2008-03-12

Putting the Web Services Specifications to REST

Dan R. Olsen
Brigham Young University - Provo

PUTTING THE WEB SERVICES SPECIFICATIONS TO REST

by

Dan R. Olsen III

A thesis submitted to the faculty of

Brigham Young University

In partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

April 2008

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Dan R. Olsen III

This dissertation has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory

Date	Dr. Phillip J. Windley, Chair
Date	Dr. Kent E. Seamons
Date	Dr. Christophe Giraud-Carrier

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Dan R. Olsen III in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	Dr. Phillip J. Windley Chair, Graduate Committee

Accepted for the Department

Date	Dr. Parris K. Egbert Graduate Coordinator

Accepted for the College

Date	Dr. Thomas W. Sederberg Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

PUTTING THE WEB SERVICES SPECIFICATIONS TO REST.

Dan R. Olsen III

Department of Computer Science

Master of Science

Web services have become a useful and effective way of sharing information over the World Wide Web. SOAP has become a popular way of providing Web services and has been the focus of the Web Services specifications. The Web Services specifications provide additional capabilities to Web Services such as security and policy exchange. Another popular form of Web services includes light-weight Web or RESTful Web services over HTTP. These light-weight Web services are currently not addressed by the Web Services specifications. In order to provide the same capabilities to RESTful Web services, the Web Services specifications will be used to extend the HTTP protocol to provide the additional capabilities.

This work will show how the HTTP protocol can be extended using existing well defined specifications to provide extra capabilities such as security to RESTful Web services.

ACKNOWLEDGMENTS

My wife Emily who has been patient through the process of getting my degree, Dr. Windley who helped to make sure I had what I needed to be able to finish my degree, and my parents Dan and Vickie for their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LISTINGS

Introduction

Using the Internet as the basis of sharing information has grown increasingly popular and specifications on how that information is transmitted are being constantly written and rewritten. The Hypertext Transport Protocol (HTTP) has proven to be one of the most popular, but for some Web services laying additional functionality onto the base specification would increase the standardization and flexibility of authentication requirements, site policies, and site specific functions. This thesis proposes repurposing protocol specifications built for use with the SOAP protocol to add features to HTTP. I will demonstrate the practicality and utility of doing so.

Hypertext Transport Protocol (HTTP)

HTTP is the transport protocol for the World Wide Web (WWW). The HTTP protocol is currently at version 1.1 and is specified in RFC 2616 [1]. The most commonly used methods that define the protocol are GET, POST, PUT and DELETE. With these methods, data can be sent, received, stored and removed from servers listening for such messages so long as they adhere to the specification.

Roy Fielding's dissertation describes an architecture that models information as resources where a resource is a conceptual mapping to a set of entities [4]. Because in-

formation is a resource, information can be accessed using a resource identifier. In HTTP, an identifier is a Uniform Resource Locator (URL) where each URL maps to a unique resource, including but not limited to, a web page, an embedded image, or a document needed for service functionality. Also, by returning information in a well-known format such as XML, each application can parse and interpret the data in unique ways. Fielding calls this architecture the Representational State Transfer architecture, better known as the REST architecture.

HTTP is incredibly flexible because of its open structure. Additional information can be added into an HTTP request using the standard headers to pass information between two devices. As the WWW has continued to grow, other, more abstract transport layers and protocols have been built on top of the HTTP protocol. One of these layers is the Simple Object Access Protocol (SOAP).

Simple Object Access Protocol (SOAP)

SOAP is a standard maintained by the World Wide Web Consortium (W3C) and is used to send messages over various transport protocols in what is called a SOAP envelope. An envelope contains all the information regarding the SOAP message and is built using XML syntax.

SOAP is used to pass information independent of the transport protocol and uses the protocol's primitives to pass the message across the network. By using the protocol's primitives, the SOAP message does not need to contain information about the machines involved in the message passing. The most common transport protocol used with SOAP is HTTP.

"A protocol is a communication service that higher-level objects use to exchange messages."[7] A protocol also has "[a] set of syntactic and semantic rules for exchanging information that includes (a) syntax of the information; (b) semantics of the information; and (c) rules for the exchange of information."[2] Table 1 shows how TCP, HTTP, and SOAP meet this definition.

Table 1: The three pieces that define a protocol in relation to TCP, HTTP, and SOAP

	TCP	HTTP	SOAP
Syntax	Connect, Send, Receive	Fixed Methods (GET, HEAD, POST, etc.)	XML, Envelope contains header and body
Semantics	Packets of information containing the message to go across the wire	A query or post event with additional information	Applications specific information using supported data types inside a SOAP envelope
Information Exchange Rules	Guaranteed delivery, Three-way handshake, an ACK must be received for each package sent	Send a valid HTTP method and receive a status message and response	Valid SOAP envelope must be sent and received, must understand flag

SOAP is used to transport information between peers much like HTTP and TCP, therefore SOAP can be thought of as an abstract transport protocol. SOAP takes a message and abstracts the information out of the message dealing with the source device and the destination device. SOAP is then just another layer on the transport stack. Just as HTTP is layered on top of TCP, SOAP is layered on top of HTTP.

