Privilege States Based Access Control for Fine-Grained Intrusion Response

Ashish Kamra[1] and Elisa Bertino[2]

[1] School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA
akamra@purdue.edu

[2] School of Computer Science, Purdue University, West Lafayette, IN, USA
bertino@cs.purdue.edu

Abstract. We propose an access control model specifically developed to support fine-grained response actions, such as request suspension and request tainting, in the context of an anomaly detection system for databases. To achieve such response semantics, the model introduces the concept of *privilege states* and *orientation modes* in the context of a role-based access control system. The central idea in our model is that privileges, assigned to a user or role, have a state attached to them, thereby resulting in a *privilege states based access control* (PSAC) system. In this paper, we present the design details and a formal model of PSAC tailored to database management systems (DBMSs). PSAC has been designed to also take into account role hierarchies that are often present in the access control models of current DBMSs. We have implemented PSAC in the PostgreSQL DBMS and in the paper, we discuss relevant implementation issues. We also report experimental results concerning the overhead of the access control enforcement in PSAC. Such results confirm that our design and algorithms are very efficient.

1 Motivation

An access control mechanism is typically based on the notion of authorizations. An authorization is traditionally characterized by a three-element tuple of the form $< A, R, P >$ where A is the set of permissible actions, R is the set of protected resources, and P is the set of principals. When a principal tries to access a protected resource, the access control mechanism checks the rights (or privileges) of the principal against the set of authorizations in order to decide whether to allow or deny the access request.

The main goal of this work is to extend the decision semantics of an access control system beyond the *all-or-nothing* allow or deny decisions. Specifically, we provide support for more *fine-grained* decisions of the following two forms: *suspend*, wherein further negotiation (such as a second factor of authentication) occurs with the principal before deciding to allow or deny the request, and *taint*, that allows one to audit the request in-progress, thus resulting in further monitoring of the principal, and possibly in the suspension or dropping of subsequent requests by the same principal. The main motivation for proposing such fine-grained access check decisions is to provide system

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 402–421, 2010.
© Springer-Verlag Berlin Heidelberg 2010

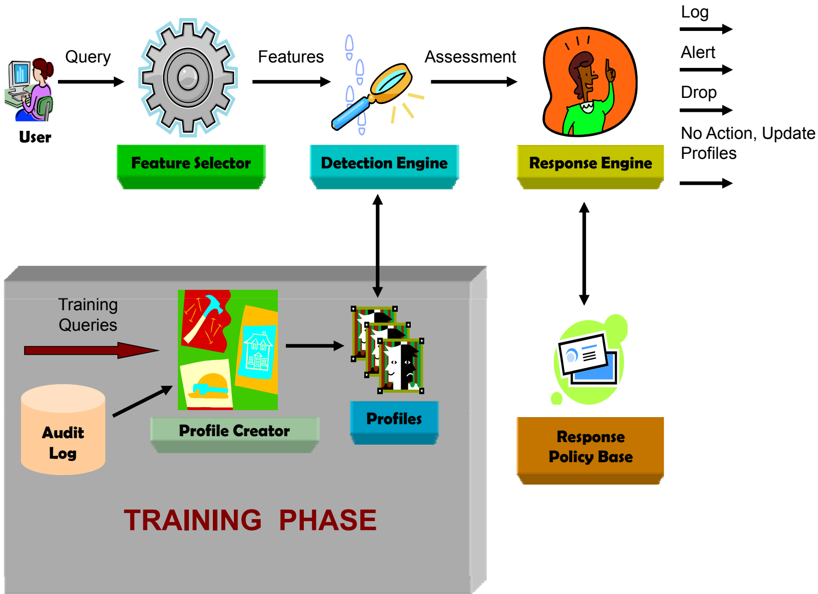


Fig. 1. Anomaly Detection and Response System Architecture

support for extending the response action semantics of an application level anomaly detection (AD) system that detects the anomalous patterns of requests submitted to it.

