Cristian Opincaru, Gabriela Gheorghe

Service Oriented Security Architecture

As Service Oriented Architectures (SOA) and Web services are becoming widely deployed, the issue of security is far from being solved. In an attempt to address this issue, the industry proposed several extensions to the SOAP protocol that currently reached different levels of standardization. However, no architectural guidelines have yet been proposed. In this paper we first outline the security challenges and the specifications that address these challenges and then present our concept the Service Oriented Security Architecture—SOSA. We argue that the different security functions (authentication, authorization, audit, etc.) should be realized as different stand-alone Web services These security services can then be chained together by means of Enterprise Application Integration (EAI) techniques such as message routing on Enterprise Services Buses (ESB). Next, we will present a prototypical implementation of this framework and describe our experiences so far. We show that by distributing the security functions, a more flexible architecture can be designed that would lower the costs associated with implementation, administration and maintenance.

1 Introduction

While Web services were designed to lower integration costs and to open new ways of doing business, many organizations are still reluctant to opening their businesses to Web services although standards like SOAP and WSDL are in place for almost a decade. One of the major factors for this is the inadequate understanding of the security risks involved and the false belief that they will have to make costly reinvestments in their security infrastructures [GFMP04]. In an attempt to make Web services a secure ground, OASIS [OASIS] has standardized a number of extensions to SOAP messaging which address different security issues related to Web services. These extensions are WS-Security, WS-Trust, WS-Federation, WS-SecureConversation and WS-Policy. In addition to the SOAP extensions, other security specifications can be used in combination with Web services—XACML [OAS05a], SAML [OAS05b] or the Digital Signatures Services [OAS07a] are some examples.

These specifications define how security techniques and mechanisms should be applied for individual SOAP messages (encodings, message exchanges, etc), but they do not define architectural guidelines for possible implementations. In this paper we address this issue by proposing an architecture for the realization of Web services security systems: the *Service Oriented Security Architecture*, *SOSA*. This architecture is based on the now popular Enterprise Services Bus (ESB) architecture. The idea behind it is to build modular security services that address well-defined security functions (i.e. authentication, authorization, etc.). Message routing techniques can be then used to combine these *security services* and to develop complex security solutions. *SOSA* builds on the idea that rather than creating a "fat" security component which is integrated in the application or in the messaging middleware (and therefore not portable and hard to reuse), it is more appropriate to build security into modular services that are platform independent, easier to test and document, and have a high degree of reusability. The remainder of this paper is structured as follows: Section 2 describes the main security issues that must be addressed in the context of Web services (references to the specifications that address these issues are made where appropriate), Section 3 presents architectural approaches for the implementation of security systems, Section 4 introduces the proposed architecture and describes its most relevant elements—communication between security services, service composition and possible security services, and in Section 5 we comment on some similar approaches. After this we present our prototype implementation, the *SOS!e* framework, in Section 6, make a functional as well as a performance analysis in Section 7 and finally present our conclusions.

2 Security Requirements for Web Services

The main security issues to be addressed by Web services, as also discussed in [GFMP04, WSF+03, W3C04], are enumerated below. Because

Web services are all about interoperability, we provide references to the specifications that address these security issues, where appropriate. Please note that, relative to the OSI network stack, Web services are located at the application layer. Therefore, in this paper we are only addressing application layer security; security at the lower layers is out of scope.

Authentication
The requester might be asked to provide credentials prior to accessing a Web service. Authentication is a key issue, since without knowing the requester's identity, other security functions cannot be accomplished—i.e. you cannot charge someone for using a service without knowing who he is. Authentication is addressed by various specifications, most importantly by WS-Security and SAML [OAS05b] (single sign on).

Authorization
Access to Web services should be restricted based on authorization policies, that is, clear conditions should be satisfied in order to allow an entity to access certain Web services. Authorization is addressed in XACML.

Confidentiality
The information flow between services must be protected. Special thought should be given to the fact that SOAP messages often pass through multiple servers before reaching their destination. Confidentiality is addressed in XML-Encryption and WS-Security.

Integrity
The information received by a Web service must be the same as the one sent by the requester. Messages must not be altered along the path. Integrity is addressed in XML-Signature and WS-Security.

Non-repudiation
The service provider should be able to prove that a requester used a certain Web service (requester non-repudiation) and the requester should be able to prove that the information he has originates from a certain service rovider (provider non-repudiation). Non-repudiation is addressed in XML Digital Signature.

Privacy
Both, service requester and provider should be able to define privacy policies. Both of them should agree on these policies prior to the actual delivery of the service. Privacy is addressed in WS-Policy and WS-SecurityPolicy.

Audit
User access and behavior should be traced in order to ensure that the established obligations are respected. Audit is enforced by audit guards, that can be both *active* and *passive* [W3C04].

Trust
Service requester and service providers should be able to determine if they trust one another. Both direct and brokered trust relationships should be taken into consideration. Trust is addressed in WS-Trust.

