Role-Based-Access-Control: A Novel Approach

Yanjie Zhou[1,a], Min Wen[2,a]

[1]College of Mathematical and Computer Science Jiangxi Science & Technology Normal University

Nanchang, China 330031

[2] Department of Civil and Architectural Engineering Nanchang Institute of Technology

Nanchang, China 330099

[a] zhouyanjie1111@126.com

Keywords: RBAC; Access Control; Internet Security; Novel Framework and Application

Abstract. We present a novel static approach to Role-Based Access Control policy enforcement. The static approach we advocate includes a novel design methodology, for applications involving RBAC, which integrates the security requirements into the system's architecture. We apply this novel methodology to policies restricting calls to methods in Java applications. We present a language to express RBAC policies on calls to methods in Java, a set of design patterns which Java programs must adhere to for the policy to be enforced statically, and a more detailed description of the checks made by our static verifier for static enforcement.

Introduction

The objectives of an access control system are often described in terms of protecting system resources against inappropriate or undesired user access. When there is a request for a resource, the system must check who triggered the request (authentication), check if that user has the permission for the request to be fulfilled (authorization) and as a result allow or deny the request (enforcement). Thus, an implementation of access control requires a specification of the rights associated to users in relation to resources (a policy). For this, several models of access control have been defined, from simple access control lists giving for each user the list of authorized operations, to more abstract models, such as the popular Role-Based Access Control (RBAC) model [1]. Our focus is on enforcement, for which there exist two main approaches, static and dynamic, with a recently emerged third approach combining the two: the hybrid approach. The static approach performs all access checks at compile time, whereas the dynamic approach performs these at run time. In short, the static approach enables policy violations to be detected earlier, facilitating debugging and reducing the impact on testing, and usually involves a lower run-time cost. However, the kinds of policies enforceable statically are not as expressive nor as flexible as those enforceable by the dynamic approach. We refer to [2] for a more detailed comparison; see also [3] for hybrid analysis of programs, although not directly applicable to our problem.

Summarizing, we propose a static solution to RBAC policy enforcement for Java programs through the use of new RBAC MVC design patterns combined with a set of static verification checks made by our static verifier. The patterns integrate roles into the program as a set of Model-View-Controller (MVC) [13] components (i.e. classes) for each role. Each role's associated MVC classes act as a role-specific interface to accessing resources – protected methods in resources are invoked in these role classes only. The flow of the program directs users to the set of role classes associated to their active role. Finally, the protected invocations are checked statically for policy compliance. We present a static verifier, which performs syntactic checks and call graph analysis to ensure the invocations to methods belonging to resource classes are made only in role classes, such invocations are permitted according the policy and role classes do not invoke methods of components belonging to other roles.

Concepts of Our Proposed approach

General and specialised flow of programs that enforce RBAC. Programs that restrict access to resources from users typically involve an initial user authentication phase, where users log in and retrieve their access rights, then allowing users to undertake user tasks which may involve accessing resources, and finally logging out of the system. We present a simplified model of the general flow of a program which implements RBAC in the left-hand side of Figure 1. In RBAC, authentication also involves retrieving and activating the role(s) associated to the user, and logging out also involves deactivating the role(s). Controlling access most commonly takes place between 'Tasks' and 'Resources', for example through a reference monitor intercepting all access requests made to resources at run time, stopping those requests which are unauthorized.

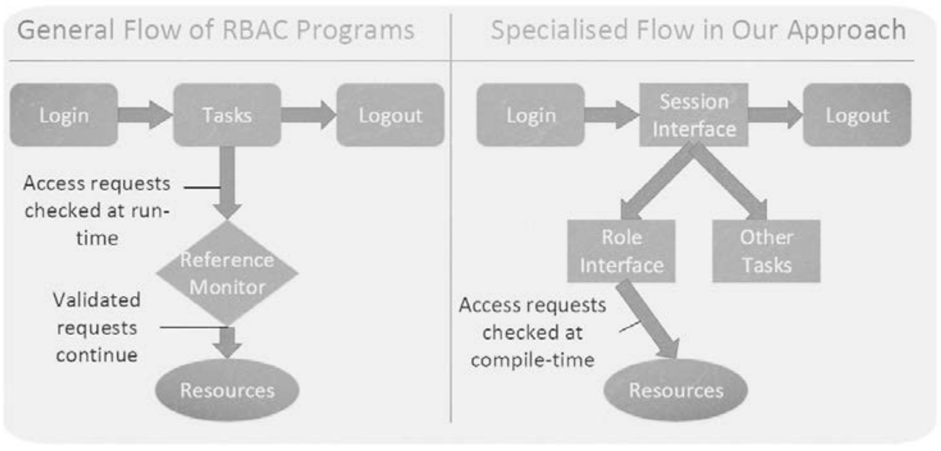
General Flow of RBAC Programs
Specialised Flow in Our Approach
Login
Tasks
Logout
Access requests checked at run-time
Reference Monitor
Validated requests continue
Resources
Login
Session Interface
Logout
Role Interface
Other Tasks
Access requests checked at compile-time
Resources

