Experiences Implementing Interoperable SOA in a Security-Conscious Environment*

Scott McGregor, Thomas Russ, Norman Wilde, James P. Gabes, Willard Hutchinson, Daniel Duhon, Ali Raza

scottwmcgregor@gmail.com, thomas.a.russ@gmail.com, nwi@uwf.edu, jgabes@acm.org, willard.hutchinson@me.com, {dwd5|ar42}@students.uwf.edu

Executive Summary

This report describes the results of an academic case study investigating the implementation of secure, interoperable Services Oriented Architecture (SOA) based web services. The report attempts to capture some of the issues faced in implementing security in interoperable SOA and to recommend some ways to overcome the inherent challenges facing secure SOA implementation.

This study was conducted as part of a two-semester capstone course in the Software Engineering Masters program at the University of West Florida. Seven different implementations were created of a service offering airline reservations; three of the implementations used the Microsoft Windows Communication Foundation (WCF) technology package, three used the Linux/Apache/MySQL/PHP (LAMP) package, and one the Java/NetBeans/Glassfish package. To demonstrate interoperability, all implementations conformed to a previously defined WSDL interface; conformance was checked using an automated functional tester. Security was provided in the form of an authentication service and identity information and message data were exchanged using SOAP messages in standardized formats. All services were deployed to the cloud using Amazon's Elastic Compute Cloud (EC2) platform.

The conclusions of the report provide a number of lessons learned touching on the security process, good design of service code and the relative ease of use of the three technology packages.

-------

1. Introduction

Today’s business environment demands security and interoperability between different information technology systems and programming languages. Services Oriented Architecture (SOA) is one approach to providing interoperability within a system of systems. Web Services (WS), as an implementation of SOA, provides a mechanism whereby businesses and other organizations can provide platform-agnostic services within their own domains or to outside consumers. Because security becomes more difficult as service providers open up service interfaces in the form of Web Services Description Language (WSDL) documents, there must be a fundamental and constant commitment to security practices from the very beginning of development. In order to develop these kinds of open and available yet secure systems with standardized interfaces that cross organizational boundaries a recommended strategy involves several key principles:
Considerations of contract-first (or WSDL-first) development
A focus in development on interoperability from the ground up
Reasonable and portable security policies that are appropriate for the business domain and the level of risk

This endeavor is complicated by several factors:
- There exists a significant lack of coherent, consolidated documentation on getting one technology to interact or inter-operate with another, at least among the technology types explored in this study.
- Data type consistency varies widely between languages. Though XML’s standardized types can ameliorate this difficulty, much work remains to be done by individual developers in order to comply with standards. Worse, data type errors may not manifest themselves recognizably, complicating the troubleshooting of interoperability issues.
- Even when developing contract- or WSDL-first, some languages present widely varying run-time WSDLs making validation and testing problematic.
- Automatic code generation utilities, while helpful in some respects, can complicate efforts to program WSDL-first by the complexity of the generated code.
- Perhaps most importantly, good security and interoperability take a lot of effort and time to develop.

Presented in this paper are experiences from an academic case study carried out during a two-semester Masters in Software Engineering capstone course. The authors certainly do not claim to have all the answers to providing secure, interoperable web services, but the hope is that the experiences herein described specifically focusing on security in SOA may help other developers and security professionals to avoid some of the pitfalls and difficulties of implementing securely developed, interoperable web services.

2. Background

2.1 Services Oriented Architecture and Contract-First Development

Services Oriented Architecture (SOA) is an approach to constructing large software solutions that are really "systems of systems" that may tie together components owned by different organizations, implemented in different languages, and running on different platforms. The components are exposed as loosely coupled services:

Technically, a service is a description of one or more operations that use (multiple) messages to exchange data between a provider and a consumer ([6], p. 300).

This kind of architecture is needed because more and more processes in business, technical and military domains require integrating activities provided by different organizations. Thus a business partnership between a company in China and one in the United States may require
sharing data on parts inventory, or a weapons system may require tying intelligence data from one vendor’s system to navigational data from another. Each organization has its existing software which must communicate to complete the process. It is impractical to require standardization of programming language or environment across the different organizations, so integrating activities necessarily requires interoperating software.

The most common way of implementing SOA uses the Web Services standards to provide standardized interfaces to reusable service components. Often, this is accomplished using a Web Services Description Language (WSDL) document, which is an XML document describing the interface and how web services can be consumed.

In practice, there seem to be two main approaches to web services development—code first and contract first. In code first development, the WSDL is generated automatically from existing code, either as a final production step or dynamically at runtime. In contract first development, the WSDL is generated (or provided) before development and code generation begin. Thus software engineers design code that conforms to the expected contract. The advantages of contract first development are that a standard WSDL can be used to develop concurrent, competing, or complementary services regardless of deployment platform (operating system, programming language, HTTP server, etc.). It is thus more likely that the resulting services will interoperate smoothly since standards compliance has been enforced from the beginning of development and there is less chance that non-portable data types and processing styles will slip into the code. The disadvantage of contract first development is that developers must work to conform code to specific WSDL interfaces, which may require more time and effort with less reliance on vendor-provided tools.

