A Two-Stage Approach to Modular Access Control For Java-Based Web Applications

Kung Chen
Dept. of Computer Science, National Chengchi University, Taipei, Taiwan
chenk@cs.nccu.edu.tw

Abstract*-Web applications are usually structured into three logical tiers: presentation, business logic, and data processing. In most of current access control frameworks for Web applications, the control is enforced at business logic or data processing level. In contrast, this paper presents a two-stage approach where the enforcement of access control is divided between presentation level and business-logic level. A flexible menu generator is used to achieve presentation-level access control by restricting user menus to functions that a user's current access-privileges permit. Other fine-grained access controls are enforced at the business-logic level using a modular scheme based on the aspect-oriented language AspectJ.*

Keywords: access control, aspect-oriented programming, AspectJ, Java, Web application.

1. Introduction

Like any applications, Web applications need access to various system resources, such as files or databases. To maintain the security of sensitive data accessed through Web applications, it is extremely important to enforce a certain degree of access control at the application-level beyond the normal protection provided by system software such as database systems. However, it is not easy to derive a robust access control implementation for Web applications. Indeed, "broken access control" is listed as the second critical Web application security vulnerability on the OWASP's top ten list [10].

The principle difficulty in designing security concern such as access control into an application system is that it is a concern that permeates through all the different modules of a system. As a result, security concerns in an application are often implemented with scattered and tangled code, which is not only error-prone but also makes it difficult to verify its correctness and perform the needed maintenance.

A better way to address this problem is to treat security as a separate concern and devise a framework where the access control logic is encapsulated and separated from the core of application [16]. This will not only improve the application's modularity but also make the task of enforcing comprehensive access control more tractable. The Java Authentication and Authorization Services (JAAS) of J2EE [13] is a well-known attempt toward such a solution. Furthermore, it takes one step forward to support declarative security where access control can be specified declaratively in a configuration file without actual coding.

While convenient and flexible for Web application development, there are also some shortcomings using solutions such as JAAS. In Web application development, it is a well-accepted practice to divide an application into three logical tiers: presentation, business logic, and data processing. JAAS-like frameworks for Web applications conduct the access control along with the invocation of an application function, hence belonging to the area of business logic. The following list the shortcomings of such approaches. First, users will see functions which they are not allowed to execute. Second, any attempted access to unauthorized functions, whether intentionally or unintentionally, will lead to business-logic level security check and incur certain amount of runtime overhead. Third, users may feel annoyed or confused with the access violation messages from time to time.

In many situations, it is possible to determine whether a particular function should be authorized to a user without actually having to try to perform it. For those operations, users should not be able to see them on the function menu in the first place. The business-logic level check should be applied to more fine-grained access control requirements only. Therefore, we argue that a two-stage approach to access control is a better way to structure the access control mechanisms. The first stage is conducted on the presentation level, and the second on the business-logic level. By dividing the enforcement of access control into two stages, we can overcome the shortcomings described above while retaining the security derived from conventional access control.

This paper presents a two-stage approach to access control for Java-based Web applications. We begin with access control requirement modeling using a flexible scheme. These requirements are transformed into access control rules for the presentation-level and the business-logic level according to their granularity. Fine-grained rules are those whose decisions involve data contents or

arguments passed. They are assigned to the business-logic level. As access control is a system-wide concern that cut across functional modules, we choose to use the emerging techniques of aspect-oriented programming (AOP) [8] as our design and implementation mechanism. We devise a modular scheme for enforcing these rules through AspectJ [9], a Java-based aspect-oriented language. All access control code is encapsulated and linked to functional modules in a low coupling way.

Rules depending mainly on user information are enforced at presentation-level using a flexible menu generator. All user functionalities provided by an application can be selectively removed from user menus, either on an individual basis or as a group, according to the rules specified in a configuration file associated with the application. If the rules are changed, the user interface automatically adapts to provide the correct functionality to each user, eliminating the need to recode the user interface.

The rest of the paper is organized as follows. Section 2 outlines our approach. Section 3 and 4 present the mechanisms we use to enforce presentation-level and business-logic level access control, respectively. Section 5 describes related work. Section 6 concludes and sketches future work.

2. Overview of Our Approach

The section outlines the main ideas behind our approach. First, it describes how we model access control requirements into rules of two different levels. Second, it illustrates how we enforce these rules in Web applications.

2.1. Access control modeling

For access control purpose, we model the interaction between a user and a Web application as a sequence of access tuples of three elements: <*user, function, data*>, indicating a user's request to execute the function on a specific data object. The access control rules of an application determine which access tuples are allowed and which must be denied. They are derived from user access control requirements.

