Towards Automatic Integration Of Or-BAC Security Policies Using Aspects

Ylies Falcone[1] **and Mohamad Jaber**[2]
[1]INRIA, Rennes - Bretagne Atlantique, France
[2]VERIMAG Laboratory, Grenoble Universities, France

Abstract— *We propose a formal method to automatically integrate security rules regarding an access control policy (expressed in Or-BAC) in Java programs. Given an untrusted application and a set of Or-BAC security rules, our method derives corresponding AspectJ aspects. Derived aspects modify the behaviour of the underlying program so as to meet the policy. Then, these aspects are weaved into the target program (using the AspectJ compiler). The result is a trusted program on which the security policy is enforced. This approach was applied in order to secure the behaviour of a travel agency application.*

Keywords: Security policy, Or-BAC, Aspect Oriented Programming, AspectJ

1. Introduction

To answer the growing security needs in software and information systems, an approach widely used nowadays is the one based on security policies. The system security is ensured via the application of a set of rules and recommendations applying to several levels (physical and logical). The complexity of systems for which the policy can apply generated lots of endeavor in order to find models for security policies. These policies have to deal more and more with distributed systems where local policies might differ on the locations.

One of the most advanced modeling approaches is the one based on access control, see [9] for a survey. It rules the access, through actions, from subjects (system entities) to objects (system resources). The plethora of formalisms to express security policies allows to express at a high level of abstraction the set of requirements a system is supposed to satisfy. Among them, the Or-BAC [1] (Organization-Based Access Control) model adds structuring concepts guided by the notion of organization. The Or-BAC model generalizes the Role-Based Access Control models and adds an organizational dimension to the policy. An organization is an entity in charge of management of a set of security rules (obligations, permissions, prohibitions). The organization affords to structure the system entities: the subjects (resp. objects, actions) are abstracted into roles (resp. views, activities). A context [7] notion is added to define circumstances under which a rule applies. A policy is then a set of relations (*e.g.* interdiction) between organizations, roles, views activities and contexts.

A classical problem is then to *enforce the security policy* on the target system. Indeed, a gap usually exists between a security model and its derived implementation. Moreover, during system development, it turns out to be difficult for system administrators to borrow the formalisms on which models rely on. Three ways, based on their approach, tend to be distinguished in access control enforcement [12]: static code analysis, monitor generation, and code transformation/rewriting. The principle of enforcing a security policy by code rewriting is to transform unsafe code (that may violate the policy) into safe one prior to its execution. Recent works [11], [15] proposed theoretical frameworks for security policy enforcement. They have notably shown that program rewriting is the most powerful technique for enforcing a security policy. The principle is to transform unsafe code (that may violate the policy) into safe one prior to its execution. Also, Aspect-Oriented Programming (AOP) [13] is an efficient technique to deal with orthogonal concerns in software development. Notably, it offers practical means to transform program in an efficient and systematic way.

a) Proposed Approach: In this paper we use Or-BAC (Organization Based Access Control) to model the security policy and Aspect-Oriented Programming as an underlying technique for its enforcement. It is possible to consider our approach as the automatic integration of security into an implementation. Starting from a system and its Or-BAC policy, we use policy rules to modify the initial application by weaving the corresponding aspects. To do so, our approach proposes a model/integration of the Or-BAC concepts in the initial application. For this purpose we combine the Or-BAC rules to generate security aspects. These program rewriters are in charge of enforcing the security policy. Thereby, our method is similar to a deployment activity for security policies.

The remainder of this article is organized as follows. We review in Sect. 2 the Or-BAC model and the Aspect-Oriented Programming paradigm. Aspect generation from an Or-BAC policy is described in Sect. 3. In Sect. 4 we present the application of our method to some aspect generation examples taken from a case study. Then, in Sect. 5 we

compare our proposal with other authors' work. Finally, we made concluding remarks and future work in Sect. 6.

