Journal of Computer Science 2 (2): 171-179, 2006
ISSN 1549-3636
© 2006 Science Publications

Developing a Secure Web Service Architecture for SVG Image Delivery

[1]Sabah Mohammed, [1]Jinan Fiaidhi, [2]Hamada Ghenniwa and [1]Marshall Hahn
[1]Department of Computer Science, Lakehead University, Thunder Bay, Ontario P7B 5E1, Canada
[2]Department of Electrical and Computer Engineering, University of Western Ontario
London, Ontario N6A 5B9, Canada

Abstract: Web Services are substantially growing and become vital for businesses and organizations. A major concern, especially for mission-critical applications is Security. This study focuses on developing Scalable Vector Graphics (SVG) as Web services. In particular, we develop a service-oriented architecture that securely manages SVG Web services using the *intermediary* design pattern. In the proposed architecture we introduced two kinds of specialized security intermediaries to enforce SVG signature/authentication and encryption/decryption. A prototype of the proposed architecture has been implemented based on Apache Axis.

Key words: Web service security, SVG, apache axis, image security

INTRODUCTION

The Scalable Vector Graphics (SVG) promises to revolutionize the Web through the introduction of standards-based vector graphics, animation and interactivity. The broad support behind SVG comes from its many advantages. SVG has sophisticated graphic features, which is naturally important for a graphic format, but it also benefits from being an XML grammar. SVG has all the advantages XML brings such as internationalization (Unicode support), wide tool support, easy manipulation through standard APIs (e.g., the Document Object Model, DOM API, Batik API) and easy transformation (e.g., through XML style sheet Language Transformations, XSLT). In the graphical arena and especially compared to raster graphics formats (such as GIF, JPEG or PNG images), SVG has the advantage of being:

* Lightweight. For many types of graphics, an SVG graphic will be more compact than its raster equivalent
* Zoomable. SVG content can be viewed at different resolutions, e.g., enlarged or shrunk without losing quality.
* Searchable. Because SVG content is XML, it becomes possible to search the content of an SVG image for text elements, comments or any kind of meta-data.
* Structured and Accessible. Graphic objects can be grouped and organized hierarchically.

It is natural for Web Services to accommodate SVG for graphical based services. In addition to being open and XML, SVG has a rich structure and preserves semantic because of its descriptive element and metadata. This richness provides an opportunity to Web Services to generate, modify or search rich graphical content[1].

Traditionally SVG has been used as a flexible imaging viewer only, which limited its' potential for advanced imaging applications. Security issues have been the main challenge in SVG applications. Indeed, there are variety of Web Service composition languages such as PDL, XPDL, BPSS, EDOC, BPML, WSCI, ebXML and BPEL4WS are available today, but there are very little work on Web Services Workflow engines that can be used for achieving security[2]. A key challenge, therefore, is to enable the interoperability between Web Services to take place seamlessly and securely. Generally there are two standards for enforcing security of the Workflow:

Security at the transport level or REST (Representational State Transfer): Security at the transport level uses the inbuilt security features of transport technologies like HTTP, IBM WebSphere MQSeries. Most first-generation Web services use this type of security and have been deployed behind a Web Site's firewall. SSL provides adequate confidentiality for such first-generation Web services. Using SSL the channel over which two parties communicate can be kept confidential - data is encrypted by the sender and decrypted by the recipient. In this direction, delivering SVG Web services can be as simple as using HTTP request then it can be incorporated into a client side view with a single DOM mutation. Prescod[3] reported many technical advantages of implementing SVG Web services built upon REST those built upon SOAP. However, using SSL to support security at this level

Corresponding Author: Sabah Mohammed, Department of Computer Science, Lakehead University, Thunder Bay, Ontario P7B 5E1, Canada

does not provide complete protection for deploying Web services that expose internal systems, especially mission-critical, over the Internet to external entities[4,5].

Security at the messaging level or SOAP (Simple object access protocol): The security at the message level is currently being extensively researched and specifications are being developed by several groups like OASIS and W3C. This involves usage of digital signatures, XML encryption, certificates. The approach is based on creating a layered architecture for the Web services security specifications[6] shown in Fig. 1. On top of SOAP the security model (WS-Security) can be constructed to provide the basis for the other security specifications, such as Web service endpoint policy (WS-Policy), a trust model (WS-Trust) and a privacy model (WS-Privacy). Collectively, these components can be utilized to provide cure interoperable Web services that are more secure than REST based approaches.

