Naming and Grouping Privileges to Simplify Security Management in Large Databases

Robert W. Baldwin

Tandem Computers
19333 Vallco Parkway
Cupertino, CA 95014

Abstract

This paper describes an extension to ANSI SQL that simplifies security management by reducing the complexity of the access controls on database objects, and by providing users with the flexibility to define administrative roles (like Auditor or Security Administrator) that match their requirements for the separation of duties. The benefit of simplified security management is improved security. The main features of this extension have been implemented in the Oracle RDBMS and have been adopted for a future version of the ANSI SQL standard. This paper focuses on major concepts and issues, not syntax and implementation. The key idea is to allow users to group and name privileges to form named protection domains (NPDs). The Clark-Wilson and Bell-LaPadula models are used to illustrate the benefits and limitations of NPDs. The main conclusion is that the naming and abstraction mechanisms provided by NPDs can simplify security management in much the same way that procedures can simplify programming.

Introduction

SQL relational databases [3] have become quite popular in part because they provide a flexible method for managing information. Unfortunately, the security system for these RDBMSs is neither flexible nor manageable. The result is poor operational security. Access controls are not set as tightly as they could be. This paper describes an extension to SQL that supports flexible and manageable security.

The SQL security system [6] is based on access control lists (ACLs). Each ACL indicates which individuals can perform each operation (e.g., select and insert) on a named object (e.g., table or view). The current ANSI SQL standard does not include the ability to grant privileges to groups of individuals. Another feature of the standard is that most operations can be granted to other users, but some, like adding a new column to a table, can only be performed by the object's owner. A privilege (an operation-object pair) can be granted "with grant option", which allows the recipient to pass that privilege on to other users, including the possibility of allowing the other users to further grant the privilege. A SQL data dictionary table records who granted which privileges to whom. Privileges are taken away from users with the revoke statement. One option of revoke is to cascade the removal of privileges to cover the case where the original recipient has passed on the privileges. The rules that govern cascading revocation are complex, and like any complex control system chosen and fixed by vendors, only a few customers will find that it meets their needs.

Managing and understanding a large collection of ACLs is hard [5]. ACLs express security controls in a low-level language that is far removed from the high-level specification of a site's security policies. This semantic gap between ACLs and security policies is the fundamental source of manageability problems in large security systems. The problems addressed in this paper are listed below.

CH2884-5/90/0000/0116$01.00 © 1990 IEEE

Difficult SQL Security Management Problems

1. Maintaining application-oriented security.
2. Reassigning security responsibilities.
3. Restricting ad hoc SQL statements.
4. Separating administrative duties.

From a security administrator's point of view, enabling a new person to run the appropriate pieces of an invoice application involves issuing a large number of grant statements that correspond to the privileges required by the application to carry out that person's job. Notice that it is not enough to control execute access to the application. In order to avoid duplication of software, applications usually support several different classes of users. This means the Security Administrator must know which table privileges are required by each class of users and carefully grant the correct ones to each new person. Consider an invoice application that has two user classes: Clerks and Supervisors. Clerks can only create invoices whereas Supervisors can both create and modify invoices. A single application program would support both Clerks and Supervisors. The only way to enforce the separation between those classes of users is to grant them different privileges on the underlying objects. However, keeping track of all the user classes, applications and privileges in a large database is so hard that one site decided to implemented a special database to manage this information. This site found that their ACL-based security configuration was too complex to maintain by hand. Instead they used a database report writer to generate the required grant and revoke statements.

Another problem with SQL security arises when the responsibilities of the Security Administrator are reassigned to another person. A very long procedure must be carefully followed to make sure that ordinary users retain their current access rights and to make sure that no database objects are inadvertently destroyed. The difficulty is that the ANSI SQL security model requires that there be a chain of grantors that leads from an object's owner to each user who can access that object. If an intermediate grantor in that chain is removed, as would be the case in changing the Security Administrator, then all users that are downstream from that grantor will lose their access. The long procedure creates alternate chains based on the new administrator before the old administrator is removed. There is no way to change the "ownership" of a link in the chain without creating a new link and removing the old one. This procedure is so difficult that in practice, customers create one (and only one) special user account that performs all the duties of the Security Administrator. This account is never removed so the long procedure is avoided. The security and accountability of this approach rests solely on good password management for that special user account.

Complex databases often have integrity constraints that can only be enforced by applications and this leads to application-oriented security requirements that cannot be expressed under current SQL. These requirements are generally of the form "this user can only access that data if he is running one of these application programs". Under ANSI SQL, the set of operations a user can perform (the user's protection domain) only depends on the name of the user. The user's protection domain is only changed when ACLs on objects are modified. To satisfy application-oriented security requirements, it must be possible for a single user to operate in different protection domains without losing track of the true identity of the user, which is required for auditing. Actually, the ANSI SQL standard does not address the issue of auditing, but many vendors provide it. Many customers have asked for a clean way to support this kind of security requirement.

The final trouble area addressed by the mechanism described in this paper is how to support administrative roles like Auditor, Database Administrator, and Security Administrator, and Operator. Usually, two customers have two different opinions about which database powers should be assigned to a role. For example, should auditors be able to delete the audit trail, or should they be able to change the kind of actions that generate audit records? Should the owner of a table be able to turn off auditing for that table? Can Database Administrators create new users? Can an Operator shutdown the database, or should his powers be limited to taking backup dumps? There is no consensus on these questions. Customers want flexibility. They do not want vendors to choose security policies for them by providing a fixed set of administrative roles.

Key Concepts

This paper describes an extension to ANSI SQL that solves these problems, and thus greatly improves the operational security of relational databases. The extension is based on the concept of a Named Protection Domain (NPD). The idea is to allow the grouping and

naming of collections of privileges and individuals. The concept of a protection domain was developed with the Multics operating system [8]. It refers to the set of all operations on objects that could be performed by a subject (process). Along with Lampson's access matrix concept [7], protection domains are often used to explain the behavior of security systems. However, Multics did not explicitly represent protection domains. They were abstractions derived from information in ACLs. The NPDs described in this paper are explicit representations of a collection of privileges. They are protection domains which are defined and named by the users of a database. In addition, NPDs can group and name collections of individual users including the possibility of groups of groups. Basically, NPDs allow a site to create opaque modular abstractions that form the basis of their security configuration. Like procedures which can simplify programming, NPDs can simplify security management.

The key idea is to group privileges, not individuals. An *object privilege* is the right to perform some operation on a particular object (e.g., select rows from the employee table). This extension allows users to build a directed acyclic graph that leads from object privileges to users. Each node in this *privilege graph* can be given a name and that node represents a *named protection domain* (NPD). For example, all the object privileges needed to run the accounts-receivable portion of a general-ledger application can be granted to an NPD called A_R. The job of enabling a user to perform the accounts-receivable task is simple: just grant A_R to that user. Alternatively, the A_R domain could be granted to an NPD called A_R_Clerks, which also includes the privileges to perform a number of other tasks. In this case, when a new clerk is hired, he is granted the A_R_Clerk domain.

