Push Architectures for User Role Assignment

Venkata Bhamidipati and Ravi Sandhu

Laboratory for Information Security Technology
Information and Software Engineering Department
George Mason University
{vbhamidi, sandhu}@gmu.edu

Abstract The basic concept of role-based access control (RBAC) is that permissions are associated with roles and users are made members of appropriate roles thereby acquiring the roles' permissions. Using RBAC to manage RBAC provides additional convenience. The administration of RBAC can be divided into three main categories namely, user-role assignment, permission-role assignment and role-role assignment. The administration of RBAC in distributed systems presents additional challenges relative to centralized systems. The central contribution of this paper is to identify some architectures for RBAC administration in distributed systems, and to present a push-based architecture for user-role assignment. We classify the architectures based on event notification, system policies, system capabilities and role classification.

1 Introduction

Role-based access control (RBAC) has recently received considerable attention as a promising alternative to traditional discretionary and mandatory access controls (see, for example, [FK92, FCK95, Gui95, GI96, MD94, HDT95, NO95, SCFY96, vSvdM94]). 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 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. Role-role relationships can be established to lay out broad policy objectives.

In large enterprise-wide systems the number of roles can be in the hundreds or thousands, and users can be in the tens or hundreds of thousands, maybe even millions. Managing these roles and users, and their interrelationships is a formidable task. A promising advantage of RBAC is it can be used to manage itself. Sandhu et al [SBM99] recently introduced a comprehensive model for role-based administration of RBAC called ARBAC97 (administrative RBAC '97).

Administration of roles in a distributed environment is more intricate when compared to a centralized environment. There are many factors which contribute to the complex nature of role administration in a distributed environment. For example all the systems may not support role hierarchies. Also, every role may not be present on every server. In this paper we identify some possible architectures for distributed RBAC. This is not a complete analysis but rather the beginning of a framework to approach practical issues in this arena. In particular we present a push architecture concerning how users are assigned to roles in a distributed environment. We assume that all the necessary authorizations that are required to perform the operation of assigning and revoking users are done by a separate model, such as the URA97 component of ARBAC97 [SBM99] or some other suitable model. Our architecture is independent of the administrative authorization model.

We emphasize that the focus of this paper is on architectures for role administration. There are numerous distributed systems issues that need to be addressed in an actual implementation but their discussion is outside the scope of this paper. We assume that a core of distributed services will be available and do not consider their detailed implementation. The central contribution of this paper is to identify some architectures possible for RBAC in distributed systems and to present a push-based architecture for user-role assignment. We classify the architectures based on event notification, system policies, system capabilities and role classification. It is not our goal to discuss all possible architectures but rather to take a first step in identifying interesting alternatives.

The rest of the paper is organized as follows. In section 2 we outline architectures possible for distributed RBAC. In section 3 we propose a push-based architecture for user-role assignment. Section 4 concludes the paper.

2 Architectures for distributed RBAC

Broadly speaking a distributed system consists of data, software and users spread across several sites connected by some form of network. We can have several possible architectures depending on how the resources, data and users are distributed and how they interact. We have identified certain architectures and we will discuss them in this section. We reiterate that these architectures are only a subset of all the possibilities. An exhaustive analysis is beyond the scope of this paper.

The administration of RBAC has essentially three components: user-role assignment (UA), permission-role assignment (PA) and role hierarchy (RH). Generally user-role assignment is a logically centralized service since the same enterprise role may occur on multiple servers. Thus assigning Alice to the Engineer role should give her access to all servers where the Engineer role is authorized. This should be transparent to the administrator who assigns Alice to the Engineer role. On the other hand permission-role assignment is typically done independently at each local system. For example, the human-resources server and the marketing-department server can independently determine what privileges are available to the Engineer role.

In this paper we treat the role hierarchy as a centralized service. A role hierarchy is mathematically a partial order [SCFY96]. Senior roles inherit permissions from junior roles. Treatment of the role hierarchy as centralized imposes the same seniority relationship at each server. Thus if the Manager role is senior to Worker that relationship will hold on all servers in the system. Clearly there are cases where this will not apply. For example, Manager may be senior to Worker in most servers but there may be some servers where Workers are senior (say those servers run by the Workers union). For simplicity we limit our discussion here to the case where the role hierarchy is uniform at all servers. We do allow individual servers to introduce additional role-role relationships between roles that are incomparable in the original hierarchy, provided the net remains acyclic.

