A Comparison of Two Approaches for Achieving Flexible and Adaptive Security Middleware

Tom Goovaerts Bart De Win Wouter Joosen

{tom.goovaerts, bart.dewin, wouter.joosen}@cs.kuleuven.be
DistriNet Research Group
Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenenlaan 200A
3001-B Leuven, Belgium

ABSTRACT

Open and dynamic business environments require flexible middleware that can be customized, adapted and reconfigured dynamically to face the changing environment and requirements. In this respect, the mechanism for composing middleware services with application code has an important impact on the kinds of adaptations that can be supported. This paper studies this problem in the context of security middleware. A bus-based architecture for integrating security middleware services is proposed and a qualitative comparison of the flexibility of the approach with an alternative AO-middleware-based approach is presented.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; D.2.11 [**Software Engineering**]: Software Architectures; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Design

Keywords

Composition, Middleware, Security Service Bus, Messaging, Aspect-Oriented Programming

1. INTRODUCTION

It is common practice to implement distributed enterprise applications on container-based middleware platforms such as Java EE or .NET. Security enforcement is one of the many services that are offered by these middleware platforms. However, due to the increasingly dynamic nature of typical modern enterprise settings, it must be possible to customize and adapt the security mechanisms, even at runtime. Vendors of middleware platforms strive to make their systems easy to configure and applicable to many different application domains at the same time. A consequence thereof is that the default security enforcement mechanisms of these systems are often too simple to use in practice.

The one size fits all model of middleware security in which the vendor provides a single black box security service for all possible environments is not (or no longer) feasible. Obviously, resorting to rebuilding security enforcement from scratch as part of the application is both insecure and costly at the same time. Instead, it is desirable to have a highly customizable security architecture that can enforce policies for many different applications. In this respect, an important challenge is how to compose the enforcement architecture with the applications in a flexible and adaptable way.

During the past decade of middleware research, techniques such as reflection [7], message interception [10] and component based development [4] have been used to make middleware more customizable and adaptable. More recently, enterprise middleware platforms such as JBoss and Spring have incorporated aspect-oriented techniques to achieve the same goal.

This paper studies the kinds of adaptations that can be supported by these flexibility mechanisms in the context of security middleware. The contributions of this paper are twofold:

1. We propose an architecture that uses a combination of message interception and a contract-enhanced message bus as a method for offering flexible security support.
2. We analyze the differences between the above approach for flexible security and an approach that is based on AO middleware.

Section 2 motivates the need for flexibility by discussing a case study and a set of adaptation scenarios. In Section 3, our architecture is briefly discussed. Section 4 investigates the advantages and disadvantages of our approach with respect to an AO middleware-based approach. Section 5 discusses related work and Section 6 concludes the paper.

2. MOTIVATION

In this section, we motivate the need for adaptation and we identify the kinds of adaptations that are analyzed in this work.

MidSec '08 December 1-5, 2008 Leuven, Belgium

2.1 Case study: a Personal Content Management System.

Nowadays, people acquire gigabytes of personal digital content (documents, pictures, videos, ...) and store it on many different locations (on desktops, hard disks, online services, ...). As the amount of content increases, its management and organization becomes very hard, which makes it a time consuming task to find specific content. Moreover, users want to be able to easily share their content with other users. The PeCMan project [6] aims at developing a platform that offers users a uniform interface for managing (reading, modifying, searching, storing, ...) and sharing their personal content that is scattered over various devices.

A deployment view of the PeCMan architecture is shown in Figure 1. The architecture follows a client-server pattern. The user interacts with the system via a client component (either a web client, a full desktop client or a client on a smartphone), via which he/she can add, remove, tag, modify or search for documents. The server component is a facade towards the system that processes incoming requests from the client. The Metadata Repository component is responsible for storing the metadata of documents. This metadata can be extracted from the documents or it can be user-generated in the form of tags and it is used for quickly finding back documents. The server-side components are deployed on an enterprise middleware platform.