Accounting, Charging
These two aspects are not primarily concerned with the security of the system, but are nevertheless tightly coupled with the other security functions described above (i.e. charging requires the service to know the identity of the requester). Most eBussiness applications require a complete A4C[1] system.

3 Security Implementation Approaches

When it comes to implementing the previously introduced security functions in the context of Web services, several architectural approaches are possible. These are graphically presented in Figure 1 and described in the following.

Embedded in the Application
In this case the security implementation is coded in the application itself (Figure 1A). The developer of the Web service is responsible for writing the code that represents the security logic. For this task he will probably chose to implement some of the functionality himself, while reusing code from third party libraries to implement other security aspects. Example of such libraries include the Java Authentication and Authorization Service (JAAS) and the security features found in Web Services Enhancements (an extension to Microsoft .NET platform). Because communication between the security system and the application is done by APIs, the performance is very good in this case. However, this approach lacks scalability and results in implementations which are complex, hard to document and which have a low degree of reusability and extensibility. These findings are backed by [Bro03].

Embedded in the Middleware
In this case security is provided by the middleware system where the Web service is executing (Figure 1B). This is the case with most application servers such as the Systinet Server for Java [SSJ] or Apache AXIS [Axis]. Here, the security aspects are handled by the application server. Before and after the Web services hosted by the middleware are invoked, the messages are inspected and the security policies are enforced. In comparison with the previous

1. A4C is an acronym for Authentication, Authorization, Audit, Accounting and Charging.

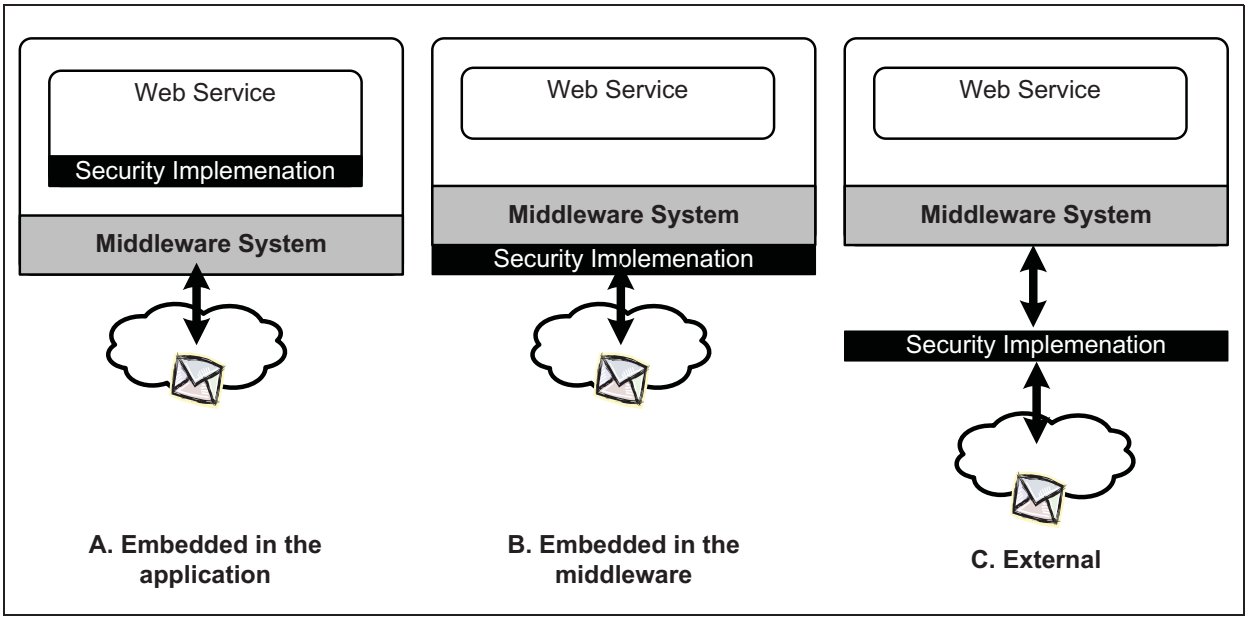

Figure 1 Approaches for the Implementation of Security Features in Web Services

approach, a noticeable improvement here is the fact that the security implementation is separated from the application logic. This leads to less complex implementations which are easier to document. Furthermore, it is possible to define security policies that cover several services which run inside the same instance of the middleware system. Nevertheless, porting the security implementation to a further middleware system is a major effort—if it can be accomplished at all. More, it is a remarkable challenge to define and enforce policies for services distributed on different middleware systems.