At any given time only one NPD is active. Activation has the effect of enabling all the privileges that are in the subtree rooted by the activated NPD. That is, although only one NPD is active, some collection of NPDs will be enabled. Notice that the user cannot enable an arbitrary collection of NPDs. The enabled NPDs must always form a subtree in the privilege graph. This restriction makes it easier for security administrators to understand the possible behaviors of the security system. It is easy to determine which privileges can be enabled at the same time, and thus easy to make sure that there are no conflicts between the privileges themselves. This design also makes it possible to express separation of duties policies that help prevent fraud by prohibiting certain privileges from being active simultaneously. The security administrators can choose the size and complexity of each protection domain. These domains can be small to conform to the principle of least privilege, or large to make the security system transparent.

A *database privilege* is one of the super-user privileges like being able to perform crash recover, or being able to read any table. NPDs can group together and name collections of database privileges to form *administrative roles*. Although most customers are satisfied with the SQL controls on data operations (e.g., select and insert operations), there is little consensus on the administrative controls. Some customers want to allow the owners of tables to be able to determine which events generate security audit records, while others want to restrict that power to a special Auditor role. Even among customers who want an Auditor role, there is disagreement about whether the Auditor should have the power to delete the audit trail (presumably after archiving it). Customers want the flexibility to define their own administrative roles. NPDs can give them that flexibility. Each site can define a set of administrative roles that dovetail with their system of human controls outside of the computer.

This paper argues that a security system which is based on grouping privileges is easier to manage than one based on grouping individuals. Privilege groups can represent the abstractions that are found in high-level security policies. In fact, groups of users are a degenerate case of this mechanism. The resulting security system is easier to manage because there is a smaller semantic gap between the high-level security policy and the abstractions used to express that policy to the security system.

The bulk of the concepts described here have been implemented in the Oracle DBMS and the basic features of NPDs have been accepted for a future version of the ANSI SQL standard. In both places the term "Role" is used to refer to a Named Protection Domain. However, "Role" is an over-used term, so to avoid confusion with other concepts, this paper uses the acronym "NPD". The relationships between NPDs, roles, groups and access classes are discussed at the end of the paper.

Outline of Paper

NPDs will first be discussed in the context of grouping the privileges for objects. The issues of naming, controlling and activating NPDs are the same for both object privileges and database privileges. A wide range of

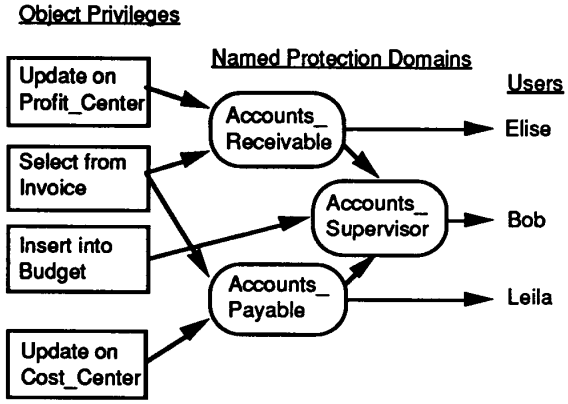
administrative roles can be built with NPDs, so the second portion of the paper discusses how various customer requirements can be supported by grouping low-level database privileges with NPDs. The last two sections describe how NPDs can be used to support the Clark-Wilson model and the Bell-LaPadula model.

Grouping Object Privileges and Individuals

This section describes how NPDs are used to group object privileges and individuals to improve the manageability of the security system. A later section describes how the same mechanism can group database privileges to form flexible administrative roles.

NPD Privilege Graph

In ANSI SQL, object privileges are directly granted to individual users. An *object privilege* is the right to perform a specified operation on a particular table or view. The operations include the normal data modification operations (select and insert) plus the data definition operations on existing objects (altering a table definition to add a new column). Operations that are not based on existing objects (creating a new table) are discussed in the section on *database privileges*. The NPD extension to SQL allows an object privilege to be granted to an NPD, which in turn may be granted to other NPDs before being granted to a user. However, as discussed in the section on NPD administration, the with-grant-option for a privilege cannot be given to an NPD. As shown in the figure below, all the object privileges needed to perform the accounts-receivable task (perhaps a dozen different privileges on various tables) can be granted to an NPD with the name Accounts_Receivable, which can then be granted to any user who needs to perform that task. In addition, the site's security policy might have a notion of an account supervisor who can handle both payable and receivable accounts, so both the Accounts_Receivable NPD and an NPD called Accounts_Payable would be granted to an NPD with the name Accounts_Supervisor. The supervisor's NPD can be used to create fraudulent transactions, so it should only be given to employees who are trusted (or at least carefully watched). As discussed in the section on NPD activation, it is also possible to add a constraint that prevents a supervisor from activating both of the sub-NPDs at the same time.



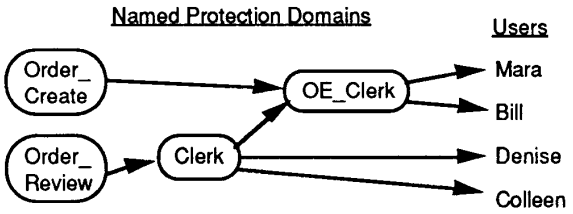
Sample Privilege Graph

Basically, object privileges can be granted to NPDs, which can then be granted to another NPD (loops are not allowed) or to a user. The SQL grant statement is extended to handle the composite privileges represented by NPDs in addition to the primitive object privileges that it currently supports. Each NPD could be said to contain a collection of object privileges and other NPDs. When an NPD is granted to a user, that user can exercise all the object privileges that are directly or indirectly contained in that NPD.

NPDs define a directed acyclic graph, called the *privilege graph*, leading from object privileges to individual users. Each node of the privilege graph can be named and thus serve as a mnemonic for the collection of privileges it represents. A careful choice of groupings and names can greatly simplify security management. Of course, as with procedures in programming languages, a poor choice of abstractions can make things very confusing. One design strategy is to create an NPD for each task performed by each DBMS application. This strategy produces a finer granularity of control than simply restricting who can execute each application. To simplify application development, DBMS applications often support different classes of users and each class is only allowed to perform a subset of all the possible tasks (this is particularly true for applications involved in on-line transaction processing). The NPD-per-task approach can handle such applications. To allow someone to perform a particular task: grant them the corresponding NPD. Similarly, determining the set of tasks that a user can perform is easy. Revoking an NPD from a user will remove that user's ability to perform the corresponding task. This strategy simplifies the security management of applications.

In practice both the NPDs and the applications must be carefully designed to make security management this easy. The design becomes difficult when the various tasks require overlapping privileges. Explicitly granting the NPDs for two different tasks may implicitly provide the privileges required to perform a third task. This situation can be recognized by examining the NPD privilege graph, but avoiding the problem may not be possible without redesigning the tasks.

NPDs can also be used to group individuals. If every user who is an order entry clerk has been granted the NPD, OE_Clerk, then granting a privilege to OE_Clerk enables all order entry clerks to use that privilege. This provides the same functionality as user groups in traditional operating systems. In addition, NPDs can create groups of groups of individuals. The figure below shows a privilege graph that allows the group of users named OE_Clerk to perform the application task of creating new orders via the Order_Create NPD. All clerks are grouped into an NPD called Clerks and they have been granted the NPD that enables them to review orders. The privilege graph embodies the security policy that all four clerks can review orders, but only two of them can create new orders. The NPD mechanism has the power to group both privileges and individuals in a unified way.