2.2 Security in Services Oriented Architecture

Since SOA by design opens up infrastructure interfaces in order to be more interoperable and platform-independent, the need for appropriately implemented security is even more critical than in more closed or “walled garden”-type systems. As with security in any system, security in SOA is concerned with confidentiality, integrity, and availability. Web services are open, accessible from outside firewalls, and publish their interfaces. Additionally, they are designed to be called by external programs which therefore subjects them to automated attackers and other security concerns. Much as the original designs of the Internet were concerned not with security but with open communication, so SOA is focused on interoperability and clear data transfer, which can complicate the process of securing SOA infrastructure and design. This obviates the critical need to consider and implement security at the very beginning of the design and development of a SOA system.

As an initial step, care must be taken to ensure that the underlying operating system, web server, and database are secured against attack to avoid end runs around fortifications of the service itself. Good deployment instructions and deployment scripts are important tools in this process for two reasons. First, the scripts and instructions can be audited by experts for their efficacy. Second, they can be improved over time to make the system more secure.

Security must be built into the system and should not be considered as an afterthought in the development process. Just as requirements are accompanied and demonstrated by use cases, they should also be accompanied by abuse cases, which are developed and improved as the system is designed, developed, tested, and fielded. Through each of these phases there are processes, tools, and checks to be performed [9]. Developers who are informed of and trained
in proper security practices write and develop code in entirely different ways from developers who are either unaware of or inattentive to the demands of secure programming [1].

A common misconception in web services is that the use of transport layer security (TLS, often referred to by the legacy protocol name SSL) makes everything safe. Transport encryption is considered by the WS-I profile to be a prerequisite to having a secure web service [10], but is not sufficient security by itself. Since web services are often compositions of other services, it is common for an intermediary to be involved in a transaction. In scenarios in which the intermediary should not have access to the data passing through, transport encryption is incapable of keeping data confidential. Also, should the intermediary alter the data, the integrity of the data cannot be guaranteed between the beginning and endpoint. Thus other measures in addition to transport encryption are required to maintain confidentiality and integrity.

Authentication helps to maintain confidentiality by keeping out unwanted users. In SOAP, authentication is typically performed by attaching a security header to the message being sent. The structure of the header is governed by the WS-Security standard [4]. The details of each language’s approach to creating, accessing, and manipulating SOAP headers vary, but the final version in all cases must conform to the WS-Security standard in order to inter-operate.

Security aspects such as encryption, hashing, validation, preventing code injection, and following good deployment practices all take time to build in. Architects and developers must attend to these details in the same way that functional requirements are addressed, so budgeting time for these activities must be part of the early planning stages to avoid schedule slips or feature reductions to meet deadlines. Encryption and hashing can be particularly time-consuming because of subtle differences in the implementations of algorithms between languages and platforms. Further, encryption and decryption by design provide little feedback to developers as they test data-- it either works or it doesn’t. Combined with SOA web services’ “black-box” behavior (i.e., data come in, something happens which can’t be seen, and different data come out), secure interoperable web services take more time to implement and debug than systems that exist within a single technology domain.

2.3 The Case Study Context

This case study was performed as a practical academic exercise at the University of West Florida in the Software Engineering Masters program’s two-semester capstone course (COT6931, Fall 2011 - Spring 2012 semesters). One instructor and seven students participated in the course; six students completed the two semesters. All students were part time with full time jobs, mostly in computer-related fields. Students were geographically distributed so all interaction was on-line using the UWF’s Elluminate online conferencing and eLearning course delivery as well as email and Google Docs. The experience level of the students varied, but only one had extensive previous experience with services computing. Accordingly most of the first semester was taken up with background reading in SOA and small scale service programming and deployment exercises. The culminating exercise was the creation of a SOA-based airline reservation system which contained several basic airline providers as well as other types of web services.

Students picked technology packages in accordance with their interests and career objectives. Three chose the Microsoft - Windows Communication Foundation package, three the Linux™ - Apache™ - MySQL™ - PHP (LAMP) package, and the instructor took the Java™ - NetBeans™ - GlassFish™ option. Each participant did contract-first development of a basic airline
reservations service using their assigned package. All services were deployed to the Amazon EC2™ cloud platform [13]. An Amazon course grant provided each participant with an account credit to cover Amazon Web Services costs.¹

The instructor provided a functional test driver (using the Java package) to exercise any of the airline reservations services. One team also provided an authentication service that would let an airline service authenticate its consumers; however, in the time available only two of the airline reservation services were able to successfully support use of the authentication service.

The starting point for contract-first development was an interface description of the basic airline service. This interface description consisted of one WSDL 1.1 file which included two XML Schema Description (XSD) files implementing an airline reservations data model. The WSDL and XSDs were developed using the WSDL editor of the Eclipse™ SOA Platform (Helios version) [7]. Four operations were defined:

1) findSeats - locate unbooked seats on flights matching origin, destination, and schedule constraints
2) bookSeat - reserve one seat on a specified flight for a specific passenger
3) getPassengerList - get the list of passengers on a specified flight
4) resetBackend - reset the database to an initial state to facilitate reproducible testing

The WSDL used the document/literal wrapped pattern [12] and Eclipse’s validation tool checked conformance with the WS-I interoperability profile [10].

All seven versions of the basic airline service were coded independently but students and the instructor collaborated in problem solving to address the many issues that arose in service implementation and deployment. By the end of the course all seven services deployed successfully to EC2 and passed the instructor’s functional test. As the semesters progressed, students recorded the issues they encountered and contributed experiences for incorporation in this report.

