Decentralized Group Hierarchies in UNIX: An Experiment and Lessons Learned

Ravi Sandhu and Gail-Joon Ahn

Laboratory for Information Security Technology (LIST)
Information and Software Engineering Department, MS 4A4
George Mason University, Fairfax, VA 22030
{sandhu,gahn}@isse.gmu.edu, www.list.gmu.edu

ABSTRACT Unix includes a simple group mechanism for access control. In this paper we describe an experiment to extend this mechanism in two significant ways that are valuable in managing group-based access control in large-scale systems. The goal of our experiment is to demonstrate how group hierarchies (where groups include other groups) and decentralized user-group assignment (where administrators are selectively delegated authority to assign selected users to selected groups) can be implemented by means of Unix setgid programs. In both respects the experimental goal is to implement previously published models, specifically RBAC96 for group hierarchies and URA97 for decentralized user-group assignment. Our results indicate that Unix has adequate flexibility to accommodate modern access control models to some extent, but that it also has critical limitations. The paper discusses how additional setgid based mechanisms could be implemented to make our implementation more scalable.

1 INTRODUCTION

Groups have been used for access control ever since the first time-sharing systems were implemented in the early 1970s. A group is a collection of users and serves as a convenient unit for granting and revoking access. Membership in a group is presumably determined by the need to share resources and information so the group provides a suitable unit for access decisions. A user or administrator can make a resource available to an entire group without having to explicitly provide access to every member. Similarly, access can be revoked from a group without explicitly revoking each member's access. Also new users can be made members of appropriate groups, thereby obtaining access to a number of resources.

Unix includes a simple group mechanism for access control [GS96]. Users belonging to a group are explicitly enumerated in either `/etc/passwd` (for the primary group) or `/etc/group` (for secondary groups). Unix notably lacks a facility for including one group in another. In practice, it is often desirable that groups bear some relationship to each other. For instance, consider a project divided into several independent tasks assigned to different teams. We can define a group for each task team so its members have common access to files relevant to the task. Since some files may pertain to the entire project we can define a project group such that members of the individual task groups are thereby also members of the project group. The project wide files are then made explicitly available to the project group alone. This is certainly more convenient than having to explicitly make such files available to every task group.[1] It is also more convenient than explicitly making every member of a task group a member of the project group. By allowing membership in a group to automatically imply membership in some other groups we can reduce the number of explicit access decisions that need to be made by users and administrators. Many commercial database management systems,

[1]Unix protection bits only allow file permissions to be granted to one group. Several Unix implementations now incorporate access control lists where a file permissions can be configured for multiple groups. Hierarchical groups are useful in either case.

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.

Let x>y signify that group x is *senior* to y, in the sense that a member of x is also automatically a member of y but not vice versa. Note that a member of x has the power of a member of y and may have additional power, hence a member of x is considered senior to a member of y. It is natural to require that seniority is a partial ordering, i.e., > is irreflexive, transitive and asymmetric. The irreflexive property is obviously required since every member of x is already a member of x. Transitivity is certainly an intuitive assumption and perhaps even inevitable. After all, if x>y and y>z then a member of x is a member of y and so should also be a member of z. The asymmetric requirement eliminates redundancy by excluding groups which would otherwise be equivalent. We write x≥y to mean x>y or x=y. If x is senior to y we also say that y is *junior* to x. For convenience we use the term hierarchy to mean a partial order.

An example of a group hierarchy for a hypothetical engineering department is shown in figure 1. By convention, senior groups are shown toward the top and junior ones toward the bottom. Transitive edges from seniors to juniors are omitted. In this example there is a junior-most group E to which all employees in the organization belong. Within the engineering department there is a junior-most group ED and senior-most group DIR.[2] In between there are groups for two projects within the department, project 1 on the left and project 2 on the right. Each project has a senior-most project lead group (PL1 and PL2) and a junior-most engineer group (E1 and E2). In between each project has two incomparable groups, production engineer (PE1 and PE2) and quality engineer (QE1 and QE2). We will use this example throughout this paper.

This example can be extended to dozens and even hundreds of projects within the engineering department. Moreover, each project could have a different structure for its groups. The example can also be extended to multiple departments with different structure and policies applied to each department.

Another limitation of Unix groups is that membership is exclusively controlled by the root account. This is a centralized model which does not scale gracefully to systems with large numbers of groups and users. More generally, it is possible to decentralize user-group assignment by allowing administrators to selectively delegate authority to assign selected users to selected groups. Our decentralization philosophy is motivated by the principle that a manager who can assign a user to work on a particular task should also have the authority to enroll that user in appropriate groups which confer the necessary permissions to work on that task. Effective decentralization of user-group assignment is one step towards making security more acceptable to end users as an enabling and empowering technology, rather than as the general nuisance it is often perceived to be.

In this paper we describe an experiment to extend the Unix group mechanism to include group hierarchies and decentralized user-group assignment by means of setgid programs. A setgid program runs with the permissions of the group associated with the program, rather than with permissions of the groups associated with the user who invokes the program [GS96]. The setgid feature allows the access control behavior of Unix to be extended in a controlled and protected manner.[3]

Although setgid has been a part of Unix for a long time we are not aware of prior experiments in using setgid programs in such a comprehensive manner to extend Unix access control. Published work on extending access control has usually focussed on changing the Unix kernel, an approach which we deemed to be outside our experimental scope. Our objective was to focus on techniques that do not require access to Unix kernel source code since this is typically proprietary and not

[2]For purpose of our example it is convenient to have a powerful senior-most group DIR. We emphasize that, in general, our model allows arbitrary hierarchies so it is *not* required that there be such a senior-most group. In many cases we would not want to have such a group. Similar comments apply to the junior-most group E.