Consider the architecture of a database specific AD mechanism using fine-grained response actions as shown in Figure 1 [8,15,14]. The system consists of three main components: the traditional database server that handles the query execution, the profile creator module for creating user profiles from the training data, and the anomaly detection and response mechanisms integrated with the core database functionality. The flow of interactions for the anomaly detection and response process is as follows: During the training phase, the SQL commands submitted to the database (or read from the audit log) are analyzed by the profile creator module to create the initial profiles of the database users. In the detection phase, for every SQL command under detection, the feature selector module extracts the features from the queries in the format expected by the detection engine. The detection engine then runs the extracted features through the detection algorithm. If an anomaly detected, the detection mechanism submits its assessment of the SQL command to the response engine according to a pre-defined interface; otherwise the command information is sent to the profile creator process for updating the profiles.

The response engine consults a base of response policies to issue a suitable response action depending on the assessment of the anomalous query submitted by the detection engine. The system supports three types of response actions, that we refer to respectively as conservative actions, fine-grained actions, and aggressive actions. The conservative actions, such as sending an alert, allow the anomalous request to go through, whereas the aggressive actions can effectively block the anomalous request.

Fine-grained response actions, supported by the extended decision semantics of our access control mechanism, are neither conservative nor aggressive. Such actions may result in either request suspension (supported by the *suspend* decision semantics) and request tainting (supported by the *taint* decision semantics).

Why do we need to extend the access control mechanism to support such response actions? Certainly, such responses may also be issued by an AD mechanism working independently of the underlying access control system. The usefulness of our approach is evident from the following scenario. Suppose we model a user request as the usage of a set of *privileges* in the system where a privilege is defined as an operation on a resource. For example, the SQL query '*SELECT * FROM orders, parts*' is modeled as using the privileges {select,orders} and {select,parts} in the context of a database management system (DBMS). After detecting such request as anomalous (using any anomaly detection algorithm), consider that we want to re-authenticate the user and drop the request in case the re-authentication procedure fails. Suppose that every time a similar request is detected to be anomalous, we want the same re-authentication procedure to be repeated. If our response mechanism does not *remember* the requests, then the request will always undergo the detection procedure, detected to be anomalous and then submitted to the response mechanism to trigger the re-authentication procedure. A more generic and flexible approach for achieving such response semantics is to *attach a suspend state to the privileges* associated with the anomalous request. Then for every subsequent similar request (that uses the same set of privileges as the earlier request that was detected to be anomalous), the semantics of the privilege in the *suspend* state automatically triggers the re-authentication sequence of actions for the request under consideration without the request being subjected to the detection mechanism. Moreover, if the system is set-up such that the request is always subjected to the detection mechanism (in case access control enforcement is performed after the intrusion detection task), more advanced response logic can be built based on the fact that a request is detected to be anomalous whose privileges are already in the *suspend* state.

In addition to supporting fine-grained intrusion response, manually moving a privilege to the *suspend* state (using administrative commands) provides the basis for an event based continuous authentication mechanism. Similar arguments can be made for attaching the *taint* state to a privilege that triggers auditing of the request in progress. Since we extend the decision semantics of our access control system using privilege states, we call it a *privilege state based access control* (PSAC) system. For the completeness of the access control decisions, a privilege, assigned to a user or role, in PSAC can exist in the following five states: *unassign*, *grant*, *taint*, *suspend*, and *deny*. The privilege states, the state transition semantics and a formal model of PSAC are described in detail in Section 2. Note that the PSAC model that we present in Section 2 is flexible enough to allow more than the above mentioned five states.

We have developed PSAC in the context of a role based access control (RBAC) system [18]. Extending PSAC with roles presents the main challenge of *state conflict resolution*, that is, deciding on the final state of a privilege when a principal receives the same privilege in different states from other principals. Moreover, additional complexity is introduced when the roles are arranged in a hierarchy where the roles higher-up in the

hierarchy inherit the privileges of the lower level roles. We present precise semantics in PSAC to deal with such scenarios.

The main contributions of this paper can be summarized as follows:

1. We present the design details, and a formal model of PSAC in the context of a DBMS.
2. We extend the PSAC semantics to take into account a role hierarchy.
3. We implement PSAC in the PostgreSQL DBMS [5] and discuss relevant design issues.
4. We conduct an experimental evaluation of the access control enforcement overhead introduced by the maintenance of privilege states in PSAC, and show that our implementation design is very efficient.

