Formal Specification of Constraints on Role-Based Access Control Policies

Nafi' Alswae'r, Khair Eddin Sabri
Computer Science Department
King Abdullah II School for Information Technology
The University of Jordan
Amman, Jordan
Email: nalswaier@souq.com, k.sabri@ju.edu.jo

Abstract

Access control policies state the authorization of users to access resources. One of the most used model is Role-Based Access Control model RBAC which assigns permissions to roles instead of users. Usually, organizations have regulations that can be implemented as constraints on policies. Therefore, to ensure the enforcement of the regulations, all stated policies in the organizations should satisfy the predefined constraints. Manual checking of the satisfaction is time consuming and subject to errors. Therefore, an automated process is essential.

In this paper, we introduce a formal specification of RBAC policies using First Order Logic (FOL). We identify constraints on these policies and formalize them in FOL. To check the satisfaction of constraints, we implement policies and constraints using Prolog programming language. Finally, we validate our work through a case study.

Keywords: Role-Based Access Control, Constraints, First Order Logic

1 INTRODUCTION

Access control models are developed to specify the authorizations of users to access resources. Several models are proposed in the literature such as Mandatory Access Control (MAC), Discretionary Access Control (DAC) and Role-Based Access Control (RBAC). RBAC model has attracted several researchers such as [1, 2, 3] due to its suitability for practical use in organizations where the access of information is given to roles instead of users. This would reduce the complexity of specifying policies.

Organizations need to make sure that policies satisfy predefined constraints. These constraints are usually regulations implemented in the organization. An example of a constraint is the prohibition of a user playing two conflicting roles such as a developer and a reviewer, as the user who develops the code should be different from the one who reviews it. Another constraint could be specifying the number of users that should have an access to a record. Other constraints could state that all records should be accessible to users and all users should be assigned to roles. Therefore, we should verify that the stated policies added by the administrator follow these constraints. In a large system, where there are many policies and constraints, the verification becomes more time consuming and vulnerable to human error. Therefore, the automated verification at the run time once a policy is added or deleted becomes essential.

The goal of this paper is developing a formal framework that allows specifying policies and constraints and then automatically verifying the satisfaction of constraints. Our main contribution is developing a logical framework that can be used as an authorization system based on predefined policies and as a validation system that can be used to enforce constraints. We extend RBAC to specify policies on type of objects as well as on objects to reduce the complexity when having a large number of objects. Finally, we implement the framework using Prolog programming language. The structure of the paper is as follows. Section 2 presents the literature review. Section 3 introduces the proposed framework. Also, it lists several constraints and their specifications. Section 4 validates the work through a case study. Finally, the conclusion is given in Section 5.

2 LITERATURE REVIEW

Many researches worked in the field of RBAC model and the enforcement of constraints. This section presents an overview of existing techniques and approaches for role-based access control. The work in [4] presented a theory

for specifying policies and constraints in first order logic. Furthermore, the paper presented a comprehensive list of constraints and used a general purpose theorem prover tool called Prover9 for proving the satisfaction of constraints. However, the RBAC model defined in the paper did not handle role hierarchy and did not distinguish between different actions on objects. The paper [10] used FOL to detect conflicts in RBAC policies. However, this work did not consider constraints other than the separation of duties. Furthermore, our formalism of RBAC is different from the model of [10] as we explicitly specify objects and their types which gives flexibility in specifying systems. The work of [7] extended RBAC model and used defeasible logic for specification, but their main goal is handling conflicts and not specifying constraints.

The work in [5] presented an approach to identify constraints using declarative language called Object Constraints Language (OCL) which is part of the Unified Modeling Language (UML). The authors of [9] presented an approach for systematically incorporating role-based access control policies into an application design model specified using UML. Their goal is the visualization of constraints. The verification of the satisfiability of constraints is not considered in these papers.

The paper [8] presented a set-based specification language and discussed the enforcement of separation of duties constraints. Other constraints are not discussed in this paper. In [6], the authors presented an algebraic approach for specifying and enforcing constrains. However, they did not consider RBAC model.