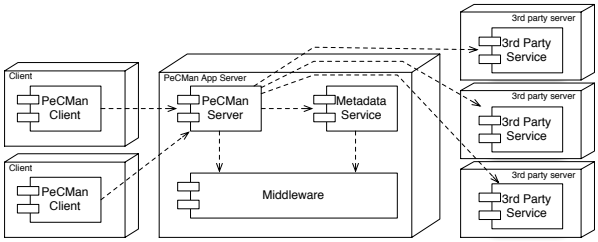

Figure 1: Deployment view of the PeCMan architecture. The dashed arrows represent dependencies.

A core feature of the system is that it is open and extensible by service providers that can enhance the core system with value-added services such as online storage, web publishing, face recognition or social network integration. It is the intention to use existing popular online services such as flickr, Youtube or Riya that fall outside of the control of the provider of the PeCMan service.

The PeCMan system is a multi-user system and it allows users to share their content with each other. Therefore, the system has the following high level security requirements:

- Users should be authenticated.
- Because of privacy reasons, access control for documents is required. The authorization policy can be influenced by an administrator or by the user itself. Authorization decisions can depend on attributes of a user or on metadata of the documents.
- Security-sensitive events should be audited.

2.2 Supporting Adaptation of Security Middleware Services

The main reason for performing adaptations to middleware security services is to deal with changes in the environment. The two most important changes are changes in the security requirements of the applications (eg. a new threat is identified) and changes in the application itself (eg. a new component is introduced). In a highly dynamic system such as PeCMan that is frequently extended with third party services, both of these classes of changes occur relatively often.

Moreover, it is no longer sufficient to merely support these adaptations at configuration or deployment time. Instead, it must be possible to reconfigure the middleware dynamically while the application is running.

In the rest of the paper, we make the distinction between a *security service* and a *security component*. By *security service*, we denote the presence of a certain security feature. A security service is implemented by a one or more *security components*, which are the software components that provide the service at runtime. The following scenarios are the adaptations that are discussed in this paper:

S1 Change a local parameter of a security component (eg. the encryption method for an audit service).

S2 Introduce new security functionality (eg. add a secure logging component).

S3 Compose/recompose a deployed security component with one or more application components. Application components depend on the security component but the security component can also depend on the application component(s) (eg. for context-based access control).

S4 Replace a security component with another one (eg. replace authorization decision engine with a different one).

S5 Compose a security component with a (new) third-party component that is deployed elsewhere.

S6 Change a security policy. Since the security policy explicitly depends on application-level concepts, a change in a security policy can require adaptations to the composition of security with the applications.

3. FLEXIBLE SECURITY ON THE SECURITY SERVICE BUS

Our approach, which is called the Security Service Bus (SSB), relies on a communication bus for providing flexible composition between application components and security components and mutually between security components. In this section, we briefly discuss our architecture and we illustrate how it supports the adaptation scenarios from Section 2.2. For a broader discussion of the SSB, we refer to [5].

3.1 Architecture

Figure 2 gives an overview of our architecture. In the bottom side of the figure, three remote security components are shown that realize the security requirements for the PeCMan system. The Authentication (ACN) component is used to verify user credentials, the PDP is used as an authorization policy engine that evaluates access requests and returns access decisions, and the Secure Logger provides auditing functionality. Each of these components is deployed

on a separate host. In order to correctly cover the security requirements, these components need to be placed in a composition with the application components and with each other.

3.1.1 Component Model

We use a simple component model for representing the various parts of the security enforcement infrastructure. There are two classes of components: *security components* and *application bindings*. We discuss each of them in this section.

Security Components.

The security functionality is offered by *security components*. Security components are reusable modules that can be invoked, managed and composed with applications and with each other. Security components have a well-defined *security interface* that specifies its core security functionality. For instance, for the PDP, it contains operations for making authorization decisions given certain contextual information. Furthermore, a security component contains a *management interface* that specifies operations that can be invoked to configure and manage the security components, such as loading a certain policy or enabling/disabling the service. In a central *registry service*, the SSB keeps track of all security components that are registered and it can invoke their functionality when it is needed. The security components themselves can use the SSB for obtaining application-level context information from application components.