Grouping individuals with NPDs

Managing Changes to the Security Configuration

A very important feature of NPDs is that they can be used to clearly separate security management into three activities: 1) defining collections of object privileges that match abstract tasks in the site's security policy, 2) defining classes of users according to the jobs they perform (as opposed to grouping users by their administrative departments), and 3) defining which classes of users can perform each task. Partitioning these three kinds of grouping simplifies the routine maintenance of the security configuration. For example, when a person changes jobs, the security administrator just grants and revokes a few NPDs to match the person's new job description. Some of the NPDs granted to that person will not need to be changed; for example, there could be an NPD called FT_Employee that is granted to all users who are full time employees. Notice that the administrator can respond to the job change without worrying about numerous select and insert privileges on tables and views. Another common change is to add a new application program or a new feature to an existing application. Such a change can be accomplished by granting a few new privileges to an existing NPD, or perhaps creating a new NPD and granting it to an existing NPD that defines some class of users. In both cases, the change can be made using just local knowledge. There is no need to review all the privileges required by all the applications.

Aspects of security administration

1. Grouping privileges into application tasks
2. Grouping individuals into classes of users
3. Assigning tasks to user classes

From the point of view of assurance, NPDs help auditors understand why some privilege has been granted. The privilege graph is an explicit representation of the reason that each privilege is granted to each user. For example, under ANSI SQL it would be quite hard to figure out why somebody has select access to the Hire_Date column of the Employee table, whereas a privilege graph could show how that operation was grouped together with others as part of an application that determines vesting in a corporate retirement account. The privilege graph clearly shows that the user needs to access the hiring date information in order to run the retirement report application. ANSI SQL has a table which indicates who granted each privilege, but not why. The NPD privilege graph, which would be visible as a SQL table, answers the why question. The NPDs represent abstractions in the site's security policy and thus they make the security configuration easier to understand, verify, or modify.

Controlling NPDs

So far we've discussed how NPDs simplify the allocation and management of access controls, and that leads to the question of how the NPDs themselves are controlled. How are they named? Who can create them? Who owns them? Who can grant an NPD to a user? Who can grant a privilege to an NPD?

Naming and Ownership of NPDs

There are two choices for naming entities in a SQL database. A name can be global to the whole database as are usernames and schema names (schemas are similar OS file directories), or a name can be local to a schema. To simplify the syntax of the grant and revoke statements, NPDs are given global names chosen from the same name-space as usernames. One advantage of this choice is that NPD names do not need to be preceded by a type indicator (a keyword like "ROLE"). The main reason for chosing global names is to make NPDs independent of the existence of any user (or schema). One of the goals of this mechanism is to reduce the security impact of adding or removing users. If the name of an NPD depended on the existence of a user, then removing that user would have the undesirable side-effect of causing a major change to the NPD privilege graph.

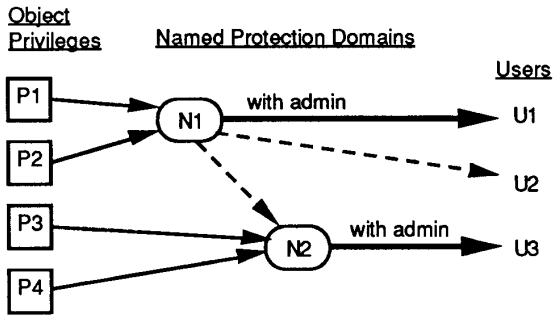
Although NPDs look like usernames, an NPD cannot own any object. NPDs cannot be used to solve the ownership problem of tables that logically belong to an application. For example, the collection of tables, views and indices used by the MIS applications are often owned by a mythical person with the username MIS. According to ANSI SQL, there are some operations that can only be performed by the owner of an object (e.g., dropping (remove) it, or altering its definition). When these operations need to be performed, some real individual must login as the mythical user MIS, and that leads to a loss of accountability. Any person who knew the MIS password could have been responsible for the actions that took place during a questionable session. Security systems should be designed to eliminate the need for multiple people to know the password for a single account.

A better solution is to avoid having privileges that cannot be granted. No operations should be restricted to just the owner of an object. It should be possible for individuals who are operating under their true user identities to perform any of the operations required to maintain or upgrade application tables. If nothing else, databases should support a course-grain "owner" access mode. As part of installing the application, this owner access mode would be granted to the users who will be responsible for maintaining the application. If the database also supports NPDs, one could grant the "owner" access mode to an NPD which is eventually granted to some set of users. However, the NPD would not own the table. Removing the NPD would not cause the table to be removed. With fine-grained controls, one NPD might contain the privilege to add columns to a table, while another NPD might have the privilege to drop (remove) the table. NPDs do not own objects.

Administration of NPDs

Granting an NPD to a user makes it possible for that user to exercise all the privileges directly or indirectly granted to that NPD. Clearly, this operation must be restricted to authorized users. In this design, a user is authorized to grant an NPD to other users or NPDs if that user has been granted the NPD "with admin option". The syntax for this is similar to the grant with-grant-option statement in current SQL, but the semantics are quite different in order to avoid the complexities of cascading revoke and inherited grant privileges. Many SQL customers have complained that cascading revoke is too hard to understand and that it does not help them solve their security management problems. The key difference between the with-admin and with-grant options is that removing an administrator does not cause a cascading change to the NPD privilege graph. If the ex-administrator has granted an NPD to other users or NPDs, those authorizations remain in effect after the NPD has been revoked from the ex-administrator.

Granting an NPD, N1, to a user, U1, makes it possible for user U1 to exercise all the privileges contained in the NPD N1. In the figure below this includes the two object privileges P1 and P2. If the with-admin-option was included in the grant, then U1 can also build new arcs (shown as dashed lines) in the NPD privilege graph. U1 could grant N1 to another user U2, which would build an arc from N1 to U2, or grant N1 to a second NPD N2, which would build an arc from N1 to N2. The NPD N2 would then contain all the privileges that were granted to N1 (P1 and P2) plus the ones it already contained (P3 and P4). However, U1 could not grant N2 to U2 because U1 does not have the with-admin privilege on N2.



Changes possible with the admin option

When an object privilege is granted to an NPD, the with-grant-option cannot be included. The main reason for this restriction is to further decouple NPDs from the complexities of cascading grants and revokes. Customers have difficulty understanding the implications of the with-grant-option and they find cascading revoke to be confusing, useless or down right dangerous. NPDs have simpler administrative semantics that are easy to understand and use.

The with-admin-option for an NPD can be granted to other NPDs. This is a debatable design decision because it introduces a large variety of ways to build confusing security configurations. This decision is consistent with the behavior of other object privileges. The "admin" option can be treated as an operation on the NPD object. Granting and revoking the admin option would be like granting or revoking select access on a table. The major benefit of this feature is that NPDs can be used to define a class of users who are responsible for maintaining a portion of the NPD privilege graph. For example, an NPD called MIS_Security could be granted the with-admin-option on all the NPDs that are part of the MIS applications. This would make it easy to designate a new person to administer the security of MIS applications: grant him the MIS_Security NPD. Even though the with-admin-option does not have cascading effects, it is still possible to create a tangled system of administration with this feature. In practice, the with-admin-option should only be granted directly to users or NPDs that represent the classes of users who control the security configuration.

