Using Dynamic Aspects for Delegating Fine-Grained Access Rights

Kung Chen
National Chengchi University, Taipei 116, Taiwan
chenk@cs.nccu.edu.tw

Abstract

This paper shows that a proper combination of instance-level aspects and dynamic deployment can be used to enhance an aspect-based access control system with dynamic and fine-grained delegation effectively in a highly modular manner. We developed a prototype implementation using the per instance interception mechanism of AspectWerkz. While workable, this mechanism still leaves much to be desired. We describe our implementation scheme and discuss the issues we encountered.

1. Introduction

Application-level access control is a typical instance of system-wide crosscutting concerns that aspect-oriented programming (AOP) aims to modularize well. Our previous work [1][2] has demonstrated the feasibility of using AOP to implement fine-grained access control for Struts-based Web applications [3]. Here we extend this line of study to include delegation of access rights, such as roles and permissions. Our goal is to investigate how we can enhance an aspect-based access control framework with proper delegation capability using AOP.

In access control systems, the delegation requirement arises when a user needs to act on another user’s behalf for accessing resources [4]. Usually this is a temporary arrangement for handling the situations such as vacations and ad-hoc information sharing among collaborating users. With delegation, the delegated user (delegatee) has the rights to carry out some functions that are originally authorized only to the delegating user (delegator). Clearly, to support such delegation requirements, all access control decisions need to take the relevant delegations into account. However, it is also conceivable that delegation requirements emerged only after the access control system was built. In such cases, it is highly desirable that we have some mechanisms to enhance the working access control system with proper support for delegation in a modular and effective way.

One way to fulfill the delegation requirements is to extend the user login module with a delegation request and authorization module that a user can choose to execute when he or she login to the system. Once activated, this module will enlarge the set of access rights of an authorized delegatee throughout that particular session [4]. However, this approach is not as flexible as we would expect. All users that intend to perform a delegated task need to end the current session and re-login to the system to acquire the delegated rights. Similarly, after finishing the delegated task, the delegatee must logout and re-login to restore his or her normal access rights. In contrast, we want a more flexible service that can support the delegation of access rights at anytime during a user session.

Apparently such a flexible delegation service is a crosscutting concern since all the code modules that require access control now need to deal with delegation. Therefore, we would like to investigate how to employ AOP to extend our aspect-based access control framework with such a dynamic delegation service while maintaining its modularity. A direct way to do this using AOP is to use inter-type declarations to extend relevant classes and define a new static aspect with a piece of around advice that blindly intercepts every access control decision and checks if a specific delegation setting should apply. This scheme may work, but it will incur significant extra runtime overhead that should have been spared. Indeed, requests from users that are not acting as delegates need not be interrupted and slowed down by this static delegation aspect. Given the dynamic nature of delegation, we set out to seek alternative implementation schemes based on dynamic aspects.

This paper reports our preliminary results and findings from this study. In short, we devise a simple yet effective scheme that use dynamic aspects to temporarily enlarge the access rights of a delegatee according to what the delegator has authorized. It thus enables user-to-user delegation of fine-grained access rights.
rights. Moreover, this scheme is a modular extension of our access control framework; no need to change it at all. As to implementation tools, the two popular dynamic AOP frameworks for Java, JBoss AOP [5] and AspectWerkz [6], now both provide a mechanism, called per instance interception, that can support our scheme to a large extent. We chose AspectWerkz for its closer similarity to AspectJ, in which our access control framework was built. However, in their current forms, both frameworks still lack some features that we need for completely realizing our scheme. Our results should also motivate these AO tool developers to further enhance their implementations.

2. System Overview

Our experimental platform is the popular MVC-based Apache Struts Framework for Web applications [3]. In a Struts-based application, every user request is dispatched to an action class by the controller according to the action mapping defined in the configuration file, *struts-config.xml*. These actions are responsible for serving user requests or passing them to the designated business components, and for returning the correct view element that the controller should forward to after finishing the user request. This view forwarding is also based on the mapping information specified in the configuration file. Figure 1 illustrates the architecture of Struts-based Web applications.

The action classes play the role of gateway between the presentation tier and the business and the data tiers. All user action classes must inherit from the class *Action* and implement an *execute* method with a specific signature. As discussed in [1], the *execute* method of a user action class is the right target for weaving access control aspects. While these aspects share a common code structure, their specific coding details will not be the same because the access constraints of each action class may be different. So any user action class whose access needs to be restricted will be associated with some proper aspects to enforce the required access control. More details about these aspects will be reviewed in Section 3.