External
In this case security is implemented outside the middleware system (Figure 1C). The Web service is loosely-connected to the security implementation through a messaging interface. This approach is taken for example by XML firewalls—these are deployed at the network perimeter and enforce security policies by processing incoming and outgoing messages. [DGFRLP04] elaborates on application firewalls. Compared to the previous two approaches, there are two major differences: the security is decoupled from the application and the two communicate by means of messages. This makes the security implementation independent of the middleware system where the protected Web service runs and results in more understandable implementations (the security aspects are not mixed with the rest of the application). Furthermore, because the security system is essentially a Web service, it comes with the advantages a Web service brings: scalability, portability, higher degree of reusability. A decrease in performance is a possible disadvantage related to this approach, because there is a significant computational effort associated with message processing, especially if the messages are XML (as is the case of SOAP).

Mixed
Of course it is possible to have mixed approaches where some security aspects are implemented in the application, some in the middleware, while others are externalized.

4 The Proposed Architecture

In this paper we build on the external security approach described earlier and propose an architecture for security systems where the security functions are realized as small modular services. We call these services *security services*. In order to have a simple, understandable and verifiable design, the principle of separation of concerns is applied. According to this principle, the security system is functionally divided into services. These services can be regarded as infrastructure services, as they can be shared by applications living in the same network. This makes the design highly reusable. Additionally, through the use of standardized messaging interfaces, overall system portability is ensured. A taxonomy of possible security services will be presented hereafter.

Enterprise Application Integration (EAI) techniques are used to “glue” the security services together with the Web services which they are supposed to protect. Because of its flexibility, the Enterprise Services Bus

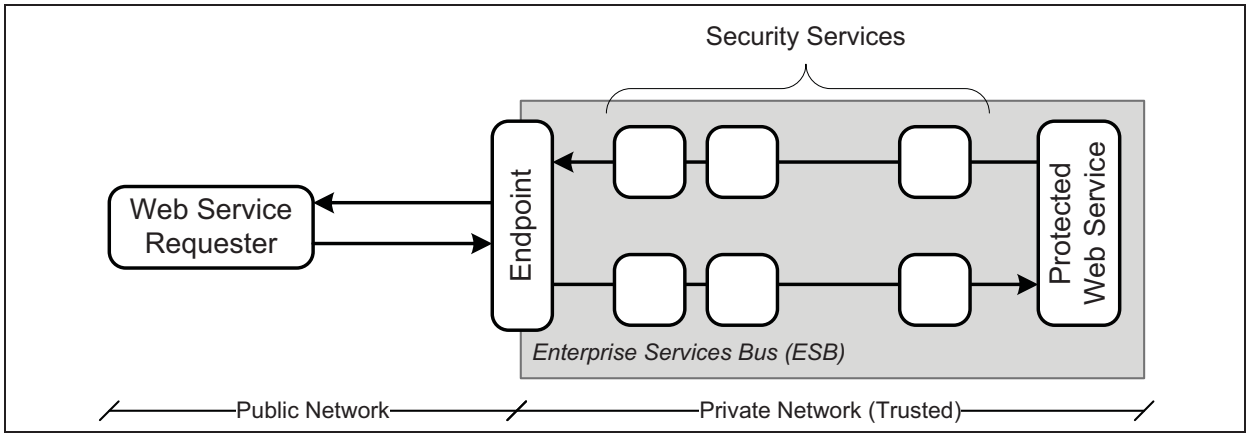

Figure 2 Security system composed of several security services

model is used. ESBs support both service orchestration and service choreography and implementations usually come equipped with simple-to-use orchestration editors and runtime environments which can easily be used to architect a security solution from security services. Probably two of the most important features of an ESB are message routing and the mediation pattern, which allow functionality to be built in the system in a totally transparent fashion.

In this paper, we consider that security services are realized as ESB mediations and that they are chained together by means of message routing. Other possibilities exist (such as for example BPEL or WS-CDL), however these are out of the scope of this paper. Mediations and message routing are enough to design scalable, extensible and easily congurable security systems.

The realization of such a system is illustrated in Figure 2. A message reaching the endpoint will be routed by the ESB through several security services before reaching the protected service. Each of these security services implements some security function and will enforce some portion of the security policy. The response message is also routed through several security services before being returned to the requester. In the proposed model, we consider that the security services trust one another and that they are located in a trusted network; scenarios such as service hijacking are out of the scope of the paper. Nevertheless, by using encryption and digital signatures, the model can be extended to include scenarios where the security services are only partially trusted (they are trusted to perform their task, but not trusted for anything else). However, this is not discussed here. Because this model is applied to Web services and Service Oriented Architectures (SOA) and because the core idea is to think of security in terms of reusable services, the model was named Service Oriented Security Architecture or *SOSA*. The following subsections present the main elements of this model, namely how security services communicate, how they are coupled together and what security functions can be implemented as services.