3 PROPOSED FRAMEWORK

Role-based access control consists of users, roles, objects, and permitted actions. In RBAC, authorization is given to roles in order to access objects instead of users. Users are assigned to roles and therefore, a user playing a role is allowed to access all the objects authorized to that role. In our model, to simplify stating policies in organizations that have thousands of objects that can be categorized, we define policies on types of objects. Therefore, a user can access all the objects of the same type. However, an authorization to a specific object and to a specific user can be expressed as well within our model. Furthermore, we define a set of actions that can be applied on objects such as read, write, etc. We define the following predicates to specify these sets.

- *user(x):* indicates that x is a user.
- *role(x):* indicates that x is a role.
- *object(x):* indicates that x is an object.
- *type(x):* indicates that x is an object type.
- *action(x):* indicates that x is an action.

For example, the predicate *user(John)* indicates that John is a user, the predicate *role(Manager)* indicates that Manager is a role in the system, the predicate *object(rec)* indicates that rec is an object, the predicate *type(security)* indicates that security is a type of objects, and the predicate *action(insert)* indicates that insert is an action that can be performed on objects.

In addition to the defined sets, we define the following predicates to define relations:

- *hasRole(x,y)* $\subseteq$ *user* $\times$ *role* indicates that the user x has the role y.
- *hasType(x,y)* $\subseteq$ *object* $\times$ *type* indicates that the object x has the type y.

For example, the predicate *hasRole(John, Manager)* indicates that John is playing the role of Manager. The predicate *hasType(rec, security)* indicates that the object rec is a security file.

A policy in our model indicates the allowed action of a role on a type of an object.

Definition 3.1. Let t be an object type, r be a role, a be an action, we specify a policy as *policy(t,r,a)*.

The policy predicate states that the role r has the permission to perform the action a on all objects of type t. For example, the *policy(security, Manager, read)* states that the manager can read files of security type. To give an access of a user to a specific object, we define the authorized predicate.

Definition 3.2. Let o be an object, u be a user, and a be an action, we specify an authorization to access an object as *authorized(o,u,a)*

The authorized predicate states that the user u is authorized to perform the action a on the object o. The authorization for users can be explicitly stated in the system but in most cases it can be inferred from *policy*, *hasRole* and *hasType* predicates as shown in the following formula:

$\forall a,r,o,t,u$ *(hasRole(u,r)* $\wedge$ *hasType(o,t)* $\wedge$ *policy(t,r,a)* $\rightarrow$ *authorized(o,u,a))*

which states that if the user a has the role r, the object o has the type t, and the policy indicates that the role r is allowed to perform the action a on objects of type t, then the user u is allowed to perform the action a on the object o. For example, *hasRole(John, Manager)* $\wedge$ *hasType(rec, security)* $\wedge$ *policy(security, Manager, Read)* $\rightarrow$ *authorized(rec, John, Read)*.

Usually, the roles can be represented as a hierarchy where the role on the top of the hierarchy has the most authority. We specify the hierarchy by defining the predicate

- *moreAuthority(x, y)* $\subseteq$ *role* $\times$ *role*

which states that the role x has more authority than the role y. Therefore, any action permitted to x is also permitted to y which is formulated as:

$\forall r_1, r_2, a, t$ *(moreAuthority(*r_1*,* r_2*)* $\wedge$ *policy(t,* r_2*, a)* $\rightarrow$ *policy(t,* r_1*, a))*

For example, if the manager has more authority than the engineer and the engineer is allowed to read all files of type bugs, then the manager is allowed to read these files.

moreAuthority(Manager, Engineer) $\wedge$ *policy(bugs, Engineer, read)* $\rightarrow$ *policy(bugs, Manager, read)*

The moreAuthority relation should be transitive and asymmetric. We enforce the transitivity property by adding the following formula to the set of policies.