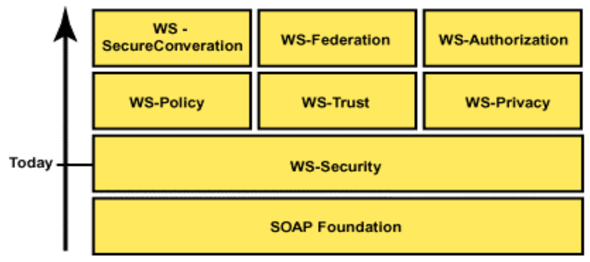


Fig. 1: Web services security specifications road map[6]

However, treating security at the SOAP level presents more challenges than that REST because[4]:

* Soap messages are independent of the transport layer. Hence, the development of the security service should accommodate and not rely on the different security degrees that might be supported by the transport protocols.
* SOAP is designed to be flexible and able to bypass firewalls. However, this is unacceptable feature from a security point of view and need to be restricted or even sacrificed.

However, one interesting aspect of extending SOAP, is to gain access to the actual network stream before it is decentralized into objects and vice versa. For instance, through these extensions encryption algorithms can be developed on top of the Web Service call. However, most of the SOAP extensions are transport Protocols dependent, especially for multimedia applications, such as RTP (<http://www.ietf.org/rfc/rfc1889.txt>.), Multicast and UDP. To deal with this issue, brokering-based architectures can be used. For example, NaradaBrokering[7] extension is based on JMS that provides Web services-sharing and collaboration, as shown in Fig. 2.

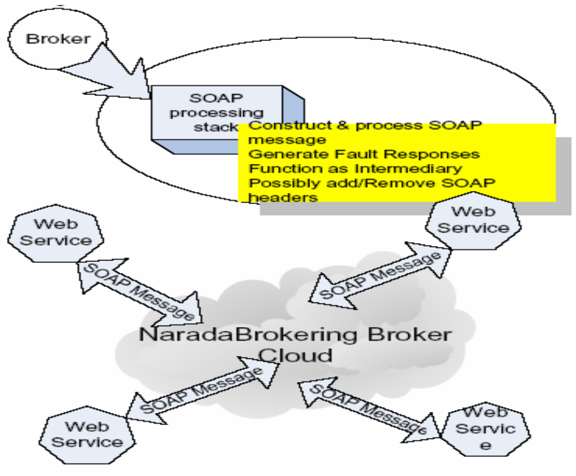


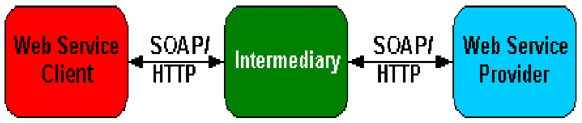
Fig. 2: Indiana University NaradaBrokering SOAP extension[7]

The NaradaBrokering SOAP extension has been used to provide transparent message-related capabilities for SVG Web services[8]. Indeed this can be extended to deal with SVG security, but NaradaBrokering SOAP lacks the widespread support. There are some recent attempts to extend SOAP via more popular publish-subscribe message brokering systems like the IBM *WebSphere Business Integration Message Broker (*<http://www-306.ibm.com/software/info/ecatalog/en_ US/products/ Z106022I73024W93. html>*) and* the Sun Microsystems JXTA (www.jxta.org) . In particular the JXTA-SOAP bridge project as developed by the Reptile (<http://reptile.openprivacy.org>) is very promising direction to support Web services security.

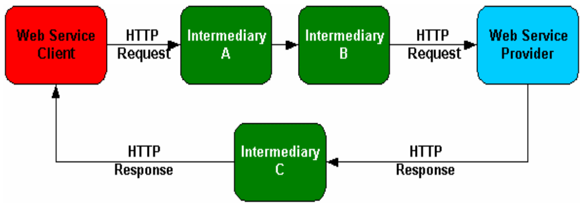
The work presented in this article is another attempt to extend SOAP using intermediary pattern and Apache AXIS Web. The security has been treated at the architecture by injecting Intermediaries in the Web service architecture supported by Apache AXIS. The Intermediaries can be programmed for securing SVG Web services (SVG signature/authentication and SVG Encryption/Decryption).

SOAP intermediaries via apache axis: In this study, we'll be using Apache Axis as our Web Services engine. Axis Web Services can be thought of, at their most basic level, as blocks of code that are assembled and executed according to a particular configuration. Even the Axis engine itself is built out of smaller pieces that are sewn together. This is why Axis is one of the most configurable and extensible engines available. The basic building blocks of a Web Services application are called *handlers or intermediaries.* Each handler represents a specific piece of functionality that is to be executed on the message-processing path.