4.1 Communication between Security Services

Each of the security services will process incoming messages in order to accomplish its task. Some tasks may require several services to cooperatively process one message (for example authorization normally requires the identification of the requester). It is clear that in this case intermediary processing results (in the previous example, the identity of the user) need to be exchanged between services. Following the patterns described in [HW03], there are two possibilities of service intercommunication. The first approach is to have the two services communicate by means of a *shared database*: after processing a message, the first service stores the intermediary results in a database, while the second one later queries this database. The second approach employs *annotations*: the first service appends the intermediary processing results to the message before dispatching it to the next service; we call this an annotated message. For the particular case of security services, this latter approach is more appropriate because the intermediary processing results normally refer to the processed message (i.e. identity attributes, authorization decisions, obligations, accounting information, etc.) and can be therefore transported together with the message. To get an idea about how annotations work, think about a document-based work flow in a company: assume that Bob (an employee) wants a new computer. For this purpose, he prepares a written request and mails it to his boss. The boss will first analyze Bob's reasons, approve the request, perhaps annotate it, and forward it to the financial department. The financial department will verify the

request, if there are enough funds (perhaps annotate it), and send it to the Infrastructure department. Here the computer is ordered and a reply is sent to Bob informing him that his new computer is on its way. The persons involved in this work flow act similar to the mediation services: first inspect the request they receive, then they approve it (they can also reject it), they may annotate it, and finally forward it along the chain.

4.2 Possible Security Services

In Section 2 of this article, the security requirements for Web services were presented. Most of these requirements can be implemented as standalone Web services. In fact, several service interfaces for security services have already been standardized by OASIS as part of the WS-* specifications. Examples for this include WS-Trust [OAS07b], WS-Federation, XACML [OAS05a] or the newer DSS [OAS07a]. All these services are defined through WSDL documents and follow a request-reply pattern. In this article, as stated earlier, we are looking instead at implementing security as ESB mediation services. With this idea in mind, we argue that the following requirements can be wrapped into possible security services:

Authentication
Two types of authentication services are possible: *verification* and *identification*. The first one will verify the credentials (keys, passwords, etc.) found in a message, while the second one is responsible for providing identity attributes. An identification service will annotate messages with these attributes so that the other services along the chain (i.e. authorization, audit, charging) can use this information.

Authorization
If we follow the XACML [OAS05a] service model, three types of authorization services are possible: *Policy Information Point (PIP)*, *Policy Decision Pointn (PDP)* and *Policy Enforcement Point (PEP)*. The task of a PIP is to annotate messages with additional attributes that the PDP may require in the decision making process. The task of the PDP is to evaluate the message, produce an authorization decision and annotate the message with this decision and with any obligations, if requested. The task of the PEP is to enforce the decisions of the PDP services and to discharge obligations.

Audit
Two types of audit services can be envisioned according to [W3C04]: services that perform passive audit such as a *logging service* and services that perform active audit such as a *notification service*.

Cryptographic Services
Encryption and *digital signing* are tasks that require significant computational power. Therefore, distributing them on more powerful processors will often be a good choice.

Accounting
If accounting represents a complex task, it makes sense to realize it as a standalone service. The task of an *accounting service* is to meter service usage and to provide input for the charging service (in the form of annotations).

Charging
If charging is done immediately (i.e. not on a periodical basis), the task of the *charging service* is to charge the requester according to the information provided by the accounting service.

Infrastructure Services
In addition to the above mentioned services, other mediation services might be useful, especially if we think about coupling different security services together. [Cha04] identifies the following three: *orchestration services*, *message transformation services* and *message storage services.*

This list of services is not complete: depending on the concrete deployment scenario, other services may be required. Furthermore, the granularity of the services is also an issue to be considered: concrete implementations may incorporate several of the above-mentioned requirements into a single service (for example instead of PIP, PDP and PEP one single authorization service), for performance reasons. Alternatively, in complex systems consisting of different realms, messages may be routed through several PDP services, each one enforcing the policy of its realm. A detailed analysis of these issues is not within the scope of the article. However, in Section 7.1 we analyze the performance of our prototype implementation and comment on the relation between the number of security services and message delay.

4.3 Putting it all together: Message Routing Patterns

The next design step is to connect the security services with the application Web service. Because the security services are realized as mediation services on an ESB, message routing patterns can be applied in architecting the security solution. Some common patterns applicable here are the following ([HW03] elaborates on message patterns):

Content-Based Routing
Messages are routed between services based on their content (for example, incoming messages are routed to appropriate identification services depending on the authentication token they contain).