Web Services

Web services make networked services available to developers to compose larger services and applications. This allows developers to spend time developing instead of re-creating information and services that already exist and are made available. Web services can also reduce the amount of redundant information across the Internet.

Fielding's architecture, which uses URLs as identifiers to information over the HTTP protocol [4], has popularized RESTful Web services. RESTful Web services use only the HTTP protocol to relay messages between two devices while skipping the need for other messaging frameworks. In the RESTful architecture, all the documents and messages can be accessed using HTTP methods and URLs. The REST approach provides a simple, light-weight implementation of Web services.

With the emergence of RESTful Web services, many Web service providers are including RESTful solutions to their services. Amazon provides a Web service that allows developers to access information using remote calls with either SOAP or REST. The RESTful version of the Amazon Web service has become more widely used than the SOAP service due to its simplicity. In fact, as of August 2004, 80% of calls to Amazon's Web service were REST calls [3].

Web Services Specifications (WS-*)

A number of Web Services (WS-*) specifications are under development by Oasis, Microsoft, IBM and other standards bodies, companies and individuals. The intent of these new specifications is to standardize Web services in regards to security, messaging and transactions. The specifications promote loose coupling between web applications

and the Web services they consume. Some of the WS-* specifications add metadata and functionality to SOAP while others add transport functionality. To date, the WS-* specifications have been used exclusively with SOAP.

Table 2: Metadata-oriented and transport-oriented specifications

Metadata	Transport
WS-Security	WS-MetadataExchange
WS-Policy	WS-ReliableMessaging
WS-Trust	WS-Discovery
WS-Enumeration	WS-Transfer

The WS-* specifications can be divided into two groups, those that are transport-oriented and those that are metadata-oriented. Transport-oriented specifications are ones that define service endpoints and other details regarding massage passing. Metadata-oriented specifications contain information about the structure of the data to be used and processed with regards to individual services. Table 2 contains a list of some, but not all, of the WS-* specifications and which of the two categories they belong to.

As an example of how the metadata-oriented specifications work, WS-Security is a specification that allows for the server and the client to communicate about the types of security that are accepted by the service. Not only does it allow for basic security information but it is also used to send information on how information should be encrypted and decrypted. Other information can also be included that describes what needs to be done by both the server and the client in order to provide a secure connection between them.

Thesis Statement

The HTTP protocol is a popular method for implementing light-weight Web services. The Web Services (WS-*) specifications add important capabilities to Web services such as security and policy exchange. Some of these specifications are designed to be transport independent but are currently only used inside SOAP. I will demonstrate that certain of the WS-* specifications can be used over other transports, specifically HTTP, by providing a methodology for adapting WS-* specifications to HTTP as well as a proof-of-concept. Using the WS-* specifications over HTTP augments it with well-defined, standards-based capabilities without compromising its simplicity.

Related Work

The World Wide Web (WWW) continues to expand and include more ways of information passing and transaction handling. As this growth continues there is an increased call for an easy way to access information and for standards to increase interoperability.

With Web services becoming more prominent across the WWW work is continually being done in providing ways to make these services communicate easily and efficiently. Large scale products and services are using Web services as the basis for information exchange and security.

Microsoft CardSpace

Identity has become an important topic when it comes to online transactions. Individuals and companies want to make sure their information is safe when being transmitted to acquire goods or services. Microsoft's CardSpace [5], formerly known as InfoCard, is one of the front-runners in providing secure ways for providing information over the WWW.

Microsoft's CardSpace is an identity meta-system that relies heavily upon Web services, in particular the WS-* specifications. CardSpace is a driving factor behind the

research compiled in this thesis. Because CardSpace relies heavily on the WS-* specifications it also heavily relies on SOAP. Since the CardSpace identity meta-system relies on the WS-* specifications the development of similar RESTful solutions could be a possibility.

Although CardSpace is based on SOAP messages the research in this thesis will show how the idea can carry over to provide a RESTful solution using the specifications that can be layered on top of HTTP [5].

Liberty Reverse HTTP Binding for SOAP

The Liberty Alliance group has proposed a binding that allows a client to expose SOAP Web services [9]. The idea of using HTTP to expose a service follows along the idea of layering functionality on to the HTTP protocol. The specification uses the HTTP headers to provide necessary information to an HTTP server that is providing the SOAP service.

The binding specification defines the `POAS` HTTP header to provide the necessary information. Along with the `POAS` header the specification provides a new `Content-Type` to inform the server of the binding. The Content-Type is defined as `application/vnd.paso+xml`.