Handlers are then stitched together into groups called *chains*. A chain is a particular processing path from the requestor to a Web Service and back again. The object which is passed to each Handler invocation is a MessageContext. A MessageContext is a structure which contains several important parts: 1) a "request" message, 2) a "response" message and 3) a bag of properties. Axis is all about processing MessagesContext via handlers and chains. The handlers and chains represent Web Services intermediaries which can be programmed. Hence, these Web services intermediaries are computational entities that can be positioned anywhere along an information stream between the Web services and are programmed to tailor, customize, personalize, or otherwise enhance data as they flow along the stream. A Web Service Intermediary therefore lies between the Web Service Client and the Web Service Provider as shown below:



However, it is possible to combine Intermediaries in several ways. As we see below, a chain of Intermediaries (A and B) intercepts the HTTP Request from the Web Service client. Another Intermediary, C, further intercepts the HTTP Response from the Web Service.



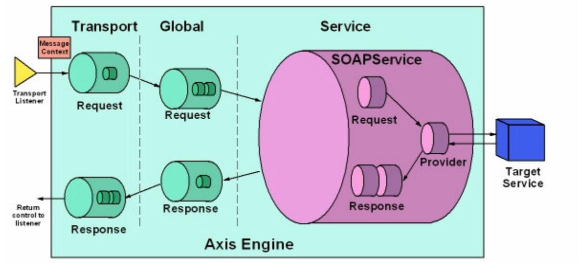
This means that Web intermediaries can be added in a horizontal as well as vertical way. Instead of adding layers to a single service, SOAP can be used to determine which service among a collection of available services is the intended recipient of a particular message and route the message accordingly. Rather than require the client application to have knowledge of two different service endpoints that provide the same functionality, the service should be able to determine from the request, which service should be used to handle the request. Hence, Axis helps in creating these intermediaries and SOAP gives us an easy way to deal with routing a message between *intermediaries*.

However, in order to extend SOAP and program the Axis Web intermediaries, it is important to know which is the Web service invocation to be used. Basically, these are two invocation models: RPC and Document style. Many good articles are available that address the differences of these two models in detail, but the main differences are quite straightforward[9]. *RPC-style* encoding is conceptually the same as other implementations architects and developers have worked with over the years, such as CORBA or RMI. In this direction SOAP RPC support complex object serialization and deserialization. As long as the object complies with the Java Bean specification, the object could be turned into XML and handled transparently to the developer. This was wonderfully seductive -- in a few simple lines of code, real business data objects were sailing over the wire with little concern for underlying implementation. But as it turns out, there are drawbacks to using complex objects over RPC. This approach often leads to integration issues. One implementation's serialization might not match another's deserialization, since the Java Bean to XML SOAP encoding process is ambiguous and not well-defined. Suddenly open technology comes to a screeching halt -- an Apache SOAP service had trouble working with .NET because of discrepancies in their implementations and thus drove the need to keep services more open. *Document-style* offers a satisfying mix of well-defined structures and interoperability. This is achieved through standard XML-Schemas for defining complex objects. In contrast to the simple SOAP encoding, XML-Schema is a rigorous and well-understood standard for defining structures. XML-Schemas provide a great deal of flexibility in defining complex structures while ensuring all the promises of Web services. Document style, which gains all the benefits of XML-Schema, seems to be the solution to all your Web service headaches. However, Document style has some trade-offs. One of the trade-off problems the programmer is faced with is increased complexity. Suddenly, the developer has the arduous task of parsing an XML document and performing the necessary transformations to populate other data beans or method requests with the incoming data. This is true both for the server and client. For the brave of heart, this means writing a custom SAX handler. And SAX handlers aren't known for being particularly user friendly or easy to maintain.

With Apache Axis we can use a hybrid of both RPC and Document-style services and hence maximizing the invocation style advantages. As with any Document-style service, you still have the task of handling the incoming XML data somehow (e.g. Transcoding and Rendering SVGs). Axis and the other surrounding Apache utilities includes a handy tools to help solve this arduous task, such as WSDL2Java API, the Java Beans Activation Framework (JAF), Xalan, Web Service Security Suite and the Apache Batik. WSDL2Java can generate both code stubs for the client and server for your methods and actual beans to model your data from the model defined in the XML-Schema. With the JAF extension, developers who use Java technology can take advantage of standard services to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it and to instantiate the appropriate bean to perform said operation(s). Xalan-Java is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. It implements XSL Transformations (XSLT) and XML Path Language (XPath). Xerces2 is the next generation of high performance, fully compliant XML parsers. The XML Security Suite is a tool that provides security features such as digital signature, encryption and access control for XML documents and finally Batik is a Java(tm) technology based toolkit for applications or applets that want to use images in the SVG format for various purposes, such as viewing, generation or manipulation.