2.1 Architectures from UA Viewpoint

Next we discuss the various architectures that are possible from UA point of view. Event notification and information can be sent to distributed servers in various ways. In general these can be placed in four categories namely push model, pull model, lookup model and credential verification model.

- Push Architecture: In push architecture the UA information is pushed to all the systems notifying the events. An example of push architecture is active channels on the Internet. They push the information to client sites.
- Pull Architecture: In pull architecture every system contacts the central repository. Each system maintains a cache where it stores this information for a certain period of time. The cache can be refreshed based on system requirements. An example of Pull Architecture is Web Browsers. The browsers keep a disk and memory cache and depending up the settings of the browser they contact the URL for the information or fetch it from their local cache.

Global Hierarchy	Restricted Roles	Classification
Non-empty	all roles	Homogeneous
Non-empty	some roles	Heterogeneous
Non-empty	none	Heterogeneous
Empty	none	Heterogeneous (federated architecture)

Table 1: Homogeneous versus Heterogeneous Role Hierarchy

- Lookup Architecture: Lookup architecture is a special case of pull architecture which does not maintain any cache. A server contacts the central system for information as needed.
- Credential based Architecture: In credential based architecture the server is presented with credentials in a verifiable form. Some of the examples are Kerberos tickets, X.509 Certificates, secure cookies, etc.

In this paper we take a detailed look at push architectures for user-role assignment. The reason for this is twofold. Legacy servers are built to stand-alone, so each maintains its own user account and user-role databases. To integrate legacy servers into distributed systems a push architecture is a natural choice. The importance of legacy systems cannot be overestimated as the Y2K problem has demonstrated. We postulate that even in the future servers which maintain their own user-role database will be required. Such servers can function even if the network has failed or is congested, so they will be deployed where justified. Thus overall push architectures are of great practical interest.

2.2 Architectures from RH Viewpoint

In this classification the architectures are distinguished by degree of homogeneity of the RBAC policies of the distributed servers. If all servers use identical role hierarchy definitions we call it a homogeneous architecture otherwise we call it a heterogeneous architecture. Another factor we consider in the classification is degree of local autonomy. Local autonomy allows servers to introduce relationship between global roles without violating any enterprise rules, that is without introducing relationship among incomparable roles that are designated as restricted. In a homogeneous architecture there is no local autonomy. If we allow complete autonomy without any restrictions and rules then every local system will have its own hierarchy definition and this system can be classified as a federated architecture. The classifications are listed in table 1.

- Homogeneous Architecture: In a homogeneous Architecture all the systems have the same hierarchy definition, for example if role x is senior to role y then it is true in all the systems. The homogeneity of the role-role relationship is maintained across all the systems. This architecture allows us to maintain a single global hierarchy. Each of the the local system may contain complete or partial set of this global hierarchy and local systems can not introduce additional role-role relationships.
- Heterogeneous Architecture: In a heterogeneous architecture the local systems can introduce relationships between roles if they are not restricted. There will be a part of global hierarchy that can not be modified and the local systems can introduce relationships between the roles which do not fall into the restricted class.
- Federated architecture: In federated architecture the role-role relationship is not same across all the systems i.e. if role x is senior to role y in one system it may not be the case in another system. There is no single global hierarchy in this kind of architecture.

Users	Roles
On all systems	On all systems
On some systems	On all systems
On all systems	On some systems
On some systems	On some systems

Table 2: User role occurrence

The push architecture for user-role assignment developed in this paper can apply to any of these architectures. Users are assigned to global roles consistent with the global role hierarchy. Extensions to the global hierarchy must be taken into account at each individual server and does not directly impact user assignment to global roles.

2.3 Architectures from System Capability Viewpoint

This classification is based on the capabilities of the system. Some systems offer support for role hierarchies whereas others do not provide such capability. If a system does not support role hierarchies, we can simulate a role hierarchy by explicit assignment of a user to all junior roles when that user is assigned to a senior role. Otherwise we can rely on the role hierarchy and limit our assignment to just the senior role. In this paper we provide user assignment algorithms for both cases.

2.4 Architectures from User and Role Occurrence Viewpoint

There can be four cases of user role occurrence on end systems as listed in table 2. The cases where all the users are present or all the roles are present are not desirable. Having all the users on all systems violates the least privilege principle. If these users have no access to resources on that system they should not have an account on the system that enables them to login. Similarly having all roles in all the systems is not appropriate as we do not wish to have roles that are not relevant to a server. For example, an Engineer role may not be needed in a sales database. In this paper we assume the case that every system does not have all the roles and all the users. By addressing this general case we clearly cover all the other cases also.

