Group Hierarchies with Constrained User Assignment in Linux

Gail-Joon Ahn[1] and Seng-Phil Hong[2]

[1] University of North Carolina-Charlotte, Charlotte, NC, USA

[2] LG-CNS, Seoul, Korea

Abstract. In this paper we investigate one aspect of RBAC administration concerning assignment of users to roles. A user-role assignment model can also be used for managing user-group assignment. We overview a constrained user-group assignment model and describe its implementation in the Linux system. Rather than set user and file rights individually for each and every user, the administrator can give rights to various groups, then place users within those groups in Linux. Each user within a group inherits the rights associated with that group. We describe an experiment to extend the Linux group mechanism to include group hierarchies and decentralized user-group assignment can be implemented by means of setgid programs.

1 INTRODUCTION

Role-based access control (RBAC) has received considerable attention as a promising alternative to traditional discretionary and mandatory access controls (see, for example, [NO95,FCK95,GI96,SCFY96,JGAS01]). In RBAC permissions are associated with roles, and users are made members of appropriate roles thereby acquiring the roles' permissions. This greatly simplifies management of permissions. Roles are created for the various job functions in an organization and users are assigned to roles based on their responsibilities and qualifications. Users can be easily reassigned from one role to another. Roles can be granted new permissions as new applications and systems are incorporated, and permissions can be revoked from roles as needed.

Sandhu and Bhamidipati [SB97] introduced the URA97 model for decentralized administration of user-role membership (URA97 stands for user-role assignment 1997). They simply focused on user-role assignment without consideration of the important constraints such as separation of duty (SOD) constraints. An example of SOD policy may be "the patent submitted to a patent authorization agency can be reviewed only by a member of its patent review committee." This simple role-based access control may not be adequate for expressing many business policies. An example of such policy is "none of the applicants of the patent is eligible to review a patent, even though the applicant is a patent review committee member." These policies, also known as SOD constraints should be dealt with user-role assignment.

Constraints are an important aspect of RBAC and are often regarded as one of the principal motivations behind RBAC. Although the importance of constraints in RBAC

Popp G. (2004).
Towards a Systematic Development of Secure Systems.
In *Proceedings of the 2nd International Workshop on Security in Information Systems*, pages 13-22
DOI: 10.5220/0002654400130022
Copyright © SciTePress

has been recognized for a long time, they have not received much attention. [AS00] recently showed that role-based authorization constraints can be expressed by the specification language called RCL 2000. We use the concept of static separation of duty (SSOD) borrowed from this work. The central contribution of this article is to describe how we can achieve this kind of constraints during user-role assignment named constrained user-role assignment as an extension of URA97.

A user-role assignment model can also be used for managing user-group assignment and therefore has applicability beyond RBAC. The notion of a role is similar to that of a group, particularly when we focus on the issue of user-role or user-group membership. For our purpose in this paper we can treat the concepts of roles and groups as essentially identical. The difference between roles and groups was hotly debated at the ACM Workshop [YCS95,San97]. There exists the consensus that a group is a named collection of users (and possibly other groups). Groups serve as a convenient shorthand notation for collections of users and that is the main motivation for introducing them. Roles are similar to groups in that they can serve as a shorthand for collections of users, but they go beyond groups in also serving as a shorthand for a collection of permissions. Assigning users to roles or users to groups are therefore essentially the same function.

The rest of the paper is organized as follows. In section 2, we review the URA97 grant model. Section 3 discusses role-based authorization constraints. Section 4 describes constrained user-group assignment (CONUGA) including implementation details. Section 5 concludes the paper.

2 OVERVIEW OF URA97 MODEL

This section reviews URA97. We often use the term group as an identical notion of role. Our description of URA97 is informal and intuitive. A formal statement of URA97 is given in [SB97]. In this section we s imply give a quick overview of the grant model which is dealing with granting a user membership in a group.

2.1 User-Group Grant Model

URA97 imposes restrictions on which users can be added to a group by whom. URA97 requires a hierarchy of groups (such as in Figure 1) and a hierarchy of administrative groups (such as in Figure 2). The set of groups and administrative groups are required to be disjoint. Senior groups are shown toward the top and junior ones toward the bottom. Senior groups inherit permissions from junior groups. We write $x > y$ to denote x is senior to y with obvious extension to $x \geq y$. The notion of prerequisite condition is a key part of URA97. User-group assignment is authorized in URA97 by the *can-assign* relation.