2.4 The Chosen Technology Packages

Each technology package involved in this project had three basic components: a programming language, an Integrated Development Environment (IDE), and a web server. In the following sections we describe how each package facilitated contract-first service development. We also discuss how to use each package’s facilities to implement security requirements since that was a particularly time-consuming issue in each case.

The approach to security in the project was in accordance with the WS-Security standard, which dictated the use of headers on the SOAP message for the purposes of authentication and confidentiality/integrity of data. Each technology package, therefore, needed to provide some approach to constructing and dealing with SOAP message security headers. Data extracted from or added to security headers had to be portably encrypted and decrypted either within the code itself (natively), or via another mechanism (e.g., MySQL).

¹ Linux is a trademark registered to Linus Torvalds. Apache is a trademark of the Apache Software Foundation. MySQL, Java, NetBeans and GlassFish are trademarks of Oracle Corporation and/or its affiliates. Amazon EC2 is a trademark of Amazon.com, Inc. Eclipse is a trademark of the Eclipse Foundation, Inc.
2.4.1 Microsoft - Windows Communication Foundation

2.4.1.1 The Technology Package

For Microsoft developers, the Windows Communication Foundation (WCF) is the preferred method for providing web services using either C# or Visual Basic (VB). The underlying desire of Microsoft is to make creation, deployment, and maintenance of WCF applications as painless as possible in the Visual Studio development environment; however, the focus seems to be first and foremost on enabling functional web services within the Microsoft garden of software, not necessarily on interoperability with other languages and systems.

Within the Microsoft development environment, WCF and Visual Studio are tightly coupled, providing a specific set of capabilities. These include a set of pre-defined project types to ease getting started, a function to specify an existing service endpoint to allow Visual Studio to automatically generate a proxy, and the ability to locally generate a client to test new WCF-based services. IntelliSense, a powerful Microsoft development tool, is also available as a way to allow statement completion when creating code elements in most file types.

Adopting a WSDL-first approach as part of this project, a set of tools were used to facilitate translation of the contract definition files into a project structure within Visual Studio. Initially, a Microsoft provided utility called SVCUTIL.exe was used to translate a WSDL file into the needed class structure within the project. As the project evolved and additional files containing XSDs were added, it became necessary to transition to a VS2010 add-in called WSCF.blue. This tool supports C# and VB code generation from a combination of both WSDL and XSD files. In addition, it has the ability to report errors in the .WSDL files via its output window. [16]

Two things were observed with the use of both tools: one was the creation of classes that weren’t needed in the context of this project, and the second was the auto-creation of configuration and properties files. In some cases, assumptions were made by the development environment/tool which went unverified or accepted by the programmer. The details of these class additions added quite a bit of confusion about whether they were really necessary to the overall implementation. Additionally, the creation of settings that were nested in multi-layered properties pages added a layer of frustration and had the potential to be a single point of failure for successful deployment of the service application.

The WCF test client, which is a tool within the Visual Studio development environment, allows users to input test parameters to the service and view the response from the service. It is a very useful tool for verifying proper service operation; however, some problems arose while using it to test deployed operations. Once deployed, services often didn’t work as they had with the test client, indicating that the problems were not necessarily in the services themselves, but with the configurations under which they were being called. The actual problem was often in either the service configuration files, or in the configuration settings that resided on the EC2 instance where the service was deployed, which the WCF test client was unable to detect.

2.4.1.2 Security

WCF relies on the System.Web.Services.Protocols.SoapHeader class to build headers. The framework uses .NET reflection to construct the XML of the header, although the developer can take direct control of the XML by implementing the System.Xml.Serialization.IXmlSerializable interface [3]. The developer injects headers by adding attributes to the methods and properties to the code.
First, the developer creates a class that inherits from SoapHeader, adding public properties to contain the values in the header and giving them XmlElement attributes to control the name of the nodes in the header’s XML. The developer opens the wizard-generated class that inherits from System.Web.Services.Protocols.SoapHttpClientProtocol, adding a [System.Web.Services.Protocols.SoapHeaderAttribute("Header")].attribute to each service method. The name “Header” must match the property name given to the instance of the header class that was created to encapsulate the header. An example of a header class is given below.

```csharp
{
    [System.Xml.Serialization.XmlElement("UsernameToken")]
    public UsernameToken _user { get; set; }

    [System.Xml.Serialization.XmlElement("BinarySecurityToken")]
    public string _hash { get; set; }

    public AuthHeader() { }
    public AuthHeader(string token, string password, string hash)
    {
        _user = new UsernameToken(token, password);
        _hash = hash;
    }
}
```

In this project, no WCF web service was able to successfully implement security authentication via the authenticator service. This was largely due to time constraints—the demands of interoperability proved much higher than anticipated and Microsoft developers spent a larger share of time working to get services that passed functional tests.

### 2.4.2 Java - NetBeans - GlassFish

#### 2.4.2.1 The Technology Package

For development and deployment of services in Java the following tools were used:

- **Programming Language**: Java with Java Development Kit JDK 1.6 including the Java API for XML Web Services (JAX-WS)
- **Integrated Development Environment**: NetBeans IDE 6.9.1
- **Application Server**: GlassFish Server 3
- **Hardware and Operating System**: Amazon Web Services Elastic Compute Cloud (EC2) with a t1.micro Linux instance

Though there were some false starts, contract first development in this environment turned out to be fairly straightforward starting from the WSDL and XSD files provided. The basic steps were:

1. In NetBeans, from the file menu choose “new project”, “Java” and “Web Application” and follow the steps in the wizard to create the new project.
2. Right mouse on the project and choose “new”, “web service from WSDL”, browse to the WSDL file and finish the steps in the wizard.
3. NetBeans then runs the \texttt{wsimport} tool to create a number of Java class files representing the data types defined in the WSDL and associated XSDs. It also generates a shell service implementation class file for the service itself, containing one method for each service operation.