When an NPD is created (see below for control of creation), the user who created it is automatically given admin authority over that NPD, and thus can designate other administrators for the NPD. The creator does not own the NPD. NPDs do not have owners. In fact, the creator's admin authority can be removed. The horn-lock problem of controlling an NPD when nobody has admin authority for it is solved by the usual recourse to an all powerful super-user (hopefully the password management of this account is good), or by restoring the system to a previous state from backup information.

Creating and Dropping NPDs

Different sites will have different policies about who can create and drop (remove) NPDs because they have different policies about how the security configuration is designed, setup and maintained. To allow flexibility, the Oracle implementation defined a database privilege that enables a user to create or drop NPDs. As discussed in the section on administrative roles, database privileges like this one can be granted to users and NPDs. The decision to make a single database privilege control both the create and drop operations on all NPDs is consistent with the current practice of using a single privilege to control both creating and dropping users. Consistency is important because usernames and NPD names are drawn from the same name-space. Further experience with NPDs is needed to determine whether a finer granularity of control is necessary.

One possible enhancement to the controls on NPDs is to restrict who can grant a privilege to an NPD. This is called the grant-to operation. Granting a privilege to an NPD changes the meaning of the NPD. For example, one does not expect the Accounts_Receivable NPD to include the ability to select salary information out of the employee table. The person who set up the Accounts_Receivable NPD had good reasons for choosing the set of privileges it contains, and that person may want to prevent others from modifying its contents. This is particularly true when NPDs are being used to represent military access classes (discussed below), which are supposed to have fixed, well defined meanings.

The question of whether to control the grant-to operation revolves around a fundamental conflict between the desires of the administrators of an NPD and the rights of the owners of tables. SQL security is based on the discretion of object owners. Owners should be able to both grant and revoke object privileges wherever they want, and by extension this includes NPDs. A table owner would be surprised if he grants a privilege to an NPD and then finds out he cannot revoke it because he no longer has the required grant-to privilege. In the interest of consistency with SQL, Oracle's implementation of NPDs does not require a separate "grant-to" authority. Anyone

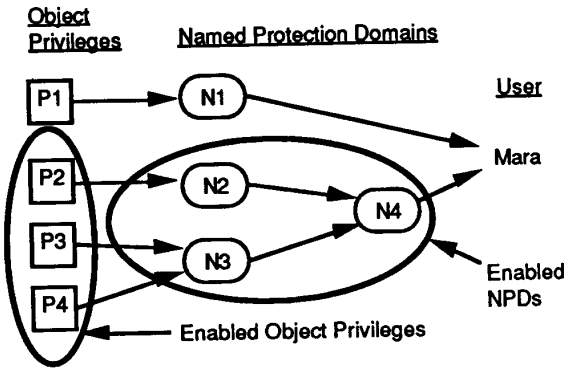
who has with-grant authority on an object privilege or with-admin authority on an NPD privilege, can grant or revoke it to any NPD or user in the database. Customer experience will indicate whether further controls are needed.

NPD activation

So far we've shown how NPDs are used to define collections of privileges out of primitive object privileges and composite NPD privileges. We've also described how NPDs are created and controlled. This section discusses how to determine which NPDs will govern the access control decisions during a a particular database session.

The issue is how to determine the effective protection domain when a user has been granted multiple, perhaps overlapping, NPDs. What is the set of all operations that he is allowed to perform? The obvious choices are: all NPDs accessible to the user are active, only one NPD is active, or a specified subset of NPDs is active. Requiring that all NPDs are active makes the security system transparent to the user and thus more convenient, but the principal of least privilege argues against this choice. The desire for application-oriented security argues for activating just one NPD, and the desire for flexibility suggests that users should be allowed to enable arbitrary collections of NPDs. The actual choice is a combination of all three possibilities.

In this design only one NPD can be activated at a time, but if that NPD contains other NPDs, those NPDs are also enabled. The user can then exercise all the privileges that are directly granted to any of the enabled NPDs including the NPD that was activated. Basically, activating an NPD corresponds to enabling a subtree in the NPD privilege graph. In the figure below, the Mara has activated N4, which enables the NPDs N2, N3, and N4. This in turn enables the object privileges P2 through P4. A mechanistic way to think of this is to associate with each DBMS session a block of security information that lists the NPDs that have been enabled. A user is allowed to perform an operation on some object if the corresponding ACL includes one of the enabled NPDs. The set of enabled NPDs includes the activated NPD and any NPD that has been directly or indirectly granted to the activated NPD.



Activation of NPDs

One unifying aspect of NPDs is that the set of object privileges that are directly granted to the user (i.e., not granted through an intervening NPD) can be viewed as privileges that are granted to an NPD with the same name as the user's name. In current SQL, a user can perform an operation if his username appears in the ACL for that operation. With this extension, an operation is allowed if either the username or the name of one of the enabled NPDs appears on the appropriate ACL.

In the full NPD security model, privileges directly granted to a user can be enabled or disabled just like NPDs (this was not included in the Oracle implementation). There is a pseudo NPD called "UserPrivs" that represents all of the user's direct privileges. If the NPD UserPrivs has been granted (directly or indirectly) to the activated NPD, then the list of enabled NPDs which are checked against ACLs will include the username. That is, enabling the pseudo NPD named UserPrivs, enables the privileges that have been directly granted to the user. This unifying trick is another reason that NPDs and usernames are chosen from the same name-space.

The resulting security model is flexible enough to support simple confinement policies. Such policies try to stop the wholesale copying of information by preventing a user who has read access to sensitive information from making a private copy of it. Of course, these kinds of policies have their limitations. A user may be able to display the sensitive information on a terminal and record it from there. However, customers who are aware of the limitations still ask for mechanisms that would help with confinement policies. Clearly such a restriction would not be possible if the user can exercise his normal ability to insert into a table that he owns. On the other hand, private copies are sometimes exactly what is desired, so the

security system must be able to enable both the NPD privileges and the direct user privileges. In this design, each NPD indicates whether the direct privileges are enabled based on whether that NPD has been granted the pseudo NPD UserPriv. If UserPriv is not enabled, then only the list of enabled NPDs is checked when searching an ACL. The user's name is not checked, so the privileges that have been directly granted to the user are not available. This strategy allows the designer of the NPD privilege graph to choose which protection domains will include the privileges that are directly granted to users.

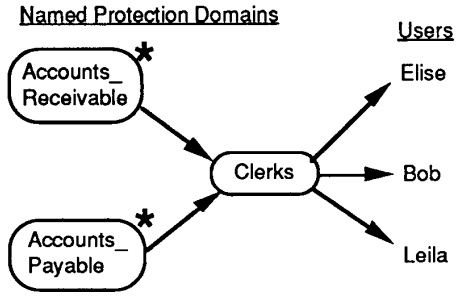
Restricting NPD Activation

An important restriction on NPDs is that only certain ones can be activated. Only activatable NPDs can serve as the tops of enabled subtrees in the NPD privilege graph.

The motivation for only allowing certain subtrees to be activated is the desire to create well-behaved opaque modular abstractions in the security configuration. A well-behaved security abstraction corresponds to some task that is meaningful to users. For example, consider two NPDs that represent creating invoices and checking customer mailing information respectively. Neither one of these NPDs should be enabled by itself. These "internal" NPDs are always combined with others to form meaningful tasks like taking orders from customers or packing shipments at a warehouse. Without security abstractions that are well-defined and well-behaved, a security administrator must do a lot of work to make sure that application programs do not encounter unexpected security errors.