Application Bindings.

Application components are bound to the SSB by means of adapter components that are called *application bindings*. The application binding has two responsibilities: it processes, transforms and enforces decisions for invocations that are intercepted by the middleware and it provides an attribute function that can be invoked by security components to obtain policy information. When a security-relevant event happens in the application component, the interceptor blocks and requests the SSB for an access decision for the event. This is illustrated in Figure 3. Likewise, whenever a security component needs a particular piece of context information, the SSB mediates the request and ensures that the application binding that offers the information is queried.

The SSB uses a data model for representing policies and policy evaluation requests that is compatible with many state-of-the-art policy technologies. Application Bindings translate concrete application requests into requests in terms of the abstract data model. The data model defines subjects, resources and their actions, and it defines which actions can be performed by a subject on a resource.

3.1.2 Composition

Security Contracts.

Besides having a general interface for performing security logic, components on the SSB can be enriched by *security contracts*. Each security contract explicitly states the types of events or requests that the particular component will exchange on the SSB. In the *required* part of the contract, the required functionality of the SSB is listed. In the *provided* part, the provided functionality is listed. Contracts are expressed in terms of the data model. Currently, there are two types of contracts: authorization contracts between enforcement points and PDPs and attribute contracts between PDPs and attribute functions. An example of a simple security contract for the Metadata Service Binding is shown below:

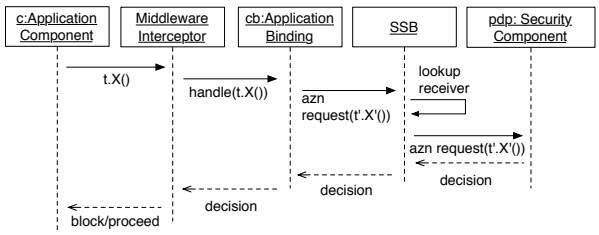

Figure 3: Sequence diagram for the enforcement of an access decision.

```
SecurityContract MetadataService {
    AttributeContract {
        provided {
            Document.owner;
            Document.sensitivity;
        }
    }
    AuthorizationContract {
        required {
            Document.read;
            Document.write;
        }
    }
}
```

This contract states that the Metadata Service provides the `Document.owner` and the `Document.sensitivity` attributes as sources for policy evaluation and that the service requires policy decisions for the abstract `read` and `write` actions.

Composing Security Services with Applications.

The composition of application components with security components happens at two levels. The application component and its binding are composed with each other by means of interception (the solid arrows in Figure 2), and the application bindings are composed with security components by means of the SSB (the dashed arrows in Figure 2). The SSB keeps track of all interfaces and contracts in the registry service. When messages are forwarded to the SSB, it automatically mediates the messages to one or more components that expect the message, according to their contract and it sends the response(s) back.

Domains.

The broadcast domain for messages is constrained by placing each component in a domain hierarchy. There is one domain hierarchy per type of contract. Within each domain, there are membership constraints on the components that can join or leave the domain. These constraints guarantee that the SSB can always validly route requests and responses. Constraints for authorization and attribute contracts require that there is always a single provider of a given attribute type or authorization decision. If there is no provider within a domain, it is possible to make a com-

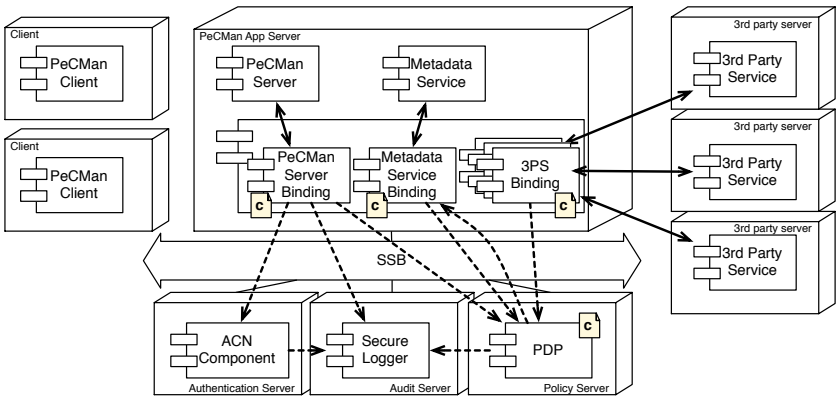

Figure 2: SSB-based Composition of the Application with Security Components.

ponent in the parent or in a child domain responsible for certain requests.