3 Push Architecture for User Role assignment

In this section we describe the user-role assignment model based on push architecture. User-role assignment is done globally and the local administrators at the local systems do the permission-role assignment. As user-role assignment and revocation is performed this information is pushed to individual servers. The assumption is that each server has its own database of users, roles and user-role assignments which it uses to enforce RBAC. Since most existing systems provide this capability a push architecture is more suited to them. As future systems get developed they may move to pull, lookup or credential based architectures. Nevertheless push architectures will be needed for some time to come, if only to accommodate legacy systems.

We assume that system provides underlying infrastructure needed for distributed processing. We present some of the features we believe should be supported by the underlying system.

- Single Sign On: The system should have single sign on capability for ease of user authentication and access.

- Network security and data encryption: The system should provide network security and data encryption to ensure confidentiality and integrity of messages and data passing over the network.
- Two Phase Commit: In order to achieve transaction semantics the system should be capable of providing two phase commit to ensure consistency.
- Location Transparency: The system should provide location transparency to end user.
- Global name resolution: There should be a name resolution service to map the names of roles and users across the systems in a consistent manner.

We have only identified the most important distributed system services here. As mentioned earlier our focus is on authorization issues and a complete discussion of distributed system issues per se is outside the scope of this work.

3.1 The Push model

In our model we have a central service called the Role Authorization Center (RAC). The administrators of RAC perform user-role assignment. We define four relations which encode the relationship between the users, roles and the databases in the distributed system. These relations are maintained by the RAC and are shown in table 3. The *user_role* relation maintains all the relationship between users and roles and it consists of user name, the database name, the *actual_role* and the *assigned_role*. The *database_role* relation maintains information about roles that are present in each system. *The role_role* relation maintains the role hierarchy relationship among the roles. The *database_user* relation maintains information about users that are present on each system.

The model deals with two operations, assignment of users to roles and revocation of roles from users. We will be discussing the algorithms used to perform these tasks below. Figure 1 shows an example global hierarchy and figure 2 gives the hierarchies in the local systems. These are exactly the same as the global one except they are limited to roles present on each local system.

3.1.1 Assigning users to roles

When a user is assigned to a role we have to execute an assignment algorithm for each of the database servers where the role is present. We have two assignment algorithms (see figure 3). The assignment algorithm 1 assigns the role to a user if the role is present in the local system. If the role is not present in the local system then we walk down the global hierarchy from the initial role as the starting point and build a list of all junior roles. After the list is built we take the senior-most incomparable roles in the list that are present in the local hierarchy and assign them to the user. In algorithm 2 when a role is assigned, we walk down the global hierarchy and build a list of all roles junior to the role. Then we assign the roles which are present in the local systems to users. This results in redundant assignment which can be useful if the end system does not support a hierarchy.

We will exemplify how the algorithms work with a small example. Let say we want to assign role PL1 (Project Lead1) to Bob. The Finance department and service department don't have PL1 role or any roles that are junior to PL1, so no action is performed on these systems. We see that Personnel department has EMP role that is junior to PL1 and Engg department has roles PE1, QE1, Eng1, and ED that are junior to PL1. If we use Assignment algorithm 1 we will find that the role that needs to be assigned to Bob in Engg Dept is only Eng1. If we use algorithm 2 the roles that need to be assigned to Bob are Eng1 and ED. In the case of Personnel department Bob will be assigned EMP role by both algorithms. Whether we use algorithm 1 or algorithm 2 the permissions that are available to Bob are the same. If we use algorithm 1 Bob gets only Eng1 role in Engg department but because of role inheritance he will inherit the permissions of ED role.

User	Database	actual_role	assigned_role
John Doe	Finance	CEO	CEO
John Doe	Personnel	CEO	CEO
John Doe	Services	CEO	CEO
John Doe	Engg Dept	CEO	CEO
Joe Black	Finance	CFO	CFO
Joe Black	Personnel	CFO	EMP
Alice	Engg Dept	Dir	Dir
Alice	Engg Dept	Dir	EMP
...	...	...	...

(a) User_ Role relation

Database	Role
Finance	CFO
Finance	Acct1
Personnel	HR Dir
Engg Dept	Eng1
...	...

(b) Database_Role Relation

Grantor_Role	Grantee_Role
CEO	CFO
CEO	VP
CIO	Eng Dir
Sales Dir	Sales
Ed	EMP
...	...