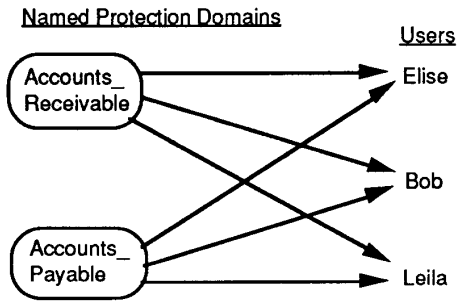
All NPDs have a boolean attribute, called Activatable, that is initialized when the NPD is created. This attribute can be modified later with an Alter statement by any user who has been granted the admin option on that NPD. In order to activate an NPD, first the user must have been granted the NPD (either directly or indirectly), and second the Activatable attribute of the NPD must be true. This attribute is not checked for any of the NPDs that are indirectly enabled by the activated NPD. In terms of the NPD privilege graph, this rule means that the root of the subtree being enabled must be activatable and there must be a path from the root NPD to the user.

One simplification of this scheme is to say that users can only activate the NPDs that are directly granted to them. This avoids the need to store a flag attribute and the corresponding syntax to initialize, view and modify it. Unfortunately, this simplification, which was included in the Oracle implementation of NPDs, interferes with the ability of security administrators to manage fine-grain user classes or detailed tasks. This point is illustrated in the next two figures. As discussed earlier, using NPDs to define user classes independently of the privileges required by applications can simplify the routine maintenance of the security configuration. If user classes exist with this simplification, a user must enable all the applications granted to a class at the same time. As illustrated below, this would mean that clerks would have to enable all the privileges of both the accounts-receivable and the accounts-payable tasks. This is a violation of the principle of least privilege.



Flag controlling NPD activatability

To avoid this violation, the privilege graph would have to be set up as shown below. The graph is more complex because the information about the user class for clerks is missing. This simplification makes it difficult to manage user classes. By making the directly granted NPDs special, this simplification forces the natural privilege graph to be flattened with a resulting loss of information and increase in complexity.



Graph without actiavability flag

The restriction that only some NPDs can be activated also arises from the desire for reuseable security abstractions. A modular abstraction can be included in many places even though it is never directly exposed to a user. The invoice creation NPD mentioned above is a good example. It would be granted to several different NPDs since creating invoices would be part of several application tasks. Whether an NPD can be activated is like the difference between an internal and external procedure. It indicates whether the abstraction is visible to the next level.

Part of making an NPD opaque is hiding the set of NPDs that it contains. That hiding would not occur if a user could activate any NPD that has been indirectly granted to him. The person who designed the relevant part of the privilege graph had a reason for grouping together the particular collection of NPDs. They represented a set of privileges that should be enabled at the same time. The activatable flag enforces the hiding of an NPD's internal structure.

The decision to only allow one NPD to be activated was chosen for several reasons. This choice allows the designers of the NPD graph to conveniently control all the possible protection domains that any user session can be in. A user can only be in one of the domains defined by the activatable NPDs. The alternative of letting each user choose the set of activated NPDs, leads to a combinatorial explosion in the number of effective protection domains. The number of combinations of NPDs makes it hard to understand the possible behavior of the security system. The activate-one approach also avoids the problem of defining compatible and non-compatible NPDs. Using the NPD graph it is easy to determine which NPDs might be enabled at the same time and detect incompatibilities. For example a site could have a policy that the Accounts_Receivable NPD may not be enabled at the same time as the Accounts_Payable NPD because that creates too many possibilities for fraud. A quick check of the privilege graph could determine whether this policy could be violated. The price of easy analysis is reduced flexibility.

Only one NPD is active at any time, so how is that NPD picked? The activated NPD can be chosen explicitly using a SQL statement or implicitly as part of invoking a database application. Linking NPD activation to the invocation of applications leads to a major improvement in application-oriented security. Implicit activation makes the security system transparent and thus easier to use. One option is to prevent a user from explicitly chosing the active NPD (using a database privilege discussed later). This option supports security requirements that prevent users from accessing a particular table unless he is running a trusted application.

Correctly supporting this requirement involves cooperation between the OS and DBMS security systems. The OS must control which applications the user can execute (including the ability to create new applications) and the DBMS must control the protection domain for each application. This strategy can even create multiple environments for executing interactive (ad hoc) SQL statements. A single interactive SQL interpreter can be given several names by the OS, and the activated NPD would depend on the name used to invoke the interpreter. This kind of NPD would grant the user the appropriate query and update access without enabling him to modify crucial tables in ad hoc ways.

The details of the linkage between applications and NPDs is operating system specific, but it can be based on the name of the application program or on OS security information. For example, the DBMS could have a table that maps username and program name pairs into an NPD name. Alternatively, there could be a mapping between OS security tokens (like Unix group names or RACF role names) and NPD names. Some DBMSs, like Oracle, already provide this sort of token translation for usernames between a DBMS session and an OS login session.

The ability to set or change the active NPD is controlled by a database privilege which may or may not be granted to the user. At the beginning of a session, the pseudo NPD UserPrivs is active. This means that the user can exercise all object and database privileges that have been directly granted to him, possibly including the database privilege required to activate NPDs. Notice that there is no deactivate statement for NPDs. If a user wants to turn off some NPD, he must pick a new one (which might be UserPrivs).

Performance and Implementation Issues

This extension should lead to an overall decrease in the amount of storage used by the DBMS security system. Although additional space is needed to keep track of the NPDs and the NPD privilege graph, less space will be needed to store the ACLs for each database object.

ACLs will be smaller because a single NPD would otherwise be represented by a large list of individual users. A large database will have more tables and views than it has NPDs, so the savings due to smaller ACLs will outweigh the space used by the NPDs.

The amount of time needed to make an access check may also decrease. Current SQL access checking scans the whole ACL looking for a single value (the ID of the user). A pre-sorted ACL can be used to avoid a linear search. In contrast, this extension requires scanning the whole ACL looking for a list of IDs that correspond to the enabled NPDs (and possibly the user's ID). Again, pre-sorting the ACL and the list of NPDs reduces the scanning time. The NPD access check would be slower if the ACLs were the same size as they are currently. However, an ACL based on NPDs is likely to be much shorter than an ACL based on individual users. The result is likely to be an overall decrease in the access checking time for large databases.

The most expensive operation associated with NPDs will be traversing the NPD privilege graph when an NPD is activated. Activating an NPD will be a common operation on sites with a strong policy for least-privilege. This operation involves disk I/O to fetch the NPD graph and CPU time to compute the enabled subtree. A good way to minimize this cost is to only examine the graph when a user logs in. When the session starts, all the accessible NPDs can be computed and cached. The cached portion of the graph would record the grant relationship between NPDs as well as the Activatable attribute. Several data structures exist that compactly represent a directed acyclic graph such as this. The currently enabled NPDs would be flagged and sorted. As long as this information is cached, activating an NPD would be relatively inexpensive. The down side of this performance-oriented strategy is that changes to the NPD privilege graph only influence new database sessions. They would not affect existing sessions unless the system included a mechanism to flush or update the cached security information.