2. Preliminaries

In this section we briefly review the Or-BAC model for security policies and the paradigm of Aspect-Oriented Programming (AOP).

2.1 Organisation-Based Access Control

The Or-BAC model [1], [7] generalizes the Role Based Access Control models and adds an organizational dimension to the policy. An *organization* is an entity in charge of a set of security rules management (obligations, permissions, prohibitions). The system operations are called *actions*. A *subject* is an active entity of the system that may realize actions inside. By opposition to the subjects, the *objects* are the non-active entities of the system (liable to subject operations). The organization affords to structure the system entities. The subjects (resp. objects, actions) are abstracted into *roles* (resp. *views*, *activities*). A *context* notion is added to define circumstances under which a rule applies.

An Or-BAC security policy is then a set of relations between organizations, roles, views activities and contexts. The relations *Obligation (org, r, a, v, c)* (resp. *Permission (org, r, a, v, c)*, *Prohibition (org, r, a, v, c)* means that the organization *org* forces the realization (resp. grants the permission, prohibits) to a subject of role *r* to realize the activity *a* on the view *v* in the context *c*.

2.2 Aspect-Oriented Programming.

An *aspect* [13] brings together joinpoints, pointcuts, advices, introduction, and declaration. A *joinpoint* is a control point in the program where aspects can act, *e.g.* methods, constructors, and classes... *Pointcuts* are elements linked to the program execution flow where it is possible to graft an aspect around. From an abstract point of view, a pointcut defines a set of joinpoints, a "cut". The *advice* defines the grafted code by the aspect into the original application. The advice are typed in order to define the place of the code insertion regarding the pointcut (*before*, *after*, *around*). *Introduction* mechanism is an other important features of an aspect. It allows one to add some methods or some attributes to a class or an interface. The *declaration* mechanism affords process instruction during the compiling step (e.g.: display some messages).

3. The proposed approach

We present here the generation of security aspects from Or-BAC rules. It relies on the following steps (see Fig. 1):

1) extraction of needed system information (class names, hierarchy,...);
2) aspect generation using the policy and the system information;
3) integration of the aspects into the original application using aspect weaving.

The technique we adopt considers that the initial system is developped without any security consideration (everything permitted). For example there is no user notion or authentification facility. Meaning that initially, only functionnal system activities are considered. The non-functionnal activities are supposed to be provided separately (e.g. authentication mechanism, user support). The specification of an Or-BAC security policy is done at an abstract level, aside any deployment condideration. We link the Or-BAC concepts with the one of Object-Oriented Programming.

3.1 Extraction of the system information

We present now the information that we need from the underlying system implementation in order to cope with aspect generation. The need for this first stage is explained by the difference of abstraction level in the concepts of Or-BAC and Object-Oriented Programming. First, let us remark that Or-BAC concepts are independent from any implementation. Roughly speaking, the purpose of any security policy is to sort the system configurations into authorized and prohibited ones. Moreover, due to the numerous development methodologies available to software developers, it is highly difficult to identify patterns of code to be linked to access control concepts. Our proposal is to let software developers identify some patterns (events) in their implementation and then link these events to those addressed in the security policy. So the purpose of this first stage is to bridge this gap. The correspondences are summarized in Tab. 1.

The first match we make is related to the actions of the Or-BAC policy (concrete level). We choose to link actions to methods (one to one) of interest in the underlying program. It would have been possible to link one action to a sequence of methods, but in this case, it is always possible to encapsulate the method sequence into one global method.

Access control policies are dealing with the application of an action by a subject on an object. One can see an access security rule as $Modality(role, activity, view, context)$ where $Modality$ states an abstract appraisal on the 4-tuple $(role, activity, view, context)$. We will consider context in the advice part of the aspect. Indeed, they are specific to Or-BAC. As so, by restricting to 3-tuple $(role, actvity, view)$ we are dealing with role-based policies, it allows our method to be applied to most of access control policies. Each appraisal on an abstract 3-tuple $(role, activity, view)$ corresponds to a set of concrete appraisals $\{subject, action, object \mid subject \in role, action \in activity, object \in view\}$. Basically we need to know how such actions are performed under the system implementation.