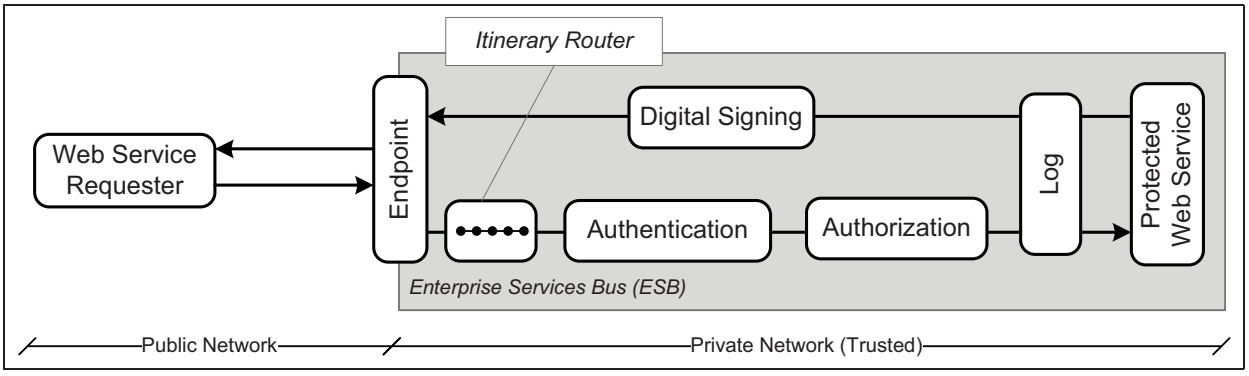

Figure 3 Security Services combined by means of message routing patterns

Itinerary Routing
A routing slip describing the itinerary is attached to the message. This is then forwarded according to the slip (for example a route may be *authentication – authorization – audit – protected Web Service*).

Splitter / Aggregator
The message flow does not necessarily need to be linear. One single request can be split (i.e. forwarded to several services that process it in parallel) and these parallel flows can be synchronized by means of an aggregator which combines the results. Imagine a message being sent in parallel to several decision services and then the authorization decisions being combined by means of AND / OR logic.

In order to illustrate how the proposed architecture fits together, an example is presented in Figure 3: after reaching the endpoint, an itinerary is attached to all incoming messages. According to this itinerary messages get first authenticated, then authorized, then logged and only then reach the Protected Web Service. On their way back, response messages are logged, digitally signed and only then returned to the requester. Please notice how services are reused: the same instance of the log service is used twice in the itinerary.

5 Similar Approaches

There are several similar approaches where security functions are implemented as standalone services. To begin with, the SOAP protocol specification [W3C03] describes intermediaries which can be either *forwarding intermediaries* (they forward the inbound message with minimal modifications) or *active intermediary* (they modify the outbound message in ways not described in the inbound message). Examples of intermediaries performing security tasks are also given in [W3C03]: a logging service is an example of a forwarding intermediary while an encryption service is an example of an active intermediary.

[Bro03] describes an architecture where security functions are implemented into proxies. A single security proxy acting as a gateway is used to secure several Web services deployed in a network. The paper compares this approach to library-based approaches. [AKT+06] enumerates the threats in the context of Web services and describes another external security approach. Here, incoming messages first pass through a *perimeter gateway* which secures several services within a network (similar to [Bro03]), then pass through a *service agent* which is attached to a particular Web service, and only then reach the protected Web service.

The security functions are divided in this case between the gateway and the agent: the first one enforces security at a coarser level (for the entire network), while the latter one does it at a finer level (for individual services).

While the above mentioned approaches do separate between security functions and the Web service to be protected, their design is not modular. In our approach, security functions are embedded within modular services, in their turn designed according to the principle of separation of concerns—different security functions should be implemented as different security services. The advantages of this approach are presented in Section 7.

An approach where security requirements are modeled into different services is presented in [HHH05]. Here, the services are combined by means of an ESB to form a security gateway. However, only authentication, authorization and cryptographic services are taken into consideration and no communication between security services is designed (in our approach security services communicate by means of annotated messages). Furthermore, no architectural analysis is made, no implementation is presented and hence, neither practical experiences nor performance analysis are described.

6 Implementation

In order to show the feasibility to this architecture, a prototype implementation was built: the *SOS!e*[2] framework. The implementation is open-source and was realized in Java. It relies on a number of open-source tools, including Apache Tomcat, Apache Axis, Apache Ant, WSS4J, OpenSAML and the Mule ESB [Mule]. It was designed for SOAP Web services and takes advantage of the SOAP processing model (security services are realized as intermediaries) and several SOAP extensions (most notably WS-Security). The framework implements message routing and the annotation-based processing model described above. On top of *SOS!e* several common security services have been developed.

Security services are realized as regular Web services based on the popular Apache Axis platform. The framework provides APIs for the manipulation of annotations. They allow the creation of new annotations, as well as the retrieval, modification and deletion of existing annotations from a message.

Annotations have been realized as SAML Attribute Assertions [OAS05b]. These can store several attribute-value pairs together with information about the author of the annotation, timestamp and other fields. SAML assertions have the advantage of being XML encoded. They are easy to attach to SOAP message headers (through the WS-Security SAML Token Profile [OAS06]) and have built-in support for digital signatures.

Message routing For this, the Mule ESB [ESB] was used. This is a 100% Java based Enterprise Services Bus implementation which supports a variety of transport mediums, message types and routing patterns. It also provides a very convenient and simple way to specify orchestration scripts and to expose these orchestrations as an endpoint.