Grouping Database Privileges

When NPDs are used to group object privileges and individual users, the result is a security configuration that is easy to understand and maintain. This section describes how NPDs can create flexible administrative roles and manage the assignment of these roles to individuals.

The key idea behind this flexibility is to separate and explicitly name all the low-level database privileges so they can be grouped and named using NPDs. Administrative roles can then be created using the same modularity and abstraction methods that are used for object privileges. For example, the database privilege that allows read access to all objects (a kind of super-user access) and the database privilege that allows shutting down the databases could be two of the database privileges that are grouped together to form the Database Administrator role. The plethora of low-level database privileges provides flexibility while the NPD grouping mechanism provides understandability and usability. In practice, a vendor would ship the database with a number of pre-defined administrative roles. These roles would simplify the installation and initial use of the DBMS and would meet the needs of many customers. However, sites that need the flexibility can create administrative roles that exactly match the separation of duties requirements of their security policy.

Administrative roles

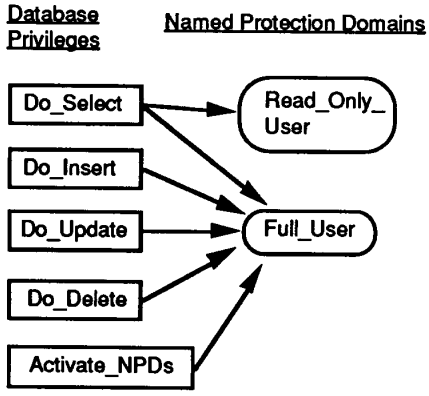
The mechanics of using NPDs as administrative roles are exactly the same as NPDs used for object privileges. The same mechanisms are used to define, name, control, and activate administrative NPDs. To simplify the control of administrative NPDs, two constraints are enforced. The main constraint is that primitive database privileges and primitive object privileges cannot be granted to the same NPD. The two kinds of privileges can only be mixed indirectly through intervening NPDs. The other constraint is that database privileges cannot be directly granted to a user. These constraints make it easier to control the grouping of database privileges. A great deal of thought must be done to build a reasonable collection of database privileges, but once that has happened, the resulting administrative NPD can be granted with the admin option to several people who can pass it on without needing to consider internal database issues.

The kinds of administrative roles that can be created with NPDs are determined by the particular set of database privileges that can be named separately. A full list of the database privileges for the Oracle database is beyond the scope of this paper. Instead, this section illustrates the kinds of database privileges that can be

identified and shows how they support customer security requirements.

One good way to choose database privileges is to create one for each kind of SQL statement. There could be a database privilege that allows the execution of the Create Table statement that would be separate from the privilege for the Create View or Create Index statements. In some database environments ordinary users are not allowed to create their own tables. This restriction could be enforced by not including the Create Table database privilege in the administrative NPD that is granted to ordinary users. This is a more uniform solution than using quotas to control table creation. A more security relevant example would be to have separate database privileges that control the Create NPD and Create User statements. These might or might not be granted to the same administrative role. One of the critical database privileges is the one that allows granting a database privilege to an NPD. Any user who can enable this privilege can do anything because he can grant himself an all powerful administrative role.

If there are database privileges that control the data modification statements in SQL (e.g., select and update), then there must be an administrative role that is granted to ordinary users. The NPD for this user-role would enable the execution of the basic statements like select and update. A user would only be allowed to select from a table if he had the database privilege for the select statement and he had specific authorization (via an ACL) to select from that table. The user-role NPD could be directly granted to the user or there could be several user-type roles that are granted to the various NPDs that the users can activate. For example, one way to control write access in an application that allows a user to enter arbitrary SQL statements is to make sure that the activated NPD does not include the database privileges for the insert, update, or delete statements. It would only contain the ability to execute select statements. Many customers have requested a read-only SQL interpreter and this feature of administrative NPDs can meet that request.



Building Administrative Roles

The details of session initialization influence the choice of database privileges that are granted to a user. After a user has authenticated himself, the pseudo NPD called UserPrivs is activated. This enables all object privileges directly granted to the user and it enables all purely administrative NPDs that are directly granted to the user (this behavior is another reason why object and database privileges cannot be directly mixed). For example if a group of database privileges is granted to the NPD, Trusted_User, and that NPD is granted to Alice, then when Alice logs in, all those database privileges will be activated. Presumably the Trusted_User NPD includes the privilege to execute the SQL statement that changes the active NPD. If not, Alice would be unable to change her protection domain either by herself or via a trusted application.

It may be desirable to control who can grant object privileges to users or to NPDs using a database privilege like the one that controls the granting of database privileges. This decision is debatable because it touches on an existing part of the ANSI SQL security model. That model allows any user who has the with-grant-option on a privilege to grant that privilege to anyone else and by extension this includes NPDs. Furthermore, the ANSI model embodies the belief that the owner of an object has an inalienable right to hand out access to that object to any user, and by extension, to any NPD. Some customers subscribe to a different bill of rights. They believe that the user who can hand out access rights to an object should not be the same as the person who can modify the object's definition (e.g., change an integrity constraint on the table). These customers want to enforce strong separation of duties

policies in order to reduce the possibilities for fraud or significant error.

A number of database privileges are needed to support the separation of duties policies that have been requested by DBMS users. For example, who can change the audit options for a table? That is, who can determine which operation will generate audit trail entries? The SQL model says that the owner of a table controls it and by extension this would include setting the audit options (actually, the ANSI SQL standard does not address auditing issues). Some prudent security administrators believe that the owner of a table is the last person they should trust to decide what gets audited. They argue that the audit trail is a global resource that should be controlled centrally. A database privilege that governs the changing of audit settings (of any object) can allow a great deal of flexibility. This privilege could be granted to the security administrators and to particular table owners. It does not have to be an all or nothing option switch.

Another way to determine possible database privileges is to look at all the operations that are available to the database's super-user (a vendor specific concept). These operations can be separated into classes that might be granted separately. For example, the Oracle super-user (called DBA) has the power to read and modify all database objects. Some sites may want to give the read-everything power to Auditors without giving them write-everything authority. The rule of thumb is that if it can be separated, someone will want it separated. Notice that there is no harm in dividing privileges too finely (though some clever programming might be needed to get the desired performance) because it is always possible to create an NPD that fixes excessive subdivisions. However, there is no way to fix privileges that are too coarse.

Control of startup and crash recovery

The final issue to discuss is how to manage database privileges when the database is not fully operational. For example, what information is checked to determine whether someone is authorized to startup a database? The authorization cannot be based on information in the database because that information is not available until the database is started. What about crash recovery and other situations which require actions that are not associated with a particular database session? What privileges can be associated with a session that is not logged in? The issue is how to determine which database privileges are enabled under these unusual circumstances. Enabling all privileges is not an acceptable solution to many customers, particularly for database startup. Often relatively untrusted people are allowed to restart a database, and those people should not be given full access to the DBMS.

