The University of Kansas

INFORMATION & TELECOMMUNICATION TECHNOLOGY CENTER
The University of Kansas

Technical Report

Implementing Web Services: Conflicts Between Security Features and Publish/Subscribe Communication Protocols

Edward Komp, Victor Frost, and Martin Kuehnhausen

ITTC-FY2010-TR-41420-19

February 2010

Project Sponsor:
Oak Ridge National Laboratory
Award Number 4000043403

Table of Contents

Table of Contents i
List of Figures i
Abstract 1
I. Introduction 1
II. Background 1
  A. TSSN 1
    1) Security Requirements 2
    2) Asynchronous Communication Requirements 3
  B. WSA-based Implementation 3
III. Implementation Conflicts 3
  A. Server Operations Obscured 4
  B. Reversal of Client and Server Roles 5
  C. Multiple Subscribers, Different Security Policies 5
IV. Resolution 5
V. Conclusions 6
Acknowledgment 6
References 6

List of Figures

Figure 1: Top-level view of the TSSN architecture. 2

Implementing Web Services: Conflicts Between Security Features and Publish/Subscribe Communication Protocols

Edward Komp, Victor Frost, *Fellow, IEEE, and Martin Kuehnhausen, Student Member IEEE*

Abstract - While on the surface the combination of software components that adhere to associated standards should lead to rapid and successful system implementation. However, issues can arise when integrating independently defined software subsystems. Here conflicts are discussed that arose when integrating elements from the Web Services Architecture[1] (WSA) led by the World Wide Web Consortium[2] (W3C), specifically publish/subscribe communication and service security. Unfortunately, the various standard components are seldom completely independent, and when separate components are jointly deployed unanticipated interactions sometimes cause significant problems at implementation time. The nature of the conflicts is discussed within the context of a specific system implementation the Transportation Security Sensor Network (TSSN) and interim solutions presented.

***Index Terms*— Eventing, Publish/Subscribe, Security, Service-Oriented-Architecture, SOA**

I. Introduction

Divide and conquer is a standard engineering practice for large, complex systems. The system is often repeatedly subdivided into (nearly) independent components, allowing groups to work independently and in parallel on subtasks. This basic principle is fundamental to large ongoing specification efforts such as the Web Services Architecture[1] (WSA) led by the World Wide Web Consortium[2] (W3C). Unfortunately, the various components are seldom completely independent, and when separate components are jointly deployed unanticipated interactions sometimes cause significant problems at implementation time.

We have experienced an example of conflict between independently defined subsystems when jointly deployed, in our design and implementation of the Transportation Security Sensor Network[3] (TSSN). In this paper we provide a brief overview of our system, followed by a detailed description of the conflicts between security features and the publish/subscribe communication protocol, and finally techniques to resolve these conflicts.

II. Background

A. TSSN

Monitoring cargo movements along trusted corridors requires coordinated application of sensing, communications, as well as the integration of shipment and other associated cargo information. To realize a trusted corridor a Transportation Security Sensor Network (TSSN) has been designed, implemented and tested in the field [4] to provide the required visibility into cargo shipments.

The system is composed of three major geographically distributed components. The Mobile Rail Network (MRN) consists of container seals that communicate over a wireless network to a reader when an event occurs, e.g., seal open, a sensornet collector node that interfaces to the reader, processes events, determines which events need to be communicated to a virtual network operations center (VNOC), and the mode of communications (e.g., GSM or satellite). The second component is the VNOC, which accepts messages from the MRN, obtains associated cargo information from a remote trade data exchange and then combines the information (e.g., nature of the event, location of the event, and cargo manifest) into an alarm message that is sent (by e-mail or SMS) to appropriate decision makers. The third component is the trade data exchange that contains the shipment information and other associated cargo information. A goal of the effort was to create technologies that will allow continuous monitoring of containers by leveraging communications networks, sensors as well as trade and logistics data within an environment composed of multiple enterprises, owners, and operators of the infrastructure.

The resulting technologies must be open and easy to use, enabling small and medium sized enterprises (SMEs) to obtain the associated economic and security benefits. Thus a standards web-based open system is preferred. The architecture developed as part of this effort uses existing software components in addition to those specifically developed for the TSSN. The infrastructure is based on W3C and OASIS Web Service specifications; including service discovery, services are described using Web Service Description Language (WSDL), and Client/Server communication based on Simple Object Access Protocol (SOAP). Figure 1 provides a top-level view of the TSSN architecture.