We can distinguish three possible ways to express how a subject applies an action on an object.

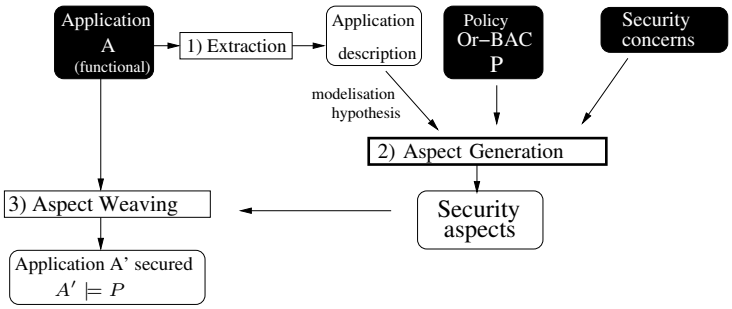

Fig. 1: Approach overview.

- In the first case we suppose that a view is related to the class notion, this means an object in the policy is related to the instance (an object) of a certain class. In this case, the funtion $ViewClass : View \rightarrow Class$ associates to each view in the policy the corresponding class (not surjective function). Thus, in the underlying system that the action is performed by the application of the method (the action) on the object. The subject is then a parameter of the action. In this case, concretely, an action a is performed by a subject s on an object o by the method call $obj.ActionMethod(a)(s)$ where obj is an instance of $ViewClass(View(o))$ corresponding to object o.
- In the second case, a role is related to the class notion; a subject is represented by the instance of a class. In this case, the funtion $RoleClass : Role \rightarrow Class$ associates to each role in the policy the corresponding class (not surjective function). Subjects of a role are instances of the corresponding class. So, in the underlying implementation, an action a is performed by a subject s on an object o by the method call $sub.ActionMethod(a)(obj)$ where sub is the instance of $RoleClass(Role(s))$.
- In the third case, the method of an action could be performed in any class. Meaning that, the implementation representation of subject and the object related to this action are parameters of the method. So an action a is performed by a subject s on an object o by the method call $x.ActionMethod(a)(o, s)$ where x is an instance of X and X the classe where $ActionMethod(a)$ belongs to.

The previously described application of the action depends on the implementation. To abstract from this detail, we denote by $apply(s, \alpha, o)$ the underlying code for the application of α by s on o, and by $EP(r, a, v)$ the set of execution points $\{apply(s, \alpha, o) \mid s \in r \wedge \alpha \in a \wedge o \in v\}$.

3.2 Aspect generation

The aspect generation uses as inputs the policy, the system information previously extracted, and additional modules. It produces aspects in two steps: the generation of pointcuts and advices. Due to space limitations, we only present the generation from prohibition rules (see Fig. 3 in Sect. 4.2 for an example).

Pointcut generation. The pointcut definition catches the set of execution points defined by $EP(r, a, v)$. For each element $apply(s, \alpha, o)$ of $EP(r, a, v)$, we generate a pointcut. Each of these pointcuts pickouts the execution of $ActionMethod(\alpha)$ (depending on the underlying implementation). To be effective it also dynamically checks at the current execution state that the α action is genuinely performed by the subject s on object o. One can see the pointcut as in-lined monitors of events in the underlying system. we generate the pointcut shown on Fig. 2 for $apply(s, \alpha, o)$.

Advice generation. The advice code performs the rule enforcement, triggered by the pointcut. It starts by verifying the context $Context$, this corresponding code depends on the implementation. It is generated from information provided by the system designer in the mapping, or is provided by an additional module. Note that the context verification can be performed only at the execution time, since it may depend on dynamically created content. Thus contexts must be assessable by a computable function. Then, if the context is verified, the advice code prevents from performing the action, and raises an exception. Otherwise, the rule should not be applied, the method $ActionMethod(\alpha)$ is proceeded normally. We generate the pointcut shown on Fig. 3.