Apache axis invocations: The traditional definition of a Service Oriented Architecture (SOA) is centered around distributed which include services, messages and the dynamic discovery of available Web Services (http://www.service-architecture.com/). With Axis Web intermediaries another component can be added to these ingredients. The invocations of the intermediaries follow certain order. The particular order is determined by two factors - deployment configuration and whether the Axis engine is a client or a server. There are two basic ways in which Axis is invoked:

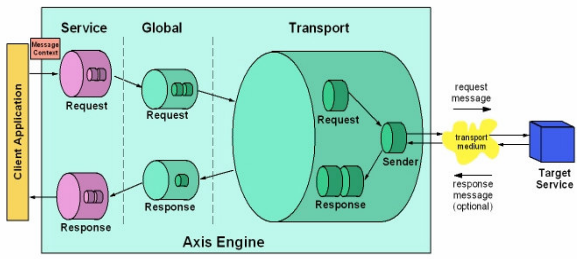
1. As a server, a Transport Listener will create a MessageContext and invoke the Axis processing framework.



Notice that the small cylinders represent Handlers and the larger, enclosing cylinders represent Chains. A message arrives (e.g. Http) at a Transport Listener. It's the Listener's job to package the protocol-specific data into a Message object (org.apache.axis.Message) and put the Message into a MessageContext. The MessageContext is also loaded with various properties by the Listener. The Transport Listener also sets the transportName String on the MessageContext , in this case to "http". Once the MessageContext is ready to go, the Listener hands it to the AxisEngine.

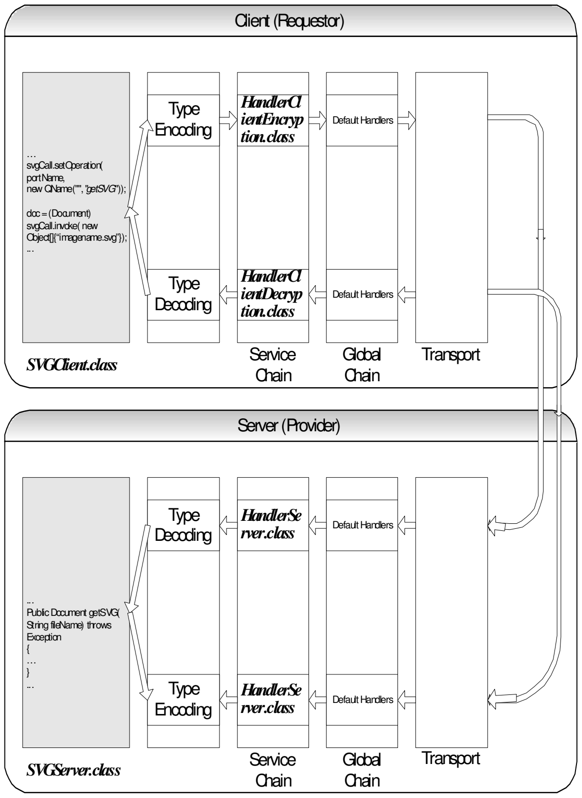
The AxisEngine's first job is to look up the transport by name. The transport is an object which contains a request Chain, a response Chain, or perhaps both. If a transport request Chain exists, it will be invoked, passing the MessageContext into the invoke() method. This will result in calling all the Handlers specified in the request Chain configuration. After the transport request Handler, the engine locates a global request Chain, if configured and then invokes any Handlers specified therein. At some point during the processing up until now, some Handler has hopefully set the serviceHandler field of the MessageContext This field determines the Handler we'll invoke to execute service-specific functionality, such as making an RPC call on a back-end object. Services in Axis are typically instances of the "SOAPService" class (org.apache.axis.handlers.soap.SOAPService), which may contain request and response Chains and must contain a provider, which is simply a Handler responsible for implementing the actual back end logic of the service. For RPC-style requests, the provider is the org.apache.axis.providers.java.RPCProvider class. This is just another Handler that, when invoked, attempts to call a backend Java object whose class is determined by the "className" parameter specified at deployment time. It uses the SOAP RPC convention for determining the method to call and makes sure the types of the incoming XML-encoded arguments match the types of the required parameters of the resulting method.

2. As a client, the message path on the client side is similar to that on the server side, except the order of scoping is reversed, as shown below.



The service handler, if any, is called first - on the client side, there is no "provider" since the service is being provided by a remote node, but there is still the possibility of request and response Chains. The service request and response Chains perform any service-specific processing of the request message on its way out of the system and also of the response message on its way back to the caller. After the service request Chain, the global request Chain, if any, is invoked, followed by the transport. The Transport Sender, a special Handler whose job it is to actually perform whatever protocol-specific operations are necessary to get the message to and from the target SOAP server, is invoked to send the message. The response (if any) is placed into the responseMessage field of the MessageContext and the MessageContext then propagates through the response Chains - first the transport, then the global and finally the service.