Definition 1. *A* ***prerequisite condition*** *is a boolean expression using the usual* $\wedge$ *and* $\vee$ *operators on terms of the form* x *and* $\overline{x}$ *where* x *is a regular role (i.e.,* $x \in R$*). A prerequisite condition is evaluated for a user* u *by interpreting* x *to be true if* $(\exists x' \geq x)(u, x') \in UA$ *and* $\overline{x}$ *to be true if* $(\forall x' \geq x)(u, x') \notin UA$*. For a given set of roles* R *let* CPR *denotes all possible prerequisite conditions that can be formed using the roles in* R*.*

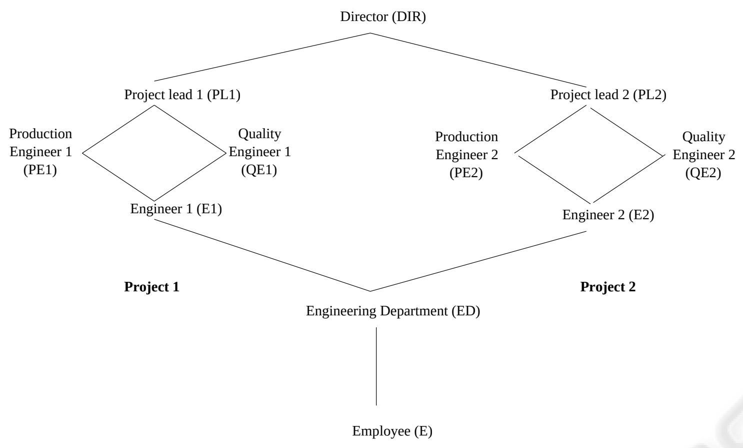

Fig. 1. An example group hierarchy

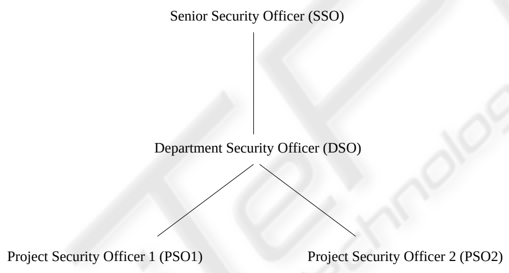
Senior Security Officer (SSO)
Department Security Officer (DSO)
Project Security Officer 1 (PSO1)
Project Security Officer 2 (PSO2)

Fig. 2. An example administrative group hierarchy

Definition 2. *The URA97 model controls user-role assignment by means of the relation* can-assign $\subseteq AR \times CPR \times 2^R$.

The meaning of *can-assign*$(x, y, \{a, b, c\})$ is that a member of the administrative role x (or a member of an administrative role that is senior to x) can assign a user whose current membership, or non-membership, in regular roles satisfies the prerequisite condition y to be a member of regular roles a, b or c.

2.1.1 Range Notation

URA97 also defines *can-assign* by identifying a range within the role hierarchy by means of the familiar closed and open interval notation.

Definition 3. *Role sets are specified in the URA97 model by the notation below*

$$
\begin{aligned}
[x,y] &= \{r \in R \mid x \geq r \wedge r \geq y\} \\
(x,y] &= \{r \in R \mid x > r \wedge r \geq y\} \\
[x,y) &= \{r \in R \mid x \geq r \wedge r > y\} \\
(x,y) &= \{r \in R \mid x > r \wedge r > y\}
\end{aligned}
$$

3 ROLE-BASED AUTHORIZATION CONSTRAINTS

Constraints are an important aspect of access control and are a powerful mechanism for laying out higher level organizational policy. Consequently the specification of constraints needs to be considered. So far this issue has not received enough attention in the area of role-based access control. [AS00] identified the major classes of constraints in RBAC such as *Prohibition Constraints* and *Obligation Constraints*, including *Cardinality Constraints*. We briefly overview these identified classes of constraints in role-based systems.

3.1 Prohibition Constraints

In organizations, we need to prevent a user from doing (or being) something that he is not allowed to do (or be) based on organizational policy. *Prohibition Constraints* are constraints that forbid the RBAC component from doing (or being) something which it is not allowed to do (or be). A common example of prohibition constraints is SOD. SOD is a fundamental technique for preventing fraud and errors, known and practiced long before the existence of computers. We can consider the following statement as an example of this type of constraint: if a user is assigned to purchasing manager, he cannot be assigned to accounts payable manager. This statement requires that the same individual cannot be assigned to both roles which are declared mutually exclusive.

3.2 Obligation Constraints

We also need to force a user to do (or be) something that he is allowed to do (or be) based on organizational policy. We derived another class of constraints from this motivation. *Obligation Constraints* are constraints that force the RBAC component to do (or be) something. The motivation of this constraints is from the simulation of lattice-based access control in RBAC. There exists a constraint which requires that certain roles should be simultaneously active in the same session. There is another constraint which requires a user to have certain combinations of roles in user-role assignment. We classify this kind of constraints as obligation constraints.