The elements in an access tuple will be modeled as three objects, *User*, *Fun*, *Data*, with various attributes that access control rules can refer to. Typical attributes for the User object include user's name, title, and roles in the organization. The attributes of the Function object include the function's full name and the arguments passed to it, while the fields of a data object being requested are the standard attributes of the Data object. Yet the specific set of attributes depends on individual application's needs. For instance, roles are usually the major attribute for the User object, as role-based access control (RBAC) [12] is the most often cited guiding principle underlying all the approaches to modeling application-level security.

To accommodate a wider range of access control requirements, we also include an application object (*App*) and a context object (*Cxt*) for specifying the constraints. The application object is instantiated from the static properties stored in a dedicated Java property file. These properties serve as specific parameters to an application for access control purpose. For example, certain functions are accessible only during working days and from specific machines. We can supply the definitions of working days and selected machines through two name-value pairs in the property file as follows.

```
WorkingDays=Mon, Tue, Wed, Thu, Fri
DedicatedMachines = 10.1.1.2, 10.1.2.2
```

The context object provides method to retrieve the time and location of access. This is the most often used contextual information for access control. Hence following the spirit of RBAC, our access control rules are expressed as follows:

<*userRole, methodName, className, Constraint*>

The *userRole* stands for the role authorized to this access tuple and the *methodName* denotes the Java method to be constrained. The *className* is the type of the data object to be protected; usually it is the same with the class of the constrained method. The *constraint* is a Boolean expression over the attributes of the three objects listed above, together with those of the application and context objects.

Example: the following is a set of access control requirements and corresponding rules for an online order management system.

C1: Only sales managers working at headquarters can delete order objects[1].

```
<"Sales" delete, Order,
   contains(User.getRoles(), "Manager")
   && equals(User.getOfficeLocation(), "HQ")>
```

C2: Orders can be printed in batch mode by sales from dedicated machines during office hours.

```
<"Sales", batchPrint, Order,
  contains(App.getOfficeHours(), Cxt.getHour())
  &&  contains(App.getDedicatedMachines(),
                    User.getclientIP())>
```

C3: Customers can list (view) their own orders.

```
<"Customer", ListOrders, Order,
   equals(User.getName(), Data.getOwner())>
```

C4: Only VIP customers can create orders whose total amount exceed $10,000.

```
<"Customer", create, Order,
```

[1] Note that a user can have multiple roles; here it requires a user have both Sales and Manager in his role set to be authorized.

User.getVIP() &&
lessEq(Fun.getArgument("total"), 10000)>

This form of access control rules is very flexible and can model a multitude of security requirements, from simple RBAC to sophisticated instance level constraints [6]. Furthermore, in our model, it is clear that rules with constraints that refer to the argument attribute of the Fun object or any attributes of the Data object must be assigned to business-logic level and get enforced during execution, while rules involve only attributes of the User object can be enforced at the presentation level.

2.2. Web Application Architecture and Access Control

We follow the popular three-tier architectural principle and divide a Web application into three logical tiers: presentation, business logic, and data processing. Furthermore, we structure the presentation tier according to the well-known Model-View-Controller (MVC) design pattern [4], and in particular follows the popular Struts framework [1]. The model components, called *actions* in Struts, encapsulate the application components in the business logic and data tiers, and the view components are those pieces of an application that display the information provided by model components and accept input. These view components are built using page-based scripting tools, JSP [14]. The controller is a special Java Servlet [15] that dispatches user requests and coordinates all the activities of the model and view components. It is the central point of control within an application. The advantages to a single controller include ease of access control, and consistent interface and flow between the tiers of our applications.

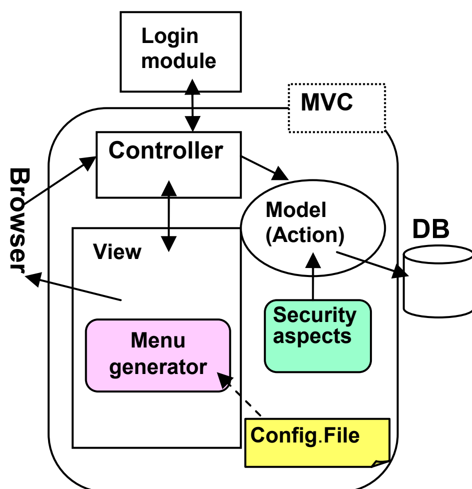

Figure 1: Web application architecture

Our presentation-level access control is achieved through a flexible menu generator. It is invoked by the controller, after verifying the user's identity, to generate a tailored function menu according to the access control rules specified in an XML configuration file. This is the first stage of control and the second stage is conducted at business-logic level by special security modules, called *aspects* in AspectJ. These aspects intercept the method calls in action classes and then enforce the access control demanded by rules derived from prior modeling stage. Figure 1 depicts the general architecture for Web applications using our approach.