Additional modules. Some activities or actions may not be initially present in the application, though they can be addressed in the policy. The implementation of these functions can be considered as an input of the aspect generation. Typical examples of such additional modules is context authentication. These additional functions are imported using some security libraries. We then use the introduction mechanism of Aspect-Oriented programming to make them available in the aspect definitions.

4. Case study : a travel agency

We depict here how we apply the previously presented technique to study the security of an application of virtual travel agency named *Travel*. To show the feasibility of our approach, we derived from the original version a

Or-BAC (abstract)	Object Oriented Implementation (concrete)
Action α	Method $ActionMethod(a)$
Activity a on view v by role r	$\{apply(s, \alpha, o) \mid s \in r \wedge \alpha \in a \wedge o \in v\}$
Context cxt	A method able to evaluate it $ContextVerification(cxt)$

Table 1: Correspondance between Or-BAC and OO concepts

```
pointcut apply_S_Alpha_O(T_S subject, T_O object) :
execution (apply(s,alpha,o) )
&& args(subject,object);
```

Fig. 2: Pointcut generated for an execution point $apply(s, \alpha, o)$

```
Object around() : apply_S_Alpha_O() {
    String s=null;
      if (VerificationContext(Context))
            s="Violating self validation rule";
      else
            s = proceed();
     return s;}
```

Fig. 3: Advice of the aspect

mutant where no security mechanism is included. Then we generated aspects from Or-BAC rules stating some security requirements the application must fulfill.

4.1 The Travel application and its security policy

The *Travel* application originates from the french national project Politess [10]. It allows the management of travel requests for its user. To record requests, a user should hold an account on the application. Validators are attributed to each user (as well a *Travel* user) who is the only person able to perform the final mission validation. In the case of a long absence, a user can delegate his access rights to another user. The delegated person can act in the name of the original user.

We formalized security requirements of the *Travel* policy using Or-BAC. For example, one of the requirement expresses the refusal of self-validation: "A validator cannot validate his own business mission: for a given mission the validator and the traveller are distinct". This could be reasonably understood and formalized in the following Or-BAC rule: $Prohibition(Travel, validator, validate, business\_mission, mission\_ownership)$. Where the context $mission\_ownership$ states that the mission is registered by the current traveler. This context can be checked at runtime.

4.2 Aspect generation for Travel

Following our generation method, the aspect generated for the previous rule is presented on Fig. 4 and Fig. 5. In this case we create the aspect with a pointcut (Fig. 4) defined by $apply(traveler, validMission, mission)$ corresponding to $EP(validator, validate, business\_mission)$. Then, we generate the code advice (Fig. 5) of type *around* where we check the context $mission\_ownership$ using the function $VerificationContext$ defined in the additional modules. If it is confirmed that we must prohibit the execution of the action *validMission* (no call of *proceed*). Otherwise, the user is authorized to execute the action (call of *proceed*).

5. Related works

The idea of using automatic rewriting techniques to enforce security policies is not new. And then some other works are similar to the one presented in this paper.

- In [5] Cuppens and al. derive AspectJ abstract aspects from requirements expressed in Nomad [6], a Metric temporal logic. They focus on availability properties: maximum execution and maximum waiting time policy to prevent the SYN flooding attack in TCP/IP protocol.
- In [8] de Oliveira and al. enforce rewrite-based security policies using AspectJ. Access control policies are formalized using rewriting systems. Policies are weaved in the underlying program using Tom, an extension of Java to define rewrite systems.

Those approaches differ from the one introduced in this paper. To the best of our knowledge our proposed approach is the first which translates directly high level access-control requirements to direct program modificators so as to enforce the property.