Next, we outline our approach to extending this access control framework to include delegation of access rights. We divide the delegation process into two stages. First, a delegating user (delegator) authorizes which users he or she wishes to pass certain access rights. Afterwards, at any point of a user session, a delegated user (delegatee) can dynamically request for the additional access rights that he or she has been given and exercises them until he or she drops them or the session ends. Moreover, since delegation is a temporary arrangement in nature, all delegators can revoke the access rights that have been delegated to other users.

To support this delegation approach, we divide our delegation framework into two parts: delegation manager and delegate factory. Figure 2 depicts their main functions and relationships between them. The delegation manager keeps the permissible delegation records in a system and is operated by the security administrator. A delegation record is a tuple of three elements: \((u_1, u_2, \text{delegatedRights})\). Here \(u_1\) is the delegator, \(u_2\) is the delegatee, and \(\text{delegatedRights}\) is the access rights that \(u_1\) has granted to \(u_2\). The rights could be roles, permissions, or any other user credentials that are delegatable to other users. During user sessions, the delegate factory accepts users’ requests that ask for more access rights or drop the delegated rights. It will then consult the delegation manager for handling these requests. If the request is acceptable, then the delegate factory will generate a proper delegate aspect for realizing the delegation of access rights or remove the delegate aspect, depending on whether its a delegating or dropping request.
The idea of our aspect factory is as follows. In general, a user’s security-related attributes, such as identity, roles and permissions, constitute the most important elements for specifying an access control constraint. The access control enforcement mechanism must obtain those attributes from the requesting user during the authorization process. If we can temporarily extend the security attributes of the delegatee to include those of the delegator, then the access control mechanism will treat the delegatee just like the delegator, thus achieving the desired delegation. To enable such temporary enhancement, we shall to attach a delegate aspect to the delegated user object that will advise all its security attribute inquiries according to the access rights authorized by the delegator. As a result, the access control codes will get the additional rights when enquiring the delegatee’s security attributes and thus endow the delegatee with the same privileges of the delegator. When the delegation is no longer needed, we can restore the normal access control by simply removing the aspect from the delegatee. In this way, we are able to enhance the access control framework with user-to-user delegation in a modular manner. Figure 3 illustrates our idea of dynamic delegate aspects.

**Figure 3: Delegate aspect overview**

To realize this scheme, we need an aspect-oriented tool that supports both instance-level aspects and dynamic deployment and undeployment of aspects. Instance-level aspects provides first-class aspect instances and instance-level advice binding under program control so that two instances of the same class can be advised by two separate aspect instances [7]. The reason we need this is twofold. First, only users acting as delegates need to be advised by the delegate aspect, so it does not make sense to associate one aspect instance for every user object, i.e., the per-object aspects are overkill. Second, as two delegated users may get different access rights, they should be advised by different aspect instances. Therefore, these delegate aspects must be instance-level aspects to provide customized advice for delegated users. Besides, since the delegation is usually temporary, we should attach a delegate aspect to a delegatee only when a delegation session begins and remove it at the end. Later we shall show that the per instance interception mechanism of AspectWerkz can satisfy our requirements of dynamic aspects to a large extent.

### 3. Static Aspects for Access Control

Since our delegation scheme is based on our previous work of access control [1][2], before presenting the dynamic aspects for delegation, we shall briefly review our approach to implementing access control using static aspects. Besides, we shall also briefly describe the relevant features of AspectWerkz when we present the aspect codes.

The simplest type of access control is indeed user authentication, where no further constraint is required. There the concern is accommodating multiple kinds of authentication scheme. Since authentication has less to do with delegation, we will omit them in the following discussion and assume that user information, including roles and permissions, will be collected into a User Account object under the user session data after the authentication is performed successfully.

Basically, every system function implemented by a user action that requires protection will be advised by an access control aspect anchored at its execute method. To support common access control requirements as well as those related to data contents, we have provided two kinds of access control aspects, namely Pre-checking and Post-filtering. The pre-checking aspects handle the cases where the information needed to evaluate the access constraint is available before executing the protected function, while the post-filtering aspects the cases where their constraint refers to data contents that will only be available after the function being executed. For example, a record deletion rule requiring that only managers with the additional “delete” permission can delete working records belongs to the first category. In contrast, a record owner rule requiring that, except managers, general staff can list only working records of their own belongs to the second one, for we must filter out unauthorized data records after obtaining the query results. Moreover, some constraints may entail the installation of both kinds of aspects on the underlying function.