3.2 Supporting Adaptation

The core flexibility mechanism of our approach is the bus architecture, which allows decisions about which component receives which request to be delayed until runtime. In this section, we illustrate how our architecture supports flexibility by illustrating how the adaptation scenarios from Section 2.2 can be realized.

S1 Changing local parameters is straightforward by directly invoking an operation on a security component.

S2 To support a new class of security components, an implementation of the component has to be deployed on the SSB. When additional guarantees on the composition are to be supported for the new class of components, a new class of security contracts has to be developed.

S3 Composition of a security component with other security components or with application bindings is automatically realized based on a combination of contract specification and domain membership. An item from the provided part of the contract causes the service to receive messages from all services that specify that they require that item, under the domain constraint. Composition with application components depends on the application binding. Making changes to this component is costly, but still remains significantly more straightforward than changing the application code itself because only public interfaces of the application components are used.

S4 Replacing an existing security component with a different one is performed at runtime in three steps. First, a new component is deployed that implements the same interface than the original component and that holds a contract that is equal or more general than the contract of the original component. Second, the original component is deactivated, and finally, the new component is activated. Due to the fact that the contract is more general than the original one, the new component is able to provide at least the same required functionality as the old one.

S5 Requests to and from external services are processed and policed by deploying an application binding that intercepts traffic to and from the remote components.

S6 In our framework, the contents of the contracts depend on the policies that are deployed within the security components. Therefore, a policy change causes a change in the contract. Whenever the change in the contract is allowed according to the domain rules, our framework guarantees that the new policy will be correctly evaluated. Our architecture can be integrated with the policy specification tool to help the policy specifier reason about whether the policy he/she is specifying can be evaluated in practice.

4. COMPARISON WITH AO MIDDLEWARE BASED COMPOSITION

Many middleware platforms offer AO support not only as a programming model, but also as a middleware customization mechanism. The AO-based customization of these popular platforms is a likely candidate for the integration of custom security enforcement with applications. In this section we analyze the integration of security enforcement using AO middleware and we compare the differences with our approach with respect to the adaptation scenarios.

The features of the AO middleware directly influence the kinds of adaptations that can be supported. More specifically, the dynamic runtime adaptations to the composition of the security enforcement components we focus on, require two important features. First, the AO model must support dynamic weaving to allow runtime deployment/undeployment of aspects. Second, it must be possible for software to change aspect compositions at runtime. Many popular AOP systems such as JBoss and Spring support these features.

4.1 AO-based Composition

There are two levels of components that need to be composed with each other (see Figure 4). First, the application

components need to be mapped to their bindings. Second, the bindings need to be composed with the security components. Both of these compositions can be implemented using AO middleware.

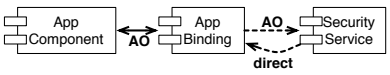

Figure 4: Composing Security Components with Application Components Using AO Middleware.

The application bindings represent the abstract data model as interfaces in one of the supported middleware languages. For instance, if the PeCMan service has the operations `updateDocument(document)`, `getDocument(document)` and `getDocumentMetadata(document)`, the application binding can have a `Document` type with `read()` and `write()` operations. AO-based composition can then map the application-level operations on the abstract operations in the binding.