Fig. 1 General and specialized flow of programs that enforce RBAC

Basic Definitions. Definition 1 (Resource): A resource is realized as a resource class containing some methods whose invocation needs to be restricted. Invocations are restricted for instances of resource classes. Definition 2 (Actions and Auxiliary Methods): An action is a method in a resource class that must only be invoked by those users with the permission to do so. An auxiliary method is a method in a resource class that is not part of the policy definition. Such methods are usually required for the correct initialization and operation of a class, and should not be invoked directly by users. Definition 3 (Permission): A permission is a pair [res;act] where res is the name of a resource and act is the name of an action of that resource. The action is allowed to be invoked on any instance of that resource class by the role which the permission is assigned to. Definition 4 (Task): We divide the concept of a user task into three groups as follows. Firstly, a role task is an operation, or business function, to be performed by an authorised user in a specific role, which could involve the invocation of one or more actions on resources. Secondly, a session task is an operation required to correctly manage the session e.g. log-in and log-out. Thirdly, another task is an operation or function that is executable by all users, regardless of the notion of role as it does not access resources (in the access control sense). Definition 5 (Session). A session is the state of the program in which an authenticated user is able to perform the three kinds of tasks in the system. The session has a user interface composed of a session-specific interface, the role-specific interface (made up of Role MVC components discussed above) of the current active role and any interfaces implementing other tasks. The session-specific interface is made up of a set of MVC components: one Session Model, one Session Controller and a set of Session View classes. The Session Model implements the session tasks which are: log-in/authentication, role

activation, log-out, calling a role-interface and calling classes that implement other tasks. The Session

Views and Controller provide the means for the user to access these session tasks. The session-specific interface is always active so that the session tasks are always available to the user. We, of course, have minimum expectations such as log-out only being available if logged-in and so forth. The session-specific interface also allows the user to interact with the system via their role by calling a role interface, or without their role thus calling other-task implementing classes. Names of session classes start with the string ‘Session’ followed by either ‘Model’, ‘Controller’ or ‘View’. For the latter, since there can be many Session View classes, any valid class identifier (in Java) is allowed to follow in the name.

RBAC Model, Controller, View and Session Patterns: The class diagrams of the patterns are shown together in Fig. 2. RBAC Model contains only packages with names containing ‘model’, describing the design of resource and role model classes. RBAC Controller adds packages with names containing ‘controller’, describing the design of role controller classes. The empty interface class ‘Role-Controller’ simply groups all role controllers to simplify the link with session classes. RBAC View adds packages with names containing ‘view.n’ (where n represents any valid package identifier in Java) to these, describing the design of sets of role view classes. RBAC Session adds the package ‘session’, to guide the implementation of two key RBAC concepts: activating a role and users having multiple roles being able to switch between them. It also adds the package ‘other’ containing other classes, linking the session classes to them.