The solution proposed here is to link a piece of the security system of the underlying operating system to pre-defined administrative NPDs. In much the same way that NPD activation can be tied to the execution of an application program, the transitional state used to start the database can be tied to a predefined administrative NPD. One of the first things that happens when a transitional state is entered is for the database to call an OS specific routine that determines which NPD to activate. The OS uses its own authentication and authorization mechanisms to pick the appropriate NPD (if any). The database will trust the resulting answer and activate the corresponding pre-defined NPD. The mapping between the names of pre-defined NPDs and the related tokens in the OS security system (e.g., Unix user groups or RACF application roles) is made by the OS specific routine. The NPD must be pre-defined because the data dictionary might not be available to lookup customer defined NPDs.

These pre-defined NPDs could either be setup by the vendor with a compiled-in table or defined by the customer as part of installing the database software. In either case the definition of these pre-defined NPDs (i.e., the set of database privileges they enable) must be available whether or not the data dictionary is available. These definitions must also be bound to the database software with high integrity, since anyone who can change these definitions can acquire full access to the database.

This solution gives customers the flexibility they need to define the powers available to a person who controls the database during a transitional state. Each customer can set up their OS security system to activate the appropriate pre-defined NPD and thus enforce their desired security policy.

The concepts and issues associated with NPDs and the privilege graph have been explained, and various examples have shown how these mechanisms can solve the difficult problems of SQL security management. The next two sections describe the relationship between the NPD security model and two existing security models (Clark-Wilson and Bell-LaPadula). Examining how the NPD model can represent those models sheds light on all three models.

Supporting the Clark-Wilson Model

The Clark-Wilson model [2] has pieces that exist both inside and outside the scope of a database system. For example, Integrity Verification Procedures (IVPs) and the certification of Transformation Procedures (TPs) are external to a DBMS. This section describes how NPDs can be used to represent and manage the access control triples in the CW model.

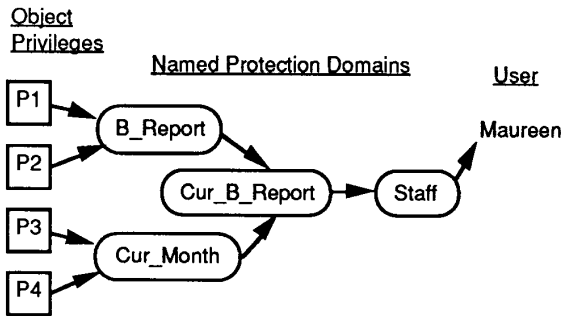
A CW triple specifies that an individual user may perform a specific application operation (Transformation Procedure, TP) on a particular data object (Constrained Data Item, CDI). NPDs can represent the three components of a CW triple and the NPD privilege graph can represent a large collection of triples in a manageable form. Basically, the arcs in the privilege graph represent the triples, while individuals nodes in the graph (NPDs) represent the components.

In its most specific form, the user component of a CW triple identifies a single user, but in this form triples suffer from many of the same problems as ACLs. If the underlying security policies calls for a group of individuals who have identical access requirements (e.g., they are all order entry clerks for the same department) then this situation would be represented by a collection of nearly identical triples. Adding or removing clerks would involve all the difficulties that are found in maintaining a large list of nearly identical ACL entries. NPDs can simplify this security management problem by explicitly representing a class of users. The class name would appear in the triple, not the individual user names. This simplifies the security administrator's job by allowing him to manage user classes separately from the allocation of access rights.

CW triples control application-level operations (TPs). A TP can be represented by an NPD that contains all the object privileges needed to perform the TP. For example, a TP that selects rows from one table and inserts rows in a second table could be represented by an NPD that contains the appropriate select and insert privileges. Notice that the NPD does not provide any guarantees that the TP performs the correct calculations. That aspect of the CW model is beyond the scope of the DBMS. Instead, the NPD guarantees that when it is active, the TP may only select rows from the first table and insert rows into the second table. In this case the security system is being used to provide a correctness check much like run-time bounds checking on array references.

The CDIs of the CW model are data abstractions that could consist of several tables and views in a DBMS. An NPD can represent a CDI by containing all the relevant object privileges on those tables and views. This collection of privileges would most likely be a subset of the full privileges for those objects. For example, the ability to rename a table would probably not be included in the NPD that represented the CDI because none of the TPs that operate on that CDI perform the rename operation. There may be a second NPD used for control operations that includes the rename privilege, but it would not be granted to most of the TPs operating on the CDI. Alternatively, if a DBMS can express specific denial of privileges (e.g., granting a No-Select privilege to a user or NPD) then more restrictive NPDs can be built out of lower-level NPDs that represent the full access rights to a CDI. The main point is that NPDs provide a convenient way to group and name the primitive operations that make up a CDI. Nodes in the NPD privilege graph can explicitly represent CDIs.

We have described how NPDs can represent each component of a CW triple (user, TP and CDI), so now the question is how to represent the triple itself. How is the relationship between the three components enforced? The answer is that the triples are represented by the arcs in the NPD privilege graph and that enforcement is done by the linkage between the invocation of applications and the activation of NPDs. For example a CW triple that allows Maureen to run the Budget Report TP on the Current Month CDI could be represented by an NPD privilege graph like the one shown below. The middle level arcs in the NPD privilege graph represent collections of CW triples. The arcs from B_Report and Cur_Month to the NPD Cur_B_Report represent the binding between a TP and a CDI. The arc from Cur_B_Report to Exec_Staff represents a collection of individual CW triples, one for each user who in a member of the executive staff.



Representing CW Triples with NPDs

The Budget Report TP uses several privileges that are independent of the particular month that it operates on. These privileges, which might include the ability to select department names from the department number table, are grouped together to form an NPD called B_Report. This NPD represents the general operation of the TP. The specific operation of producing a current budget report is represented by the Cur_B_Report NPD which is granted the privileges of both the B_Report NPD and the Cur_Month NPD, which represents the CDI for this triple. The Cur_Month NPD is granted the appropriate select access on the tables and views that hold information about the current month. Thus the Cur_B_Report NPD represents the pairing of a TP with a CDI. The final component of the triple, the user, could be represented by an arc from Cur_B_Report to Alice. However, a structured approach to the design of the security configuration would be to explicitly represent the reason that Alice is allowed to produce current budget reports. For example, Alice might be a member of the executive staff and like all other staff members she is allowed to analyze current data. The NPD Exec_Staff represents this class of users. The arc from Cur_B_Report to Exec_Staff expresses a high-level policy that would otherwise be represented implicitly by a collection of CW triples, one for each member of the executive staff. Once again, NPDs are being used to directly represent a high-level security policy and thus improve the manageability of the security configuration.

Supporting the Bell and LaPadula Model

By themselves, NPDs cannot support the full Bell and LaPadula (BLP) security model. For example, this design does not include any labelling of newly created tables. Furthermore, the controls would have to be at the level of whole tables or whole columns, not individual rows or elements, because the underlying object privileges provided by SQL are at that level of granularity. However, it is interesting to look at the relationship between NPDs and BLP access classes and to consider the problem of information flow control.

An access class (e.g. Secret: Nato-Crypto) represents a collection of read and write privileges. If a subject's clearance (trustworthiness plus need-to-know requirements) dominates the classification of an access class (sensitivity plus topic labels), then the subject can read all the objects that are labeled with that access class. This is a form of privilege grouping. The subject is granted a collection of read privileges because he has been granted a particular clearance. A similar relationship exists for write access. When viewed this way, the grouping of privileges that exist in a particular BLP configuration can be translated into a matching NPD privilege graph.