Listing 1 presents the code skeleton for the Precheck aspect and the implementation of the record deletion rule as a subsaspect of Precheck in AspectWerkz.

AspectWerkz has basically the same model for AOP as AspectJ with join points, pointcuts, advice, introduction and aspects being the essential elements.
But it does not extend Java with a new aspect language. Instead, it uses plain Java classes as aspects and relies on a set of class libraries, XML or javadoc tags, and a customized class loader to realize aspect-oriented programming. Since aspects are classes, naturally advice becomes ordinary methods in a class, except that the signature of advice must follow a certain pattern. There are also three kinds of advice, namely, before, after and around. The around advice takes one more argument of the JoinPoint class, which provides the proceed method for resuming the execution of the advised method. The check method of the Precheck aspect in Listing 1 is an example of around advice.

### Listing 1: The Precheck aspect and the DeleteRecord aspect

```java
public abstract class Precheck {
    abstract protected boolean constraint(Request request);
    private Forward forwardToErrorPage(…)
        { // redirect to error handler }
    public Object check(JoinPoint jp, …)
        throws Throwable    { //around advice
        if (constraint(request))
            return jp.proceed(); // granted
        else return forwardToErrorPage(…);
    }
}
pub class DeleteRecord extends Precheck {
    protected Boolean constraint(Request request) {
        UserAccount user=(UserAccount)
            request.getSession().getUserAccount();
        Collection roles = user.getRoles();
        Collection permissions =
            user.getPermissions();
        return (roles.contains("manager") &&
            permissions.contains("delete");
    }
}</aspectwerkz> // aop.xml descriptor file
...<aspect class="* DeleteRecord">
    <pointcut name="DeleteRecordFun" expression="*DeleteRecordsAction.execute(…)"
        />
    <advice name="check(JoinPoint joinPoint, …)"
        type="around"
        bind-to="DeleteRecordFun AND args(…)"
       />
</aspect> ... </aspectwerkz>
```

The definition of Precheck aspect follows the template advice pattern [8], where the specific access constraint is left to concrete aspects that inherit it. For example, the DeleteRecord aspect is a subsaspect of Precheck that enforces the record deletion rule.

Another major difference between AspectJ and AspectWerkz is that pointcut designators in AspectWerkz are decoupled from the aspect definition and put into an XML descriptor file, `aop.xml`. The XML descriptor in Listing 1 contains a pointcut definition called `DeleteRecordFun` and an aspect mapping. The aspect mapping declares that the DeleteRecord class will be treated as an aspect in the system, while the advice binding connects the check method as around advice to the previously defined pointcut. Like AspectJ, all aspects are by default static and have a single instance per VM wide.

Listing 2 presents the code skeleton for the Postfilter aspect and the implementation of the record owner rule as a subsaspect of Postfilter.

### Listing 2: The Postfilter aspect and the ListRecords aspect

```java
public abstract class Postfilter {
    abstract protected Boolean constraint(Request request, Object data);
    abstract protected Collection getRS(Request request);
    public Object filter(JoinPoint jp, …)
        throws Throwable    { //around advice
        Object forward=(Forward)
            jp.proceed();
        Collection col = getRS(request);
        Iterator i = col.iterator();
        While (i.hasNext()){
            Object record = i.next();
            if (!constraint(request, record))
                i.remove(); //unauthorized records
        }
        return forward;
    }
}
pub class ListRecords extends Postfilter {
    protected boolean constraint(Request request, Object record) {
        UserAccount user = (UserAccount)
            request.getSession().getUserAccount();
        Collection roles = user.getRoles();
        return (roles.contains("manager") ||
            user.getName().equals(record.getOwner());
    }
    protected Collection getRS(Request request) {
        …
    }
}</aspectwerkz>
```

The Postfilter aspect is designed to handle access control constraints that involve the contents of the data to be accessed. Once installed, it will intercept the query results before they are returned to user and it will apply the filtering constraint to each individual data record in the query results. Consequently, unauthorized data records will be filtered out. The definition of specific filter constraints is left to concrete aspects that inherit it. For example, the
ListRecords aspect in Listing 2 is such a subaspect used to enforce the record owner rule.