(c) Role_Role Relation

Database	User
Engg Dept	John Doe
Engg Dept	Alice
Finance	John Doe
Finance	Joe Black
Personnel	Joe Black
...	...

(d) Database_User Relation

Table 3: Relations for Push Model

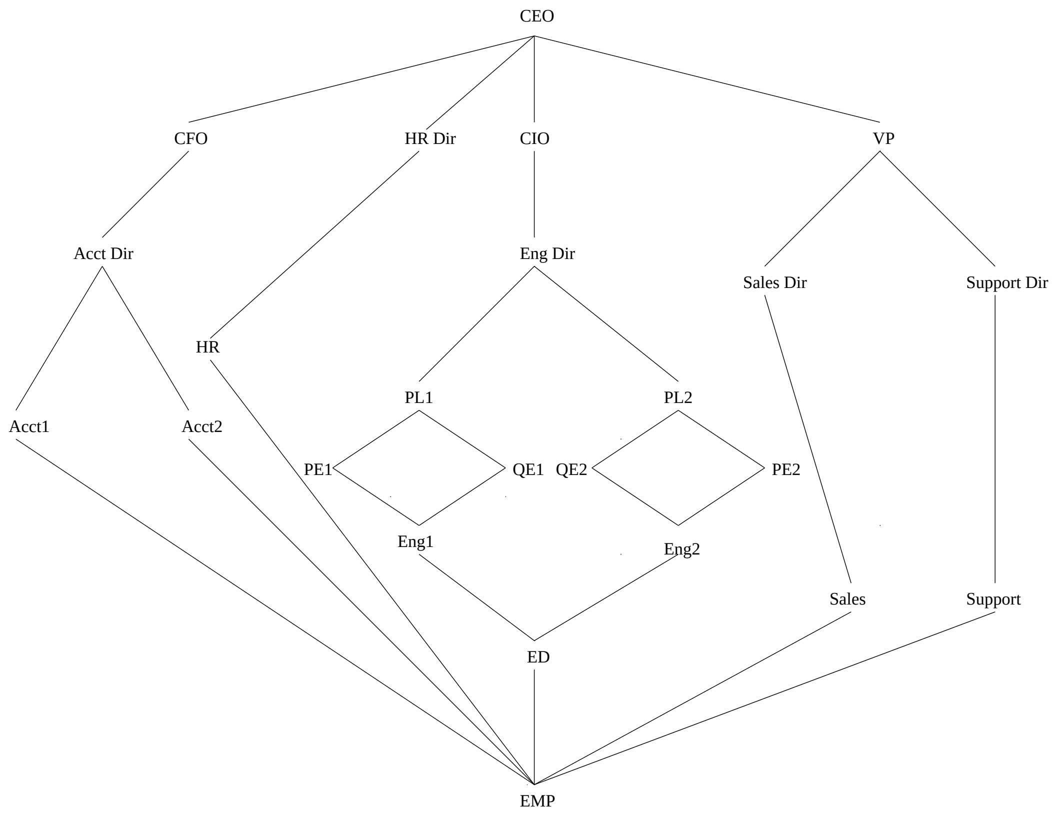

Figure 1: Global Hierarchy

Once the appropriate assign algorithm is executed and the role list is built we need to do following operations.

- If the user who is being assigned role(s) in a particular system does not exist in a local server then the user account needs to be created.
- The user role assignment should be performed at the local database servers.
- The relations in the RAC should be updated to reflect the changes.

Next we prove that algorithm 1 correctly selects the seniormost incomparable roles to be assigned to the user at each local database server.

Theorem 1 *Assign Algorithm 1 is correct.*

Proof: In order to assign only the senior most incomparable roles the while loop in the assign algorithm 1 has to maintain the following invariants.

Invariant 1 $r \in \text{candidate\_set} \Rightarrow \text{actual\_role} \geq r$

Invariant 2 $r \in \text{assign\_roleset} \Rightarrow r \in \mathit{Database} \wedge \text{actual\_role} \geq r$