The dynamics of the NPD and BLP models are different. The NPD graph must explicitly represent all the access classes that exist (or will exist), whereas the BLP model can dynamically create a new access class by combining existing category labels in new ways. This problem would not be encountered often because creating new objects (tables or views) is rare in most DBMS applications. When a new table is needed, a security administrator can add it and its new access class (if any) to the privilege graph. Another limitation is that all the objects that fit a particular access class must be named in the NPD graph. There are no labels which can be dynamically read from objects. The object's name determines its class. These restrictions mean that any specific BLP configuration can be represented, though small changes to the configuration (even creating a new object for an existing access class) require that the security administrator be involved. NPDs do not improve the manageability of BLP based security configurations.

NPDs can support mandatory control that cannot be changed at the discretion of ordinary users. In the BLP model an ordinary user cannot choose what he has access to, and likewise with NPDs, a user cannot choose which NPDs are granted to him. The administration of the security configuration can be segregated from ordinary users. These non-discretionary controls can even extend to the owners of tables. As mentioned earlier, an owner of a table can only grant access to that table if he has already been granted the

database privilege that allows him to execute the grant statement. Thus an NPD based security system could be set up to enforce a policy where all access rights are managed by an explicit set of security administrators. Ordinary users and even table owners would not be authorized to modify the security configuration by granting or revoking access to their tables.

Perhaps the most important feature of BLP is its ability to control information flow. It can guarantee that copies of information are protected just like the original. This is a very strong confinement property. The UserPriv NPD can help confine the flow of information, but only at the expense of restricting where copies can be made. For example, to control the flow of salary information, all the NPDs that grant read access to salaries must also control where the user may write when that access is enabled. Controlling write means controlling the insert, update and even delete operations on existing tables and controlling the creation of new tables. It is not necessary to forbid these operations, but any place where salary information could be written must be as well protected as the original salary information. Thus the whole burden of preventing undesired information flows is on the designer of the NPD privilege graph and the DBMS tables. The security system does not provide on-the-fly enforcement.

NPDs provide some of the characteristics of the BLP model, but they cannot fully support that model by themselves. Like access classes, NPDs provide a way to group and name collections of access rights. The NPD privilege graph and the allocation of database privileges provides a non-discretionary way to manage security and it is flexible enough to represent a variety of global security policies. The BLP global policy, which is based on the dominance relation, can be represented using two NPDs per access class (read and write access are separated). The major weakness of an NPD based implementation of BLP is the lack of explicit labels to solve the information flow problem. The NPD privilege graph can be analyzed to make sure that there are no undesirable information flows, but such flows cannot be prevented automatically.

Summary

This paper has described a simple extension to ANSI SQL that greatly improves the manageability and flexibility of DBMS security. The key idea is to allow the grouping and naming of privileges to form Named Protection Domains, NPDs. NPDs are an explicit representation of the protection domain concept [8] that is often used to explain the behavior of security systems that are based on access control lists. By allowing NPDs to be granted to users and to other NPDs, high-level abstractions can be formed that simplify the allocation of privileges to users. NPDs can even group users into classes according to the tasks they perform. The grant relationship between NPDs creates a directed acyclic graph that leads from low-level privileges to users. The privilege graph explicitly represents the reason why a privilege is granted to a user. Knowing the reason or purpose of an authorization makes it easier to understand, modify or verify the security policy that the database is enforcing.

NPDs can create flexible administrative roles. Two sites can support two different policies regarding the privileges available to auditors. These administrative roles are collections of low-level database privileges (like the ability to archive the audit trail, or the ability to change the kinds of events that generate an audit trail entry). The privilege grouping powers of NPDs can be used to define such collections. Each site can define collections of database privileges that form meaningful administrative duties and give these collections mnemonic names. This flexibility leads to higher operational security because each site can express its true security policy. The system of controls that exists outside of the computer can be dove-tailed to the controls inside the database. There are no gaps due to the problem of forcing the external controls to match the internal controls chosen by a DBMS vendor.

NPDs do not have the ownership and cascading change problems of the current ANSI SQL grant-revoke model. The administration of NPDs is completely separated from the allocation and grouping of privileges. The person in charge of security can be changed without the need for a complicated procedure of re-granting privileges. This solution avoids the loss of individual accountability that happens when a single account is shared by several people to do security management. There is no with-grant-option for NPDs and the with-grant-option on an object privilege cannot be granted to an NPD, so changes made to the NPD privilege graph do not cascade. Revoking an NPD only causes one localized change. This makes security management less difficult and less dangerous.

The activation model for NPDs allows a user to choose his effective protection domain, but the choice must be made from a limited set defined by the security administrators. Only one NPD can be activated at a time,

but the activation process enables the privileges of all the NPDs that are contained in the activated one. This is a cross between a restrictive security system that only allows a user to be in one group, and an open system that allows the user to enable an arbitrary set of groups. With NPDs, the security administrators choose which sets of NPDs correspond to meaningful protection domains. These sets are represented by subtrees in the NPD privilege graph and the roots of these subtrees are specially marked NPDs. Only the marked NPDs can be activated.

NPDs can solve application-oriented security problems if the invocation of programs is linked to the activation of NPDs. The set of privileges needed to run an application can be grouped together into an NPD, and those privileges get enabled when (and perhaps only when) the application program is invoked. This feature allows a site to enforce complex integrity constraints using trusted applications. The set of privileges available to a user when he executes ad hoc SQL statements can also be carefully controlled so he cannot violated the integrity constraints enforced by the trusted applications. The security system limits what can be done by the ad hoc statements by limiting which NPDs can be activated by these statements.

NPDs provide a general way to create named abstractions in a security system. Just as procedures can simplify programming, NPDs can simplify security management.

Acknowledgements

Several people at the Oracle Corporation helped turn the author's ideas into a commercial product: Ken Jacobs, Andy Larson, Bill Maimone, Mark Moore, Gordon Smith, and Linda Vetter.

References

1. Bell,D.E and LaPadula,L.J. "Secure Computer Systems: Mathematical Foundations", MTR-2547, Vol 2, MITRE Corp., Bedford MA, November 1973.

2. Clark,D.D. and Wilson,D.R., "A Comparison of Commercial and Military Computer Security Policies", Proc. 1987 IEEE Symposium on Security and Privacy, April 1987.

3. Date,C.J., "An Introduction to Database Systems", Vol. 1, Addison-Wesley, Reading MA, 1986.

4. Department of Defense Computer Security Center, "Trusted Computer System Evaluation Criteria", DOD 5200.28-STD, December 1985.

5. Downs,D.D., Rub,J.R., Kung, K.C., and Jordan,C.S., "Issues in Discretionary Access Control", Proc. IEEE Symposium on Security and Privacy, 1985.

6. Griffiths,P.P. and Wade,B.W., "An Authorization Mechanism for a Relational Database System", ACM TODS vol. 1, no. 3, September 1976.

7. Lampson, B.W., "Protection", Proc. Fifth Princeton Symposium on Information Sciences and Systems, March 1971. Reprinted in Operating Systems Review vol. 8, no. 1, January 1974.

8. Saltzer,J.H. and Schroeder,M.D., "The Protection of Information in Computer Systems", Proc. of IEEE vol. 63, no. 9, September 1975.