4. Dynamic Aspects for Delegating Access Rights

This section illustrates our approach to supporting fine-grained delegation through dynamic aspects. The first part of this section describes how we represent delegatable rights in a generic manner and the second part presents how we derive from this representation scheme the proper delegate aspects for enlisting the access rights of a delegated user at runtime.

4.1. Fine-Grained Delegatable Rights

An important issue to consider while designing a delegation framework is the granularity of delegatable access rights. For example, in role-based systems [9], while roles are assigned to users, it is often the case that a user must delegate to another user only part of the permissions in a role that he or she qualifies for, as demonstrated by the work of permission-based delegation [10]. Furthermore, for systems that require instance-level access control constraints, such as the record owner rule, we have to provide a more refined type of delegatable access rights.

For example, according to the record owner rule, except managers, a general staff can view their own working records. What if all managers were unavailable on some working day while the auditor must examine the working records of a particular manager? It seems that we can prescribe a delegation rule which delegates the manager role to a designated staff so that he or she can act as the manager and list the requested records. But this rule is indeed more than necessary and can lead to security breaches, for all we want to delegate is the right to examine the working records of a particular manager, not all the rights of a manager. On the other hand, if we could have more types of delegatable right, such as user names, we will be able to gain more precise control over such fine-grained delegation requirements. Therefore, to accommodate various kinds of delegation requirements and access control constraints, we propose to embody all delegatable rights in a generic class, DelegatedRights, as shown by Listing 3.

Listing 3: Generic structure of the DelegatedRights class

```java
package tw.edu.nccu.cs.delegationManager;
class DelegatedRights {
    private Collection roles;
    private Collection permissions;
    private String name;
    // ... other security-related attributes
    //constructor
    DelegatedRights(Collection rs, Collection ps, String n) {
        roles = rs;  permissions = ps;
        name = n;
    }
    Collection getRoles() { return roles; }
    Collection getRoles() { return roles; }
    String getName() { return name; }
    // ...
}
```

4.2. Delegate Factory

The delegate factory is the sole interface between the underlying application and the delegation manager. It accepts delegating requests and dropping requests from the application through the following service functions.

```java
Collection AvailableDelegators (UserAcct delegatee);
boolean delegatingReq(UserAcct delegatee, String delegator);
boolean droppingReq(UserAcct delegatee);
```

The availableDelegators(..) function returns a list of permissible delegators for a particular user, as
determined by the delegation manager. The delegatingReq(...) will attempt to realize the delegation for the requester, the delegatee. If the request is granted by the delegation manager, this function will construct the delegate aspects according to the right object it received from the manager. At any point, a user can act as a delegate for only one delegator. After finishing the delegated tasks, the delegatee can relieve the mission by calling the droppingReq(User delegatee) function.

At the core of the delegate factory is a systematic procedure that transforms the delegated rights to dynamic aspects for implementing the requested delegation. This procedure examines the contents of the right object returned from the delegation manager. For each non-null attribute in the right object, this procedure will create a delegate aspect instance and attach it to the delegatee user object. This special delegate aspect will enhance or replace the corresponding security attribute of the delegatee with that of the right object. For example, if a right object returned by the delegation manager has the value of <roles={manager}, permissions={delete}, null>, then one instance of the two aspects defined in Listing 4 will be created with proper initial values respectively and attached to the delegatee at specified pointcuts.

Listing 4: Delegate aspects

class RoleDelegate implements AroundAdvice {
    //interceptor
    private Collection roles;
    RoleDelegate(Collection rs) {
        roles = rs;
    }
    Collection getRoles() { return roles; }
    public Object invoke(JoinPoint jp) {
        //advice body
        return //enlarge the roles collection
            jp.proceed().addAll(getRoles());
    }
}
class PermissionDelegate implements AroundAdvice {
    //interceptor
    private Collection permissions;
    PermissionDelegate(Collection ps) {
        permissions = ps;
    }
    Collection getPermissions() {
        return permissions;
    }
    public Object invoke(JoinPoint jp) {
        //advice body
        return //enlarge the permission set
            jp.proceed().addAll(getPermissions());
    }
}