[3]Many programs in a typical Unix installation that could be run setgid to an appropriate group are run setuid to root. This latter practice is very dangerous and is the root cause (no pun intended) of many security penetrations in Unix [GS96]. Our experiment uses setgid programs in the spirit of least privilege to avoid this vulnerability.

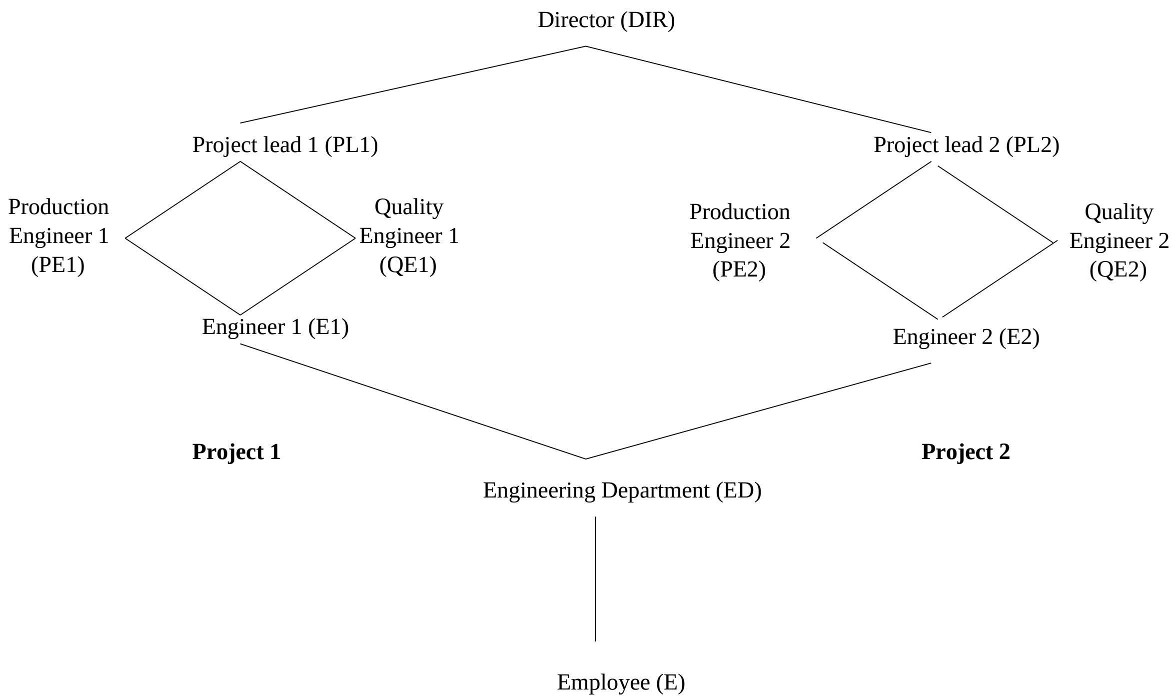


Figure 1: An example group hierarchy

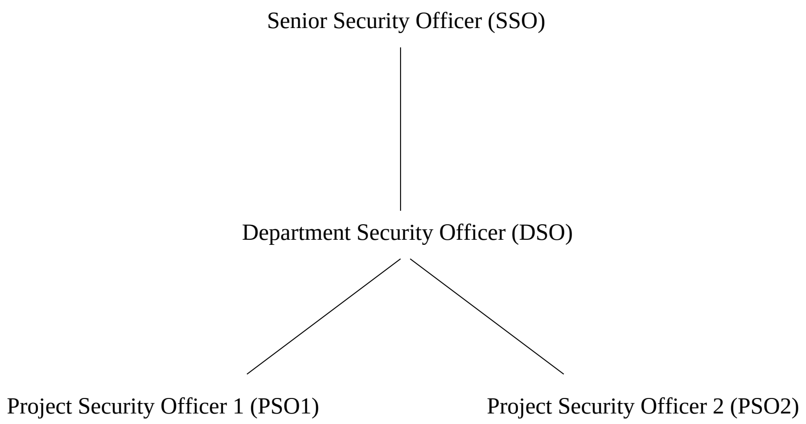


Figure 2: An example administrative group hierarchy

available to most installations. Modifying the Linux kernel for our experiments was ruled out, since we would like our solution to apply to commercial off-the-shelf Unix. Some examples of extending the access control behavior of Unix by means of kernel modifications are given in [BSS$^+$95a, BSS$^+$95b, FH97, Mey97, STCYYS94].

One of the difficulties inherent in the kind of experiment we describe here is that we need models for group hierarchies and for decentralized user-group assignment before the experimental implementation on Unix can be attempted. If these models are designed as part of the experiment there will always be a question as to whether the model was designed (deliberately or inadvertently) to facilitate a Unix setgid-based implementation. Fortunately we were able to use previously published models for our experiment to avoid this possibility of bias in model design. Our model for group hierarchies is based on the RBAC96 model for role-based access control [SCFY96].[4] The model for decentralized user-group assignment, called URA97, is adapted from [SB97]. Neither model was designed with Unix setgid programs in mind. There are numerous papers in the literature on hierarchical groups and alternate models for this purpose including [FWF95, HDT95, NO95, RBKW91, San88].