3. Presentation-Level Access Control

This section shows how we enforce access control at the presentation level by a flexible menu generator. Due to space limitation, the reader are referred to [3] for a full description of the menu generator.

3.1. Overview

The menu generator consists of three major components: menu rendering, menu service, and rule engine, as illustrated in Figure 2.

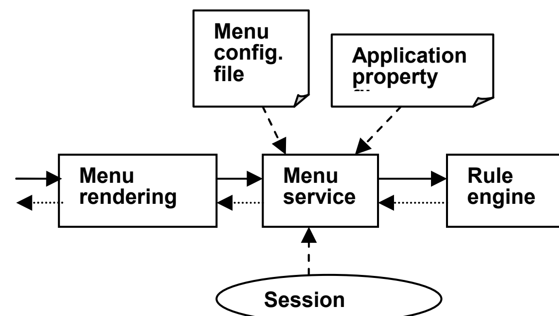

Figure 2: Menu generator structure

The menu rendering component is the driver of the generator. It is invoked by the controller servlet after the login module. Its major task is to render the XML function menu it gets from the menu service component into HTML and send it back to client browser. The menu service component is responsible for processing the menu configuration file and application property file to generate the menu contents. The specific menu items available to a user are determined by the access control rules stated in the configuration file. These rules may refer to user and application specific attributes as well as context information, but not any data-specific attributes. The menu service component extracts the rules and related information from the configuration file and then passes them to the rule engine for evaluation. The results dictate what to be included in a menu for a user. The rule engine is a JavaScript interpreter

enhanced with some utility functions and classes for implementing access control constraints.

3.2. Function grouping and menu configuration

In our design, all the functions of an application system are organized into a hierarchy and represented by a *tree-based* menu to a user. One access control rule can be associated with an individual function or a group of functions organized under the same ancestor node. The constraints in a rule determine the accessibility of the associated function or functions; only functions whose rule constraints are satisfied will be displayed in a user's menu. The tree structure of an application system's functions and the associated access control rules are both specified in an XML-based menu configuration file. Table 1 shows the major parts of the DTD for the menu configuration file.

Table 1: Major parts of the menu.dtd

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT MenuTree
    (GlobalDeclaration? ApplicationSystem) >
<!ELEMENT ApplicationSystem (Application*) >
<!ELEMENT Application
    (Display, (FunctionGroup | Function)*, Rules) >
<!ELEMENT FunctionGroup
    (Display, (FunctionGroup | Function)*) >
<!ELEMENT Function (Display) >
<!ELEMENT Rules (Rule*) >
<!ELEMENT Display (DisplayText*)>
<!ELEMENT GlobalDeclaration (#PCDATA) >
<!ELEMENT DisplayText (#PCDATA) >
<!ELEMENT Rule (#PCDATA) >
<!ATTLIST Rule path CDATA #REQUIRED>
…
```

The MenuTree is the *root* element of the menu configuration. Under it, there is one mandatory ApplicationSystem element and one optional GlobalDeclaration element. If present, the GlobalDeclaration element defines global objects that can be referenced in all rules in this configuration. For example, the application property object (app) should be declared here if needed. Our menu configuration models a four level hierarchy comprising ApplicationSystem, Application, Function Group, and Function. There must be only one ApplicationSystem element, though we can define multiple Application elements under ApplicationSystem. In turn, we can define multiple FunctionGroups and/or Functions under each Application Element. Functions are the leaf nodes in the menu tree, yet FunctionGroups can be nested to support a deep hierarchy of functions.

In addition, Application Element has a child element *Rules* that is comprised of multiple *Rule* elements. The *path* attribute of the Rule element refers to the hierarchical path of element(s) that the rule tries to govern. The accessibility of an element (Function or FunctionGroup) is defined in a *cascading* manner:

- If the accessibility of the element is explicitly defined in the configuration, use the defined accessibility;
- Otherwise uses the accessibility of the nearest ancestor that is explicitly set.
- If no ancestor has accessibility explicitly defined, make the element always accessible.

This is a very flexible approach for grouping related functions for access control purpose.

3.3. Access control rules and constraint evaluation

An access control rule element in the menu configuration file contains two parts: a function path and a constraint. The function path refers to a single function or a group of functions organized as a tree. The constraint is derived from the rules of the access control modeling described in Section 2, and transformed into a JavaScript expression which evaluates to true or false. False constraints imply inaccessibility of the specified function or functions. For instance, constraints C1 and C2 are transformed to the following form.