$\forall r_1, r_2, r_3$ *moreAuthority(*r_1*,* r_2*)* $\wedge$ *moreAuthority(*r_2*,* r_3*)* $\rightarrow$ *moreAuthority(*r_1*,* r_3*)*

The above formula states that if r_1 has more authority that r_2 and r_2 has more authority than r_3, then we can conclude that r_1 has more authority than r_3. The asymmetric property can be enforced as a constraint as discussed in the next section.

3.1 Constraints

In this section, we state different constraints that can be found in an organization or are essential for all RBAC models. Then, we represent them using our proposed framework. These constraints should be inferred from the policies.

3.1.1 Hierarchy

This constraint is on the hierarchy of roles. The relation should be asymmetric. $\forall x, y$ *(moreAuthority(x,y)* $\wedge$ *moreAuthority(y,x)* $\rightarrow x = y$*)*

The constraint states that the only case that x has more authority than y, and y has more authority than x is that x and y are equal.

3.1.2 Separation of Duties

This constraint states that two conflicting cases should not occur such as a user should not be assigned to two conflicting roles. To formulate this constraint, we define a predicate called *conflicts(x)* to indicate all the roles that are in conflicts. Then, we state the constraint as follows $\neg\exists u, r_1, r_2((hasRole(u,r_1) \wedge (hasRole(u,r_2) \wedge conflicts(r_1) \wedge conflicts(r_2) \wedge r_1 \neq r_2))$. This constraint states a user u who is playing the roles r_1 and r_2 where both roles belong to the same *conflicts* set is not permitted.

Similarly, we can define constraints such that two specific roles should not be able to read the same type of objects and no role can perform two conflicting actions.

3.1.3 Cardinality

This constraint is applied on the relations defined previously as follows:

hasRole(x,y)

- Each user should have a role $\forall x \exists y$ *(user(x)* $\rightarrow$ *(role(y)* $\wedge$ *hasRole(x,y)).*
- Every role should have a user assigned to it $\forall x \exists y$ *(role(x)* $\rightarrow$ *(user(y)* $\wedge$ *hasRole(y,x))*
- At least two users should be assigned to a role. For example, at least two different users should be involved in the quality assurance process and therefore, at least two different users should be assigned the Quality Assurance (QA) role $\exists x, y$ *(hasRole(x, QA)* $\wedge$ *hasRole(y, QA)* $\wedge$ $x \neq y$*).*

hasType(x,y)

- Each object should have exactly one type. We can represent this constraint using two formulas. The first one states that each object has a type $\forall x \exists y$ *(object(x)* $\rightarrow$ *type(y)* $\wedge$ *hasType(x,y)).* The second one states that no object has two different types $\neg\exists x, y, z$ *(hasType(x,y)* $\wedge$ *hasType(x,z)* $\wedge$ $y \neq z$*)*

policy(t,r,a)

- This constraint is on the cardinality of roles accessing an object type. For example, as a regulation of an organization and for confidentiality reason, all objects of type security should be accessible for reading by only one role. $\exists r_1$*(policy(security,*r_1*,read)*$\wedge(\forall r_2$ *policy(security,*r_2*,read)* $\rightarrow r_1 = r_2$*)).*

authorized(o,u,a)

- This constraint is on the cardinality of users accessing an object. For example, for quality assurance, at least two different users should review the record rec. $\exists u_1, u_2$*(authorized(rec,*u_1*,review)* $\wedge$ *authorized(rec,*u_2*,review)* $\wedge$ $u_1 \neq u_2$*).*

3.1.4 Prerequest

It specifies that a policy should exist in order to add a new one. For example, a user playing a role chair should be already playing the role instructor. $\forall$*u(hasRole(u,Chair)* $\rightarrow$ *hasRole(u,Instructor))*

4 CASE STUDY

As a case study, we are taking a part of the JIRA tool used to track and plan software [11]. We state policies and constraints, and implement them using Prolog programming language. Then, we verify automatically the satisfaction of constraints based on the defined policies.

