the decrypted data in the communication. The SSL dump can decrypt the message with the supplied private key.

To test proxy support, I installed “Squid” proxy server in the Linux machine at my home network. The port 3128 is the proxy server port. The port 18081 is the real host port.

Following is the beginning message of the communication:

```
New TCP connection #8: 192.168.0.102(32797) <-> 192.168.0.102(3128)
0.0012 (0.0012) C>S
-----
CONNECT 192.168.0.102:18081 HTTP/1.0
-----

New TCP connection #9: 192.168.0.102(32798) <-> 192.168.0.102(18081)
0.0022 (0.0010) S>C
-----
HTTP/1.0 200 Connection established
-----

8 1 0.2726 (0.2704) C>S SSLv2 compatible client hello
Version 3.1
cipher suites
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA
SSL2_CK_3DES
TLS_DHE_DSS_WITH_RC4_128_SHA
TLS_RSA_WITH_RC4_128_SHA
TLS_RSA_WITH_RC4_128_MD5
SSL2_CK_RC2
SSL2_CK_RC4
SSL2_CK_RC464
TLS_DHE_DSS_WITH_RC2_56_CBC_SHA
TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
TLS_RSA_EXPORT1024_WITH_RC2_CBC_56_MD5
TLS_RSA_EXPORT1024_WITH_RC4_56_MD5
TLS_DHE_RSA_WITH_DES_CBC_SHA
TLS_DHE_DSS_WITH_DES_CBC_SHA
TLS_RSA_WITH_DES_CBC_SHA
```

```
SSL2_CK_DES  
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA  
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA  
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA  
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5  
TLS_RSA_EXPORT_WITH_RC4_40_MD5  
SSL2_CK_RC2_EXPORT40  
SSL2_CK_RC4_EXPORT40
```

From the messages above, we can see the client connect to the proxy server first. It sends `SSL_Client_Hello` after it receives “HTTP/1.0 200 Connection established” message.

## 6.3 Test results

From the above tests, we can see both SSL communication and SSL proxy support work well in the testing program.

## 7 Conclusions

gSOAP is a very useful tool in implementing web services using SOAP protocol. In this project, I added more functionality to this toolkit. The major functionalities I added include SSL support (both RSA and DH) and SSL proxy support.

By working on this project, I learned the principles and application of Secure Socket Layer (SSL). I gained tremendous understanding of the security problems on the Internet and how to prevent attacks on the Internet. I learned how to add SSL to SOAP/XML web applications using openSSL toolkit.

I also learned how to work on a large project and how to cooperate with others. I gained more problem solving capabilities by examining codes in a large project and locating critical sections that relate to the problem.

## 8 References

1. Eric Rescorla: SSL and TLS –Designing and Building Secure System (2001)
2. Carlisle Adams, Steve Lloyd: Understanding Public-Key Infrastructure (1999)
3. Michael J.Young: XML Step by Step (2000)
4. Elizabeth Castro: XML for the World Wide Web (2001)
5. Robert van Engelen: The SOAP Stub and Skeleton Compiler For C and C++.  
<http://www.cs.fsu.edu/~engelen/soap.html>
6. <http://www.openssl.org/docs/>
7. <http://freecerts.entrust.com/>
8. <http://www.tldp.org/HOWTO/Firewall-HOWTO-2.html>
9. <http://java.sun.com/j2se/1.4/docs/api/javax/net/ssl/package-summary.html>
10. <http://www.rtfm.com/puretls/>
11. <http://www.w3schools.com/default.asp>
12. <http://www.jmarshall.com/easy/http/>
13. <http://msdn.microsoft.com/>
14. <http://www.rpbouret.com/xml/NamespacesFAQ.htm>
15. <http://www.whatis.com>