```
<rule path="/OrderManagement/delete">
    contains(User.getProperty("roles"), "Sales") &&
  contains(User.getProperty("roles"), "Manager") &&
   equal(User.getProperty("officeLocation"), "HQ") ;
</rule>

<rule path="/OrderManagement/batchPrint">
     contains(User.getProperty("roles"), "Sales") &&
     contains(App.getProperty("OfficeHours"),
              Cxt.getHour())                    &&
     contains(App.getProperty("dedicatedMachines"),
              User.getProperty("clientIP");
</rule>
```

To evaluate the constraints, we developed a rule engine out of an open source JavaScript interpreter, Rhino [11]. The menu service component will pass the constraint expression of a rule to the rule engine

for evaluation. There are a few utility functions and classes implemented as a library of the rule engine to assist the specification of access control constraints. The functions, “contains”, “equals” shown above are such utility functions. We have also provided a *Cxt* object to supply date-time functions.

4. Business-Logic Level Access Control

After reviewing the concept of AOP and basics of AspectJ, this section illustrates our scheme for enforcing business-logic level access control through encapsulated code in AspectJ.

4.1. AOP and AspectJ

AOP addresses the issues of implementing a crosscutting concern through a new kind of modules, called *aspect,* and new ways of module composition. In AOP, a program consists of many functional modules, e.g. classes in OOP, and some aspects that captures concerns that cross-cuts the functional modules, e.g. security. The complete program is derived by some novel ways of composing functional modules and aspects. This is called *weaving* in AOP. Weaving results in a program where the functional modules impacted by the concern represented by the aspect are modified accordingly. Figure 3 illustrates the weaving process for AspectJ [9][2].

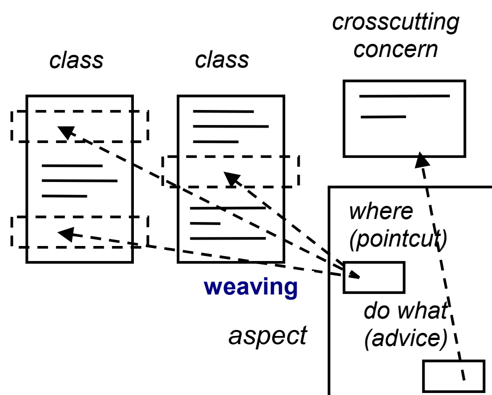

Figure 3: Aspect weaving in AspectJ

To facilitate the weaving process, a set of program *join points* are introduced to specify where an aspect may cross-cut the other functional modules in an application. Typical join points in AspectJ are method execution and field access. A set of join points related by a specific concern are collected into a *pointcut.* Code units called *advices* in an aspect are tagged with a pointcut and determine how the application should behave in those crosscutting points. There are three kinds of advices in AspectJ: *before*, *after*, and *around*. The before advice and the after advice are executed before and after the intercepted method, respectively. The case for the around advice is more subtle. Inside the around advice, we can choose to resume the intercepted method or skip it. A single piece of advice can be woven into multiple modules of an application through a pointcut and thus implement a crosscutting concern.

4.2. Access control using AspectJ

From the description above, it is clear that AOP lays a very good foundation for implementing highly adaptable yet fine-grained access control. The basic idea is as follows. Given the rules of the form: <*userRole, methodName, className, constraint*>, transform the constraints into Java code using the advices of security aspects, and choose proper pointcuts corresponding to the program points around executing the method specified in the rule. Since we are dealing with fine-grained access control requirements depending on function arguments or data contents, we will use *around advices* to enforce the constraints. For example, the constraints C3 and C4 in Section 2 can be enforced by the following aspect with two pointcuts and two around advices.

```
aspect OrderManagement
  pointcut ListOrders():
     execution(public List Order.ListOrders());
  pointcut VIPOrder(double total):
     execution(public
               boolean Order.createOrder(double total))
     && args(total);
  List around(): ListOrders() { // only one’s own order
    Set roles = sessionStore.getUserRoles();
    if (!roles.contains(“Customer”))…//throws exception;
    String uname = sessionStore.getUsername();
    List orderList = proceed(); // get the orders
    Iterator i = orderList.iterator();
     while (i.hasNext()) { // order filtering
        Order o = (Order) i.next();
        if ( !(uname.equals(o.getOwner()) ) i.remove();
     }
     return orderList;
  }
  boolean around(total): VIPOrder(total) {
   … // role checking similar to the code above
    if (! roles.contain(“VIP”) && total > 10000)
        return false; // not VIP, creation fails
```

[2] Since version 1.1, AspectJ also supports byte-code weaving.