In our implementation (refer to Figure 2), a proxy to the protected Web service is exposed in the public network. The Mule ESB is configured to route incoming messages through the necessary security services, before finally invoking the protected Web service. If a request-reply message exchange pattern is used (i.e. the call is not asynchronous), the same happens to the response message.

On top of the *SOS!e* framework, several security services have been prototypically built:

- Two *authentication services* which are able to authenticate users based on username, password and X509 certificates. If the verification is successful, an LDAP repository is contacted, user attributes are retrieved and the message is annotated with these attributes.
- A simple *authorization service* which performs authorization based on simple rules.
- Two *audit services*: one logging service and one alert service. The first one stores messages persistently (in whole or only parts of them), while the second one can be configured to send emails if certain criteria are met (these are specified through XPath expressions relative to the SOAP envelope).
- Two *cryptographic services*, one for encryption and one for digital signing. The parts of the message to be encrypted / digitally signed are also specied through XPath expressions relative to the SOAP envelope.
- Currently an accounting and a charging service based on PayPal [PayPal] are under development.

7 Analysis and Evaluation

It is well known that complexity is security's biggest enemy: as a system becomes more complex, it is more difficult to observe the flaws and back door opportunities that are created. *SOSA* splits security into small functional components that can be separately developed, thus reducing the complexity and allowing the components to be better tested (unit tests can be used). Moreover, having these components decoupled one from another triggers the ease of their reuse in different applications. Services are combined by means of message routing patterns. This allows for a clear design which is also easy to document. Because services are running inside an Enterprise Services Bus, orchestrating the security services is a matter of configuration which does not require an expert, as most ESBs are equipped with graphical editors and specialized tools for such purposes.

Additionally, the fact that the security services are completely decoupled makes the upgrades to the security system faster and less costly. New services can be introduced without affecting the existing ones, by simply altering the path of messages. It is not even necessary to stop the system, the modifications can be done at runtime, by simply temporarily redirecting the message flow (a technique often used when upgrading web servers).

2. SOSIE – Service Oriented Security, an Implementation Experiment. The name is inspired from the French word *sosie* (*look-alike* in English), because a proxy of the protected service is exposed through the framework.

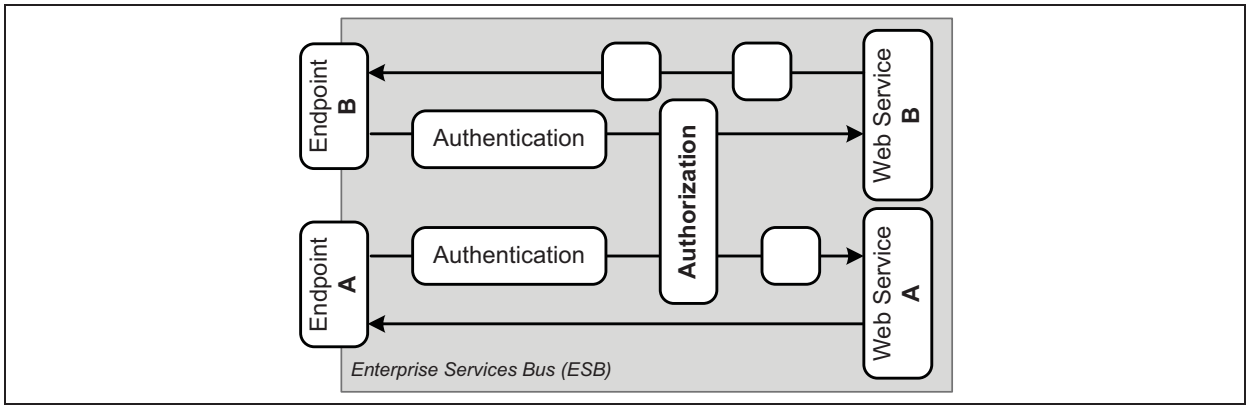

Figure 4 Security service being shared by two security system deployments

Since the security services are not bound to the application that they guard and also independent one from another, they can be developed in any programming language and can run on any operating system. This will reduce the costs associated with implementation because programmers will be able to choose the APIs and platforms which are most suitable for their project. For example, in an application where user information is stored in Microsoft Active Directory and authorization is based on XACML, the authentication service might be developed in C# (because C# has better support for Active Directory), while the authorization service might be implemented in Java (because Sun offers a free XACML implementation on SourceForge).

Another advantage of this architecture is that the same instance of a service can be used in several applications, thus making the administration and deployment of new services simpler. Figure 3 shows how the same instance of a security service can be invoked several times in an itinerary. Moreover, Figure 4 shows how the same security service can be used in two different deployments: in this example the same authorization service is used for both services A and B (the other security services, like the authentication, are different).

Furthermore, sharing security services also solves some well known security issues: sharing the authentication service leads to single-sign-on and sharing the authorization service leads to federated access control.

7.1 Performance Analysis

As a possible disadvantage to *SOSA* we see a decrease in throughput and higher latencies due to additional network traffic and overhead resulting from XML parsing (each security service must process the message content). In order to determine the feasibility of *SOSA*, performance tests were carried out against the *SOS!e* prototype implementation.