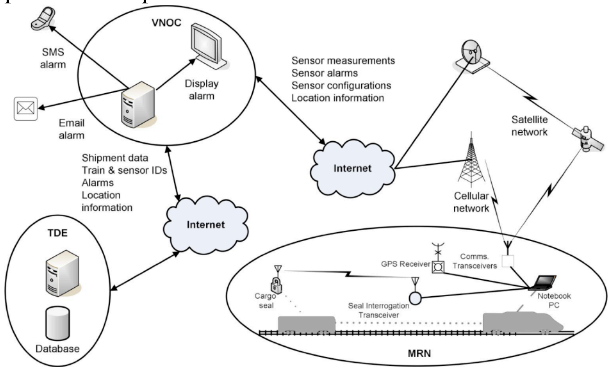

Figure 1

The TSSN has been tested in the field where the VNOC was located in Lawrence, Kansas, alarms generated on the train were sent to the MRN located in the locomotive, which forwarded relevant alarms to the VNOC. When the VNOC received an alarm, it contacted the TDE located at Overland Park, Kansas. to get the cargo information. The VNOC then combined the sensing and the shipment information to generate the alarm messages that were then communicated to a set of interested parties. See [4] for a discussion of TSSN field trials.

1) *Security Requirements*

Our application includes services and clients hosted by independent businesses often exchanging sensitive data. Therefore, robust and flexible security features were an integral portion of our design. It was clear that a single system wide security policy would be entirely inadequate to satisfy our goals.

As described in the following section, the Web Services Security[5] specifications provide support for the range of capabilities required in our system definition.

The Web Services Security model includes many features that are particularly important to our application:

- Does not rely on a single, application-wide security policy. Each service is allowed to define a security policy to match the sensitivity of data exchanged and other system constraints, such as bandwidth limitations.
- Provides a mechanism for each service to publish its security policy in a formal standard syntax, so that clients do not require details of service implementation.
- Does not demand (though allows for) a single, centralized supplier of authorization.
- Permits each easy extension of both services and clients.

In our distributed application some services must operate within tight processor and/or bandwidth constraints. A general comprehensive security policy, predicated on relatively large bandwidths among members, probably would not fit within the operating constraints of some services in our system. However, even in constrained environments, it is not acceptable to ignore security issues. The structure defined in WS-Security (WSS) specification allows each service to define precisely the amount of security, and implementation guidelines, to a fine degree. The WSS specification supports the attachment of security constraints at a variety of levels within the service definition including, service-wide policy, specific policy per operation, and distinct policies for different connections to the same service. The specification further defines how overlapping policies are to be integrated. This flexibility in defining the security policy for a service allows one to precisely define the level of security for a wide variety of services and to address changing demands if additional capabilities (operations) are added to an existing service.

The security policy(ies) enforced by a service can be specified in the WSDL (Web Services Definition Language) [6] description that formally defines a web service interface. The security policy is a critical aspect of a service interface that must be understood by any potential client. In earlier system implementations this information was generally expressed by any number of informal and ad hoc techniques, such as extra documentation and/or direct person-to-person communication between service provider and client. Web Services Policy 1.5 [7] defines a formal language to precisely define a service's security requirements, and to incorporate it into the web service interface definition. This is a major advancement for facilitating inter-operation of a service and its clients. This is particularly important when the service author and clients belong to independent business organizations.

2) Asynchronous Communication Requirements

A second fundamental aspect of our system is the ability to contact a diverse group of users when specific sensor events occur. For example, if a security lock is compromised on a container, we may need to immediately contact the carrier of the container, public first responders in the area in which the event occurs, as well as the owner of the specific container. Asynchronous notification does not directly fit the traditional client-server model. For many applications, some variation of client polling can be used in place of asynchronous notification. This approach is widely used in web applications, for example with RSS feeds. However, for our TSSN application, timely delivery of alerts and alarms is critical to utility of the system. Some portions of the TSSN network have very limited bandwidth and possibly long communication delays, further discouraging the usage of polling.