Invariant 3 $r_1, r_2 \in \text{assign\_roleset} \Rightarrow r_1 \neq r_2$

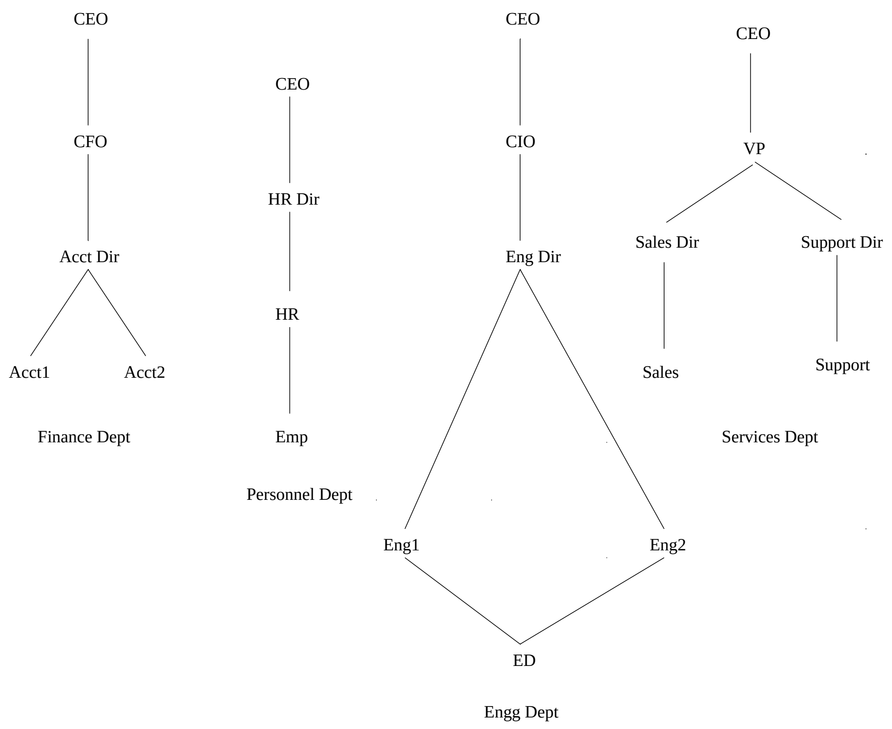

Figure 2: local Hierarchies

Invariant 4 $r \in$ assign_roleset $\Rightarrow \neg$ $(\exists\, r^{'} \in DB)$ actual_role $> r^{'} > r$

Invariant 5 $r_1,\, r_2 \in$ candidate_set $\Rightarrow r_1 \neq r_2$

The proof of these invariants is by induction on the number of iterations of the while loop in algorithm 1. For the basis case consider the first iteration of the algorithm. The conditional else statement is satisfied and all the incomparable immediate children of the *actual_role* are placed in the *candidate_set*. Since the roles that are placed in the *candidate_set* are children of *actual_role* and incomparable, invariants 1 and 5 are satisfied. Other invariants are trivially satisfied as the *assign_roleset* is empty. Assume the invariants are true after K iterations and consider K+1 iteration. Invariants 1 and 5 will be true as the roles present *candidate_set* after the Kth iteration were immediate children of *actual_role* and incomparable, so any roles that are present in the *candidate_set* are children of the *actual_role* and incomparable. The invariant 2 will hold true as the roles in the *candidate_set* are children of *actual_role* and because of conditional if statement in the algorithm. Since the roles in the *candidate_set* are incomparable the invariants 3 and 4 will be true.

Theorem 2 *Assign Algorithm 1 will terminate*

Proof: It is easy to see that the algorithm terminates when the *candidate_set* becomes empty. Consider the case when the *actual_role* is present. The algorithm will terminate after one iteration. Let us consider a case when the *actual_role* is not present. We can see that during every iteration of the loop the candidate set is decremented by one role. When ever the conditional else is satisfied the candidate set can increase by n number of roles. Since the role hierarchy is acyclic and number of roles is finite, due to invariant 5 the addition to *candidate_set* will stop after all the roles that are junior to the actual role are visited. After this point every iteration of loop will result in decrement of *candidate_set* by one role and eventually the *candidate_set* will become empty and the algorithm will terminate.

3.1.2 Revoking Roles from users

The Revoke operation does the revocation of a role from a user. The roles that need to be revoked from a user are determined by executing revoke algorithm. In revoke algorithm we search the *user_role* relation and build a list of all the tuples that contain the user and have the same *actual_role* value as the role that is being revoked. Once the list is built we delete the tuples from the *user_relation* that matches the tuples in the list. For every *assigned_role* in the list we see if the role was assigned to user by some other role assignment, if it is true then we delete those tuple(s) from the list. Once the algorithm is executed the following operations need to be performed.

- Databases in the revoke list should be notified of the roles that need to be revoked from the user.
- Each local system should perform the revocation of roles from users. After the revocation if the user doesn't have any roles in the local system then he should be dropped from the local system.
- The relations in the RAC should be updated to reflect the changes.