The second kind of composition that can be performed is the composition between the application binding and the security components. On AO middleware, this can be realized by defining an AO-based composition between these components in which an invocation to the appropriate security operation is inserted before the execution of the security-sensitive operation on the binding. For instance, before the invocation of `read()` and `write()` operations, an invocation to the `checkPermission` operation of the authorization component is inserted. The pointcut of the aspectual composition controls when security functionality is invoked. Other security components can be bound by including the relevant application joinpoints in the composition for that security component.

We have illustrated one possible AO-based composition approach for integrating security components. However, in more advanced scenarios, such as when mappings get more complex or when application bindings need to maintain state, it might be better to introduce new adaptation aspects between the composition. Nevertheless, we believe that in most cases, the approach mentioned above is sufficient.

4.2 Supporting Adaptation

In this section, we revisit the adaptation scenarios for the case of AO composition.

S1 Adapting local parameters of security components is performed similarly to our approach by invoking an operation on the security component.

S2 To support a new class of security components, a new security component implementation should be provided.

S3 (Re-)Composition of a security aspect with other security components or with application components is realized by setting or changing the pointcut of the security aspect for that component. The kinds of compositions that can be performed depend on the joinpoint model of the AO middleware. Due to the very powerful composition model of AOP, it is hard to control or restrict which security component can be composed with which application component.

Table 1: Summary of Flexibility Support for Both Approaches. Legend: support for adaptations is well (+), neutral (0), bad (−) or bad when a new contract type needs to e defined (NC:-).

	S1	S2	S3(AB)	S3(SS)	S4	S5	S6
SSB	+	+ NC:-	-	+	+	+	+
AO	+	+	+	0	0	+	-

S4 Replacing an existing security component by one that adheres to the same interface is done by undeploying the composition for the original component and replacing it with an advice for the new component that has the same pointcut(s) as the original aspect.

S5 Since external services fall outside of the control of the policy enforcing party, proxy components for the external services are needed. If these proxies are not explicitly part of the application, they typically exist internally at the middleware level as an endpoint for the communication protocol. By weaving the security aspects to these proxy components, security policies can be enforced.

S6 AO-based composition of security enforcement does not take into account the effectively required information from the policies. Therefore, it is hard to support policy changes. If the new policy needs new application-level attributes, the policy evaluating component still needs to invoke the application binding directly.

4.3 Comparison

In this section, we compare how well the adaptation scenarios can be supported in both approaches. Table 1 summarizes this. The most straightforward adaptions S1 and S2 are supported well in both approaches. However, when a new contract type needs to be defined on the SSB, substantial manual effort is needed, with the gain of having additional guarantees on compositions with the new class of security components. Changes to the composition between the application component with its binding (S3(AB)) are harder to support using the SSB approach. In the best case, only the message interceptions need to be reconfigured, but it is also possible that the binding component itself has to be changed. The composition of the binding with the security components (S3(SS)) is more flexible and easier to change at runtime using the SSB approach (although this depends on the runtime changes that are possible on the AO middleware), and it makes it also easier to reason about the compositions. The replacement of a security component (S4) is possible in both approaches, but in typical AO middleware platforms, it still requires manual intervention. Intercepting traffic towards external services (S5) is easy using interception, but the translation to the abstract policy model limits its flexibility. In the AO approach, this can only be supported when there is a local proxy for the remote component. The security contracts ensure that changes in a security policy (S6) are easier to support using the SSB.

The conclusions from this comparison are twofold. First, for the composition between application components and their bindings, AO is the more powerful alternative over middleware-level interception. Simple AO platforms often

lack flexibility, but when the platform is powerful enough to support runtime programmatic changes to the composition, the AO-based approach is the more powerful option. Therefore, when binding an application that is deployed on AO middleware, AO is the superior implementation technology.

On the other hand, for the composition between the binding and the security components, the bus-based approach is preferred. While the level of flexibility of some modern AO-based platforms has become adequate for performing adaptations, to the best of our knowledge, there is no platform that is capable of taking the effective policies into account in the composition. As a result, it is harder to reason on whether policies can effectively be evaluated. The inclusion of this information (in the form of contracts or in another form) in the AO middleware composition process is therefore an interesting opportunity for further research.