In principle, all that remained was to fill in the code in each one of the methods in the implementation class. In practice, it is inadvisable to put any more code than necessary directly into the service implementation class since it may be regenerated by the IDE after changes and any work done in the file could be lost. Similarly it was found that using the data types generated by \texttt{wsimport} directly was problematic since they may also be regenerated so code that references them could be impacted. To localize the effects of changes, the layered architecture shown in table 1 was adopted.

Table 1 - Suggested Layered Design of the Service

<table>
<thead>
<tr>
<th>Layer</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Internet Interface Layer</strong></td>
<td>Each method:</td>
</tr>
</tbody>
</table>
| Consists of the \texttt{wsimport} generated service implementation class with one method for each operation offered by the service | 1. Gets and saves the web service context containing configuration and other information needed for handling message headers.  
2. Calls the corresponding method in the Implementation Layer |
| **Implementation Layer**     | Each public method handles:                                                    |
| Consists of one class with one public static synchronized method for each service operation plus numerous private helper methods. | 1. Throttling to limit the server resources consumed in case of accidental or deliberate (denial of service) overloads.  
2. Logging of each call to one of the service operations.  
3. Authentication of the client, if the service is running in secure mode.  
4. Conversion between the data types used in the service WSDL and internal data types.  
5. Invoking methods in lower layers to carry out each operation.  
6. Handling all exceptions found during processing. |
| **Business Logic and Data Layers** |                                                                                         |
| Numerous classes providing the detailed code for implementation of each operation. |                                                                                     |
This architecture provided the ability to handle WSDL/XSD spelling corrections and a change in a Date/Time data type with almost no code change. This approach also made the service more modular and easier to modify.

Deployment of the service to GlassFish was also straightforward. For testing on a workstation NetBeans can be configured with a local copy of GlassFish so that any change to the code is immediately compiled and pushed out to the server. NetBeans allows deployment to a remote server (e.g., Amazon EC2) by simply right mouse-clicking on the project and choosing the “clean and build” menu option. This creates a war archive file with everything needed on the server. GlassFish’s Admin Console or command line interface may then be used to upload and deploy the war file.

2.4.2.2 Security

In the Java JAX-WS environment there are several ways of manipulating message headers. The simplest is to add the specification of the header to the WSDL and then regenerate the service code. The methods in the generated code then have parameters for accessing header contents. However specifying the security header in the WSDL could potentially provide an adversary with very useful information about the security protocols in use so in this project the header was described in separate documentation circulated to service developers.

This project used a callback mechanism from JAX-WS that gives a more general way of handling SOAP headers. The programmer attaches to the service a HandlerChain consisting of one or more objects implementing the SOAPHandler interface. The handleMessage() methods in these objects will be invoked during processing and at that time the message headers may be created or accessed. The procedure for adding the HandlerChain is somewhat different for the client and server sides (see [5] sections 6.10, 6.11 and 7.17).

2.4.3 Linux - Apache - MySQL - PHP

2.4.3.1 The Technology Package

The developers in this project used standard PHP, MySQL for database management, and Apache for a web server, which was generally deployed for production on Linux EC2 instances as part of Amazon’s cloud. As of 2004, PHP provides native support for SOAP messages via SoapServer and SoapClient objects and methods. Over the last several years, this support has matured to match or exceed earlier add-on solutions like NuSOAP and other alternatives, which were largely created to fill the gap in PHP’s early lack of SOAP functionality.

The development of contract-first PHP code in this project was relatively straightforward and simple. The constructor for SoapServer objects in PHP allows for the original WSDL file to be passed as one of the parameters, which then requires the developer merely to write handling code for the appropriate methods described in the WSDL. Beyond that, PHP handles SOAP calls to the service on its own.

To test the passing of SOAP messages, a tool called soapUI was found to be very useful. When provided the WSDL Uniform Resource Locator (URL), soapUI was able to pass SOAP message requests to the service and provide instant feedback as to the content of the SOAP message responses. This tool was very useful in troubleshooting deficiencies in data types being exchanged between consumer and provider. SoapUI served as an initial test client before more robust testers were available.
PhpMyAdmin, MySQL Workbench, a command-line, or some combination of the three were used as the interface to the MySQL database. MySQL Workbench provided an easy to use graphical interface used for query and management purposes. PhpMyAdmin also provided a similar interface via a web browser.