In AspectWerkz, the aspects above are called interceptors, which are single-advice aspects that can be dynamically instantiated and deployed to a selected instance of classes that have been declared as Advisable in the XML descriptor file. The two delegate classes qualify as such interceptors because they both implement the AroundAdvice interface, which is one of the designated interceptor interfaces. The steps to deploy an interceptor instance are quite simple: create an interceptor instance and attach it to a given object instance using the aw_addAdvice method, as illustrated below for a RoleDelegate interceptor:

```java
RoleDelegate rp =
    new RoleDelegate(rights.getRoles());
((Advisable)delegatee).aw_addAdvice("execution(* UserAccount.getRoles())", rp);
```

Here the target object is the delegated user object. Note that the first argument passed to aw_addAdvice is the pointcut where the second argument, the interceptor, should advise.

Now with these interceptors attached to the delegated user object, the access control aspects, when inquiring for the delegatee’s security attributes inside the constraint(...) method, will get the values enlarged by the delegator’s access rights and thus endow the delegatee more access rights, as shown by the invoke(.) method of both classes in Listing 4.

The delegate factory is also responsible for removing these delegate aspects upon request. To support this task, the factory maintains a hash table of all effective delegation sessions that maps a delegated user object to the set of delegate interceptors and their pointcuts associated with it. When the delegation session calls for an end via the dropping request, the factory will invoke the aw_removeAdvice to undeploy all the delegation interceptors one by one for the designated user. For example, the following is the code the factory used to remove the RoleDelegate interceptor from the delegatee.

```java
((Advisable)delegatee).aw_removeAdvice("execution(* UserAccount.getRoles())",
            RoleDelegate.class);
```

5. Discussion and Future Work

Instance-level aspects and dynamic deployment are both new advancements to the AOP community. We have illustrated that a proper combination of these two features can be used to enhance an aspect-based access control system with fine-grained delegation in a modular manner. However, the per instance interception mechanism we adopted from AspectWerkz (Version 2.0) is an ad-hoc combination of the two features that is barely able to meet our requirements. We now discuss the specific issues of AspectWerkz we encountered in our study.

The same issues also apply to JBoss AOP 1.3.
First, per instance interception supports only aspects with a single piece of advice. This is why we cannot use a single delegate aspect with multiple pieces of advice to implement the delegation of access rights. It is true that there are some design tradeoffs that need to be made if we use multiple-advice delegate aspects. For example, there will be some extra runtime overhead due to the unnecessary activation of delegate advice. Nevertheless, as a general AO tool, AspectWerkz should make their instance-level aspects as general as possible so they can support the widest range of applications possible.

Second, per instance interceptors do not support context-sensitive pointcuts such as \texttt{within()} and \texttt{cflow()}. Yet it is very likely that we need to use such pointcut designators to exercise finer control over the activation of the associated advice. For example, to maintain user accountability, we must separate audit identity from access identity. Hence, in treating username as a delegable right, we would like to restrict the activation of the username interceptor to access control codes by appending a \texttt{cflow} designator to the pointcut of the interceptor as follows:

\begin{verbatim}
"execution(* UserAccount.getUserName()) 
&\& cflow(call(* *.constraint(..)))"
\end{verbatim}

However, this cannot be done in AspectWerkz. So, once the interceptor is deployed, any code in the application that asks for the delegates’ username would get the username of the delegator. In particular, the logging aspect will also get confused and mix access identity with audit identity, which will make accountability hard to achieve in presence of delegation.

To overcome this deficiency of AspectWerkz, we have devised a workaround for our delegation framework that can achieve the effects of \texttt{cflow} pointcuts by using the inter-type declarations mechanism of AspectWerkz (called mixins). We sketch it as follows. We add an extra tag field to the \texttt{UserAccount} class. This tag will be set to true upon entering an access control aspect and will be unset when exiting the aspect. Based on this flag, the delegate interceptors are modified accordingly so that they will only change the results of security attribute inquiries for those users whose tag field is true. As a result of these changes, we are able to restrict the scope of those delegation interceptors to only access control modules. However, this workaround scheme needs to change the underlying access control aspects.

To conduct some performance benchmarking on our implementation scheme and compare it with the one using static aspects, which is sketched in the first section. Admittedly, dynamic aspects will incur a certain amount of overhead due to runtime weaving. However, as we argued earlier, the static implementation scheme also has some obvious performance penalties caused by unneeded interception. Therefore, it would be interesting to see how they compare with each other under different scenarios.

6. References