5. RELATED WORK

One of the core mechanisms for making security middleware more flexible and adaptable is by designing them in a component-based way. Beznosov motivates this in detail in [2]. In this work, the component-based architecture of RAD [3] is discusses and is applied to ASP.NET. The Java Authorization Contract for Containers [11] specifies interfaces for plugging external policy engines in Java EE containers. In these systems, flexibility is provided, but only statically in the form of configuration. We believe that future middleware should be adaptable at runtime in reaction to dynamic interactions. This is the main reason why we chose a message bus for security component composition.

McDaniel [8] proposes a flexible security policy enforcement architecture for the Antigone group communication system. To the best of our knowledge, this is the only work that combines a component-based security enforcement architecture with a bus-based communication model. However, in this architecture, the composition logic is essentially scattered over all components on the bus. There is little or no support for reasoning on policy evaluation.

An important drawback of the AO approach is that it is harder to reason on the composition of the security components. Several solutions to this problem have been described. In Caesar [9] and Jasco [12], for instance, the dependencies between aspects can be described based on component technology. In Caesar, the dependencies are made explicit via provided and expected interfaces in an Aspect Collaboration Interface (ACI). In Jasco, this is more implicit, but it can be supported via systematic construction [1].

The authors of [13] propose the access interface (ACI) as a contract between authorization services and the applications. The view connector defines maps the application code to the ACI and is implemented using AOP. Since the ACI is roughly equivalent to our abstract data model, this work provides one possible approach for implementing an AO-based composition between an application and a binding.

6. CONCLUSION

In this paper we have studied the impact of two mechanisms for achieving flexible composition of middleware services with the applications in the context of security. Based on a comparison of both approaches in which we analyzed their support for a set of common adaptation scenarios, we conclude that both approaches are complementary. The conciseness of the AO-based approach makes it more appropriate at the application side of the composition, whereas the large runtime flexibility of the bus-based approach make it more appropriate at the security component side.

In future work, we want to study other qualities of the composition mechanisms, such as security and performance. Furthermore, we want to make the comparison quantitative by comparing concrete implementations of both approaches.

7. ACKNOWLEDGEMENTS

This work is partially funded by the IBBT PeCMan project.

8. REFERENCES

[1] S. O. D. Beeck and J. Gregoire. A contract language for aspectual components. Master's thesis, Katholieke Universiteit Leuven, 2005.

[2] K. Beznosov. On the benefits of decomposing policy engines into components. In *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 183–188, New York, NY, USA, 2004. ACM.

[3] K. Beznosov, Y. Deng, B. Blakley, C. Burt, and J. Barkley. A resource access decision service for CORBA-based distributed systems. *Proceedings of the 15th Annual Computer Security Applications Conference*, page 310, 1999.

[4] M. Fleury and F. Reverbel. The JBoss extensible server. In M. Endler and D. Schmidt, editors, *Middleware 2003 — ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *LNCS*, pages 344–373. Springer-Verlag, 2003.

[5] T. Goovaerts, B. De Win, and W. Joosen. Infrastructural support for enforcing and managing distributed application-level policies. *Electronic Notes in Theoretical Computer Science*, 197(1):31–43, Feb. 2008.

[6] IBBT. PeCMan project (Personal Content MANagement). http://projects.ibbt.be/pecman, 2007.

[7] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.

[8] P. McDaniel and A. Prakash. A flexible architecture for security policy enforcement. *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, 2, 2003.

[9] M. Mezini and K. Ostermann. Conquering aspects with caesar. pages 90–99, 2003.

[10] P. Narasimban, L. Moser, and P. Melliar-Smith. Using interceptors to enhance corba. *Computer*, 32(7):62–68, Jul 1999.

[11] Sun Microsystems. Java authorization contract for containers (JACC) version 1.0.

[12] D. Suvée, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. pages 21–29, 2003.

[13] T. Verhanneman, F. Piessens, B. De Win, and W. Joosen. Uniform application-level access control enforcement of organizationwide policies. *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 431–440, 2005.