The example of figure 1 is taken from [SB97]. URA97 distinguishes between regular groups and administrative groups. Figure 2 shows a hierarchy of administrative groups. The senior-most group is the senior security officer (SSO). Junior to SSO is a department security officer group (DSO) and two project security officer groups (PSO1 and PSO2). These administrative groups are authorized to grant and revoke membership of users in the regular groups of figure 1, as we will see shortly. (For simplicity, URA97 assumes that control of membership in the administrative groups is centralized.)

The rest of the paper is organized as follows. In section 2, we discuss how to implement group hierarchies in Unix. In section 3 we review the URA97 model and discuss its implementation in Unix. Implementation details are described in section 4. Section 5 discusses the lessons learned from our experiment and describes how additional setgid based mechanisms could be implemented to make our implementation more scalable. Section 6 concludes the paper.

2 GROUP HIERARCHIES

As we have mentioned Unix does not have the notion of hierarchy between groups. We show how group hierarchies can be simulated in UNIX. The basic idea is that when a user is added to a senior group the assign program automatically adds the user to all junior groups. Similarly, when a user is removed from a senior group the revoke program automatically removes the user from appropriate junior roles.[5]

Each account in Unix has a symbolic name that identifies it. When adding a new user account to the system, the administrator assigns a user identification number (UID) which should be unique (Unix itself does not enforce uniqueness). Internally, the UID is the system's way of identifying an account. We assume that each account has a single human user associated with it. In practice a single user could have multiple accounts and a single account could be shared by multiple users. We assume the system administrator enforces a single user per account and a single account per user policy. Therefore we will use the terms user and account as essentially synonymous.

The administrator assigns each user to one or more groups. The file `/etc/group` lists all defined groups and is public information; all users may read it, but only the superuser is allowed to modify it. Group membership is given in `/etc/passwd` (for the primary group) or `/etc/group` (for secondary groups). In this paper we will focus on the `/etc/group` file. Extension to the `/etc/passwd` file is straightforward and would be tedious to describe.

[4] 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.

[5] Discussion of scalability issues related to this approach is given in section 5.

DIR::	47:	
PL1::	48:	Alice
PL2::	49:	
PE1::	50:	Alice
PE2::	51:	
QE1::	52:	Alice
QE2::	53:	
E1::	54:	Alice
E2::	55:	
ED::	56:	Alice
E::	57:	Alice, Dave, Eve

(a) `/etc/group`

DIR::	47:	
PL1::	48:	Alice
PL2::	49:	
PE1::	50:	
PE2::	51:	
QE1::	52:	
QE2::	53:	
E1::	54:	
E2::	55:	
ED::	56:	Alice
E::	57:	Alice, Dave, Eve

(b) `/etc/explicit`

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	-

(c) `/etc/grouphr`

Table 1: The example group hierarchy of Figure 1

Table 1(a) shows the file `/etc/group` corresponding to the list of groups in figure 1. Each row gives the group name, the group ID and a list of group members. For convenience we enumerate group members by symbolic name. Unix actually uses UIDs for this purpose. The `/etc/group` file shows the group membership of each user. 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(c). 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. In table 1(c), the first row has the null symbol because the group director(DIR) does not have any parent. The `/etc/grouphr` file can be easily constructed for any group hierarchy.

Using `/etc/grouphr`, we can find all seniors and juniors for a group by respectively chasing the parents and children. For example for the PE1 group of table 1(c) we can construct the seniors and juniors list as follows.

Group	Seniors	Juniors
PE1	PL1, DIR	E1, ED, E

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. A user can simultaneously be an explicit and implicit member of the same group.[6] For example, Alice can be an explicit member of ED and PL1, in which case she is also an implicit member of ED (by virtue of membership in PL1). To simulate a group hierarchy we maintain 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 keep information about explicit membership only. An example is shown in table 1(b). The format of this file is same as `/etc/group`. Alice has explicit memberships for PL1, ED and E. Alice also has implicit membership for all groups junior to PL1, i.e., PE1, QE1, E1, ED, and E, as shown in table 1(a). If Alice's explicit membership is revoked from E there will be no change in `/etc/group` but `/etc/explicit` will be changed to remove her from E. Suppose after that Alice is further revoked from PL1 we will have the following result.

DIR::	47:	
PL1::	48:	
PL2::	49:	
PE1::	50:	
PE2::	51:	
QE1::	52:	
QE2::	53:	
E1::	54:	
E2::	55:	
ED::	56:	Alice
E::	57:	Alice, Dave, Eve

`/etc/group`

DIR::	47:	
PL1::	48:	
PL2::	49:	
PE1::	50:	
PE2::	51:	
QE1::	52:	
QE2::	53:	
E1::	54:	
E2::	55:	
ED::	56:	Alice
E::	57:	Dave, Eve

`/etc/explicit`

In summary, to simulate group hierarchies in UNIX, we use `/etc/group`, `/etc/explicit`, and `/etc/grouphr` files. The `/etc/group` file shows all group membership including implicit and explicit group memberships. The `/etc/explicit` file just has information about explicit group membership and the `/etc/grouphr` file keeps the structure of the group hierarchy. Modifications to these files are made by setgid programs as discussed in section 4.

The above example illustrates why we need the `/etc/explicit` file in addition to `/etc/group`. The `/etc/grouphr` file is kept separate from `/etc/group` so that the structure of `/etc/group` does not need to be changed. Modifying the `/etc/group` file might break existing Unix system programs which use this file so it best not to tamper with it.