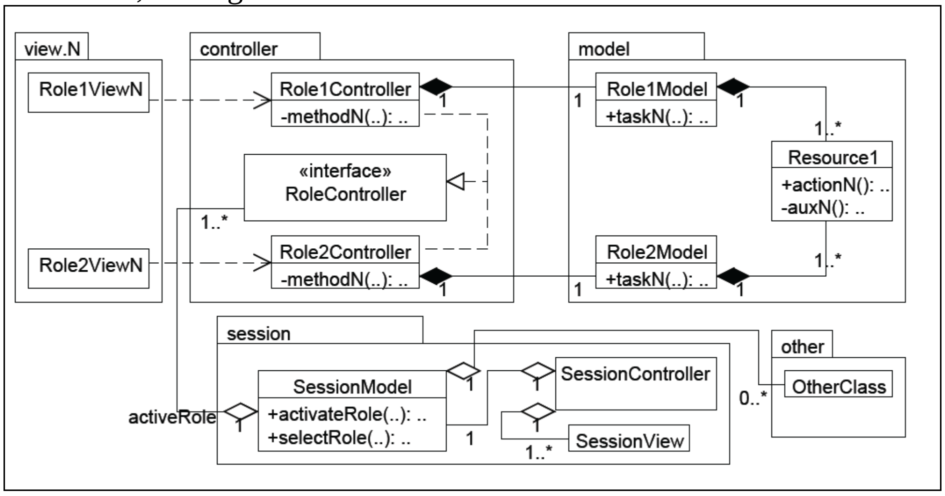

Fig. 2 UML Class Diagram of RBAC Model

Result and Implementation

Our implementation consists of a JPol policy parser, produced using the ANTLR Works tool [4], and a static analysis program which are both part of a plug-in we have produced for the Eclipse Integrated Development Environment (IDE). Eclipse plugins are able to use the Java Development Tools (JDT) Application Programming Interface (API) provided by Eclipse, whose benefits include simplifying static code analysis. In Java, there are three ways to invoke a method; either invoking a (‘static’) method on a class e.g. ‘ClassName.methodOne()’, invoking a method on a variable e.g. ‘x.methodOne()’ or invoking a method on the object returned by another method call e.g. ‘x.methodOne().methodTwo()’. Using JDT we can get the type binding for variables and method invocation expressions, and so we can check if a resource’s actions are being called or if one role’s components invoke another role’s components. This is sufficient to implement all the static checks discussed before. We have tested our plug-in on a simple doctor’s surgery web database application implemented in Java Enterprise Edition (JEE). The tool outputs helpful error messages in Eclipse’s editor window, consisting of the class name and line number where the error occurs, the kind of error that has occurred (e.g. ‘Invocation not permitted’) and a description of why that error could have occurred.

Formal approaches for the verification of properties of access control policies usually rely on purpose-built logics or rewrite-based techniques . In this paper, we have focused on verifying that a program enforces a policy, rather than on proving properties of the policies. Bodden et al. [3] enforce security properties in programs using a hybrid approach. They generate code for run-time checks, then perform compile-time analysis to eliminate some of these. In their approach, the access control enforcement of (static) roles would not be possible at compile-time, because they cannot determine, at compile-time, the access requests that each role can make. Our design pattern solves this. Therefore, in their approach, a static RBAC policy would be enforced dynamically.

Conclusion and Future Work

We have described a new system to statically check that a target program respects its RBAC policy. If the program successfully passes the static verifier's checks, then when using the program, the logged in user can only call those methods that have been authorized for the role currently activated for them. Therefore, no run-time access checks are needed. In future work, we will develop a hybrid approach for policies with dynamic conditions, in-lining code in the program to check these at run-time. This hybrid approach would utilize our concept of implementing the groupings which access rights/users are assigned to in the policy (roles in this paper) as a set of MVC components, and then statically verifying static groups whilst dynamically verifying dynamic groups. The result would allow static parts of the policy to be enforced statically, whilst still allowing dynamic policies to be expressed and then enforced dynamically.

Furthermore, we will consider systems where a policy is defined as a combination of existing policies, extending the approach in order to allow programmers to combine validated RBAC implementation without re-doing all the static checks and also use some mathematical method to implement our work[5-7].

References

[1] Piero A. Bonatti & Pierangela Samarati (2004): Logics for Authorizations and Security. In Jan Chomicki, Ron van der Meyden & Gunter Saake, editors: Logics for Emerging Applications of Databases, Springer Berlin Heidelberg, pp. 277–323, doi:10.1007/978-3-642-18690-5 8.

[2] Kevin W. Hamlen, Greg Morrisett & Fred B. Schneider (2006): Computability Classes for Enforcement Mechanisms. ACM Trans. Program. Lang. Syst. 28(1), pp. 175–205, doi:10.1145/1111596.1111601.

[3] Karsten Sohr, Michael Drouineaud, Gail-Joon Ahn & Martin Gogolla (2008): Analyzing and Managing Role-Based Access Control Policies. IEEE Transactions on Knowledge and Data Engineering 20(7), pp. 924–939, doi:10.1109/TKDE.2008.28.

[4] Jeff Zarnett, Mahesh Tripunitara & Patrick Lam (2010): Role-based Access Control (RBAC) in Java via Proxy Objects Using Annotations. In: Proceedings of the 15th ACM Symposium on Access Control Models and Technologies, SACMAT '10, ACM, New York, NY, USA, pp. 79–88, doi:10.1145/1809842.1809858.

[5] Glenn E. Krasner & Stephen T. Pope (1988): A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80. J. Object Oriented Program. 1(3), pp. 26–49.

[6] Xu B, Wang X H, Wei W, et al. On reverse Hilbert-type inequalities[J]. Journal of Inequalities and Applications, 2014, 2014(1): 198.

[7] Khalid Zaman Bijon, Ram Krishnan, and Ravi Sandhu. Risk-aware RBAC sessions. In Information Systems Security, pages 59–74. Springer, 2012.