Fortunately, the Web Services Architecture includes the WS-Eventing [8] specification that defines a protocol for one Web service ("subscriber") to register interest in another Web service ("publisher"). In this document we will refer to this capability as the "publish/subscribe communication protocol". The publisher disseminates information to the subscriber(s) by sending one-way messages. The specification further defines mechanisms to dynamically add and remove members from the subscriber list; and support for complex publisher-subscriber topologies by allowing an "event source (to) delegate subscription management to another Web service"[8].

B. WSA-based Implementation

Adopting the Web Services Architecture as the basis for the TSSN dramatically reduced the effort we needed to both design and implement the infrastructure. Simply asserting that all services and clients would be WSA compliant eliminated the need to design and provide implementation for significant portions of our system. For example, our design simply stated that an alarm service would provide notification of the occurrence of an alarm condition via a publish/subscribe protocol (using WS-Eventing). We did not need to design the actual message protocols, and determine how they would be realized over various transport layers; nor address implementation issues of buffering, timeouts, etc.

Furthermore, a variety of open source and vendor-supplied implementations of various portions of the WSA are available. We chose to use the open source, Apache Axis2 [9], web services software stack in our implementation of TSSN. This middleware dramatically reduced the amount of code to be written by our team. In particular, the Axis2 program suite includes modules providing implementation for the two specifications, WS-Security and WS-Eventing, that are the subjects of this paper.

The Axis2 module, Rampart, is "a module based on Apache WSS4J to provide WS-Security features"[10]. This module places handlers in the pre-dispatch phase of Axis2 message handling, with independent configurations for each service on the server side. In addition, the Rampart module can automatically extract and enforce the security policy(ies) specified in the service definition. This feature ensures that the security requirements of a service always match the service's published security policy. In addition, the service provider is able to alter the security policy enforced for the service operations by simply modifying its service definition – without making source code changes.

The Axis module, Savan, is an "implementation of WS-Eventing specification …designed as a general publisher/subscriber tool"[11]. When any service engages the Savan module, it provides implementation for a number of additional operations including Subscribe, Unsubscribe, Renew, and GetStatus.

III. Implementation Conflicts

In the design process, we did not specifically address how security issues for the publish/subscribe communications would be handled. Security issues for servers and clients were considered independently in the context of the WS-Security specification and implementation. This separation is an example of the divide and conquer engineering practice described earlier. The decision to consider these aspects of the design independently was concretely supported by the web services architecture documentation. The WS-Eventing submission begins with the statement: “This specification specifically relies on other Web service specifications to provide secure, reliable, and/or transacted message delivery and to express Web service and client policy.” [12]

Preliminary implementation tests were encouraging. We created a simple client/server test case. First, the service interface was formally defined in WSDL 2.0 (Web Service Description Language)[6]. In this first version, no security policy was defined in the WSDL for the service; and programmers generated a successful implementation for the service and a client based on the formal interface definition. In the next step, a security policy was attached to some operations defined for the service. The service was rebuilt against the modified WSDL definition, and immediately enforced the stated security policy on the selected operations. The (unmodified) client was able to access only the operations for which no security policy was enforced. With minor modifications to the client, to provide the necessary security credentials demanded by the modified service, the client was able to invoke all service operations.

In summary, this test indicated that a service could be implemented without regard to the service policy demanded for its operations. In addition, different security policies could be assigned at a very fine grain down to different operations within the same service.

Unfortunately, when we proceeded to integrate these concepts into the implementation of TSSN we encountered serious unexpected difficulties. The TSSN includes services that are also clients to other services. For example, the VNOC alarm processor service sends email messages containing sensor alarm notification coupled with relevant cargo information. In order to do so, this service must solicit information from both the MRN and TDE services. So, the VNOC alarm processor service also performs a client role to these two other services. Because alarms occur asynchronously, the MRN alarm processor uses the publish/subscribe protocol for exchanging information with its clients. It was very difficult to realize independent security policies for the VNOC alarm service and MRN alarm service for which it is a client.

This problem arises because of the interactions between the WS-Eventing specification (for the publish/subscribe communication) and WS-Security for the two services. For this discussion we label a service that is also a client to another service using the publish/subscribe protocol, as a *service with asynchronous client*.

A. Server Operations Obscured

When a service engages the Savan module to support WS-Eventing, Savan inserts handlers into the pre-dispatch phase of operations destined to the service, and implicitly provides features of the WS-Eventing specification for this service. The task of these pre-dispatch handlers is to detect and handle operations defined in the WS-Eventing specification. These operations include: Subscribe, Unsubscribe, Renew, GetStatus, however, these operations never appear in the WSDL for the user-defined service. So, these operations are “obscured”, that is, do not appear in the external, WSDL definition of the user-defined service. This has major impact on the specification and implementation of security for these operations as described in the following paragraph.

