(7) Constraints for Role-Based Access Control

Fang Chen and Ravi S. Sandhu

George Mason University, Fairfax, VA
fchen@issc.gmu.edu

1.0 Introduction

Role-based access control (RBAC) has been discussed a lot in recent days, for example [SAND96a..MOHA94]. Although there are many variations, there is some consensus about the basic architecture. That is, all permissions are assigned to roles, rather than directly to users, and roles are then assigned to users. When a user creates a session, the session gets all permissions of the roles the user activates for it. For convenient role management, roles can be structured in hierarchies whereby senior roles are more powerful, having more permissions, than junior roles. Another important aspect of RBAC is constraint enforcement. In this paper we investigate a practical way for a Database Administrator (DBA) or Security Officer to specify constraints. For brevity, we use SCS to stand for Security Constraint Specifiers, which can be either DBAs or Security Officers.

The basic idea to apply constraints is to lay out higher level organizational policy. A typical example is that of mutually disjoint roles. Once certain roles are declared to be mutually exclusive, there need not be so much concern about the assignment of individual users to roles. The latter activity can then be delegated and decentralized without fear of compromising overall policy objectives of the organization.

Constraints can be specified at either system level or application level, with or without being event triggered (i.e., the occurrence of certain events causes certain constraints to be applied. The general approach to specifying constraints should nonetheless be common to all these cases.

To specify constraints, one needs appropriate languages as well as some system functions. The language should be simple and intuitive, so it can easily be used without much training. Also, the language should have flexibility and richness to address most security requirements. At the same time, the language needs a formal foundation so as to be unambiguous and precise.

There are several ways to specify constraint. One was to treat them as invariants that should hold at all times, another is to treat them as preconditions for functions such as adding a role to a user. The former has theoretical advantage but is less efficient in real systems, partially due to using quantifiers like $\forall$ and $\exists$. For the latter, the problem is how to specify a constraint that can be shared by several objects, rather than each object having its own specific constraints.

In ongoing work at George Mason University (GMU), we are developing techniques to specify constraints that can both be used as preconditions shared by several objects and serve as system-wide invariants. Some simple conversions from invariants to preconditions may be used to enforce these constraints.

2.0 RBAC Structure

We are not going to discuss which RBAC structure is the best for addressing security policy, because our major focus here is on constraints. However, RBAC structure is the foundation that must be well defined before SCS can specify constraints, whether the constraints are system-level or application-level.

Here we give our own RBAC structure for purpose of our later discussion. This RBAC structure can also have many of variations. We give only one here to illustrate how to specify constraints without loss of generality.

Our RBAC structure includes role, permission, user, and session, which are given in object-oriented style.

2.1 Role

We define a class Role with attributes as follows:

- Role-id, identifying the role
- Permission set (PS), containing references to all permissions objects assigned to this role
- User set (US), containing references to all user objects holding this role
- Parent role set (PRS), containing references to all direct senior roles
- Child role set (CRS), containing references to all direct junior roles

The class Role also has functions such as implies “etc.” as add/delete/change parent role, add/delete/change child role, add role to user, and remove role from user.

2.2 Permission

The class Permission is defined with the following attributes:

- Permission-id, identifying the permission
- Operations, defining the action of the permission
- Target list, specifying parameters as objects on which operations apply

- Role set (RS), containing references to all role objects holding this permission

The functions within class permission can be add permission to role, remove permission from role, and others.

2.3 User

Class User has attributes as follows:

- User-id, identifying the user

- Role set (RS), containing references to all role objects hold by the user

- Session set (SS), containing references to all active session objects of the user

This class has no functions directly associated with it because the appropriate functions are defined for the Role and Session classes. Strictly speaking, all of our classes should have create and delete functions, but these can be ignored for our present purpose.

2.4 Session

The attributes in class Session are as follows:

- Session-id, identifying the session

- User, referencing the user object submitting this session

- Role set (RS), containing references to all role objects taken by this session

Relevant functions can be add role to session, drop role from session, and others.

2.5 Discussion

Although the RBAC system could be quite complicated, the RBAC system state is, nonetheless, decided by all those attribute sets in the RBAC structure, i.e., RS, US, SS, PS, PRS, and CRS. Any change in these sets will lead to an RBAC state change. Constraints in RBAC give restrictions to RBAC states, known as invariants, as well as to state changes, known as preconditions.

One may argue that attributes such as user-id should also be included in the RBAC state. However, compared with those attribute sets, user-id is less significant for the entire RBAC state. Similarly, for session-id, of course, we can, if required, define a general RBAC state which includes all attributes in the RBAC structure.