2.4.3.2 Security

In PHP, headers are built as an array of SoapVar arrays. The PHP SOAP extension turns these into an XML tree. In this project a PHP client class inherited the PHP SoapHeader class and overloaded its constructor as follows.

```php
class WsseAuthSoapHeader extends SoapHeader
{
    function __construct ($clientUsername, $clientPassword, $callerUsername, $callerPassword, $ns = 'http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd')
    {
        $securityItems = array();
        $clientItems = array();
        $clientItems[] = new SoapVar($clientUsername, XSD_STRING, NULL, $ns, 'Username', $ns);
        $clientItems[] = new SoapVar($clientPassword, XSD_STRING, NULL, $ns, 'Password', $ns);
        $securityItems[] = new SoapVar($clientItems, SOAP_ENC_OBJECT, NULL, $ns, 'UsernameToken', $ns);

        $callerItems = array();
        $callerItems[] = new SoapVar($callerUsername, XSD_STRING, NULL, $ns, 'Username', $ns);
        $callerItems[] = new SoapVar($callerPassword, XSD_STRING, NULL, $ns, 'Password', $ns);
        $securityItems[] = new SoapVar($callerItems, SOAP_ENC_OBJECT, NULL, $ns, 'UsernameToken', $ns);

        $hash = hash('sha256', $clientUsername . $clientPassword . $callerUsername . $callerPassword);
        $securityItems[] = new SoapVar($hash, XSD_STRING, NULL, $ns, 'BinarySecurityToken', $ns);

        $securitySV = new SoapVar($securityItems, SOAP_ENC_OBJECT, NULL, $ns, 'Security', $ns);
        parent::__construct($ns, 'Security', $securitySV, FALSE);
    }
}
```

PHP doesn’t yet offer a standardized way of reading a SOAP message header, so the service developer has to use the DOMDocument XML methods to read the header and extract the necessary data [8]. References to $this in the following example assume that the main elements of the header are properties of the class containing “parseHeader()”.

```php
public function parseHeader()
{
    $dom = new DOMDocument();
    if ($dom::loadXml($source))
    {
        $xpath = new DOMXPath($this);
```

$this->hash = $xpath->evaluate($xPathPrefix . 'ns2:BinarySecurityToken');

$userTokenXpath = $xPathPrefix . 'ns2:UsernameToken[position()=1]/';
$this->clientUsername = $xpath->evaluate($userTokenXpath . 'ns2:Username');
$this->clientPassword = $xpath->evaluate($userTokenXpath . 'ns2:Password');

$userTokenXpath2 = str_replace('1', '2', $userTokenXpath);
$this->callerUsername = $xpath->evaluate($userTokenXpath2 . 'ns2:Username');
$this->callerPassword = $xpath->evaluate($userTokenXpath2 . 'ns2:Password');
}
}

From there it was a matter of dealing with encrypting or decrypting data as needed using the appropriate key. PHP’s mcrypt library was sufficient for doing the encrypting and decrypting locally and other PHP methods provided utilities for hashing and base64 encoding/decoding.

3. The Security Process

In this project, security was addressed to the degree possible within the given time constraints. Some early training was provided to give developers a chance to attend to the key code-based security vulnerabilities of validation and code injection. Additionally, a service was developed to demonstrate interoperability issues that arise during authentication across disparate technical platforms.

Authentication was performed by inserting a header containing authentication credentials consisting of two UsernameToken elements and a BinarySecurityToken element [4]. The UsernameToken enabled two endpoints to send messages through an intermediary while ensuring that the sender and the recipient were who they claimed to be. Two UsernameTokens were included: one was for the client and the other was for the intermediary. Each Password field of the UsernameToken held a one-time password encrypted by a shared key held only by the authenticator and the client. The client passed these credentials in encrypted form to the intermediary, which could then forward them to the authenticator to ask “is this actor who they claim to be?” without the intermediary being able to gain access to the credentials. The BinarySecurityToken element held a hash of the credentials passed in to enable detection of any corruption of the data in transit. The encryption step provided an additional data integrity check.

The approach taken did not encrypt the messages, but did serve to illustrate the techniques used to encrypt messages, specifically the use and structure of a Security node in the header, encryption of header contents, and hashing. Applications with confidential messages would need to encrypt the message in the envelope as well as the header. This project uses AES/128/ECB/PKCS7 mostly because there were algorithms on hand in the technologies chosen; however, it is recommended to encrypt with public key encryption or at least with a stronger version of AES such as Cipher Block Chaining. Public key encryption has the advantage of ensuring that the identity of the sender is known because the central server does not have to hold the sender’s private key.
Authentication and message integrity are only part of the security picture. The code behind the service is where exploits take place. While good coding practices are documented in many different places, few developers are aware of the practices and their importance. Training the developers in this project increased their adherence to the guidelines such as those described by the United States Department of Defense Application Security Checklist [2]. Since code review was only performed at the end of the project, only limited improvement in developers’ security awareness was gained. Discussions at the end of the project revealed that more improvement might have been achieved through the use of midpoint code reviews. This again highlights the fact that the development process directly affects software security.

Although there are many aspects to good coding practices, two in particular are both critical and underutilized. These two practices are input validation and prevention of code injection. Automated attackers can throw great volumes of data at a public interface, trying combinations that no friendly consumer of the service would ever deliberately attempt. Systems must also account for the fact that even friendly consumers might have defects that would also cause problems for the service. All data coming in from a user should be filtered through a validation step. Another type of deficiency is the potential for code injection, whether xpath, path, or SQL injection. Developers should use third party parsers and XML firewalls for XML instead of writing their own, should not copy user-completed data directly into paths, and should use both prepared statements and stored procedures to perform database actions. About forty other such rules exist and each one should be well understood by developers.