For tests, mid-class computers were used (Pentium 4 2.8GHz CPU, 2GB RAM, connected via 100MBit network). The results of these tests, together with the testing environment are displayed in Figures 5 and 6. We tried to determine the influences of the number of security services on the most relevant performance parameters—the throughput (TP) and the round-trip-time (RTT) for one message.

Our testing methodology is similar to the one described in [UT06]. In order to only measure the overhead introduced by our framework, "dummy" security services were used these are services that implement no security functionality. The protected Web service was a very simple one: the purpose was to have this one respond faster than the security framework (otherwise this one would have influenced the results).

For the measurements we used Apache JMeter. We considered two different configurations: Configuration A, where all the security services were hosted on the same machine as the Mule ESB and Configuration B where each security service was hosted on a different machine. For each of these configurations, we tested two use-cases: one where the services were only forwarding the messages and one where the services were processing the annotations existing in the message and adding new annotations.

Throughput

As seen in Figure 5b, the TP decreases significantly when the security framework was introduced between the client and the protected service (almost 60%). This is due to latencies introduced by the Mule middleware. However, if the number of security services is increased, the effect on TP is little. Furthermore there is no difference if annotations are used or not. This shows us that annotations have no visible influence on throughput.

Round Trip Time

As expected (see Figure 6), the RTT increases linearly with the number of security services introduced

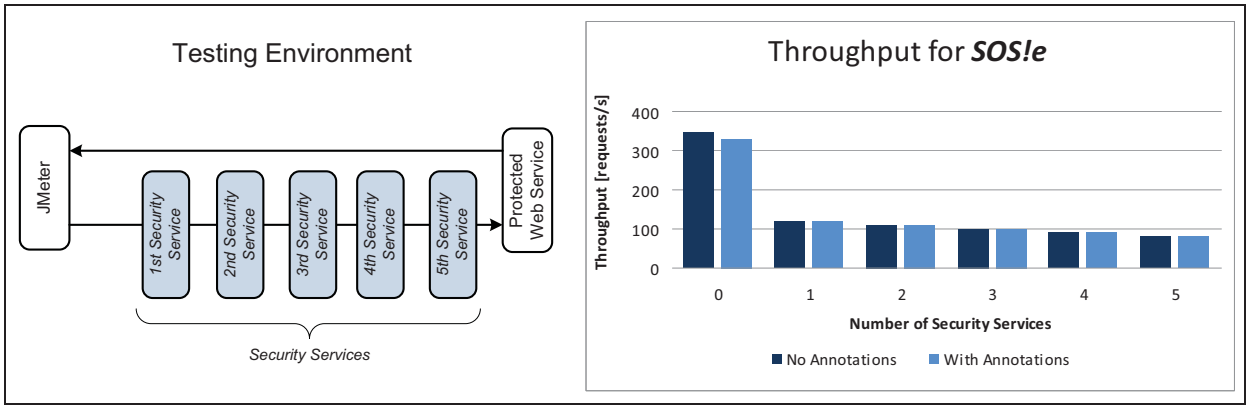

Figure 5 a.Testing environment b.Throughput for the SOS!e framework

between the requester and the protected service. The processing of annotations leads to a slight increase in latency.

In conclusion, we see that the framework has significant influence on the performance (both TP and RTT). The decrease in performance increases with the number of security services introduced between the requester and the protected service.

Whether or not the *SOS!e* framework is appropriate for a given scenario depends on the particular performance requirements of this scenario. In those cases where the RTT must be low, the *SOS!e* framework is inappropriate. However, in those cases where the RTT may be higher or if the interactions are asynchronous, *SOS!e* fits well.

Furthermore, we have to take into consideration that *SOS!e* is only a prototype implementation, which is not optimized. We are aware that more efficient implementations can be envisioned for the proposed architecture.

8 Conclusions

In this paper we presented an architecture for security systems protecting Web services the Service Oriented Security Architecture. We showed that realizing the security functions into modular, stand-alone security services results in less complex and more flexible designs for security systems. In addition to this, the presented approach has several other advantages (see Section 7).

We also presented a prototype, open-source implementation to *SOSA* , the *SOS!e* framework, and showed our experiences with this framework so far. In Section 7.1 we presented the results of performance tests that were run against our implementation, and showed that even though both RTT and throughput are affected by the fact that messages are routed through several security services, there are numerous application scenarios in which such an architecture fits well.