3.3 Cardinality Constraints

Another constraint is a numerical limitation for the number of users, roles, and sessions. For example, only one person can fill the role of department chair; similarly, the number of roles (sessions) an individual user can belong to (activate) could be constrained.

4 CASE STUDY: CONSTRAINTS AND USER-GROUP ASSIGNMENT

Most of role-based constraints work have focused on separation of duty constraints which is a foundational principle in computer security. As a security principle, SOD is used to formulate multi-user control policies, requiring that two or more different users be responsible for the completion of a transaction or set of related transactions. The purpose of this principle is to minimize fraud by spreading the responsibility and authority for an action or task over multiple users, thereby raising the risk involved in committing a fraudulent act by requiring the involvement of more than one individual. A frequently used example is the process of preparing and approving purchase orders. If a single individual prepares and approves purchase orders, it is easy and tempting to prepare and approve a false order and pocket the money. If different users must prepare and approve orders, then committing fraud requires a conspiracy of at least two, which significantly raises the risk of disclosure and capture.

Although separation of duty is easy to motivate and understand intuitively, so far there is no formal basis for expressing this principle in computer security systems. Several definitions of SOD have been given in the literature. We have the following definition for interpreting SOD in role-based environments [AK01].

> **Role-Based separation of duty** *ensures SOD requirements in role-based systems by controlling membership in, activation of, and use of roles as well as permission assignment.*

Separation of duty constraints can be determined by the assignment of individuals to roles at user-assignment time. Consider the case of initiating and authorizing payments. The separation of duty constraints could require that no individual who can serve as payment initiator could also serve as payment authorizer. This could be implemented by ensuring that no one who can perform the initiator role could also be assigned to the authorizer role. This static separation of duty can apply to the user-role assignment. Therefore, we adapt the grant model in URA97. User u can be explicitly assigned to role r_i where (u,r_i) $\in UA$. Also user u can be implicitly assigned to role r_j where $(\exists r_i \preceq r_j)[(u, r_j) \in UA]$. Let CR be a set of roles which are needed to be in static SOD. CR is said to be a conflicting role set. The static SOD requirement is that the same user cannot be assigned explicitly or implicitly to more than one role in CR.

We can enforce static SOD as we check each assignment task with a given CR. We have AT-SET (assignment time set) table which includes SOD sets used to enforce SOD requirements at assignment time. The example of AT-SET table with CR is described below. This table tells us that role $pay\_initiator$ and $pay\_authorizer$ are conflicting each other so a user cannot be assigned to both roles.

SET-NAME	ELEMENT
CR$_1$	{ pay_initiator, pay_authorizer }

Whenever System Security Officer (SSO) does assignment tasks, each assignment task should be checked with AT-SET table and satisfy the constraints in the table. Figure 3 describes an algorithm which achieves desired behavior of CONUGA. There are

Grant Algorithm

Let *invoker* be an initiator of user-role assignment and let *assign_DB* have three attributes such as *assign_DB*.`admin`, *assign_DB*.`cond` and *assign_DB*.`range` to construct a table as shown in Table 1.

```
invoker_role_set ← Membership(invoker)
target_role ← role to be assigned
user ← user to which target_role is assigned
assign_DB ← can-assign relation table
CR_set ← AT_SET table
grant_Flag ← false
assign_role_set = $\phi$

While (assign_DB $\neq$ EOF)
    if invoker_role_set exists in assign_DB.admin then
     if target_role exists in assign_DB.range then
      user_role_set ← Membership (user);
      if user_role_set exists in assign_DB.cond then
       grant_Flag = true;
       return;
      endif
     endif
    endif
End

if grant_Flag = true then
    assign_role_set ← JuniorList (target_role);
    if assign_role_set $\cap$ CR_set = $\phi$ then
      do the assignment of role in assign_role_set;
    else
      exit;
    endif
endif
```

Procedure `Membership` (*user*)
Take all assigned roles to a user

Procedure `JuniorList` (*role*)
Take all junior roles to a specified role in role-hierarchies

Fig. 3. Grant Algorithm in CONUGA

two procedures called `Membership` and `JuniorList`. `Membership` procedure allows us to have all assigned roles to a user and `JuniorList` procedure returns all junior roles to a specified role by walking down the hierarchy. This grant algorithm checks *can-assign* table and AT-SET table to enforce constrained user assignment.

