Comparison Study of Aspect-oriented and Container Managed Security

Paweł Słowikowski, Krzysztof Zieliński
AGH University of Science and Technology
keywords: aspects, AspectJ, security, J2EE

Introduction

The goal of the paper is to present that aspect-oriented security provides very powerful mechanisms which could enhance container managed security. Security of container based systems usually depends on security mechanisms provided by containers in which the components run. The security functionality of containers is often limited and hard to modify. Aspect-oriented programming (AOP) allows weaving a security aspect into a component based application providing additional security functionality or introducing completely new security mechanisms. Implementation of security with AOP is a flexible method to develop separated, extensible and reusable pieces of code called aspects.
In this paper we discuss some specific security solutions based on Java 2 Enterprise Edition (J2EE) architecture [1] and JBoss 3.0 application server [2], aspect-oriented programming with AspectJ 1.1 [3], Java Authentication and Authorization Service API (JAAS) [4] and Resource Access Decision Facility (RAD) [5]. First, the security requirements in component based systems have been discussed. Next, container managed and aspect-oriented security mechanisms are presented in more details. Then, to demonstrate and compare techniques of implementation of different security mechanisms we will use a simplified pseudo-banking application (written with EJB).

Security requirements in component based systems

Security is a very important issue in enterprise-class information systems, especially in portal/Internet applications. It is related to a possibly huge quantity of sensitive resources in the system and potentially unlimited users' access to the application. A violation of security may cause catastrophic consequences for organizations whose security-critical information may be disclosed without authorization, altered, lost, or destroyed.
There are many standards of security functionality requirements for applications [6,7,8]. In this paper the focus is on the most important security functions:

- identification and authentication: the process of establishing and verifying the claimed identity of user,
- access control: the prevention of unauthorized use of a resource, including the prevention of use of a resource in an unauthorized manner,
- accountability: the property that ensures that the actions of an entity may be traced uniquely to the entity,
- audit: an independent review and examination of system records and activities in order to test for adequacy of system controls, to ensure compliance with established policy and operational procedures, and to recommend any indicated changes in control, policy, and procedures.

In the following sections it will be shown how the above mentioned security issues are addressed in component based applications with container managed security and how they can be addressed by applying aspect-oriented programming.

Container managed security

Container managed security is a type of security mainly applied in component systems. Standard examples of such applications are the ones developed in the J2EE architecture. The container implements the security infrastructure and the component is free of security logic. This implies that security functionality of application is limited to security provided by the container. The application servers allow managing security policy on the level of user and his role in a system on the one hand, and on the level of access control to invoking methods on the other hand. There are no standard mechanisms to control access to the methods depending on method's arguments or to hide parts of the result. The security is on the level of classes, but not on the level of instances. It is also not possible to force access control based on other properties such as the time of day or physical location.
The standard security in the J2EE architecture is declarative security. The information about application's security configuration is located in the deployment descriptor external to the application. Since the security mechanisms are not hard coded in the component, the security policy of application can be set up during deployment without the need to modify the code of the components. The following example illustrates how to enforce an EJB container to restrict access to `BankEJB` only to principals (users) that are permitted the Client role. The component `BankEJB` doesn't require to be changed or recompiled.

```
<session>
        <ejb-name>Bank</ejb-name>
        <ejb-class>BankEJB</ejb-class>
...
        <security-role-ref>
                <role-name>Client</role-name>
        </security-role-ref>
...
</session>
```

Listing 1 Part of an application descriptor providing declarative security

Declarative security can be completed with programmatic security. Programmatic security allows adding some code providing security inside the code of components. The code is able to obtain security information from a container but that information is limited to the principal and his security role. Implementation of security directly inside the component is time-consuming and requires the provider of container's business logic to have excellent knowledge of security mechanisms and security policy the application should satisfy. Some J2EE product providers extend security functionality of their application servers but in that case applications utilizing new features are not portable to other application servers and components are mostly not reusable. Both declarative security and programmatic security don't support the cooperation with application servers other then the J2EE ones.

Aspect-oriented security

Security is hard to implement correctly with traditional programming techniques. AOP offers a new approach to the problem. The AOP concept assumes separation of crosscutting concerns and moving them into separate and independent modules called aspects. Object-oriented technique allows separating concerns that could be mapped to concrete objects. It is hard to separate more abstract and complex properties of the system that cut across many other units of modularity. An example of such concern is security. Aspects allow us to precisely and selectively define objects, methods and events the security should be applied to.
AspectJ is an aspect-oriented extension to Java. It enhances Java language with aspects features such as possibility to define events in the execution of program (called pointcuts) and actions (called advices) that execute at join points picked out by a pointcut. AspectJ also

allows to modify classes and their hierarchy by adding new members to classes and alters the inheritance relationship between classes. Aspects can affect both source code and compiled classes. The second case is very important if we want to apply aspects to components which are packages of compiled classes. That solution gives us possibility to enhance existing or add completely new security functionality to the component without having to modify sources. The code implementing security is separated from implementation of components' business logic. Implementing security for J2EE applications with AspectJ we can use functionality provided by the container as well as any available external security library.