3 DECENTRALIZED GROUPS

Unix centralizes user-group assignment and revocation entirely in hands of the root account. However, this simple approach does not scale to large systems. Clearly it is desirable to decentralize user-group assignment to some degree so that expensive system administrators do not need to spend valuable time on routine tasks. In particular we can use administrative groups for this purpose. For convenience we define administrative groups as distinct from regular groups. (Of course, Unix does not make this distinction and it must be enforced by the system administrators.)

[6] This is a property of the RBAC96 and URA97 models on which our experiment is based. There are other models, such as [FB97, NO95] which do not permit this.

Sandhu and Bhamidipati [SB97] recently introduced the URA97 model for decentralized administration of user-role membership (URA97 stands for user-role assignment 1997). Since 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, we will adopt this model. This section reviews URA97 and describes our approach to implementing it in Unix. In our review of URA97 we will use the term group rather than role. Our description of URA97 is informal and intuitive. A formal statement of URA97 is given in [SB97].

3.1 User-Group Assignment

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`. An example of `/etc/can_assign` with prerequisite groups is given in table 2. We put a colon between the columns to indicate the boundary. The first row authorizes the administrative group PSO1 to assign users to groups in the range [E1,PL1]. A group range is specified by giving a junior and senior group. The range includes all groups between these two endpoints. The [and] brackets indicate that respectively the junior and senior end point is included in the range, whereas the (and) brackets indicate the end point is excluded. Thus [E1,PL1] consists of E1, PE1, QE1 and PL1, while [E1,PL1) omits PL1.[7] The prerequisite group specifies which users can be assigned by PSO1 to groups in the authorized range. Only those users who are already members of ED can be assigned by PSO1 to [E1,PL1]. The other rows of table 3 are similarly interpreted.[8]

Table 3 illustrates the more 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 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.

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.

3.2 User-Group Revocation

URA97 authorizes revocation by the *can_revoke* relation which we store in the `/etc/can_revoke` file. An example is shown in table 4. The meaning of each row in `/etc/can_revoke` is that a member of the administrative group can revoke membership of a user from any regular group in group range.

[7] The reader may recognize this as standard mathematical notation for open and closed intervals.

[8] It should be noted that administrative groups in URA97 are organized in a hierarchy such as shown in figure 2. As discussed in section 2, this hierarchy is implemented in `/etc/group` by members of a senior group are also added to all junior roles. Also note that the concept of a session in which only some groups of a user are activated is not supported by our Unix implementation. In Unix all groups of a user are activated every time the user logs into the system.

Administrative Group	Prerequisite Group	Group Range
PSO1:	ED :	[E1,PL1]:
PSO2:	ED:	[E2,PL2]:
DSO:	ED:	(ED,DIR]:
SSO:	E:	[ED,ED]:

Table 2: Example of `/etc/can_assign` with Prerequisite Groups

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]:

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

Administrative Group	Group Range
PSO1:	[E1,PL1):
PSO2:	[E2,PL2):
DSO:	(ED,DIR):
SSO:	[ED,DIR]:

Table 4: Example of `/etc/can_revoke`

We would typically expect some correlation between the range authorized for an administrative group in `/etc/can_assign` and in `/etc/can_revoke`, but this is not required by the model.

URA97 defines two notions of revocation called *weak* and *strong*. Weak revocation is straightforward and has impact only on explicit membership in the group in question. Strong revocation requires revocation of both explicit and implicit membership. Strong revocation of U's membership in x requires that U be removed not only from explicit membership in x, but also from explicit (or implicit) membership in all groups senior to x. Strong revocation therefore has a cascading effect upwards in the group hierarchy. In URA97 strong revocation is effectively equivalent to a series of weak revocations. Strong revocation is a convenient operation for administrators even though it can logically be accomplished by multiple weak revokes.

Let us consider the example shown in table 4 and interpret it in context of the hierarchies of figures 1 and 2. Let Bob be a member of PSO1, and let this be the only administrative group he has. Bob is authorized to revoke membership of users from groups E1, PE1 and QE1. Table 5(b) illustrates whether or not Bob can strongly revoke membership of a user from group E1 based on table 5(a). The effect of Bob's strong revocation of each of these users from E1 is shown in table 5(c). Bob is not allowed to strongly revoke Eve and Frank from E1 because they are members of senior groups outside the scope of Bob's revoking authority. If Bob was assigned to the DSO group he could strongly revoke Eve from E1 but still would not be able to strongly revoke Frank's membership in E1. In order to strongly revoke Frank from E1, Bob needs to be in the SSO group. The general rule is that strong revocation takes effect within the revocation range authorized for an administrative group.

URA97 further defines two options for strong revocation. The options are called *drop* and *continue*. In table 5(a), Bob is not allowed to strongly revoke Eve and Frank from E1 because they are members of senior groups outside the scope of Bob's revoking authority. At this step, we can choose one of two options. With the drop option strong revocation takes no effect. Otherwise we can strongly revoke a user from groups inside the scope of Bob's revoking authority. For example assume that we choose drop option for strong revocation of Eve from E1 and choose continue for strong revocation of Frank from E1. The result will be as shown in table 6.

The strong revocation of Eve from E1 takes no effect because we chose the drop option but the strong revocation of Frank from E1 takes partial effect. Frank still has group membership for PL1 and DIR groups outside the scope of Bob's revoking authority. We emphasize that the effect of strong revocation can be achieved by a series of weak revocations, but it is a convenient operation to have in both variations (drop and continue).

We use the setgid feature of Unix 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.