4.1 Implementation Details

Every account in Linux contains a group membership list indicating which groups the account belongs to. Users belonging to a group are explicitly enumerated in either `/etc/passwd` (for the primary group) or `/etc/group` (for secondary groups). Many commercial database management systems, such as Informix, Oracle and Sybase, provide facilities for hierarchical groups (or roles). Commercial operating systems, however, provide limited facilities at best for this purpose.

To maintain the group hierarchy we use the file `/etc/grouphr` to store the children and parents of each group. The group hierarchy of Figure 1 is represented in `/etc/grouphr` as shown in Table 1. The first column gives the group name, the second column gives the (immediate) parent groups of that group, and the third column gives the (immediate) children. The null symbol "−" means that the group has no parent or child as the case may be. Using `/etc/grouphr`, we can find all seniors and juniors for a group by respectively chasing the parents and children.

We say a user is an *explicit* member of a group if the user is explicitly designated as a member of the group. A user is an *implicit* member of a group if the user is an explicit member of some senior group. To simulate a group hierarchy we use information about explicit and implicit membership in `/etc/group`. If Alice belongs explicitly or implicitly to a group she will be added to that group's member list in `/etc/group`. However, `/etc/group` is not sufficient to distinguish the case where Alice is both an explicit and implicit member of some group from the case where she is only an implicit member of the group. For this purpose we introduce another file `/etc/explicit` that keeps information about explicit membership only.

In order to enforce separation of duty constraints, we maintains `/etc/at_set` which includes conflicting roles. This table also can contain conflicting users and permissions. Table 2 illustrates how we can accommodate such sets to support constrained user assignments.

There are two issues that need to be addressed in decentralized management of group membership. Firstly we would like to control the groups that an administrative group has authority over. Recall figures 1 and 2 which respectively show the regular and administrative groups of our example. We would like to say, for example, that the PSO1 administrative group controls membership in project 1 groups, i.e., E1, PE1, QE1 and PL1. Secondly, it is also important to control which users are eligible for membership in these groups.

URA97 addresses these two issues respectively by means of a *group range* and a *prerequisite group* or more generally a *prerequisite condition*. URA97 has a *can_assign* relation which we store in the file `/etc/can_assign`. We put a colon between the columns to indicate the boundary. Table 3 illustrates the general case of `/etc/can_assign` with prerequisite conditions. Let us consider the PSO1 rows. The first row authorizes PSO1 to assign users with prerequisite group ED into E1. The second one authorizes

Table 1. The example group hierarchy of Figure 1

Group Name	Parent Group(s)	Child Group(s)
DIR	-	PL1, PL2
PL1	DIR	PE1, QE1
PL2	DIR	PE2, QE2
PE1	PL1	E1
QE1	PL1	E1
PE2	PL2	E2
QE2	PL2	E2
E1	PE1, QE1	ED
E2	PE2, QE2	ED
ED	E1, E2	E
E	ED	-

Table 2. The example AT-SET table: `/etc/at_set`

Set Name	Elements
conf-roles-1	QE1, QE2
conf-roles-2	PE1, PE2
conf-roles-3	PL1, PL2

Table 3. Example of `/etc/can_assign`with Prerequisite Conditions

Administrative Group	Prerequisite Condition	Group Range
PSO1:	ED :	[E1,E1]:
PSO1:	ED $\wedge$ $\overline{\text{QE1}}$:	[PE1,PE1]:
PSO1:	ED $\wedge$ $\overline{\text{PE1}}$:	[QE1,QE1]:
PSO1:	PE1 $\wedge$ QE1:	[PL1,PL1]:
PSO2:	ED:	[E2,E2]:
PSO2:	ED $\wedge$ $\overline{\text{QE2}}$:	[PE2,PE2]:
PSO2:	ED $\wedge$ $\overline{\text{PE2}}$:	[QE2,QE2]:
PSO2:	PE2 $\wedge$ QE2:	[PL2,PL2]:
DSO:	ED:	(ED,DIR):
SSO:	E:	[ED,ED]:
SSO:	ED:	(ED,DIR]:

PSO1 to assign users satisfying the prerequisite condition that they are members of ED but not members of QE1 to PE1. Taken together the second and third rows authorize PSO1 to put a user who is a member of ED into one but not both of PE1 and QE1. The fourth row authorizes PSO1 to put a user who is a member of both PE1 and QE1 into PL1. Note that, a user could have become a member of both PE1 and QE1 only by actions of a more powerful administrator than PSO1. The rest of table 3 is similarly interpreted.