JIRA is a tracking system project founded by Atlassian and widely used by companies to track their projects. JIRA tracking system is based on tickets created by the employees themselves. Ticket types and the actions that can be performed on the tickets depend on the business of the project itself. In our case study, we take part of the most used ticket types and cover the necessary policies and constraints. First, we specify and implement policies in Prolog. We implement all the predicates given previously. Assume that we have the following types of tickets:

1. Story: A ticket created by the Project manager. It contains a new work with a full description for the team to work on it.
2. Bug: A ticket created by the Quality control team. It contains a full description of a functional defect in a new code or in the existing project.
3. Security: A ticket created by the security team. It contains a full description of a security defect on the production project.

```
ticket(story).
ticket(bug).
ticket(security).
```

Assume that we have four records which are rec1, rec2, rec3, and rec4.

```
object(rec1).
object(rec2).
object(rec3).
object(rec4).
```

The records rec1 and rec2 are of type story, the record rec3 is of type bug, and the record rec4 is of type security.

```
hastype(rec1,story).
hastype(rec2,story).
hastype(rec3,bug).
hastype(rec4,security).
```

Assume that we have the following roles:

1. Software Quality Control Engineer
2. Security Engineer
3. Software Engineer
4. Software Engineer Manager
5. Software Engineer Director
6. Product Manager

```
role(qa).
role(security_team).
role(engineer).
role(enginnering_manager).
role(engineering_director).
role(product_manager).
```

Role hierarchy is applied on some of the above roles as described below:

1. Software Engineer reports to Software Engineering Manager
2. Software Engineering Manager reports to Software Engineering Director.
3. Product Manager reports to Software Engineering Director.

Also, we need to implement the transitive property of the moreAuthority relation.

```
more_authority1(engineering_manager,
                engineer).
more_authority1(engineering_director,
                engineering_manager).
more_authority1(engineering_director,
                product_manager).
more_authority(A, B) :-
                more_authority1(A, B).
more_authority(A, C) :-
               more_authority1(A, B),
               more_authority(B, C).
```

Assume that there are six users, which are Nafea, Salma, Husni, Ahmad, Haitham and Zaid.

```
user(nafea).
user(salma).
user(husni).
user(ahmad).
user(haitham).
user(zaid).
```

Assume that Nafea is a Software Quality Control Engineer, Salma is a Software Engineer as well as Software Engineer Manager, Ahmad is a Software Engineer, Zaid is a Software Engineer Director, Haitham is a Product Manager, and Husni is a Security Engineer.

```
hasrole(nafea,qa).
hasrole(salma,engineer).
hasrole(salma,enginnering_manager).
hasrole(ahmad,engineer).
hasrole(zaid,engineering_director).
hasrole(haitham,product_manager).
hasrole(husni,security_team).
```

Assume that we have the following actions:

1. Create: Create a new ticket in the system.
2. Start: Start working on a ticket.
3. Review: Review the process on a ticket.

```
action(create).
action(start).
action(review).
```

Assume we have the following policies:

1. Product Manager Role can create story tickets.
2. Software Quality Control Engineer can create bug tickets.
3. Security Engineer can create security tickets.
4. All tickets can be reviewed by the Software Quality Control Engineer.
5. Software Engineer can start working on all tickets.
6. Security Engineer can start working on security ticket.

```
policy(story,product_manager,create).
policy(bug,qa,create).
policy(security,security_team,create).
policy(story,qa,reviwe).
policy(bug,qa,review).
policy(security,qa,review).
policy(story,engineer,start).
policy(bug,engineer,start).
policy(security,engineer,start).
policy(security,security_team,start).
```

We include into the specification, the ability of a role to access all the records that can be accessed by a role lower in the authority hierarchy.

```
policy(A,X,C) :- more_authority(X,Y),
                 policy(A,Y,C).
```

Finally, we define the authorized predicate.

```
authorized(O,U,A):- policy(T,R,A),
                    hasRole(U,R),
                    hasType(O,T).
```

Next, we specify constraints. Below we define several constraints and their implementation in Prolog.