```
        else return proceed(); // resume execution
    }
}
```

Note that "*args(total)*" is also an AspectJ pointcut that captures the argument(s) passed to the intercepted method, and the call to "*proceed()*" in an around advice resumes the intercepted method.

5. Related Work

Most access control frameworks for distributed client/server systems focus on enforcing the control on server-side components [2]. As far as we know, only Goodwin et al. [6] also proposed two levels of access control: command-level and resource-level. However, both are conducted at the business-logic tier; and it does not use aspect-oriented programming. Applying AOP to security concerns is pioneered by [16][17]. Georg et al.[5] focuses on the use of aspects for modeling and weaving in security concerns.

6. Conclusion and Future Work

In this paper, we have argued the advantages of developing a two-stage approach access control framework for Web applications and described our work toward this goal. We presented a flexible user menu generator for presentation-level access control and illustrated how AspectJ can enforce business-logic level access control in a modular manner. By dividing the enforcement of access control into two stages, we can overcome the shortcomings of single-stage approaches while retaining the same level of security very little extra efforts.

We have built a prototype system using this approach, and the preliminary findings show that it is a feasible one. However, it also inspires us to further explore the reuse mechanisms of AspectJ to improve the organization of the security aspects. Specifically, AspectJ allows aspect inheritance, abstract aspect, and abstract pointcut. We can write an aspect without any reference to a join point by declaring the pointcut abstract. A sub-aspect then extends the abstract aspect and defines the concrete pointcut. We will use this ability of AspectJ to build an aspect framework [16] for better structuring the access control aspects. The goal is to capture the generic part of an access control aspect using abstract aspects, and leave the rule-specific part, such as constraints and pointcuts, to concrete aspects extended from the framework. This will not only reduce the programmatic efforts of writing security aspects, it will also greatly improve the maintainability and extensibility of our approach.

References

[1] The Apache Struts Web Application Framework: http://struts.apache.org/

[2] K. Beznosov, and Y. Deng., "Engineering Application-level Access Control in Distributed Systems," in *Handbook of Software Engineering and Knowledge Engineering*, vol. 1, S. K. Chang, ed.: World Scientific Publishing, 2002.

[3] K. Chen and C.S. Chang, "A Flexible Presentation Level Function Access Control Framework Web Application", submitted for publication, Oct. 2004.

[4] Gamma, Helm, Johnson and Vlissides: *Design Patterns.* A. W. L., 1995. ISBN 0-201-63361-2.

[5] G. Georg, I. Ray, and R. France, "Using Aspects to Design a Secure System," *Proc. of the 8th IEEE International Conference on Engineering of Complex Computer Systems*. December 2002.

[6] R. Goodwin, S.F. Goh, and F.Y. Wu, "Instance-level access control for business-to-business electronic commerce," *IBM System Journal*, vol. 41, no. 2, pp. 303-17, 2002.

[7] JavaScript : http://www.mozilla.org/js/

[8] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in ECOOP '97, LNCS 1241, pp. 220-242.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold, "Getting Started with AspectJ", *Communications of ACM,* vol. 44, no. 10, pp 59-65, Oct. 2001.

[10] Open Web Application Security Project: *The Top Ten Most Critical Web Application Security Vulnerabilities.* http://www.owasp.org/documentation/topten

[11]. Rhino: JavaScript for Java, http://www.mozilla.org/rhino/

[12] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. "Role-based access control models," *IEEE Computer*, 29(2):38–47, 1996.

[13] Sun Microsystems, Java Authentication and Authorization Service (JAAS), http;//java.sun.com/products/jaas/index.jsp

[14] Sun Microsystem, JavaServer Pages Technology (JSP): http://java.sun.com/products/jsp/

[15] Sun Microsystem, Java Servlet Technology : http://java.sun.com/products/servlet/

[16] B. Vanhaute, B. De Win, B. De Decker, "Building Frameworks in AspectJ", ECOOP 2001, *Workshop on Advanced Separation of Concerns*, pp.1-6.

[17] B. De Win, B. Vanhaute, and B. De Decker, "Security Through Aspect-Oriented Programming," *Advances in Network and Distributed Systems Security*, Kluwer Academic, pp. 125-138, 2001.

[18] B. De Win, B. Vanhaute and B. De Decker, "How aspect-oriented programming can help to build secure software," *Informatica* 26(2), pp. 141-149, 2002.