The revoke algorithm is independent of the assign algorithm that was used to assign the role to user. Each assign algorithm does the appropriate bookkeeping in the *user_relation.* So we need only one revoke algorithm.

4 Conclusion

In this paper we outlined some architectures possible in a distributed environment for RBAC administration and we developed a push-based model for user-role assignment. Our model is generic and it can accommodate different kinds of assignment structures. An authorization model can sit on top of our model to perform checks on the assignment authorization (for example, a model like URA97). We do not impose restrictions on the authorization model that needs to be used. A possible extension of the current work is to introduce a centralized control of permission assignment using packaged abilities. Each administrator of the local system can package the permissions into abilities(permission only roles) and export them to the RAC. The administrators of the RAC are responsible for assigning these abilities to appropriate roles by means of a push architecture. Finally we emphasize that future systems may move towards a pull, lookup or credential-based architecture but the need for push architectures will remain due to the large number of deployed legacy systems which inherently require a push architecture.

References

[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.

[FK92] David Ferraiolo and Richard Kuhn. Role-based access controls. In *Proceedings of 15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, MD, October 13-16 1992.

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

[Gui95] Luigi Guiri. A new model for role-based access control. In *Proceedings of 11th Annual Computer Security Application Conference*, pages 249–255, New Orleans, LA, December 11-15 1995.

[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.

[MD94] Imtiaz Mohammed and David M. Dilts. Design for dynamic user-role-based security. *Computers & Security*, 13(8):661–671, 1994.

[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.

[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.

[SBM99] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security*, 2(1), February 1999.

[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.

[vSvdM94] S. H. von Solms and Isak van der Merwe. The management of computer security profiles using a role-oriented approach. *Computers & Security*, 13(8):673–680, 1994.

```
Procedure Mark(role)
Take the role as the starting point and walk down the hierarchy. mark all nodes visited during the walk
Procedure Unmark(role)
Take the role as the starting point walk down the hierarchy and unmark all nodes visited during the walk

Assign Algorithm1
$actual\_role$ $\leftarrow$ Role to be assigned
User $\leftarrow$ User to which role is being assigned
DB $\leftarrow$ Database
$assign\_roleset \leftarrow \emptyset$
$candidate\_set \leftarrow$ {actual_role}
Unmark($actual\_role$)
while $candidate\_set \neq \emptyset$ do
    r $\leftarrow$ get a member from candidate_set
    $candidate\_set \leftarrow candidate\_set$ - {r}
    if r exists in DB
    then
      $assign\_roleset \leftarrow assign\_roleset \cup$ {r}
      Mark{r}
    else
      $R^{'}$ $\leftarrow$ Immediate Children of r
      $R^{''}$ $\leftarrow$ Unmarked members of $R^{'}$
      $R^{'''}$ $\leftarrow$ { r $\in R^{''}$ | no senior of r is in $candidate\_set$}
      $candidate\_set \leftarrow candidate\_set \cup R^{'''}$
end
do the assignment of roles in the $assign\_roleset$

Assign Algorithm2
BEGIN
$actual\_role$ $\leftarrow$ Role to be assigned
User $\leftarrow$ User to which role is being assigned
DB $\leftarrow$ Database
Mark($actual\_role$)
$assign\_roleset$ $\leftarrow$ all the nodes marked visited
END
do the assignment of the roles in the $assign\_roleset$
```

Figure 3: Assign Algorithms

```
Revoke Algorithm

BEGIN
$i \leftarrow$ number of tuples in $user\_role$ relation
r $\leftarrow$ role to be revoked
u $\leftarrow$ user from which the role is being revoked
while $i \neq 0$
do
        $i \leftarrow i$ - 1
        if $user\_role$(i){$actual\_role$, user} = {r,u}
        then
           $revoke\_set \leftarrow user\_role$(i)
           $user\_role \leftarrow user\_role$ - $user\_role$(i)
end
$j \leftarrow$ number of tuples in $revoke\_set$
while $j \neq 0$
do
        $j \leftarrow j$ - 1
        $k \leftarrow$ number of tuples in $user\_role$ relation
        while $k \neq 0$
        do
           $k \leftarrow k$ - 1
           if $revoke\_set$(j){$assigned\_role$, database, user} = $user\_role$(k){$assigned\_role$, database, user}
           then
              $revoke\_set \leftarrow revoke\_set$ - $revoke\_set$(j)
end
do the revocation of roles in the $revoke\_set$.
END Revoke Algorithm
```

Figure 4: Revoke Algorithm