An other approach is the so called Monitoring Oriented Programming (MOP) [2] which is a programming paradigm in wich the monitor synthesis is realised from a user-defined specification, using Aspect-Oriented Programming. This environment is implemented in a tool, Java-MOP [3], based on AspectJ. The objective of this framework is to monitor the underlying program. So, it differs of our approach, indeed the monitoring approach aims mainly to detect misbehaviors, not prevent them.

6. Conclusion and future works

This article presents an aspect generation technique from an Or-BAC security policy. This method allows to develop a system without taking into consideration any security concern. Then, from the security policy we generate a set of security aspects. The aspects are then integrated into the initial application using the aspect weaver. We have presented the method for Or-BAC which is a generalization of a majority of Role Based Access Control policy models.

```
pointcut valider1() :
    execution (String DiagImpl.validMission(int , int) )
        && if(getRole((Integer)thisJoinPoint.getArgs()[0].intValue())== Validator
        && getView((Integer)thisJoinPoint.getArgs()[1].intValue())== business_mission)
```

Fig. 4: Pointcut of the aspect generated for rule preventing self-validation

```
Object around() : valider1() {
    String s=null;
      if (VerificationContext(mission_ownership))
            s="Violating self validation rule";
      else
            s = proceed();
     return s;}
```

Fig. 5: Advice of the aspect generated for rule preventing self-validation

We generate aspects described with AspectJ, which is one of the most famous AOP implementations. We claim that the proposed method is general and it can be applied to several RBAC-like policies and aspect languages.

This work opens several research perspectives that we are currently investigating. First of all, it seems to us possible to extends the presented method with underlying program semantics consideration. Also, we plan to integrate this method into an Or-BAC policy deployment tool integrating in MotOr-BAC [4]. In addition it seems interesting to combine this approach with a test generation approach [14] from an Or-BAC policy. Another working direction is to formally analyze possible interactions between weaved aspects. Notably, we are looking to how to formally prevent conflicts between aspects.

References

[1] A. Abou El Kalam, R. E. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)*, June 2003.

[2] F. Chen, M. D'Amorim, and G. Roşu. Checking and correcting behaviors of java programs at runtime with java-mop. In *Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*, pages 3–20, 2005.

[3] F. Chen and G. Roşu. Java-mop: A monitoring oriented programming environment for java. In *Proceedings of the Eleventh International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS'05)*, volume 3440 of *LNCS*, pages 546–550. Springer-Verlag, 2005.

[4] F. Cuppens, N. Cuppens-Boulahia, and C. Coma. MotOrBAC : un outil d'administration et de simulation de politiques de sécurité. In *Security in Network Architectures (SAR) and Security of Information Systems (SSI), First Joint Conference*, June 6-9 2006.

[5] F. Cuppens, N. Cuppens-Boulahia, and T. Ramard. Availability enforcement by obligations and aspects identification. In *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security (ARES'06)*, pages 229–239, Washington, DC, USA, 2006. IEEE Computer Society.

[6] F. Cuppens, N. Cuppens-Boulahia, and T. Sans. Nomad: A security model with non atomic actions and deadlines. In *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 186–196, Washington, DC, USA, 2005. IEEE Computer Society.

[7] F. Cuppens and A. Miège. Modelling Contexts in the Or-BAC Model. In *19th Annual Computer Security Applications Conference (ACSAC '03)*, 2003.

[8] A. S. de Oliveira, E. K. Wang, C. Kirchner, and H. Kirchner. Weaving rewrite-based access control policies. In *ACM Conference on Computer and Communication Security*, November 2007.

[9] S. di Vimercati, P. Samarati, and S. Jajodia. Policies, models, and languages for access control, 2005.

[10] French National Project. RNRT Politess, 2007. http://www.rnrt-politess.info.

[11] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[12] Kevin W. Hamlen. *Security Policy Enforcement by Automated Program-rewriting*. PhD thesis, Cornell University, Aug. 2006.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[14] K. Li, L. Mounier, and R. Groz. Test purpose generation from or-bac security rules. In *31st Annual IEEE International Computer Software and Applications Conference*, 2007.

[15] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 2007.