Although HTTP can already send SOAP messages without any extension or inclusion of HTTP headers the specification states that "[t]he primary difference from the normal HTTP binding for SOAP is that here a SOAP *request* is bound to a HTTP *response* and vice versa." [9]

The idea of this binding is similar to the work done in this research. Using the headers of an HTTP message, Liberty can now provide an additional extension to HTTP to expose the SOAP services.

However, in this work I am taking a set of specifications and instead of using them with SOAP they will be used to work with RESTful Web services. The specifications will actually be extending HTTP with adding additional headers instead of providing a way to expose an already existing service.

SOAP 1.2 and GET

With SOAP 1.0 requests were only able to be passed using the POST method across HTTP. With the limited method use within SOAP, many questions arose about why a request for information is not done using the GET method as specified in the HTTP specification.

The HTTP GET method is safe and idempotent. A request made with the GET method should provide the same information each time a request is made with the specific URL. This follows the specifications idea of a safe method. The GET method follows the idea of being idempotent by guaranteeing that the request causes no side effects.

Since a Web service request often only requests information, including SOAP Web services, the new 1.2 SOAP specification provides a way of making a request for information using SOAP without using the POST method [8].

“SOAP 1.2 introduces Message Exchange Patterns (MEPs) and a new HTTP binding. By combining the two, you can finally implement a Web service that replies to GET requests. MEPs document the interaction patterns between the client and the server.

The SOAP Request-Response MEP is a typical Web service interaction: The client sends a request to the server, and the server replies" [8].

Once again, the SOAP specification itself is being geared toward layering functionality on top of HTTP in a way that uses more of HTTP's flexibility. This continues to support the idea of how simple it is to layer functionality on top of HTTP.

Methodology

The goal behind using the HTTP envelope with the WS-* specifications is to provide the same functionality and options to RESTful Web services that come with using the specifications with SOAP. For example, WS-Security can be used to pass information that will allow the RESTful service to check for authenticity of the message. The specifications currently use the SOAP header to pass the WS-* specification information but can also be passed in a HTTP header. Inserting the information in the HTTP header will allow RESTful Web services to leverage the WS-* specifications within the standard HTTP protocol.

To accomplish the goal, the following steps will be used to augment a given WS-* specification in HTTP.

1. Determine if the specification is metadata-oriented.
2. Identify each metadata item.
3. Determine what metadata is relevant to a RESTful solution and what is only an extension to SOAP.
4. Determine if the metadata can be included in an HTTP header or should be stored in a document and referenced in a header.
5. Determine what existing HTTP headers are suitable for storing the metadata.

6. Determine what new HTTP headers are needed to store the metadata.
7. Determine an appropriate format for passing metadata in the headers.

The following sections describe each of these steps in detail and provide examples from the WS-Security and WS-Policy specifications to illustrate how each step is accomplished.

1. Determine if the specification is metadata-oriented

The transport of metadata is the focus of this work. Before any implementations can be done, there must be a distinction between what is considered metadata-oriented specifications and what are not.

In order to determine which specifications will work with the methodology of this thesis, the specifications are separated into two groups. The first group is the metadata-oriented group, and the second group is the transport-oriented specifications. The specifications that are transport-oriented will not be included because they extend the functionality of SOAP for transport information. For example, WS-MetadataExchange provides details on how metadata can be exchanged.

Breaking the specifications into groups allows attention to be placed specifically on the metadata-oriented specifications which can be evaluated and implemented using headers as a method of providing transport information. Looking at the metadata-oriented specifications independently of the transport-oriented specifications can help clarify what transport information is necessary as well as being able to make specific evaluations of each specification.

While the metadata-oriented specifications can be used to augment the HTTP protocol with additional functionality, the transport-oriented specifications are more SOAP specific. Because of the number of metadata-oriented specifications, only two of the specifications will be focused on in this work: WS-Security and WS-Policy.

WS-Security and WS-Policy are good examples of metadata-oriented specifications. They allow data about security and policies to be sent between the service and the client. Information involving what security methods a service uses are not specific to extending SOAP and can be used across other transport protocols. This is an example of what is meant by metadata-oriented information.

2. Identify each metadata item

In order to determine how a specification can be used with a RESTful service the different components of the specification must be understood and not just the specification as a whole. This will help determine what metadata is needed and what is not. It is important that the implementation of each specification does not contain things that are part of other specifications.

A majority of the specifications use ideas and technologies that are already specified in other specifications. It is important that those specifications are taken into consideration and not assumed to be part of the specification being reviewed. This includes taking into account WS-* specification as well as other specifications such as the specification for digital signature [10]. In many cases the WS-* specifications depend on WS-Security for secure communication between the client and the service being invoked. Because of this dependence, the WS-Security specification will be one of the main specifications focused on in this work.