Developing an axis SVG secure image delivery SOA system (ASSIDS): In the two types of invocations, Axis job is simply to pass the resulting MessageContext through the configured set of Handlers, each of which has an opportunity to do whatever it is designed to do with the MessageContext. Thus, the SOA system developed in this article consists of two invocation components: the Axis client (service requestor) and Axis server (service provider). Utilizing the Axis client invocation, a user can search and obtain a list of SVG files available for download from the Axis server. Once the client decides on one of the available SVG files, the client can then securely download and view that SVG file. Axis handlers/intermediaries on both the client side and server side automatically encrypt and sign the body tag of every message exchanged by the client and server (except for the first pair messages exchanged, which are used by the client to obtain the servers public key). Figure 3 illustrates the general layout of our SOA.



```
…
svgCall.setOperation(
portName,
new QName("", "getSVG"));

doc = (Document)
svgCall.invoke( new
Object[]{"imagename.svg"});
…
```

```
…
Public Document getSVG(
String fileName) throws
Exception
{
…
}
…
```

Fig. 3: General layout of our SOA

The Class SVGserver provides three methods which can be called remotely. To deploy such class on the Web you need to follow the Apache Axis Web site[10]. The *constructor* creates an org.w3c.dom.Document containing an X509 certificate (which contains the servers public key) and assigns it to the instance reference variable *certDoc*. The getCertificate() method simply returns the org.w3c.dom.Document referenced by certDoc. The getFileList() method returns a string array in which each entry is the name of a file within the directory ASSIDS\ jakarta-tomcat-5.5.9\bin\svg. The getSVG(String fileName) method parses the SVG image indicated by the fileName argument into an org.w3c.dom.Document. It then returns this document. All SVG files are contained within the directory ASSIDS\ jakarta-tomcat-5.5.9\bin\svg. The basic structure of the SVGServer class is illustrated below:

```
public class SVGserver
{
        Document certDoc;
        public SVGserver() {…}
        public Document getCertificate(String arg) {…}
        public String[] getFileList(String dirName) {…}
        public Document getSVG(String fileName) throws
Exception {…}
}
```

When the service requestor (class SVGClient) invokes an operation provided by the SVG web service (For Example, method getSVG() in class SVGServer), both the data provided as an argument to the invocation (For example, imagename.svg) and the method name (For example, getSVG) are encoded in XML format. The result of this process is a complete SOAP envelope that's ready for delivery to the service provider. Before the envelope is sent to the provider, it must first traverse a pair of handler chains. A handler chain is a list of handlers, each of which is given a chance to process the message in whatever way it sees fit. There are two of these chains, one specific to the particular service requestor and one that's global for the entire Axis runtime on that requestor side. As shown in Fig. 2, the two service handlers on the requestor side are HandlerClientEncryption and HandlerClientDecryption. After the envelope has been processed by the two chains, it's sent to a transport protocol(HTTP) that takes care of delivering the message to the service provider.

After the service provider transport receives the message, the Axis engine then gets the envelope out of the transport and passes it through a pair of handler chains. This time the order of the handler chains is reversed. Moreover, the service handler on provider side is called HandlerServer. After the envelope exits the service-specific chain, the contents of the message undergo type decoding, which creates Java objects that correspond to the data in the envelope. These objects are then passed to the method name specified in the message. Once the service code has executed and produced a result, the entire process reverses itself. The return type is encoded in XML format and so on. The details of the HandlerServer class is illustrated below:

```
public class HandlerServer extends BasicHandler
{
static KeyInfo currentCertificate;
```

```
 public void invoke(MessageContext msgContext)
throws AxisFault
      {
            …
                  if(!msgContext.getPastPivot())//Request
message
                  {
            …
                  }
                  else//Response message
                  {
            …
                  }
}
}
```

As previously shown, class HandlerServer is deployed as both a request flow service handler and response flow service handler. Using the same class for both handlers allows the response message processing code to utilize information extracted from the request message (more details on this later). The method msgContext.getPastPivot()) allows the handler to determine whether its invoke method has been called during request message processing or response message processing.