4 IMPLEMENTATION DETAILS

To implement URA97 in Unix we use several reference files introduced in the previous sections and set their permission bits as shown in table 7. 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` and `/etc/grouphr` in section 2, and `/etc/can_assign` and `/etc/can_revoke` in

/etc/group			/etc/explicit		
DIR::	47:	Frank	DIR::	47:	Frank
PL1::	48:	Frank, Eve	PL1::	48:	Frank, Eve
PL2::	49:		PL2::	49:	
PE1::	50:	Frank, Eve, Dave, Cathy	PE1::	50:	Frank, Eve, Dave, Cathy
PE2::	51:		PE2::	51:	
QE1::	52:	Frank, Eve, Dave	QE1::	52:	Frank, Eve, Dave
QE2::	53:		QE2::	53:	
E1::	54:	Frank, Eve, Dave, Cathy	E1::	54:	Frank, Eve, Dave, Cathy
E2::	55:		E2::	55:	
ED::	56:		ED::	56:	
E::	57:		E:	57:	

(a) `/etc/group` and `/etc/explicit` prior to strong revocation

User	E1	PE1	QE1	PL1	DIR	Bob can revoke user from E1
Cathy	Yes	Yes	No	No	No	Yes
Dave	Yes	Yes	Yes	No	No	Yes
Eve	Yes	Yes	Yes	Yes	No	No
Frank	Yes	Yes	Yes	Yes	Yes	No

(b)Status prior to strong revocation

/etc/group			/etc/explicit		
DIR::	47:	Frank	DIR::	47:	Frank
PL1::	48:	Frank, Eve	PL1::	48:	Frank, Eve
PL2::	49:		PL2::	49:	
PE1::	50:	Frank, Eve	PE1::	50:	Frank, Eve
PE2::	51:		PE2::	51:	
QE1::	52:	Frank, Eve	QE1::	52:	Frank, Eve
QE2::	53:		QE2::	53:	
E1::	54:	Frank, Eve	E1::	54:	Frank, Eve
E2::	55:		E2::	55:	
ED::	56:		ED::	56:	
E::	57:		E:	57:	

(c) `/etc/group` and `/etc/explicit` after strong revocation

Table 5: Example of Strong Revocation

/etc/group			/etc/explicit		
DIR::	47:	Frank	DIR::	47:	Frank
PL1::	48:	Frank,Eve	PL1::	48:	Frank,Eve
PL2::	49:		PL2::	49:	
PE1::	50:	Eve	PE1::	50:	Eve
PE2::	51:		PE2::	51:	
QE1::	52:	Eve	QE1::	52:	Eve
QE2::	53:		QE2::	53:	
E1::	54:	Eve	E1::	54:	Eve
E2::	55:		E2::	55:	
ED::	56:		ED::	56:	
E::	57:		E:	57:	

Table 6:

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`

Table 7: The permission of reference files

section 3. For simplicity all these files in our implementation are owned by root. We assume that the `rbac` group has no members.

There is one procedure each for assigning a user to a group, doing a weak revoke of membership and doing a strong revoke of membership. In our implementation a user invokes the procedure call to grant or revoke a group from or to another user. The procedure calls are as follows.

- assign(user, tgroup)
- weak_revoke(user, tgroup)
- strong_revoke(user, tgroup)

The parameters user and tgroup (target group) specify which user is to be assigned to tgroup, or to be weakly or strongly revoked from tgroup. If the strong_revoke operation fails because it is not authorized by `/etc/can_revoke` the user will be asked for a choice of the drop or continue options as discussed earlier.[9]

[9] In a more robust implementation we would like this option to be designated as part of the command line call to strong_revoke, so that it might be called from within programs and shell scripts without requiring user intervention.