It is important to review the entire specification. The specifications are often broken down into sections and by taking each section in turn and determining what the individual pieces of the specification does is helpful because you can begin to see what is needed for the general specification and what is used to enhance SOAP.

As mentioned above, there may be other non WS-* specifications that a given specification may depend on. However, often times there are packages already built that take care of the functionality of outside specifications. An example is the XML-Signature specification which is used by WS-Security but WS-Security does not dictate how XML-Signature works. Separating the dependent specifications from the WS-* specification helps to clarify what belongs to the specification and what does not.

3. Determine what metadata is relevant to a RESTful solution and what is only an extension to SOAP

Once the metadata items have been identified it is easier to determine what items provide metadata specific information and what is information used to extend SOAP. In some instances all the information in the specification is metadata specific and may not contain information that extends SOAP. WS-Security and WS-Policy are both examples of specifications that are metadata based information.

4. Determine if the metadata can be included in an HTTP header or should be stored in a document and referenced in a header

An important aspect of solving this problem is to use HTTP headers and general XML documents to implement the enhancements described in the WS-* specifications. Consequently, we must separate the metadata into two groups. The first group is meta-

data that can be sent within the HTTP headers. Other metadata might be more appropriately stored in an XML document that is retrieved to provide the necessary information. In the latter case, the URI of the document would be passed in the header. These two methods are called the "direct" and "indirect" methods respectively.

Header Information

Header information refers to metadata that can be passed using the HTTP headers. The headers will be used by the service to provide metadata related to the WS-* specification. An example would be a header that informs the service that WS-Security is being used. Listing 1 gives an idea of what the headers might look like when WS-Security is being used.

Listing 1: The HTTP headers from a request using WS-Security

```
1  GET /cgi-bin/dan/temp_conv.cgi?tempc=32 HTTP/1.1
2  Host: teton.cs.byu.edu
3  User-Agent: Mozilla/5.0
4  Accept: text/xml,application/xml,application/xhtml+xml,…
5  Keep-Alive: 300
6  Connection: keep-alive
7  X-WSSecurity: <Security> … </Security>
```

Using the HTTP headers to pass metadata is useful when a request needs to contain information the service requires. This is often the case with WS-Security. The service needs the WS-Security metadata when the request is made in order to process the request properly.

XML Document Information

There are times when a specification provides metadata that needs to be retrieved. In these situations we need to provide an XML document instead of the XML metadata in

the headers. An example would be WS-Policy. WS-Policy provides a description of the policies the Web service allows. In this case the client can request that XML document to receive the proper WS-* information.

Passing the WS-Policy metadata in the header could be done but it is easier to provide the client with URL to such an XML document and allow the client to pull the document to receive the policies allowed by the service. Providing the URL to the document allows for changes to be done to the document instead of the actual implementation of the service.

5. Determine what existing HTTP headers are suitable for storing the metadata

Once the proper metadata is extracted from the specification it must be decided what headers will hold which metadata. In some instances there may be existing HTTP headers that can be used to pass information to the service.

An example of using an existing header is a timestamp. The `Date` HTTP header could be used to send the information that allows the service to know what time the request was made. This may be useful when a service only allows a request that was sent within a certain time period.

6. Determine what new HTTP headers are needed to store the metadata

In some instances there are no existing HTTP headers that can be used to pass the necessary information. The HTTP specification allows for new headers to be created by using the `X-` prefix. Figure 1 shows the `X-WSSecurity` header which was created in order to pass WS-Security specific metadata. Additional headers may be needed based on

the information that needs to be passed for the specification being used. Most often headers will need to be created to describe the metadata that is being passed through the header much like the `X-WSSecurity` which lets the service know that the information associated with that header is WS-Security specific.

7. Determine an appropriate format for passing metadata in the headers

Once the metadata has been extracted and the proper HTTP headers have been determined the next step is to decide in what format the metadata will be sent. In the WS-* specifications the data is passed in an XML format. However, by using the HTTP headers to send the metadata it may not be necessary to send data in XML.

In the case of a timestamp in the `Date` header there would be no point in wrapping the date up into XML if the request was a direct request between the client and the service. Instead you would just pass the date straight through as you normally would in the `Date` header. If the date is important and the request is passed through different locations then wrapping the date in a separate header would be desirable to preserve the original request's timestamp.

On the other hand, WS-Security aggregates multiple pieces of metadata. In this case keeping the XML format is worthwhile. In other cases there may be other forms of formatting that can be used to best describe the specification's information. JSON has become a popular way of passing data that can be easily parsed with publicly available libraries. Understanding the metadata within the specification is important so that it can be properly formatted in a way that best suits the purposes of the service.