The SVGclient class is fairly simple. Most of its code involves setting up the GUI. The GUI listener methods are responsible for performing remote procedure calls using Axis. The body of this class is illustrated below:

```
public class SVGclient implements ActionListener,
SVGDocumentLoaderListener,
      GVTTreeBuilderListener, GVTTreeRendererListener,
ItemListener
{
      …
      private void init() throws Exception { ... }
      public void actionPerformed(ActionEvent ae) { … }
      public void itemStateChanged(ItemEvent ie) { … }
      …
}
```

Most of the Axis related code has been explained[8] except for the code related to client side handlers which will be illustrated briefly in the following sequel:

```
init()
{
…
reqHandler=new HandlerClientEncryption();
respHandler=new HandlerClientDecryption();
….
listCall.setClientHandlers(reqHandler,respHandler);
//org.apache.axis.client.Call
…
}
```

The init() will invoke the two client handlers(HandlerClientEncryption and HandlerClientDecryption). These handlers will perform the client side encryption and decryption of SVG SOAP Message. The details of these handlers are given briely bellow:

```
public class HandlerClientEncryption extends BasicHandler
{
KeyInfo keyInfo;
public void setKeyInfo(Document doc) throws Exception {
… }
…
}

public class HandlerClientDecryption extends BasicHandler {
… }
```

The combined function of these two classes used on the client side of ASSIDS is similar to that of class HandlerServer. The main difference regards how the X509 certificate is obtained. Before any other methods of class SVGserver can be remotely called, the getCertificate() method must be called. This method returns an org.w3c.dom.Document containing an X509 Certificate (which contains the server's public key). Using the setKeyInfo() method, class SVGclient can inform the class HandlerClientEncryption the public key of the server. Obviously, the messages generated by getCertificate() remote procedure call cannot be encrypted (or at least on the client end). Therefore, the getCertificate() method is deployed as a separate Axis web service in which none of the previously mentioned handlers are deployed.

The techniques used for SVG security: The SVG security is enforced through they way the communication is made between the SVGServer and SVGclient classes. There are two scenarios in this direction:
In the case of a request message:

1. The SOAPEnvelope object (extracted from msgContext) will be converted to an org.w3c.dom.Document.
2. The contents of body tag within this Document will be decrypted.
3. The signature contained in the header will be verified (On the client side of ASSIDS, the body tag was signed).

The sign method of class Secure always includes an X509 certificate along with the signature it produces. The verify method of class Secure extracts this X509 certificate and returns it within a KeyInfo object (and uses it to verify the signature as well). The static variable currentCertificate is set equal to this object. The public key in the object referenced currentCertificate is later used to encrypt the response message.

4. The Document will be converted back to a SOAPEnvelope object which will replace the original SOAPEnvelope object.

In the case of a response message:

1. The SOAPEnvelope object will be converted to an org.w3c.dom.Document.
2. The contents of body tag within this Document will be encrypted.
3. The body tag will be signed.
4. The Document will be converted back to a SOAPEnvelope object which will replace the original SOAPEnvelope object.

During this type of communication the SVG security is maintained via using the class Secure which provides six static methods (encrypt,decrypt,sign,verify,…) that provide SVG Security support for the previously mentioned service handlers. Moreover, the class KeyAndCertKeyInfoResolver is instantiated within the sign method of class Secure. It is used to extract the public key contained within an X509 certificate(X509 certificates are sent along with all signatures created by ASSIDS).

```
public class Secure
{
public static void encrypt(Document doc, KeyInfo keyInfo, String entry) throws Exception { … }
public static void decrypt(Document doc, String file, String ksPass, String alias, String aliasPass) throws Exception { … }
public static void sign(Document doc, String file, String ksPass, String alias, String aliasPass) throws Exception { … }
 public static KeyInfo verify(Document doc) throws Exception { … }
public static SOAPEnvelope getDocumentAsSOAPEnvelope(Document doc,MessageContext msgContext,String type) throws Exception { … }
public static Document getSOAPEnvelopeAsDocument(SOAPEnvelope env, MessageContext msgContext) throws Exception { }
}
```

The encrypt() method encrypts the contents of the body tag using 3DES symmetric encryption algorithm. The 3DES key used is randomly generated and is encrypted using RSA asymmetric encryption before being sent to the receiver along with the encrypted body tag. The RSA public key used is extracted from the KeyInfo object referenced by the keyInfo parameter. The entry parameter is equal to the alias of the public\private key pair. The decrypt() method decrypts the contents of the body tag using a private key extracted from the java keystore file specified. Parameter ksPass specifies the password needed to access the keystore. Parameter alias is the alias of the private key and parameter aliasPass is the password needed to access the private key. The sign() method signs the body tag using a private key extracted from the java keystore file specified. All other parameters are similar to those of the decrypt method. An X509 certificate containing the public key is sent along with the signature. The verify() method verifies the signature found in the header of the SOAP Envelope using the public key that was sent along with the signature. The remaining two methods do what their names suggest. In fact, within method getDocumentAsSOAPEnvelope() is the only time that the Apache XML Security library is used by ASES. Both of these methods are based upon example code that can be found in samples\security\SignedSOAPEnvelope.java of Axis Release 1.2.