To understand the many rules and guidelines that must be followed to write secure code, developers need training. There should be training at the start of projects, on initial hire, and at various points during the development cycle. Ideally, this training is informed by having a security expert review the code in development to identify team weaknesses to eliminate them. Also, needed security patterns should be identified, codified into the architecture, and communicated to the team as part of the training. One example of training in secure coding practices is the use of stored procedures for SQL. In this project, before any developer had created a database, developers were instructed to make sure “the application uses prepared or parameterized statements, is not vulnerable to SQL Injection, does not use concatenation or replacement to build SQL queries, and does not directly access the tables in a database.” Despite that wording, a majority of developers wrote their database code as embedded SQL until advised otherwise in an early informal peer review. This experience demonstrates that peer review should be performed very early in the development process in addition to the customary final peer review performed after software is fully functional to help foster each developer’s ability to write more secure code. Developers in this project made these mistakes more out of ignorance than deliberate action, demonstrating that there remains a critical need for adequate security training for developers in today’s information technology environment [1].

4. Identified Issues and Recommendations

Contract-first development of secure, interoperable web services presents some very distinct challenges. In the course of this project, several technology-specific issues were discovered as well as some general lessons learned. Herein are presented explanations of those issues and some possible methods to mitigate such issues in future projects.
4.1 Microsoft

One of the newer technologies in the Microsoft tool box for the development of Interoperable services is Windows Communication Foundation (WCF). This technology proved to have a very steep learning curve that was mostly due to the lack of resources available in regard to the scope of this project (i.e. using the Microsoft technologies to create services that are interoperable with other technologies).

The WCF framework provides a developer with powerful tools that can be used to develop interoperable web services. One such tool was the WCF Test Client that was used extensively in the beginning stages of the project. The WCF Test Client allowed us to test calls to the service while in development without having to build a client to do so. However, due to coding changes that were required to enable communication with the Java test client, the WCF test client was rendered useless and we were unable to utilize that resource.

A second tool that was an incredible help in WCF services development was the Visual Studio 2010 built in WCF Configuration Tool. A key component to deploying a WCF Web Service is the web.config file (a file written in the markup language XML) that contains all the configuration instructions for the web service: namespace declarations, endpoint configurations, security settings, and debug assistance. While all of this can be done manually within the web.config file, the WCF configuration tool provides the programmer with a GUI to make the necessary configuration changes and automatically updates the web.config file accordingly.

4.1.1 Java Client Interoperability

It should be noted that the most prevalent interoperability issues faced by the Microsoft developers in this project mainly had to do with making WCF work with the Java technology. Very early on in development, a solution was found that worked perfectly with PHP technologies; however, when using a Java client with WCF-built services, cryptic exceptions were thrown that led the Microsoft developers to believe a DateTime issue was the source of the problem. While the DateTime format inconsistency may have been part of the issue, a more solid resolution was eventually uncovered in a namespace declaration that was essential for Java to effectively communicate with the WCF web service. This very specific code change for Java severely impacted development because it was not clear what needed to be changed initially.

4.1.2 Scarce Documentation

A lack of consolidated documentation on interoperability complicated development. This is to be expected since interoperability is important to individual programmers, businesses, and to a certain extent consumers, but is not in the interest of environment providers (like Microsoft) who are more motivated to keep programmers and consumers within their walled gardens of service. Though there was a modicum of documentation support for java and PHP, there was little to no information available in regard to making WCF web applications interoperable with other technologies. To overcome these challenges, much study and time were needed to adequately solve many of the issues encountered. While there was plenty of documentation on web services using WCF within the Microsoft environment, there was very little on getting WCF to interoperate with other languages and types of web services.

4.1.3 Data Types
Because of the contract-first approach of this project, the data types specified in the XSD files did not coalesce well with .NET. The first encountered issue was that there was no way to randomly generate a number that was of type “long”. To overcome this problem, a separate “RandomLongNumberGenerator” class had to be created as a helper class in the application for the booking number.

Another major issue realized was that the DateTime data type is very specific in .NET: DateTime is an actual data type that did not return the date and time in the format specified in the airlineData.xsd file (i.e., standard XML format). To compensate for this, Microsoft’s DateTime value had to be converted to a string value, and returned as a string to the Java client to maintain interoperability with that particular technology.

### 4.1.4 The Difficulty of .NET
Many of the issues experienced by the Microsoft developers were due to general unfamiliarity with the technology as a whole. The libraries, namespaces, and rich classes caused issues with learning how to use the technology to accomplish the goals of this project. Everything from coding, XML, SQL Server, and C# was not an easy task. It became apparent that developing a WCF web service contract-first was not the general way this technology was intended to be used due to the limited information available on the topic (see 4.1.2).

### 4.2 Java
The Java environment proved to be generally supportive of contract first development. The wsimport tool takes a WSDL and creates a set of “bean” classes representing the data structures with getter and setter methods for the important fields. The classes created for the service itself are rather more obscure and it is sometimes necessary for the programmer to work with these classes when, for example, manipulating headers, logging SOAP messages, and changing the endpoint where a service is deployed. Fortunately most of the necessary tricks of the trade are published in works such as [5] or may be found online.

#### 4.2.1 IDE Tester Challenges
One difficulty with the NetBeans IDE was that its built in “test web service” option never seemed to work. In the projects view it is supposed to be possible to open the “Web Services” tab, click on a specific service, and choose “Test Web Service”. An html input form was created and opened, but the SOAP request and response never seemed to be generated correctly. The easy work-around was to use Eclipse’s tester instead.