Comparison of security implementation techniques

This section presents examples of implementation of authentication, access control, accountability, and audit for J2EE application. The examples serve to compare container managed and aspect-oriented security.

Identification and authentication

In our examples authentication is based on JAAS. In order to authenticate the client application we have to initialize the security manager: configure a login module, create an instance of `LoginContext`, and log in [9]. Listing 2 shows implementation of authentication in pure Java. Authentication takes place just after running the client application. The code marked in gray is required to authenticate the client of `bank` component.

```
class BankClient {
       LoginContext lc = null;

       public static void main(String[] args) {
// Callback to get username and password. Required by LoginContext
              AppCallbackHandler handler = new AppCallbackHandler( "scott", "echoman" );

              try {
                     lc = new LoginContext( "Bank", handler );
                     lc.login();
              } catch( LoginException e ) {
// ...
              }
// ...
              BankHome homeBank = (BankHome) ctx.lookup( "ejb/Bank" );
              Bank bank = homeBank.create();
              System.out.println( bank.getAccountInfo( "bill" ) );
// ...
              try {
                     lc.logout();
              } catch( LoginException e ) {
// ...
              }
       }
}
```

Listing 2 Authentication of a client application with pure Java

We can achieve analogous functionality with AspectJ as shown in the following listing.

```
class BankClient {
       public static void main(String[] args) {
// ...
              BankHome homeBank = (BankHome) ctx.lookup( "ejb/Bank" );
              Bank bank = homeBank.create();
              System.out.println( bank.getAccountInfo("bill" ) );
// ...
       }
}

aspect BankAspect {
       LoginContext lc = null;
       pointcut mainExecution():
               execution( public static void main( .. ) );

// Login before execution of main()
       before(): mainExecution() {
               AppCallbackHandler handler = new AppCallbackHandler( "scott", "echoman" );
               try {
                      lc = new LoginContext( "Bank", handler );
                      lc.login();
               } catch( LoginException e ) {
// ...
               }
       }

// Logout after execution of main()
       after() returning: mainExecution() {
               try {
                   lc.logout();
               } catch( LoginException e ) {
// ...
               }
       }
}
```

Listing 3 Authentication of a client application with AspectJ

The part of application providing the authentication mechanism is placed in the aspect. The code providing security is separated from the remaining code of application.

Access control

Access control is a server-side mechanism. After invoking a method the decision has to be made whether the method is allowed to be executed or not. In the case of container managed and declarative type of security the decision is made by container based on an application descriptor as it was shown in Listing 1. The container that makes a decision about access to the component allows for principal identifying the client, client’s security roles, and roles required to get access to the component or its methods. It is not possible to base access control on values of method’s arguments, the time of the day, or the physical location of the caller. In that case it is possible to apply programmatic security and use some limited security information obtained from the container but then access control has to be hard coded inside the component.
Let’s presume that we have an external access control server with regard to the application server and we want to base authorization on that server. The authorization server is compatible with RAD specification, so an authorization decision depends on: the resource we want to access, operation on the resource, and credentials of the request’s caller. The following listings show how RAD based access control could be implemented with and without aspects:

```
class BankEJB implements SessionBean {
       public String getAccountInfo( String customer ) {
// access control
// get instance of RADConfig to prepare parameters for access_allowed()
               RADConfig radConfig = RADConfig.getInstance();
// prepare an instance of ResourceName to identify the customer’s account
               ResourceName resourceName = radConfig.getResourceName(
RADConfig.CUSTOMER_ACCOUNT, customer );
// prepare the operation that is performed on the resource
               String operation = radConfig.getOperation( "getAccountInfo" );
// prepare user’s security credentials
               SecAttribute[] attributeList = radConfig.getSecAttributes( SecAttribute.USER,
context.getCallerPrincipal().getName() );
// get reference to the RAD service
               RAD rad = RADLocator.getInstanceOfRAD();
// make an access decision
               if ( !rad.access_allowed( resourceName, operation, attributeList ) ) {
// do something if access has been denied
```

```
               }
// ...
// main functionality
       }
}
```