3.0 Specifying Constraints in RBAC

In this section we outline a practical way to specify constraints. The specified constraints can both be used as preconditions shared by several objects and also serve as invariants. Some simple conversions from invariants to preconditions may be used to enforce these constraints. For the convenience of description, we first define several global functions.

3.1 Global Functions

The global functions defined here are to be used in specifying constraints. These functions are defined with overloading.

- Role-set: permission $\rightarrow$ role-set, i.e., mapping a permission to the permissions RS attribute
- Role-set: permission-set $\rightarrow$ role set, i.e., $\cup_{permission \in \text{permission-set}}$ role-set*(permission)*
- Role-set: user$\rightarrow$ role set, i.e., mapping a user to the user's RS attribute;
- Role-set: user-set$\rightarrow$ role-set, i.e. ,$\cup_{user\text{-}set}$ role-set*(user)*
- Role-set: session$\rightarrow$ role-set, i.e., mapping a session to the session's RS attribute
- Role-set: session-set$\rightarrow$ role-set, i.e.,$\cup_{session \in \text{session-set}}$ role-set*(session)*

Similarly, we can define functions such as permission-set and user-set. Also, we have the following:

- Parent-role-set: role $\rightarrow$ role-set, i.e., mapping a role to the role's PRS attribute
- Parent-role-set: role-set $\rightarrow$ role-set, i.e.,$\cup_{role \in role\text{-}set}$ parent-role-set*(role)*
- Parent-role-set-at: role, number $\rightarrow$ role-set, i.e.,
  $\underbrace{\text{parent} \leftharpoondown \text{role} \leftharpoondown \text{set}(\ldots\text{parent} \leftharpoondown \text{role} \leftharpoondown \text{set}(\text{role})\ldots)}_{\text{number}}$
- Parent-role-set-up: role, number $\rightarrow$ role-set, i.e., $\cup_{1 \leq i \leq number}$ parent-role-set-at*(role, i)*
- Parent-role-set-up: role $\rightarrow$ role-set, i.e., parent-role-set-at*(role, i)*

We can have functions such as child-role-set, child-role-set-at, and child-role-set-down. Also, we can define function permission-set-down to give all permissions a role can have, both directly and indirectly.

3.2 Language for Specifying Constraints

The language we use to specify constraints is based on set theory. The basic components are as follows:

- Set description for role-set, permission-set, user-set
- Set operator: union ($\cup$), intersection ($\cap$), difference ($-$)
- Relation operator: $\neq$, $\subset$, $\subseteq$, $\supset$, $\supseteq$, $\in$, $\notin$
- Logic operator: and ($\wedge$), or ($\vee$), not ($\neg$), implication ($\rightarrow$)
- Function: set element count ($\|$), global functions, system functions, security officer defined functions, and non-deterministic functions

Two new related non-deterministic functions, oneelement and allother, are introduced to replace explicit quantifies.

- Oneelement: set $\rightarrow$ element, i.e., get one element from set
- Allother: set $\rightarrow$ set, i.e., get set by taking out one element

They are related by context, because for any set$_s$ {oneelement(s)} $\cup$ allother(s) = s, and at the same time, neither is a deterministic function. In the next subsection, we use several examples to illustrate how to use these two functions to specify constraints.

3.3 Examples

3.3.1 Conflicting Roles for Some Users

- Role set: $R = \{r_1, r_2, \ldots, r_n\}$
- User set: $U = \{u_1, u_2, \ldots, u_m\}$
- Check-condition: oneelement(R) $\in$ role-set(oneelement(U)) $\rightarrow$ allother(R) $\cap$ role-set(oneelement(U)) = $\emptyset$

This condition can be an invariant because functions “oneelement” and “allother” are non-deterministic, so their values are implicitly quantified over the entire set. It can also be used as a precondition when a role, say, r_1 is to be assigned to a user, say u_1. Only the right hand of the implication is used as the precondition, which can be shared by all role objects in R for all user objects in U. For this specific case, $R - \{r_1\}$ will substitute “allother(R)”, and u_1 will substitute “oneelement(U).” (The descriptions for the following cases are similar to this one.)