#### 4.2.2 Date and Time Types
An annoying Java language issue was the convoluted handling of dates and times. In SOAP messages the XML standard date and time types are used. Java does provide a class called javax.xml.datatype.XMLGregorianCalendar which seems to be mapped to several of the XML types. In the Java library the XMLGregorianCalendar class has little support so the code became rather cluttered with complex conversions to and from Java’s preferred java.util.GregorianCalendar class.

#### 4.2.3 Security Headers
Another problem was in generating and accessing security headers on SOAP messages. As mentioned previously, the eventual solution was to attach a message HandlerChain to the service object. It did, however, take a long time and many false starts to figure this out since the
javadoc documentation and online comments were particularly obscure. It was not possible to fall back on exploratory test examples since the API for SOAP messages consists mainly of interfaces, not classes. The actual classes and objects come from the GlassFish application container at runtime. Thus SOAP message objects can only be created inside GlassFish, which makes debugging and direct observation difficult because GlassFish is intended for production, not development.

4.2.4 Encryption and Decryption

Difficulty was also encountered with encryption which was needed for the security headers on SOAP messages. The Java Cryptography Architecture provides a very flexible, but very complex, set of encryption capabilities. The architecture is very general so that in each environment there may be different Provider classes, each of which knows how to handle different variants of the many different cryptographic algorithms. For the security header, it is essential that encryption is following exactly the same variant on the service producer and service consumer side; otherwise the content will not be decoded correctly.

What is not clearly documented is which variants of which algorithms are guaranteed to be available. Thus initially the Advanced Encryption Standard (AES) algorithm with 256 bit keys was adopted for this project, and most general references indicated that this combination should be widely available. However when this combination was attempted using Java the code yielded incomprehensible error messages. The eventual discovery of an obscure table at the end of an obscure “Standard Algorithm Name Documentation” page (http://docs.oracle.com/javase/6/docs/technotes/guides/security/StandardNames.html) yielded a clue: in this table only 128 bit keys were mentioned and thus one had to deduce that 256 bit keys were NOT supported. This led to the reduction of key sizes for this project which enabled java encryption and decryption to work with the authentication service. It is the unfortunately typical state of documentation on the web that it can be very hard to answer simple but important questions such as which encryption algorithms and key sizes are actually available on which platforms.

4.3 PHP

For the most part, PHP was the friendliest of the languages for contract-first development in this project. In order to develop a web service using PHP based on an existing WSDL, the programmer needed only point an instance of a SoapServer object to the existing WSDL file, after which it automatically handled SOAP calls to the service by referencing the appropriately mapped method calls. The run-time WSDL would simply be the originally specified WSDL, which also simplified evaluation of the run-time interface. The documentation for SOAP in PHP exists in the online PHP manual; however, it can be incomplete or scattered at times.

4.3.1 Case-sensitivity

Case-sensitivity issues proved to take a lot of time for PHP development. Even though PHP is a case-sensitive language, in some instances, a PHP method would work with the wrong case on one platform, but not work when the same code was ported to another. This was found to be true of both methods that implement a SOAP operation defined in the WSDL as well as methods of PHP classes. Specifically, automatic translation of some methods from the uppercase method name to the lowercase was seen in Windows instances of PHP, but when the code was copied to a Linux or Mac instance, there were execution errors in the code.
There were also inconsistencies in case handling within a single platform. PHPMyAdmin, which was used to export database tables, converted all stored procedures, tables, and column names to lowercase. When the resulting exported scripts were run in MySQL on Linux, the databases were incompatible with the PHP code that called the stored procedures with mixed-case names. In contrast, these scripts worked without error in MySQL on both the Mac OS and Windows. Scripts exported from Sequel Pro on the Mac worked well on all three OS implementations of MySQL because the exported SQL was mixed-case.

4.3.2 Ambiguity due to Loose Typing
In general, PHP was the more “forgiving” technology package, which had advantages and disadvantages. The obvious advantages were in quick and easy set up of services which functioned appropriately and passed tests. If implemented correctly, these PHP-based services would interoperate with other languages and types of web services with little effort on the developer’s part.

The drawback to the ease of implementation in PHP was also in its “forgiveness” however. If PHP ran across an error in parameters or datatype, the default behavior was to figure it out if at all possible, or just skip parts of the program logic if necessary and continue functioning. This sometimes had the side effect of suppressing error messages that would have made more strict technology packages fail, which made some aspects of troubleshooting problematic. For instance, some PHP methods, when passed more parameters than specified in their constructor code, would silently ignore additional parameters and continue processing. While this made for forgiving code, it sometimes made pinpointing errors across and between different technology environments difficult.

To mitigate these issues, attention must be paid to code precision and exactness. PHP has methods for warning about particular issues with code implementation which may not manifest until services are deployed outside of a test environment (specifically when they begin to interoperate with other types of services), so attention and care should be paid to heeding language warnings early in development.

Another excellent method for mitigating these kinds of problems is early and frequent code reviews by other developers.

4.4 General Interoperability Issues

4.4.1 Differing Run-time WSDLs
A discouraging aspect of contract-first development is that the runtime WSDL and XSDs generated from the final code in some languages had quite a different appearance from the original WSDL and XSDs that were the starting point. Files were split up differently, namespace definitions were moved around and namespace prefixes changed, line endings were sometimes eliminated, and in general it was very difficult to be sure that the final interface definition was the same as the starting interface definition.