The Secure class utilizes the Apache WSS4J API for SVG Signature and Encryption[11,12]. WSS4J can be used as a library to provide an API for WS-Security processing. However, the WSS4J is primarily used to sign and verify SOAP Messages with WS-Security information. The WSS4J contains the following classes and primitives:

* WSSecurityEngine <http://ws.apache.org/ws-fx/wss4j/>
* WSBaseMessage <http://ws.apache.org/ws-fx/wss4j/>
* WSSignEnvelope <http://ws.apache.org/ws-fx/wss4j/>
* WSAddTimestamp <http://ws.apache.org/ws-fx/wss4j/>
* WSEncryptBody <http://ws.apache.org/ws-fx/wss4j/>
* WSAddUsernameToken<http://ws.apache.org/ws-fx/wss4j/>
* WSSAddSAMLToken<http://ws.apache.org/ws-fx/wss4j/>

The way the WSS4J is used in signing an SVG SOAP Message can be illustrated as follows from the client and server sides:

Using WSS4J at the client side

1. A client application wants to add a security token to a SOAP message for authentication. The client authors the complete SOAP Body, passes the SOAP Envelope along with a security token to WSS4J and asks WSS4J to attach the security token to the SOAP message. There are many types of tokens (e.g. Binary Certificate, Username Tokens, Timestamps, SAML Tokens). In our case we used the X509 certificate, the client application will simply pass the X509 certificate byte array to WSS4J. WSS4J will author the complete WSS compliant XML/SVG structure of the security token, wrap the security token inside a WSS header and return the completed WSS message back to the client application. The X.509 standard defines what information can go into a certificate and describes how to write it down (the data format). All X.509 certificates have the following data, in addition to the signature: (Version, Serial Number, Signature Algorithm Identifier, Issuer Name, Validity Period, Subject Name, Subject Public Key Information).

2. A client application wants to sign a portion of a SOAP message using a private cryptographic key associated with an X509 certificate. The client application has already added the certificate to the SOAP message inside a WSS header as a security token. The client application asks WSS4J to sign a particular portion of the WSS message using the private key associated with the certificate.
3. A client application wants to encrypt a particular portion of a SOAP message. The client application first adds the certificate of the intended recipient of the message as a WSS token. It then provides the fragment identifier and asks the application to produce XML encrypted version of the message. WSS4J produces the encrypted portion of the message and returns the completed message back to the requesting client application.

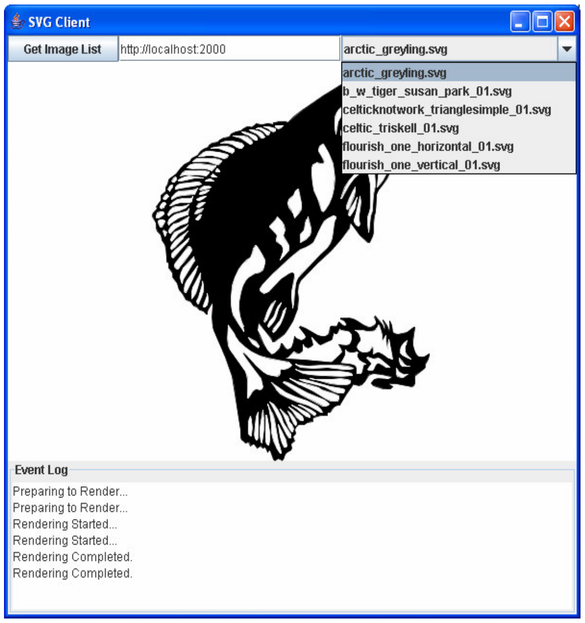
Using WSS4J at the server side

1. A server SVG application has received a WSS message with a number of security tokens. The XML/SVG firewall needs to extract a list of all tokens from the WSS message. So it asks WSS4J to extract the tokens from the WSS message. WSS4J extracts all the tokens and returns them to the requesting server application.
2. The server will need to provide a secret key to WSS4J for decryption of encrypted portions of a WSS message. If the author of the WSS message used a public key to encrypt a portion of the message, the corresponding private key is required to decrypt the encrypted portion. Similarly, if the client signed a portion of the message using a session key inside a Kerberos ticket, the server will need a password to decrypt the ticket and extract the session key, without which the server cannot verify the signature. Therefore, the server may provide a secret to WSS4J corresponding to each token in a WSS message.
3. The server also ask the WSS4J to provide a list of all XMLDS signatures in the WSS message. After WSS4J has provided a list of all signatures, the server SVG application ask WSS4J to process any individual signature.
4. Similarly, the server also ask WSS4J to provide a list of all encrypted elements in a WSS message. After WSS4J has provided a list of all encrypted elements, the server application can ask WSS4J to decrypt any individual encrypted element.