An example of using another format such as JSON would be if you simply needed to pass a set of information such as a username and password. In that case you could simply send the information in the headers as shown in Listing 2

Listing 2: Formatting using JSON

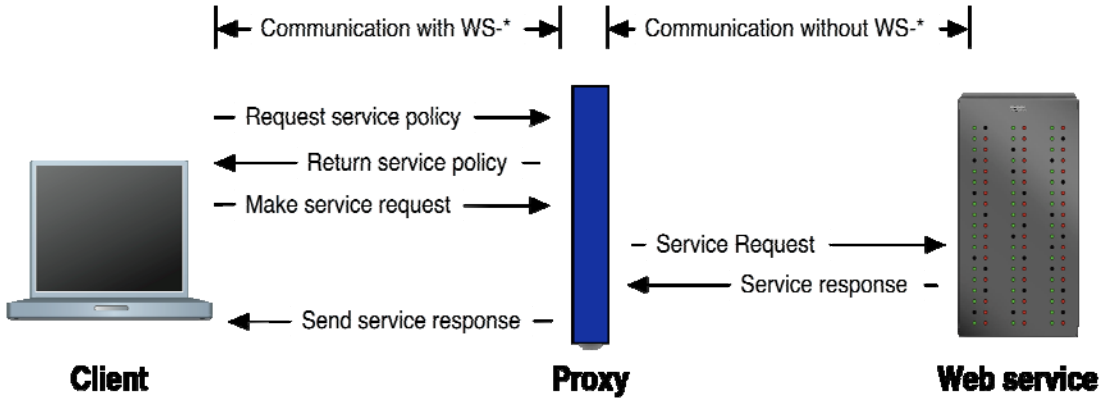
```
1 { "username" : "danolsen", "password" : "eclab" }
```

Example Applications

To show how the WS-* specifications work with a RESTful Web service, I built a proxy in front of the Amazon E-Commerce Service (Amazon ECS) that takes WS-Security information and determines whether or not to forward the request to the service. Two examples will be provided which will use WS-Security and WS-Policy to provide security for the Amazon ECS.

Figure 1 shows the communication between the client, the proxy and the Amazon ECS. The communication shown in the figure is being done over HTTP without the use of SOAP. The proxy takes care of the WS-* processing that is needed for the examples. Before the client can make a request to the service it must ask the proxy what policies it requires in order to pass a request on to the Amazon ECS.

Figure 1: A diagram that shows the communication between the client and proxy as well as the communication between the proxy and the Amazon ECS.



The proxy uses a configuration file in order to know what services are available as well as provides the necessary information needed for service processing. Listing 3 contains an example of what a configuration file might look like.

Listing 3: An example file showing configurations for two different services.

```
 1  <Services>
 2    <service>
 3      <name>AMAZON</name>
 4      <url>http://ecs.amazonws.com/onca/xml</url>
 5      <wspolicy>.../amazon_policy.xml</wspolicy>
 6    </service>
 7    <service>
 8      <name>TEMPCONVERT</name>
 9      <url>http://dan-olsen.org/tempconvert</url>
10      <wspolicy>/tempconvert/policy.xml</wspolicy>
11    </service>
12  </Services>
```

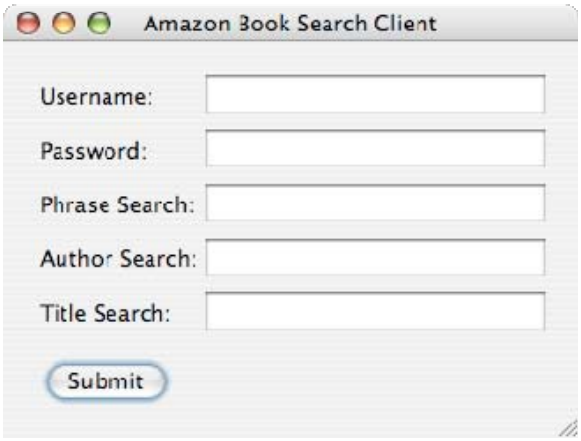
The proxy uses the `<name>` element to identify which service the client is requesting. The name listed here is concatenated to the proxy's URL to create the service endpoint. For example, the Amazon service endpoint is the proxy's URL concatenated with "amazon" (`http://localhost:3232/amazon`). In the case of the second service in Listing 3 the service endpoint is `http://localhost:3232/tempconvert`.

Line 4 provides the proxy with the base URL that is used by the Amazon ECS. By providing the base URL for the service the variables from the request URL can easily be extracted and then appended to the base URL for making the actual request between the proxy and the Amazon ECS.

The `<wspolicy>` element contains the path of the WS-Policy document. The client requests the policy by taking the proxy's URL and concatenating the service name and then the policy. For the Amazon example the request URL is `http://localhost:3232/amazon/policy`.