Table 4. The permission of reference files

PERMISSION	OWNER	Setgid	GROUP	FILE NAME
U:rw- G:rws W:--x	root	YES	rbac	assign
U:rw- G:rws W:--x	root	YES	rbac	weak_revoke
U:rw- G:rws W:--x	root	YES	rbac	strong_revoke
U:rw- G:rw- W:r--	root	NO	rbac	`/etc/group`
U:rw- G:rw- W:r--	root	NO	rbac	`/etc/explicit`
U:rw- G:rw- W:r--	root	NO	rbac	`/etc/can_assign`
U:rw- G:rw- W:r--	root	NO	rbac	`/etc/can_revoke`
U:rw- G:rw- W:r--	root	NO	rbac	`/etc/grouphr`
U:rw- G:rw- W:r--	root	NO	rbac	`/etc/at_set`

Assignment of a user to a group in URA97 means explicit assignment. Implicit assignment to junior groups happens as a consequence and side-effect of explicit assignment. In other words `/etc/can_assign` applies only to explicit membership.

We use the setgid feature of Linux to enforce this behavior. The setgid (set group ID or SGID) file access modes provide a way to grant users access to which they are not otherwise entitled on a temporary, command level basis via a specified program. When a file with SGID access is executed, the effective group ID of the process is changed to the group of the file, acquiring that group's access rights for duration of the program contained in this file. Using setgid a user who is working as an administrative group can read and write the reference files: `/etc/group`, `/etc/explicit`, `/etc/grouphr`, `/etc/can_assign` and `/etc/can_revoke`. Thereby we can enforce desired behavior of URA97 with respect to different administrative groups.

To implement CONUGA in Linux we use several reference files introduced in the previous sections and set their permission bits as shown in table 4. The three procedures assign, weak_revoke and strong_revoke are setgid to the special group `rbac` defined for this purpose. These procedures can read and write the five reference files. We previously described the structure of files `/etc/group`, `/etc/explicit`, `/etc/grouphr`, `/etc/at_set`, `/etc/can_assign` and `/etc/can_revoke`. For simplicity all these files in our implementation are owned by root. We assume that the `rbac` group has no members.

In our implementation a user invokes the procedure call to grant or revoke a group from or to another user. The parameters specify which user is to be assigned to target_group, or to be weakly or strongly revoked from target_group. This implementation is convenient for administrative groups since they only need to define the group hierarchy and the relations `/etc/can_assign` and `/etc/can_revoke`. These procedures are called at the Linux command line prompt as follows.

```
[usage] assign username target_group
[usage] weak_revoke username target_group
[usage] strong_revoke username target_group
```

5 CONCLUSION

In this paper we have described how to extend the Linux group mechanism supporting constrained user group assignment model that is useful in managing group-based access control. When a user is assigned to a group the system checks constraints including prerequisite conditions and conflicting role set, and *automatically* adds the user to all junior groups to the group. We have extended the URA97 model and implemented it in Linux by means of setgid programs. Our result indicates that (static) separation of duty constraints can be determined by the assignment of individuals to groups at user-group assignment time and this behavior can be achieved by accommodating sophisticated access control model to some extent.

References

[AK01] Gail-Joon Ahn and Kwangjo Kim. CONUGA: Constrained User Group Assignment. *Journal of Network and Computer Applications*, 24(2), April 2001.

[AS00] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and System Security*, 3(4):207–226, November 2000.

[FCK95] David Ferraiolo, Janet Cugini, and Richard Kuhn. Role-based access control (RBAC): Features and motivations. In *Proceedings of 11th Annual Computer Security Application Conference*, pages 241–48, New Orleans, LA, December 11-15 1995.

[GI96] Luigi Guiri and Pietro Iglio. A formal model for role-based access control with constraints. In *Proceedings of IEEE Computer Security Foundations Workshop 9*, pages 136–145, Kenmare, Ireland, June 1996.

[JGAS01] James Joshi, Arif Ghafoor, Walid G. Aref, and Eugene H. Spafford. Digital government security infrastructure design challenges. *IEEE Computer*, 34(2):66–72, February 2001.

[NO95] Matunda Nyanchama and Sylvia Osborn. Access rights administration in role-based security systems. In J. Biskup, M. Morgernstern, and C. Landwehr, editors, *Database Security VIII: Status and Prospects*. North-Holland, 1995.

[San97] Ravi Sandhu. Roles versus groups. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*. ACM, 1997.

[SB97] Ravi Sandhu and Venkata Bhamidipati. The URA97 model for role-based administration of user-role assignment. In T. Y. Lin and Xiaolei Qian, editors, *Database Security XI: Status and Prospects*. North-Holland, 1997.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[YCS95] Charles Youman, Ed Coyne, and Ravi Sandhu, editors. *Proceedings of the 1st ACM Workshop on Role-Based Access Control, Nov 31-Dec. 1, 1995*. ACM, 1995.