Listing 4 RAD based access control with pure Java

```
class BankEJB implements SessionBean
{
       public String getAccountInfo( String customer ) {
// main functionality
       }
}

privileged aspect BankAspect {
       pointcut bankMethodsInvocation( String customer ):
               execution( String getAccountInfo( String ) ) && args( customer );

       Object around( String customer, SessionBean obj ): bankMethodsInvocation( customer ) &&
               this( obj ) {
// access control
               RADConfig radConfig = RADConfig.getInstance();
               ResourceName resourceName = radConfig.getResourceName(
RADConfig.CUSTOMER_ACCOUNT, customer );
               String operation = radConfig.getOperation(
thisJoinPointStaticPart.getSignature().getName() );
               SecAttribute[] attributeList = radConfig.getSecAttributes( SecAttribute.USER,
((BankEJB)obj).context.getCallerPrincipal().getName(); );
               RAD rad = RADLocator.getInstanceOfRAD();
               if ( !rad.access_allowed( resourceName, operation, attributeList ) ) {
// do something if access has been denied
               } else
                       return proceed( customer, obj );
       }
}
```

Listing 5 RAD based access control with AspectJ

As in the case of authentication with AspectJ, the code providing access control using this technique is separated and independent from the rest of application. Any change to the security mechanism would require a modification only in the aspect code. If we implemented access control without aspects and had to change existing security mechanism, we would have to modify code of every component we had secured earlier.

Accountability and audit

Accountability and audit serve to collect and analyze the activity of the information system. They aim at detection of security violations and defining their causes. Although accountability and audit are very important parts of security, there is no standard mechanism to solve the issue. Usually accountability is reduced to logging to a text file and audit is omitted at all. The J2EE containers don't provide any standard functionality related to accountability and audit. The problem could be solved with programmatic security but this would require modification of component's sources. Accountability and audit belong to the category of concerns that can be easily implemented with aspects. The problem can be solved similarly to the access control issue. The following listing shows how accounting could be implemented with usage of aspects. The methods will be logged if they throw any exception of a given type.

```
aspect BankAspect {
       pointcut bankMethods():
               execution( public * bank.*( .. ) ) && this( SessionBean );

// Log information after throwing BankSecurityException from SessionBeans which belong to the
bank package
       after() throwing (BankSecurityException e): bankMethods() {
               Log log = Log.getInstance();
               log.write( e );
```

```
            }
}
```

Listing 6 Accountability with AspectJ

Analogously audit could be implemented but in that case the information should be directed to some audit module i.e. an intrusion detection module.

The brief comparison of a container-managed and aspect-oriented security is shown in the following table:

	Container managed security		Aspect-oriented security
Security mechanism	Declarative security	Programmatic security	
Identification and authentication	Limited to container functionality	Limited to container functionality	No modification of component's code required
Access control	Limited to container functionality	Required modification of component's code	No modification of component's code required
Accountability	No standard solution	Required modification of component's code	No modification of component's code required
Audit	Not supported	Required modification of component's code	No modification of component's code required

Table 1 Comparison of container managed and aspect-oriented security

Conclusions

Both container managed and aspect-oriented security have its pros and cons. Container managed security frees the application developer from implementing security mechanisms by himself and leaves definition of application's security properties to the security expert. Application‘s security properties are defined only in the application deployment descriptor. On the other hand we are limited to security functionality of the container. It is often not enough to enforce more complicated and dynamic security polices. We can additionally use programmatic security but it requires to mix security logic and business logic in each secured component. The final code is mangled and usually difficult to maintain. More flexible and extensible solution is to use AspectJ to implement security. This allows creating separated flexible module responsible for security and weaving it to the non-secured application. That approach doesn't require any modification of application's sources to introduce security. The best solution seems to be combination of container managed and aspect-oriented security. The both types of security can be perfectly complementary.

References

[1] J2EE page at Sun web site, http://java.sun.com/j2ee
[2] JBoss web site, http://www.jboss.org
[3] AspectJ web site, http://www.aspectj.org
[4] JAAS page at Sun web site, http://java.sun.com/products/jaas/
[5] *Resource Access Decision Facility Specification*, OMG, 2001. http://ww.omg.org/docs/formal/01-04-01.pdf
[6] *Security Functionality Requirements*, National Institute of Standards and Technology, 1992
[7] *Technical Security Standard for Information Technology (TSSIT)*, Government of Canada, 1997.
[8] *Information Technology Security Evaluation Criteria (ITSEC)*, Department of Trade and Industry, London, 1991.
[9] Scott Stark. *Integrate security infrastructures with JbossSX*, JavaWorld, April 2001. http://www.javaworld.com/javaworld/jw-08-2001/jw-0831-jaas.html