The difference was important because many tweaks and adjustments were needed as the service was implemented. In this project, many such changes were required in the Microsoft environment in particular. Ideally, if the final runtime interface definition were textually identical to the original, there would be some assurance that tweaks and adjustments had not caused
any fundamental changes in the service description, however, there were so many differences that no such textual comparison was possible.

It would be very useful to have a tool that would compare service interface descriptions, perhaps by reducing them to some standard canonical form. It could then be established that service A and service B do actually have the same interface, despite any apparent differences between them. Though theoretically this “standard canonical form” is the entire point of the use of XML in general and WSDLs in particular, the realities of different software environments again complicated inter-operability.

### 4.4.2 Debugging Services with Distributed Ownership

SOA is intended to let organizations collaborate, so it is intrinsic to SOA that the producer and consumer services are likely to have different ownership. This can greatly complicate tracking down problems, however, whether caused by interoperability issues or simply by coding errors.

In debugging a classic desktop or web application, one programmer probably has access to all the code. Thus he can set up tests in which one module sends test data to another, and he can watch in a debugger or otherwise how that second module processes the incoming data. However in SOA, as we encountered in this project, the two modules may be owned by two different organizations, so the programmers involved are in different cities and possibly in different time zones.

Suppose a problem is reported of the form “When service B calls service A with the following data the result is wrong.” The owner of service B only knows that something seems to be wrong, but has no way of knowing if the cause is something wrong with the data sent or with the way that A is being invoked, or even if the message is actually reaching service A. The owner of service A cannot reproduce the test since he has no access to the code for service B and probably could not run it if he had it since its language and environment may not be used at his organization. Probably the programmer of service A is not even at his desk when B is running tests so setting up a joint debugging session is likely to cause costly delays.

Problems of this nature were encountered at several points in the project and a lot of time was lost as a result. Problems that in a conventional system might have been resolved in a few hours could take days to work out. The solution that gradually evolved was messy and certainly nothing new to experienced service developers: provide lots of logging and a way of sharing the logs.

Our tester program for the basic airline service only became useful when we added logging of all test and response SOAP messages. The log was made visible along with the test results so programmers could see what messages were being sent, what messages were being returned, and what exceptions were being thrown. Programmers for the airline services then also added logging to capture incoming messages and to trace internal execution.

Sharing of logs obviously creates a significant security hole. Mitigation strategies for these kinds of issues should include:

- Extensive logging at configurable levels for the service.
- Some moderately secure mechanism should be provided in the test environment so that consumers can get information from producer logs and vice versa.
4.4.3 Encryption
This project used MySQL’s AES_ENCRYPT and AES_DECRYPT methods for backend processing of encrypted data in the authentication service. This proved problematic because of the lack of existing documentation of those methods in any consolidated place online. Combined with the difficulty of troubleshooting interoperability issues in general and SOAP message interchanges specifically, encryption and decryption between any of the technology packages was difficult. Further, because of time constraints, security was curtailed on the authenticator service by shortening key lengths and standardizing some aspects of SOAP data so as to allow basic interoperability between tech packages/services. While security is always a trade-off between strength and ease of use, better up-front security design enables stronger security with less of an impact on usability.

As indicated above, accessibility of server logs from the authentication service was crucial to debugging encryption/decryption issues. Early specification of exact encryption standards and protocols would also be helpful to mitigating conflicts and problems implementing security within each different technology package.

5. Conclusions
This report has described some experiences from an academic case study on the programming of secure, interoperable web services. While such services are needed in many domains, there is little published documentation on how to manage the numerous issues that arise in bridging technology packages such as Microsoft, LAMP and Java while complying with reasonable security requirements. No single case study can identify all such issues, but we hope our experience can help other developers avoid at least some of the pitfalls that we encountered.

Some of the key lessons could be summarized as follows:

1) Requiring contract-first development, interoperability, and security compliance significantly complicates the time and effort required to develop web services. In comparison with earlier student service programming exercises that did not have these requirements, we encountered many more glitches and significant time loss. Project schedules need to plan for such contingencies.

2) It is important to define up front a security process, which should include a security checklist, a SOAP security header format, and the adoption of encryption and hashing schemes known to be available on all platforms. It should also mandate early (not just final) code inspections to verify that developers have internalized the principles established in the checklist.

3) Of the three technology packages we used, LAMP with PHP was definitely the easiest to get working quickly. The Java and NetBeans environment was somewhat more time consuming and the Microsoft package was the most challenging. This appreciation obviously applies only to the effort needed for initial deployment; there may be strong organizational and life-cycle reasons for choosing other environments in any specific project.

4) The operations and data types used in the service interface should not be used internally but should be mapped to internal functions and data types. (Also suggested in [6], p. 40). A layered architecture such as that described in section 2.4.2.1 is very useful to isolate as much as possible of the service code from future changes in interfaces.
5) Services should be programmed with extensive and configurable logging, particularly of the actual SOAP messages being sent and received. To address the difficulties of debugging when services have different owners, if possible test environments should be set up so that service consumers can view the logs of service providers and vice versa.

Though there are undeniable benefits to contract-first web service development, it comes with an inherent set of challenges. Further, SOA implementations as a whole possess some unique security challenges that may not be an issue in other types of software. The specific experiences in this case study pointed to the clear conclusion that incorporation of early, robust planning of security as part of the overall structure of SOA web services is crucial to functional yet secure interoperable web services.

Acknowledgement
We would like to thank Amazon.com for providing an AWS course grant which covered almost all student use of Amazon Web Services facilities during this study.

References