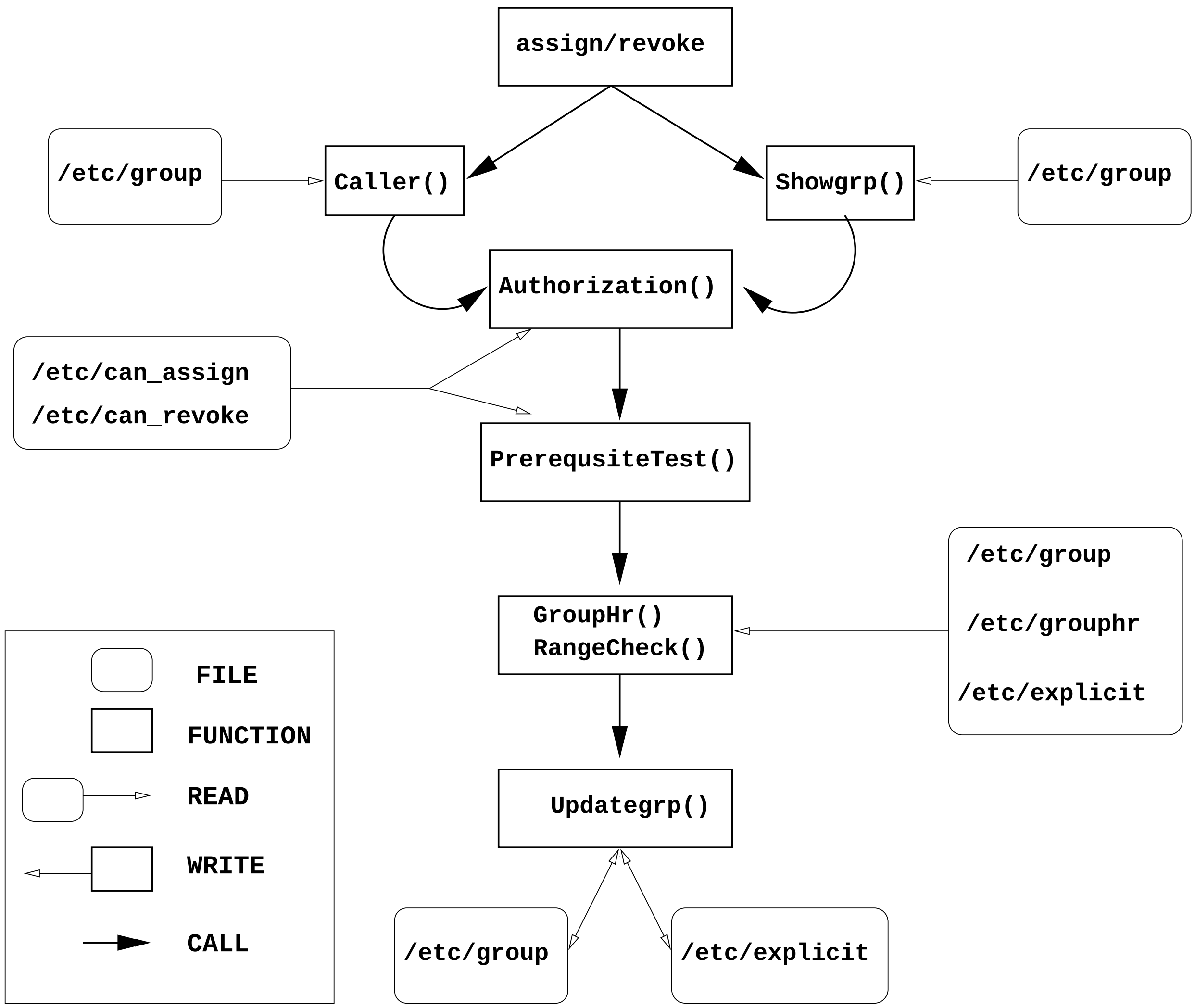


Figure 3: Data Flow Diagram

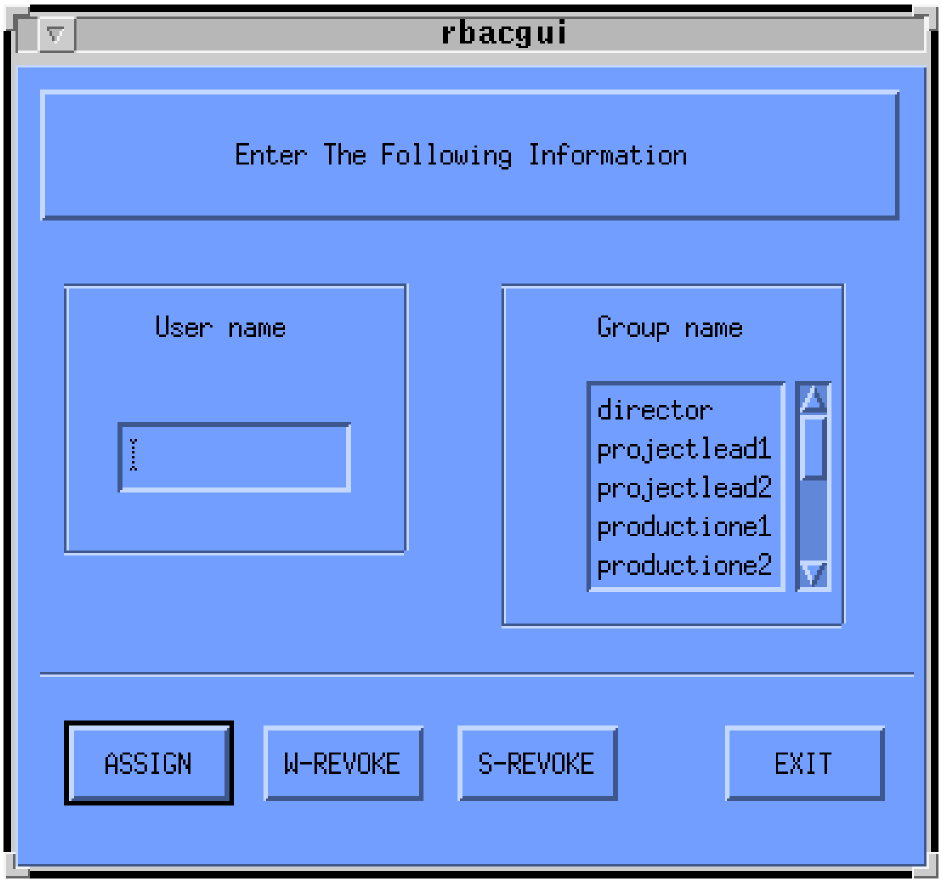


Figure 4: User Interface:RBACGUI

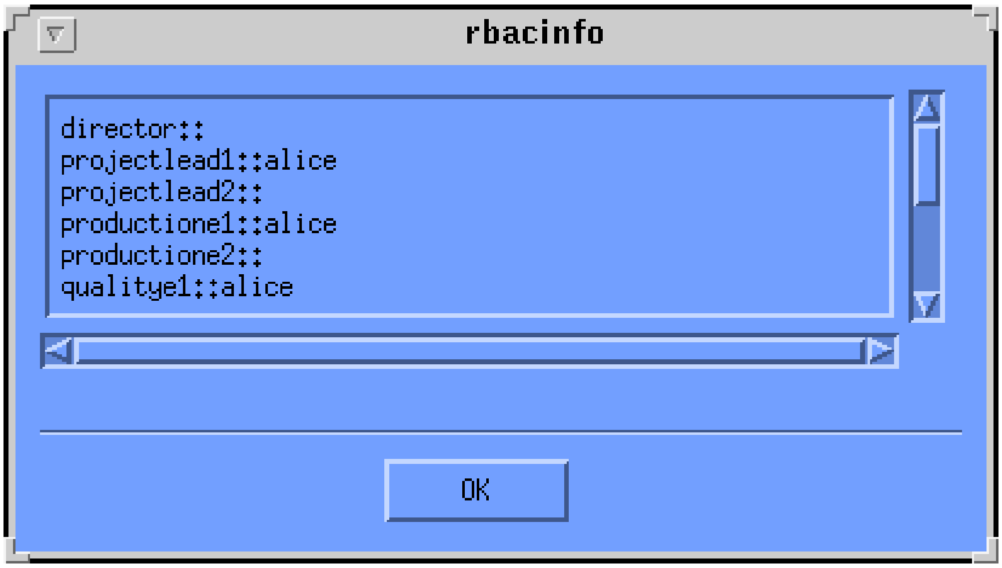


Figure 5: User Interface:RBACINFO

All three procedures follow the basic steps shown in figure 3. The diagram shows the data flow and the relationship between functions and files. Each procedure call include several functions. The description for each function is as follows.

- `caller()`:
  returns a list of all groups to which the user belongs (explicitly or implicitly)
- `showgrp()`:
  returns a list of all administrative groups to which the invoker belongs (explicitly or implicitly)
- `authorization()`:
  checks the invoker's authorization with respect to `/etc/can_assign` or `/etc/can_revoke`
- `PrerequsiteTest()`:
  checks whether the user satisfies the prerequisite condition
- `GroupHr()`:
  return the senior and junior list for tgroup
- `RangeCheck()`:
  checks if strong_revoke is authorized and offers option of drop or continue
- `Updategrp()`:
  updates `/etc/group` and `/etc/explicit` files as appropriate

In general authorization needs to be tested for multiple rows in `/etc/can_assign` or `/etc/can_revoke`. In such cases the authorization and PrerequisiteTest procedures are called repeatedly for each row.

These procedures are called at the Unix command line prompt as follows.

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

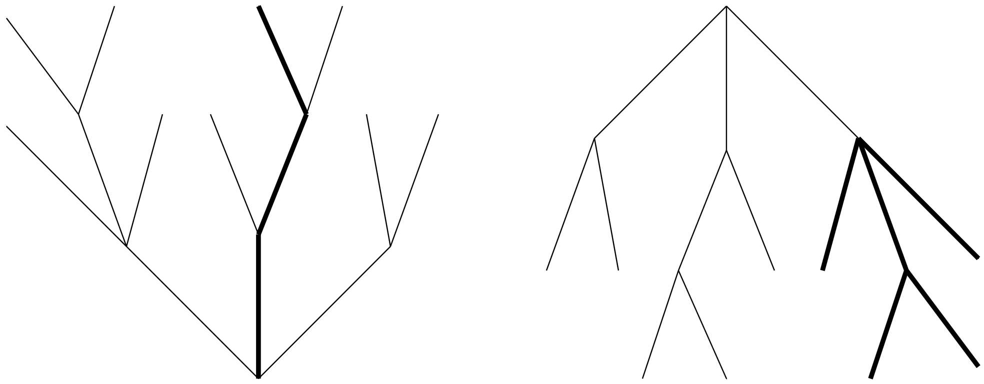
In order to make our implementation more convenient we developed two graphical user interfaces (GUIs) which interact with these procedures to do user-group assignment and revocation. The graphical user interfaces are illustrated in figure 4 and 5 and are called RBACGUI and RBACINFO respectively. They were developed using Motif programming. RBACGUI is used to initiate user-group