As demonstrated by our early test cases, using WSDL to formally define the service interface, including the security policy(ies) associated with its operation set, is a very powerful abstraction. Clearly, providing a standard, public definition of the interface promotes interoperability. In addition, middleware tools (we use, Apache Axis2) provide support for securing messages. If a service (client) engages the Axis2 Rampart module, handlers are inserted in the pre-dispatch phase to satisfy the security policy expressed in the service WSDL. The Rampart module automatically handles signing messages, encrypting/decrypting messages, applying and verifying timestamps, etc. This significantly reduced the coding effort for servers and clients. Perhaps, even more importantly, modification of security policy attached to service operations may require no modification of the service or client code, since securing messages can be handled by the underlying Rampart module.

Since these eventing operations do not appear in the WSDL, there is no place to express the specific security policy for these operations. This has adverse effects for both the publisher (service) and subscriber (client).

Although the service author cannot attach security policy to specific WS-eventing operations in the WSDL, it is possible to attach a security policy at the service level. Specification at this level, attaches the same security policy to every operation of the service. Sometimes this is an acceptable compromise, but in general, one wants the finer level of security control that WS-policy provides for other services. In our application, we require higher levels of security for an entity requesting subscription to a sensor's alarm events, than to other more general status operations to determine if a sensor is active, etc.

For a service with asynchronous client assigning a service level security policy implicitly applies the same policy to both the service side *and* client side operations. This effectively requires this service to utilize the security policy of the service supplying it with asynchronous input for its own operation set. This is completely counter to the WS-Security intention to allow each service independently define its own security policy.

On the client (subscriber) side, obscuring these additional service operations also makes it difficult to ensure that the client satisfies the security policy enforced by the server. The service stub generator, used to generate a framework for a client for the service incorporates code to direct the Rampart module on the client-side to satisfy the security policy appearing the service WSDL for each visible operation. Since the eventing specific operations do not appear in the service WSDL, this behavior is not available.

On the client side, the explicit server eventing operations are further hidden from the client author. The client code must include an external library (provided with the Savan module) that provides an abstract functional interface providing the WS-Eventing capabilities. Internally the Savan module invokes the Subscribe (and related) operations provided by the service. Unfortunately, Savan completely ignores security policy required by the service it contacts.

B. *Reversal of Client and Server Roles*

For the delivery of asynchronous messages from server to client, Savan effectively reverses the roles of the participants. The server (publisher) delivers a notification to the client (subscriber) by invoking a client operation. The name of the client operation corresponds to the notification type. So, for notification, the server requests the named operation to be performed by the client.

From the perspective of security management, the client must enforce the appropriate security policy for the incoming operation message. The client now controls the level of security for the delivery of asynchronous messages. It is much more appropriate for the server, who is the owner of the information being distributed, to control the level of security used for this message transfer.

C. *Multiple Subscribers, Different Security Policies*

As described in the previous section, the Savan module permits the client (subscriber) to define the security policy for the delivery of asynchronous messages. In addition to mis-assigning security responsibility, this role reversal may make it (nearly) impossible for the server to successfully deliver notification to all clients. Multiple clients may subscribe to a server for the same notification (a core feature of WS-Eventing), and each could enforce a different security policy in its implementation of the associated notification operation. The Savan module provides no support for delivering notification to different subscribers with different security policies, so special care must be taken to ensure that every client specifies the same security policy for notification operations.

IV. RESOLUTION

As described in section III, use of the Savan module to implement WS-Eventing seriously restricts the independence of security policy for servers and clients.

Initial attempts to specify security policy for publishing services failed. The current release of Savan for subscription support, ignores security policy required by the service it contacts. Since these operations are invoked below the level of the user-written client code, there is no obvious way to ensure that the service provider's security requirements are satisfied. In order to satisfy the security requirements for subscription in our system, we modified the Savan module, and added a new method to allow the client to transfer to Savan the service security policy.