3.3.2 Conflicting Roles for Sessions of Some Users

- Role-set: $R = \{r_1, r_2, \ldots, r_n\}$
- User-set: $U = \{u_1, u_2, \ldots, u_m\}$

- Check-condition: oneelement(R) $\in$ role-set(session-set(oneelement(U))) $\rightarrow$ allother(R) $\cap$ role-set(session-set(oneelement(U))) = $\varnothing$

3.3.3 Prerequisite Roles for Some Roles with Respect to Some Users

- Role set: $R_1 = \{r_{11}, r_{12}, \ldots, r_{1p}\}$

- Role set: $R_2 = \{r_{21}, r_{22}, \ldots, r_{2q}\}$

- User set: $U = \{u_1, u_2, \ldots, u_m\}$

- Check-condition: oneelement(R_2) $\in$ role-set(oneelement(U)) $\rightarrow R_1 \subseteq$ role-set(oneelement(U))

3.4 Discussion

The relationship between invariants and preconditions is one to many, because to keep one invariant satisfied, different operations should have different preconditions. In example 3 above, if we want to add r_{21} to u_1, we need to check whether $R_1 \subseteq$ role-set(u_1); however, if we want to remove r_{11} from u_1, we need to check if $R_2 \cap$ role-set(u_1) = $\varnothing$.

Although there are many different operations, the essential point for specifying constraints is to take care of RBAC system state changes. In other words, whenever an attribute set is to be changed, for any possible constraint (invariant) *Ct*, the following issues have to be considered:

- Whether *Ct* applies to this change

- How to check *Ct* for this potential change, i.e., how to convert *Ct* to a precondition for this state update

The basic changes to an RBAC state are:

- Add an item to an attribute set

- Delete an item from an attribute set

Any other change can be seen as a sequence of these two basic ones.

However, one may wish to do two state changes at the same time, with legal initial and final states, i.e., satisfying all constraint (invariants), but not the intermediate state. For example, a system may require that exactly one manager must exist at any time. In this case, if we need to change manager from one user to another using either delete-add or add-delete, the intermediate state is illegal. Therefore, we have to either introduce a transaction concept or expand basic RBAC state changes to more complicated ones. Unfortunately, with the former, using precondition will become impossible, while with the latter, more work has to be done to convert invariants to preconditions. This issue is left for future research.

4.0 Logical Foundation

Any constraint (invariant) written in our language can be easily translated to a first-order predicate logic-based language. The rule to convert a constraint *Ct* to such a language is as follows:

- If function oneelement(*S*) or allother(*S*) appears in *Ct*, convert *Ct* to $\forall s \in S\ (Ct)$
- Replace oneelement(*S*) with *s*
- Replace allother(*S*) with $(S - \{s\})$

For example:

$$\text{oneelement}(R) \in \text{role-set}(\text{oneelement}(U)) \rightarrow \text{allother}(R) \cap \text{role-set}(\text{oneelement}(U)) = \emptyset$$

can be converted to

$$\forall r \in R\ \forall u\ U\ (r \in \text{role-set}(u) \rightarrow (R - \{r\}) \cap \text{role-set}(u) = \emptyset)$$

Therefore, it is clear that all constraints written in our language can also be written in a first-order predicate logic-based language. However, in general, a constraint written in first-order predicate logic is not always easy to be taken as a precondition for an operation in RBAC. Our language, to some extent, forces constraints to be written in a way that is easier to be treated as a precondition.

Can all useful constraints be written using our language? This is a hard question, because in general, constraints are open to all aspects. However, any specific constraint can be seen as a condition consisting of several components, some of which are RBAC state related, while others are not. For example, some system may require inclusion of date or time information into constraints, which, of course, is not RBAC state related. Therefore, the original question can be converted to the one that is, with the given RBAC structure and facilities (global functions), can all useful RBAC state-related constraints be written using this language?

We must emphasize that even global functions are open to SCS. Although we can provide most necessary global functions, we may not be able to take care of all their specific needs. Therefore, we should allow SCS to add more global functions they need by themselves. From this view point, our language is a very open one.

5.0 Conclusion

In this paper, we outline our recent progress on constraints for RBAC, which can be summarized as follows:

- Define an RBAC structure
- Provide a set of overloaded global functions

- Propose a new language with two new non-deterministic functions to specify constraints

- Discuss the logic foundation problem underlying the new language

This entire language development needs substantial further work. Consequently, our description is preliminary and tentative at this point.

References

[FERR92] David F. Ferraiolo, Richard Kuhn, “Role-based Access Controls.” *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, Baltimore, MD, 13-16 October 1992, 554-563.

[MOHA94] Imtiaz Mohammed, David M. Dilts, “Design for Dynamic User-Role-Based Security,” *Computer and Security*, 13(8), 1994, 661-671.

[SAND96a] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman, “Role-Based Access Control.” *IEEE Computer*, 29:2, February 1996, 38-47.

[TING92] T. C. Ting, S. A. Demurjian, and M. Y. Hu, “Requirements Capabilities and Functionalities of User-Role Based Security for an Object-Oriented Design Model,” *Database Security V: Status and Prospects*, C. Landwehr and S. Jajodia (Eds.), North-Holland, 1992 275-296.