assignment and revocation instead of typing the above as command line procedure calls. There are three buttons for doing user-group assignment and revocation; ASSIGN, W-REVOKE (weak revoke), and S-REVOKE (strong revoke). RBACINFO allows the invoker to consult the reference files frequently. Figure 5 shows the information of `/etc/group`. This window will be opened after typing `rbacinfo group` at the Unix command line prompt.

This implementation is convenient for administrative groups since they only need to define the group hierarchy and the relations *can_assign* and *can_revoke*.

5 LESSONS LEARNED

Our experiment indicates that the extensibility provided in Unix be means of setgid programs does enable implementation of the two extensions described here, without changing the Unix kernel or any of its system programs.

The implementation of the URA97 model is quite faithful and does not present significant limitations. The implementation of group hierarchy by explicitly making a member of a senior group to be



(a) Inverted Tree Hierarchy (b) Tree Hierarchy

Figure 6:

a member of all junior groups in `/etc/group` does raise a scalability issue. Many implementations of Unix limit the number of groups activated in a process to a fairly small number such as 32 or 16. The NFS system only accommodates 16 groups so we can take that as a benchmark number due to the popularity of NFS.

Consider the scalability issue in the hierarchy of figure 1. For the leader of a single project we are reasonably safe. However, even with just two projects the DIR group has 10 juniors. If we have hundreds of projects the DIR group cannot be handled by this technique. True scalability can only be achieved if the Unix kernel is modified to directly support hierarchical groups. However, there are still some useful alternatives available without kernel modification.