CONCLUSION

Traditionally Web Services have been deployed in client-server like settings. Services are generally hosted within specialized containers and service requestors directly access these services. With this type of approach, security is basically achieved via the transportation layer (e.g using SSL). The major downside to this approach is that it can't protect documents outside the network the transport layer safeguards. However, when a distributed middleware substrate, such as Apache Axis and Web Services leverage each other's capabilities the resultant system can facilitate the development. This article used the Apache Web intermediaries to implement the W3C XML Signature and Encryption specifications to protect SVG Web services. For this purpose the SVG image delivery system is composed of two Apache Axis components (SVGClient and SVGServer) and incorporates variety of APIs that helps in signing/verification, encrypting/decrypting and for rendering SVG images. The developed system is part of an ongoing research to construct a secure SVG medical imaging consultation system[13]. There are many other challenges and concerns remain to be answered: (1) What are the mechanisms to discover and invoke Web Services securely and dynamically using a UDDI Client, for example and (2) What are the mechanisms for optimizing the workflow performance by choosing Web Services from those available.

Figure 4 shows a screen shot from the developed AXIS SVGClient.


SVG Client
Get Image List
http://localhost:2000
arctic_greyling.svg
arctic_greyling.svg
b_w_tiger_susan_park_01.svg
celticknotwork_trianglesimple_01.svg
celtic_triskell_01.svg
flourish_one_horizontal_01.svg
flourish_one_vertical_01.svg
Event Log

```
Preparing to Render...
Preparing to Render...
Rendering Started...
Rendering Started...
Rendering Completed.
Rendering Completed.
```

Fig. 4: A screen shot from the developed AXIS SVGClient

REFERENCES

1. Vincent Hardy, 2003. Using SVG to create compelling user interfaces for web services. XML Europe 2003 Conf., London, England May 5-8. http://www.idealliance.org/papers/dx_xmle03/papers/02-04-05/02-04-05.pdf

2. Lican Huang *et al.*, 2005. Dynamic Invocation, Optimisation and Interoperation of Service-Oriented Workflow, Cluster Computing and Grid (CCGrid2005), May 9-12, Cardiff, UK.
3. Paul Prescod, 2003. Anatomy of dynamic SVG web services. SVG Open 2003 Conf., Vancouver, Canada, Jul. 13-18. http://www.svgopen.org/2003/papers/AnatomySVGWebServices/
4. Evans, S. and O. Dowling, 2002. Is SSL enough security for first-generation web services? Published at WebServices.org <http://www.webservices.org/index.php/article/articleview/529/1/1/>, Jun. 18. <http://www.vordel.com/news/articles/02-07-18.html>
5. Gopalakrishnan, U. and R.K. Ravi, 2003. Web services security: Part 1. IBM Res. J., 25 Feb., http://www-106.ibm.com/developerworks/webservices/library/ws-sec1.html
6. Microsoft-IBM Joint White Paper, Security in a Web Services World: A Proposed Architecture and Roadmap, April 7, 2002, http://msdn.microsoft.com/library/default.asp?url=/library/enus/dnwssecur/html/securitywhitepaper.asp
7. Fox, G., S. Pallickara and S. Parastatidis, 2004. Towards flexible messaging for SOAP based Services. SCI2004 High Performance Computing, Networking and Storage Conf. Pittsburgh, PA, USA, Nov. 6-12.
8. Qiu, X., B. Carpenter and G. Fox, 2003. Collaborative SVG as a web service. SVG Open 2003 Conf., Vancouver, Canada, Jul. 13-18.
9. Gibbs, K., B.D. Goodman and E. Torres, 2003. Create web services using apache axis and castor. IBM Res. J., 30 Sep., http://www-106.ibm.com/developerworks/webservices/library/ws-castor/
10. Leung, T.W., 2004. Professional XML Development with Apache Tools: Xerces. Xalan, FOP, Cocoon, Axis, Xindice. Wrox Press.
11. Siddiqui, B., 2003. Using XSS4J for XML encryption. O'Reilly Web Service J., Nov. 25. http://webservices.xml.com/pub/a/ws/2003/11/25/jwss.html
12. Siddiqui, B., 2004. Signing message with XSS4J. O'Reilly Web Service J., Jun. 02. http://webservices.xml.com/pub/a/ws/2004/06/02/wss4j.html
13. Mohammed, S. and J. Fiaidhi, 2005. Developing secure transcoding intermediary for SVG medical images within peer-to-peer ubiquitous environment. IEEE 3rd Ann. Conf. Communication Networks and Services Research Conf. (CNSR2005), Halifax, Nova Scotia, Canada, May 16-18.