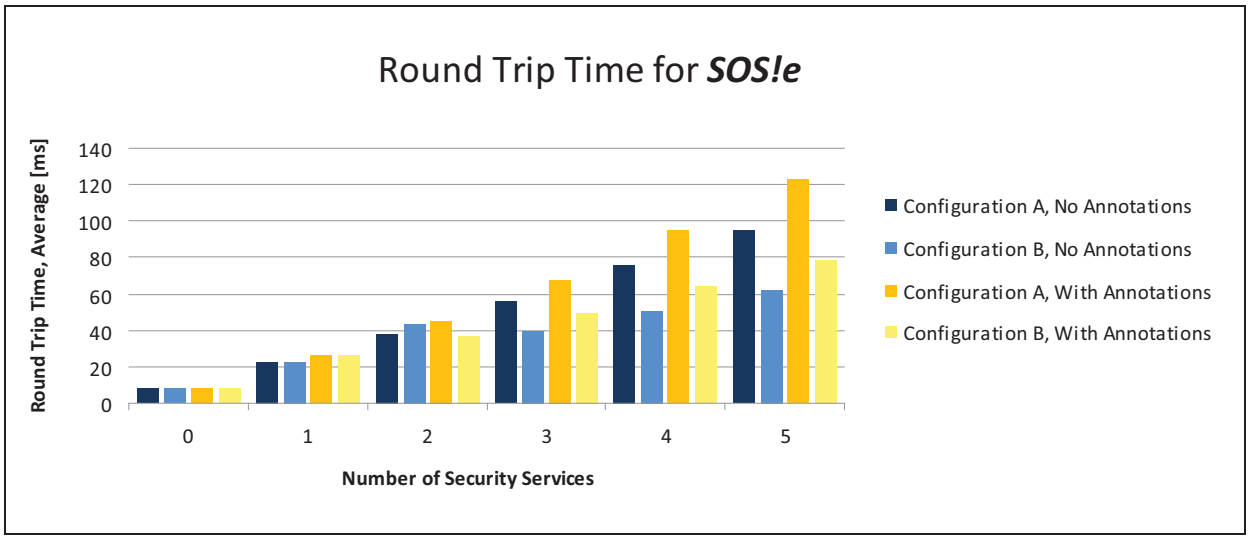

Figure 6 Round Trip Time for the SOS!e framework

References

[ACT+06] Mohamad Afshar, Nickolaos Kavantzas, Ramana Turlapati, Roger Goudarzi, Barmak Meftah, and Prakash Yamuna. Best Practices for Securing Your SOA: A Holistic Approach. *Java Developer's Journal*, June 2006.

[Bro03] G. Brose. Securing Web Services with SOAP Security Proxies. *Proc. Int'l Conf. Web Services (ICWS'03)*, pp. 231–234, 2003. [Cha04] David A. Chappell. *Enterprise Service Bus*. O'Reilly, 2004.

[DGFRLP04] N. Delessy-Gassant, E.B. Fernandez, S. Rajput, and M.M. Larrondo-Petrie. Patterns for application firewalls. In *Proceedings of the Pattern Languages of Programs (PLoP) Conference*, 2004.

[GFMP04] C. Gutierrez, E. Fern´andez-Medina, and M. Piattini. Web Services Security: is the problem solved? *Information Systems Security*, 13, pp. 22–31, 2004.

[HHH05] Heather Hinton, Maryann Hondo, and Dr. Beth Hutchison. Security patterns within a service-oriented architecture. *IBM white paper*, November 2005.

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[OAS05a] OASIS. eXtensible Access Control Markup Language v2.0, February 2005.

[OAS05b] OASIS. Security Assertions Markup Language V2.0 -Core, March 2005.

[OAS07a] OASIS. Digital Signature Service Core Protocols, Elements, and Bindings Version 1.0, February 2007.

[OAS07b] OASIS. WS-Trust 1.3, March 2007.

[UT06] K. Ueno and M. Tatsubori. Early Capacity Testing of an Enterprise Service Bus. *Proceedings of the IEEE International Conference on Web Services (ICWS'06)-Volume 00*, pages 709–716, 2006.

[W3C03] World Wide Web Consortium W3C. SOAP Version 1.2 Part 1: Messaging Framework, June 2003. [W3C04] World Wide Web Consortium W3C. Web Services Architecture, February 2004.

[WSF+03] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for Grid services. *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, pp. 48–57, 2003.

[OASIS] Organization for the Advancement of Structured Information Standards -http://www.oasis-open.org (20 June 2007)

[SSJ]Systinet Server for Java, http://www.systinet.com/products/ssj/overview (12 June 2007)

[Axis] Apache Axis, http://ws.apache.org/axis (11 March 2007)

[Mule] Mule Enterprise Service Bus, http://mulesource.com (10 July 2007)

[PayPal] PayPal, http://www.paypal.com (20 June 2007)

[Jmeter] Apache JMeter, http://jakarta.apache.org/jmeter/ (20 June 2007)

Cristian Opincaru

University of the German Armed Forces
Werner-Heisenberg-Weg 39
85577 Neubiberg
Germany
cristian@opincaru.ro

Gabriela Gheorghe

University of Trento
ICT International Doctorate School
Via Sommarive 14
38100 Povo
Italy
gabriela.gheorghe@disi.unitn.it