The RBAC96 model [SCFY96] has the notion of a session which we have not mentioned so far. In a session a user can activate a subset of the groups to which they belong. In other words not all groups are activated all the time. The motivation for this stems from the least privilege principle. Thus if Bob is a project leader for multiple projects, say 1, 2 and 3, he need not activate all his project lead groups simultaneously. Instead he can activate PL1, PL2 and PL3 in separate Unix shells (probably on separate windows on a workstation). So we can accommodate scalability to some extent by allowing a choice of which project lead groups are activated. In our context we may limit this to only one project lead group at a time.

This idea of working on a single project in a window can work up to a point. Consider Alice who is a member of the DIR group. If there are hundreds of projects in her department she can never really activate the DIR role. Instead she can activate project lead roles for individual projects as needed.

This leads us to the following approach for accommodating scalability beyond that indicated above. We need to depart from RBAC96 and recognize due to the 16 group at a time limitation of Unix-NFS we must interpret membership in a senior group as the ability to activate a junior group whenever. In other words in our modified model activation of a senior group does not automatically lead to activation of junior groups. The activation is done only when requested. Thus Alice can log on with the DIR group activated and then decide which groups junior to DIR to activate in a given window. It may be possible to do this by means of setgid programs.

To provide another perspective on the scalability issue consider the inverted tree and tree hierarchies of figure 6. The inverted tree does not present a problem since activation of a senior group amounts to activation of a single path (such as shown in bold-face in the figure) from leaf to root. The inverted tree is a commonly occurring hierarchy. For example, Novell Netware supports it [SGE94]. In object-oriented terms it corresponds to a generalization-specialization hierarchy with no multiple inheritance (although the root is shown at the top versus our convention of showing it at the bottom). For the tree hierarchy, however, we have the scalability problem because a senior is the root of a sub-tree of junior groups which be sizable in number due to the fan-out at each group.

To summarize, scalability of our approach can be handled in Unix by introducing the notion of a session and redefining the meaning of the group hierarchy of RBAC96 as indicated above. It may be possible to do this by means of setgid programs.

6 CONCLUSION

In this paper we have described our experiment to provide two useful extensions to the Unix group mechanism by means of setgid programs. First we have added hierarchical groups by means of explicit assignment to junior groups. When a user is assigned to a senior group the system *automatically* adds the user to all junior groups. Similarly, when a user's membership is revoked from a group, revocation from appropriate junior groups is *automatically* carried out. This behavior is adapted from the RBAC96 model. Secondly we have adapted the URA97 model for decentralized user-group assignment and implemented it in Unix. Our implementations use setgid programs to enforce authorization to add and remove users from groups. Our results indicate that Unix has adequate flexibility to accommodate modern access control models to some extent. We also indicated how additional setgid based mechanisms could be utilized to make our implementation more scalable. The implementations are available in the public domain.

Acknowledgement

This work is partially supported by the National Science Foundation at the Laboratory for Information Security Technology at George Mason University.

References

[BSS+95a] Lee Badger, Daniel Sterne, David Sherman, Kenneth Walker, and Sheila Haghighat. Practical domain and type enforcement for Unix. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 66–77, Oakland, CA, May 1995.

[BSS+95b] Lee Badger, Daniel Sterne, David Sherman, Kenneth Walker, and Sheila Haghighat. A domain and type enforcement UNIX prototype. In *Proceedings of USENIX Unix Security Symposium*, 1995.

[FB97] David Ferraiolo and John Barkley. Specifying and managing role-based access control within a corporate intranet. In *Proceedings of 2nd ACM Workshop on Role-Based Access Control*, pages 77–82. ACM, Fairfax, VA, November 6-7 1997.

[FH97] Christian Friberg and Achim Held. Support for discretionay role-based access control in acl-oriented operating systems. In *Proceedings of 2nd ACM Workshop on Role-Based Access Control*, pages 83–94. ACM, Fairfax, VA, November 6-7 1997.

[FWF95] Eduardo B. Fernandez, Jie Wu, and Minjie H. Fernandez. User group structures in object-oriented database authorization. In J. Biskup, M. Morgernstern, and C. Landwehr, editors, *Database Security VIII: Status and Prospects*. North-Holland, 1995.

[GS96] Simon Garfinkel and Eugene Spafford. *Practical Unix and Internet Security (2nd edition)*. O'Reilly & Associates, Inc., 1996.

[HDT95] M.-Y. Hu, S.A. Demurjian, and T.C. Ting. User-role based security in the ADAM object-oriented design and analyses environment. In J. Biskup, M. Morgernstern, and C. Landwehr, editors, *Database Security VIII: Status and Prospects*. North-Holland, 1995.

[Mey97] William Meyers. RBAC emulation on trusted DG/UX. In *Proceedings of 2nd ACM Workshop on Role-Based Access Control*, pages 55–60. ACM, Fairfax, VA, November 6-7 1997.

[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.

[RBKW91] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1), 1991.

[San88] Ravi Sandhu. The NTree: A two dimension partial order for protection groups. *ACM Transactions on Computer Systems*, 6(2):197–222, May 1988.

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

[SGE94] Marvin Schaefer, Gary R. Grossman, and Jeremy J. Epstein. Using a semiformal security policy model: 2C a C2 better. In *Proceedings of 17th NIST-NCSC National Computer Security Conference*, pages 153–164, Baltimore, MD, October 11-14 1994.

[STCYYS94] Barbara Smith-Thomas, Wang Chao-Yeuh, and Wu Yung-Sheng. Implementing role based, Clark-Wilson enforcement rules in a B1 on-line transaction processing system. In *Proceedings of 17th NIST-NCSC National Computer Security Conference*, pages 56–65, 1994.