1. Each user should have a role.

   ```
   forall(user(X),hasrole(X,Y)).
   ```

2. Each role should have at least one user assigned to it.

   ```
   forall(role(Y),hasrole(X,Y)).
   ```

3. The Software Engineer role is a pre-request for the Software Engineer Manager role.

   ```
   forall(hasrole(X,engineering_manager),
          hasrole(X,engineer)).
   ```

4. No user can be a Software Quality Control Engineer and a Software Engineer

   ```
   \+((user(X),
       hasrole(X,qa),
       hasrole(X,engineer))).
   ```

5. Two different users should be able to start working on the record rec4.

   ```
   authorized(rec4,X,start),
   authorized(rec4,Y,start),
   X\=Y.
   ```

6. Two different users belonging to two different roles should be able to review the record rec4.

   ```
   authorized(rec4,X,review),
   authorized(rec4,Y,review),
   X\=Y, hasRole(X,Z),
   hasrole(Y,W),
   Z\=W.
   ```

7. The asymmetric property of the moreAuthority relation.

```
\+((more_authority(X,Y),
    more_authority(Y,X),
    X\=Y)).
```

8. Each object has exactly one type.

```
forall(object(X),hastype(X,Y)),
\+((object(X), hastype(X,Y) ,
    hastype(X,Z) , Y\=Z )).
```

9. An Engineer should not be able to review records

```
\+(policy(X,engineer,review)).
```

5 CONCLUSION

In this paper, we introduce a theory for specifying RBAC policies in First Order Logic. The theory allows specifying policies on types of objects as well as individual objects to reduce the complexity of specifying policies. Furthermore, we identify a comprehensive list of constraints that can be applied on policies. We validate our work by implementing the theory in Prolog programming language and check the satisfaction of constraints automatically through the program. As a future work, we are extending theory to represent the delegation of policies and identify more constraints.

REFERENCES

[1] Q. Ni, A. Trombetta, E. Bertino, and J. Lobo, "Privacy-aware role based access control," in *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '07. New York, NY, USA: ACM, 2007, pp. 41–50.

[2] W. Kuijper and V. Ermolaev, "Sorting out role based access control," in *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '14. New York, NY, USA: ACM, 2014, pp. 63–74.

[3] E. Bertino, P. A. Bonatti, and E. Ferrari, "Trbac: A temporal role-based access control model," *ACM Transactions on Information and System Security*, vol. 4, no. 3, pp. 191–233, Aug. 2001.

[4] K. E. Sabri, "Automated verification of role-based access control policies constraints using prover9," *International Journal of Security, Privacy and Trust Management*, vol. 4, no. 1, pp. 1–10, 2015.

[5] G.-J. Ahn and M. E. Shin, "Role-based authorization constraints specification using object constraint language," in *Proceedings of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, ser. WETICE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 157–162.

[6] K. E. Sabri and H. Hiary, "Algebraic model for handling access control policies," in *The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016)*, ser. Procedia Computer Science, vol. 83, 2016, pp. 653–657.

[7] K. E. Sabri and N. Obeid, "A temporal defeasible logic for handling access control policies," *Applied Intelligence*, vol. 44, no. 1, pp. 30–42, Jan. 2016.

[8] J. Crampton, "Specifying and enforcing constraints in role-based access control," in *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '03. New York, NY, USA: ACM, 2003, pp. 43–50.

[9] I. Ray, N. Li, R. France, and D.-K. Kim, "Using uml to visualize role-based access control constraints," in *Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '04. New York, NY, USA: ACM, 2004, pp. 115–124.

[10] A. Schaad, "Detecting conflicts in a role-based delegation model," in *Proceedings of the 17th Annual Computer Security Applications Conference*, ser. ACSAC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 117–.

[11] R. Sagar, *Mastering JIRA*. Packt Publishing, 2015.