Figure 2 shows the client interface used for the examples. The client is built to be able to perform requests based on which policy is preferred by the proxy. If the preferred policy of the proxy is to have the user authenticate, the client will use the provided username and password fields. On the other hand if the proxy prefers that the client create a digital signature of the variables with the request then the client will ignore the username and password and only use the phrase, author and title fields to build the query.

Figure 2: User interface of the client used in the example applications



The first example uses WS-Security to sign the service request in order to provide integrity to the request. The first example (see "Example 1" below) will only use the

phrase, author, and title search fields. The second example (see “Example 2” below) creates an authentication front-end for the ECS to only allow authenticated users to receive results for the ECS. In the second case the username and password provided by the interface will be used.

Before anything can be done with WS-Security, the client must obtain the Web service’s security policies specified according to WS-Policy. The client sends a request for the server’s policies and they are returned in an XML document.

Listing 4 shows the policy document used in the examples. The `Preference` attribute found on lines 3 and 11 lets the client know which policy the Web service prefers. For the first example, since the preference on line 3 is set to 1, the client will send a digitally signed request to provide integrity. The second example assumes lines 3 and 11 are swapped, so the client uses the password authentication front-end to authenticate to the server.

Listing 4: The Policy document used in the examples

```
 1 <Policy>
 2   <ExactlyOne>
 3     <All Preference="1">
 4       <Integrity>
 5         <Algorithm Type="AlgSignature" URI="...#rsa-sha1" />
 6       </Integrity>
 7       <SignedParts>
 8         <URLVariables />
 9       </SignedParts>
10     </All>
11     <All Preference="2">
12       <WSSUsernameToken11 Type="PasswordDigest" />
13     </All>
14   </ExactlyOne>
15 </Policy>
```

In WS-Policy the `<SignedParts>` tag specifies those parts of the message that must be signed. We introduced the `<URLVariables/>` tag, shown on line 8 of Listing 4, since the WS-* specifications do not define a way for the URL or pieces of the URL to be signed or encrypted. The tag lets the client know that there will need to be a signature applied to the URL encoded variables.

Once the client knows what policy is required the WS-Security information for the request can be generated.

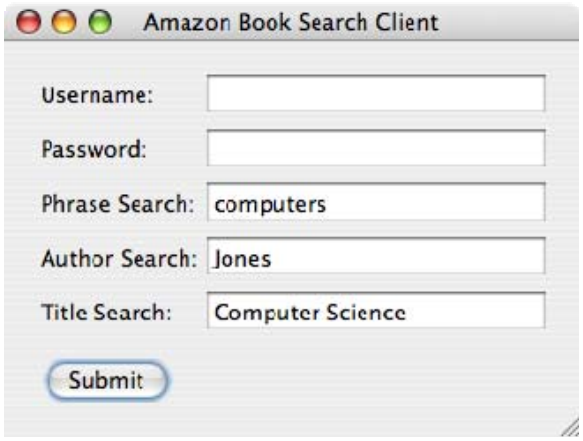
Example Application One

In this first example the Amazon ECS Web service will be used to do a search using input from the user. This example will use WS-Security to sign the URL variables that will be passed to the service. This shows how message integrity can be accomplished with this RESTful call to the Web service.

Before the WS-Security information can be generated by the client it will need to know where to make the request. In this case there is a base URL that points to the Amazon ECS service proxy. In order to pass the information to the proxy the URL is generated using encoded variables that are normally used to make a request to the ECS service.

Figure 3 shows how the user submits a search request using the client interface and Listing 5 shows those URL encoded variables that are generated using the user's input.

Figure 3: User interface with the user search information for example one



Now that the request variables are created the client can begin generating the WS-Security information. In order to generate a signature the client need to have an encryption key pair. The keys will be used to sign and verify the signature generated for the request. In the break down of the WS-Security specification it is not the goal of the specification to worry about the exchange of security keys. Therefore, this example does not address the key exchange; instead it assumes the client and server already have the appropriate keys.

Listing 5: The URL encoded variables needed by the Amazon ECS services

```
1  request?SearchIndex=Books&Service=AWSECommerceService
       &Author=Jones&Title=Computer%20Science
       &version=2006-06-07&Keywords=computers
       &Operation=ItemSearch&SubscriptionId=...
```

Listing 6 shows the outcome of signing the request and generating the WS-Security header. At line 12 is the SHA1 digest that is signed using the RSA algorithm and the signature is then located at line 18. The WS-Security information can now be inserted into the X-WSSecurity header defined earlier.