If every subscriber to a server providing asynchronous notification is purely a client, then many of security policy conflicts can be resolved, though, informally, by the server describing the security policy it will apply to notification messages it sends; and expect all clients to respect this agreement. Unfortunately, this workaround significantly diminishes many of the advantages of WS-Security related to formally publishing security policy in the service description (in WSDL). In our experience, both server and client were required to embed in its implementation details of this security policy.

In the case of more complex clients, such as a server with asynchronous client, the challenges became more complex. For services of this type, it is necessary to distinguish those operations that define the interface of this service to its clients, from the operations added to handle the notifications sent to it. We devoted large amounts of time to testing and debugging to eventually generate services meeting specific security constraints for our application. The goal of generating a solution with a flexible and extensible set of security policies was compromised at the critical juncture where multiple services communicate asynchronously.

V. Conclusions

We were able to successfully field the TSSN as a set of web services and ensured that all links satisfied our security constraints. But to do so, we had to require that otherwise independent services shared the same security policy for their respective operations; and we had to embed security policy decisions directly into the implementation of some services. These compromises restrict the flexibility and extensibility of TSSN.

To fully resolve these issues, we believe that aspects of the WS-Eventing specification, and the Savan module implementation, in particular, need to be reviewed from the perspective of interactions with WS-Security.

The specifications should more clearly identify who is responsible for the definition of security policy for the delivery of messages from publisher to subscriber. The publisher of the information invokes an operation of the subscribing (client) service, thereby deferring the security policy to the operation owner, or client. However, since the publisher is the actual owner of the information being disseminated, it seems that it should maintain responsibility for the service policy to enforce when delivering messages.

In addition, the implementers of the Savan module should review the implementation strategy for the WS-Eventing specifications to ensure that it respects the security policies expressed in the WSDL definition of any service that activates the Savan module. In addition, consideration should be be given to make all WS-Eventing operations, such as subscribe, and renew, explicit in the WSDL interface for each service activating the Savan module. This is important to allow the pusblishing service advertise (and enforce) specific security policy for these operations.

Acknowledgment

This work was supported in part by Oak Ridge National Laboratory (ORNL) Award Number 4000043403. This material is also partially based upon work supported while V. S. Frost was serving at the National Science Foundation.

References

[1] David Booth, et al. "Web Services Architecture, W3C Working Group Note." http://www.w3.org/TR/ws-arch/, W3C 2004. October 2009.

[2] *W3C*. http://www.w3.org/, W3C 2009, October 2009.

[3] Martin Kuehnhausen, "Service Oriented Architecture for Monitoring Cargo in Motion Along Trusted Corridors," Master's thesis, University of Kansas, July 2009.

[4] D. T. Fokum, V. S. Frost, D. DePardo, M. Kuehnhausen, A. N. Oguna, L. S. Searl, E. Komp, M. Zeets, D. D. Deavours, J. B. Evans, and G. J. Minden, "Experiences from a Transportation Security Sensor Network Field Trial." to appear Proc. 3rd IEEE Workshop on Enabling the Future Service-Oriented Internet-Towards Socially-Aware Networks (EFSOI 09), Honolulu, Hawaii, USA, Nov. 2009.

[5] Anthony Nadalin (ed.) et al. "Web Services Security: SOAP Message Security 1.1." http://docs.oasis-open.org/wss/v1.1/, OASIS Open, Feb. 2006.

[6] Roberto Chinnic (ed.) et al. "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," http://www.w3.org/TR/wsdl20/, W3C 2007, October 2009.

[7] Asir S Vedamuthu (ed.) et al. "Web Services Policy 1.5 – Framework." http://www.w3.org/TR/2007/REC-ws-policy-20070904/, W3C September 2007, October 2009.

[8] Don Box (ed.), et al."Web Services Eventing (WS-Eventing), http://www.w3.org/Submission/WS-Eventing/, W3C March 2006, October 2009.

[9] "Apache Axis2/Java - Next Generation Web Services," http://ws.apache.org/axis2/, Apache Software Foundation 2009, October 2009.

[10] "Securing SOAP Messages with Rampart", http://ws.apache.org/axis2/modules/rampart/1_0/security-module.html, May 04, 2007.

[11] "Apache Savan/Java", http://wso2.org/projects/savan/java. WSO2 Inc., 2009.

[12] Don Box (ed.), et al. "Web Services Eventing (WS-Eventing): Introduction." http://www.w3.org/Submission/WS-Eventing/, W3C March 2006, October 2009.