Listing 6: X-WSSecurity header generated for the first example's request

```
<Security>
   <Timestampe Id="T0">
      <Created>
         2007-01-08T18:10:59Z
      </Created>
   </Timestamp>
   <Signature>
      <SignedInfo>
         <SignatureMethod Algorithm="...#rsa-sha1" />
         <Reference>
            <DigestMethod Algorithm="...#sha1"
            <DigestValue>
               UlzjVS+3w3klGaC7oXDsmRw1TkA=
            </DigestValue>
         </Reference>
      </SignedInfo>
      <SignatureValue>
         PFN0YXJOUH...
      </SignatureValue>
      <KeyInfo>
         <KeyName>
            ex_mykey
         </KeyName>
      </KeyInfo>
   </Signature>
</Security>
```

When the proxy receives the request from the client it checks to make sure the request contains information regarding any of the preferences contained in the policy document as shown in Listing 4. If the request from the client meets exactly one of the preferences the proxy continues to process the request. If the request does not contain any of the required information then an error is sent. In the first example it takes the URL encoded variables from the request URL and generates the SHA1 hash and verifies the enclosed signature. If the signature is verified then the request is passed to the Amazon ECS. The results that return from the Amazon ECS are then forwarded to the client.

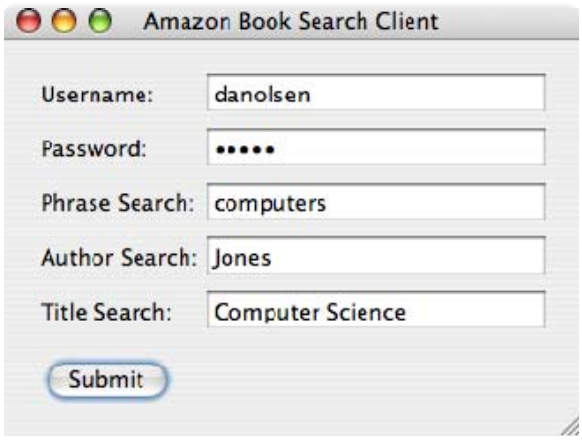
If the proxy is unable to verify the signature an error message is returned to the client application.

Example Application Two

For the second example the URL needs to be generated with encoded variables as shown in Listing 5. These variables will always need to be generated in order to provide the information the Amazon ECS needs.

In this example the policy requests that the client authenticate with the service in order to receive the desired results. The user must enter a username and password, as shown in Figure 4, which is registered with the service. The policy in Listing 4 tells the client that it expects a username token with a password digest. When using a username token in WS-Security a service has the option to request the username and password or to request the username and a password digest. This example uses the password digest in order to provide a more secure way of using a username/password authentication front-end.

Figure 4: The user interface for example two including the username and password



The password digest is created by concatenating the creation date, a nonce, and the user's password, in that order, and using the SHA1 algorithm to generate the digest. Listing 7 shows how the `X-WSSecurity` header will look in this situation.

Listing 7: `X-WSSecurity` header generated for the second example's request

```
 1 <Security>
 2    <UsernameToken>
 3       <Username>
 4          danolsen
 5       </Username>
 6       <Password Type="...#PasswordDigest">
 7          GgfdV2m90dTb7jJjX0YxIG6IHpg=
 8       </Password>
 9       <Nonce>
10          MTk0MzYwODE2NjYyNzk50A==
11       </Nonce>
12       <Created>
13          2007-01-09T17:28:35Z
14       </Created>
15    </UsernameToken>
16 </Security>
```

Once the client sends the request to the proxy the proxy in turn takes the username given in the header. The proxy then does a lookup on the given username and determines if the password was valid. Once the proxy finds the matching password it generates a digest the same way the client did. Once the digest is generated then the proxy compares the digest it generated with the digest given in the header sent with the request. If the digests match then the proxy considers the user authenticated. If the digest does not match the proxy sends back an error message informing the client of why the user was not authenticated.

Once the proxy authenticates the user then the proxy will take the URL encoded variables and append them to the service URL and forward the request to the Amazon

ECS service. Once the proxy receives the results from the Amazon ECS it returns the results to the client.

Other Situations

Of course the two situations discussed above do not account for all possibilities within the WS-Security and WS-Policy specifications. There may be situations where the service may require a part of the WS-Security message to be signed instead of the URL encoded variables. There is also the option of signing the URL encoded variables as well as one or more parts of the `X-WSSecurity` header. Signing a piece of the WS-Security header can be done in the same manner as demonstrated in the first example application.

Another situation may ask for the client to encrypt the URL encoded variables instead of just signing them or requiring the user to authenticate with the service. In this case the client would be sure to encrypt the variables before appending them to the end of the proxy service URL. The client would also provide the proper information in the `X-WSSecurity` header.

Analysis

In the examples given in the previous section we showed that the WS-Security and WS-Policy specifications can be applied to an existing RESTful Web service and that it is possible to take the WS-* specifications and use them as a foundation to providing the same extensibility to RESTful Web services. This is significant because it shows that an extra level of security without having to write a specification from scratch in order to do so.

The first example showed that the WS-Security specification can be used with a RESTful service to allow the client to sign variables in the request. In a SOAP based service, portions of the body can be digitally signed. The signature is included as part of the SOAP body in a WS-Security header. Similarly, we demonstrated how URL variables can be signed, and the digital signature included in a WS-Security HTTP header. In both cases, a signature of the necessary variables is included in a WS-Security header.

The second example showed that the WS-Security specification can be used with a RESTful service to support user authentication. Authentication information is generated and stored in the `UsernameToken` structure in the WS-Security header. This is the identical information that is passed in a SOAP based service.

Although the examples did not show all aspects of the WS-Security and the WS-Policy specifications, they demonstrate how the methodology could be applied to the remainder of the specifications. For example, in the cases of encryption of the variables we would see something similar to the signature design. Instead of encrypting a SOAP body, the client would encrypt the URL encoded variables and the proxy would decrypt them before sending the request to the Amazon ECS service.

By implementing a proxy in front of the Amazon ECS we were able to show how WS-Security based signatures and authentication can be used with a RESTful Web service call. For security reasons, the proxy should be co-located with the service on an internal network or possibly on the same machine so that the request between the proxy and the service cannot be hijacked. Using HTTPS would be an option for providing a secure connection between the proxy and the service if they are located on separate machines. This is a security risk that could be found in the examples shown above since the proxy was located at a separate location than the service.

In this work we chose to use a proxy architecture to provide the WS-* functionality for demonstration purposes. However, there may be other architectures that would be appropriate. A WS-* library could be built that allows for a RESTful Web service to built on top of it and allow it to use the WS-* library to perform necessary checks and manipulations depending on the WS-* specifications being used.

Conclusions

Contributions

Although the WS-* specifications were created for SOAP based Web services, we can leverage the WS-* specifications to add functionality to RESTful Web services. We have shown simple examples of how this would work using a proxy to provide WS-* specification processing. The examples in this paper show that the WS-* specifications can be used to provide the same Web service enhancements to a RESTful Web service that SOAP Web services enjoy.

Future Work

The work in this thesis presented WS-Security and WS-Policy and showed how they would be used with a RESTful Web service. Although the work was limited to the two specifications, this idea applies to all of the meta-data oriented specifications mentioned earlier in this paper.

In order to show how the specifications could be used with an existing RESTful Web service a proxy was used to show how this could be achieved. This work is not lim-

ited to just proxy based services but could be expanded and included within an HTTP server and configured easily within the configuration files of a given HTTP server.

In addition to providing security to a Web service there could be a way of exposing what services are provided by a given proxy. In Listing 3 was the configuration file for the proxy. Future work could include a way for the user to select which of the services it wants to use. The client could provide some sort of drop down menu after discovering which services are available.

A general purpose WS-* library could be created in order to be able to process all the metadata-oriented specifications. This would allow a single library to be used to provide any WS-* specifications the service desires to use. However, smaller libraries that are application specific may be preferred due to performance issues.

Service discovery could include extending HTTP with the WS-Discovery Specification which provides a way of providing service discovery.

References

[1] T. Berners-Lee, R. T. Fielding, H. F. Nielsen, J. Gettys, and J. Mogul. (1997, January) Hypertext Transfer Protocol - HTTP/1.1. [Online]. Available: ftp://ftp.ietf.org/rfc/rfc2068.txt

[2] Carnegie Mellon Software Engineering Institute. (2006, January) SEI Open Systems Glossary. [Online]. Available: http://www.sei.cmu.edu/opensystems/glossary.html

[3] B. DuCharme. (2004, August) Amazon's Web Services and XSLT. [Online]. Available: http://www.xml.com/pub/a/2004/08/04/tr- xml.html

[4] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, Univeristy of California, Irvine, 2000.

[5] Microsoft. (2006, June) Introducing InfoCard. [Online]. Available: http://msdn.microsoft.com/winfx/reference/infocard/default.aspx

[6] OASIS. Web Services Security: SOAP Message Security 1.1. [Online]. Available: http://www.oasis- open.org/committees/download.php/16790/wss- v1.1- spec- os-SOAPMessage

[7] L. L. Peterson and B. S. Davie, Computer Networks: A Systems Approach, 3rd ed. Morgan Kaufmann, 2003.

[8] B. Marchal. (2004, March) Tip: SOAP 1.2 and the GET Request. [Online]. Available: http://www-128.ibm.com/developerworks/xml/library/x-tipgetr.html

[9] Liberty Alliance Project. (2005) Liberty Reverse HTTP Binding for SOAP Specification. [Online]. Available: http://www.projectliberty.org/liberty/content/download/1219/7957/file/liberty-paos-v1.1.pdf

[10] OASIS. (2007) Digital Signature Service Core Protocols, Elements, and Bindings Version 1.0. [Online]. Available: http://docs.oasis-open.org/dss/v1.0/oasis-dss-core-spec-v1.0-os.html