The rest of the paper is organized as follows. Section 2 presents the details of PSAC and its formal model; it also discusses how a role hierarchy is supported. Section 3 presents the details of the system implemented in PostgreSQL, and the experimental results concerning the overhead introduced by the privilege states on the access control functions. Section 4 discusses the related work in this area. We conclude the paper in Section 5.

2 PSAC Design and Formal Model

In this section, we introduce the design and the formal model underlying PSAC. We assume that the authorization model also supports roles, in that RBAC is widely used by access control systems of current DBMSs [11,4,7]. In what follows, we first introduce the privilege state semantics and state transitions. We then discuss in detail how those notions have to be extended when dealing with role hierarchies.

2.1 Privilege States Dominance Relationship

PSAC supports five different privilege states that are listed in Table 1. For each state, the table describes the semantics in terms of the result of an access check.

A privilege in the *unassign* state is equivalent to the privilege not being assigned to a principal; and a privilege in the *grant* state is equivalent to the privilege being

Table 1. Privilege States

State	Access Check Result Semantics
unassign	The access to the resource is not granted.
grant	The access to the resource is granted.
taint	The access to the resource is granted; the system audits access to the resource.
suspend	The access to the resource is not granted until further negotiation with the principal is satisfied.
deny	The access to the resource is not granted.

granted to a principal. We include the *deny* state in our model to support the concept of negative authorizations in which a privilege is specifically denied to a principal [9]. The *suspend* and the *taint* states support the fine-grained decision semantics for the result of an access check.

In most DBMSs, there are two distinct ways according to which a user/role[1] can obtain a privilege p on a database object o:

1. **Role-assignment:** the user/role is assigned a role that has been assigned p;
2. **Discretionary:** the user is the owner of o; or the user/role is assigned p by another user/role that has been assigned p with the GRANT option[2].

Because of the multiple ways by which a privilege can be obtained, conflicts are natural in cases where the same privilege, obtained from multiple sources, exists in different states. Therefore, a *conflict resolution* strategy must be defined to address such cases. Our strategy is to introduce a *privilege states dominance* (PSD) relation (see Figure 2). The PSD relation imposes a total order on the set of privilege states such that any two states are comparable under the PSD relation. Note the following characteristics of the semantics of the PSD relation. First, the *deny* state overrides all the other states to support the concept of a negative authorization [9]. Second, the *suspend*, and the *taint* states override the *grant* state as they can be triggered as potential response actions to an anomalous request. Finally, the *unassign* state is overridden by all the other states thereby preserving the traditional semantics of privilege assignment.

The PSD relation is the core mechanism that PSAC provides for resolving conflicts. For example, consider a user u that derives its privileges by being assigned a role r. Suppose that a privilege p is assigned to r in the *grant* state. Now suppose we directly deny p to u. The question is which is the state of privilege p for u, in that u has received p with two different states. We resolve such conflicts in PSAC using the PSD relation. Because in the PSD relation, the *deny* state overrides the *grant* state, p is denied to u.

We formally define a PSD relation as follows:

Definition 1. (**PSD Relation**) Let n be the number of privilege states. Let $S = \{s_1, s_2 \ldots s_n\}$ be the set of privilege states. The PSD relation is a binary relation (denoted by $\preceq$) on S such that for all $s_i, s_j, s_k \in S$:

1. $s_i \preceq s_j$ means s_i overrides s_j
2. if $s_i \preceq s_j$ and $s_j \preceq s_i$, then $s_i = s_j$ (anti-symmetry)
3. if $s_i \preceq s_j$ and $s_j \preceq s_k$, then $s_i \preceq s_k$ (transitivity)
4. $s_i \preceq s_j$ or $s_j \preceq s_i$ (totality) □

2.2 Privilege State Transitions

We now turn our attention to the privilege state transitions in PSAC. Initially, when a privilege is not assigned to a principal, it is in the *unassign* state for that principal. Thus,

[1] From here on, we use the terms *principal* and *user/role* interchangeably.

[2] A privilege granted to a principal with the GRANT option allows the principal to grant that privilege to other principals [2].

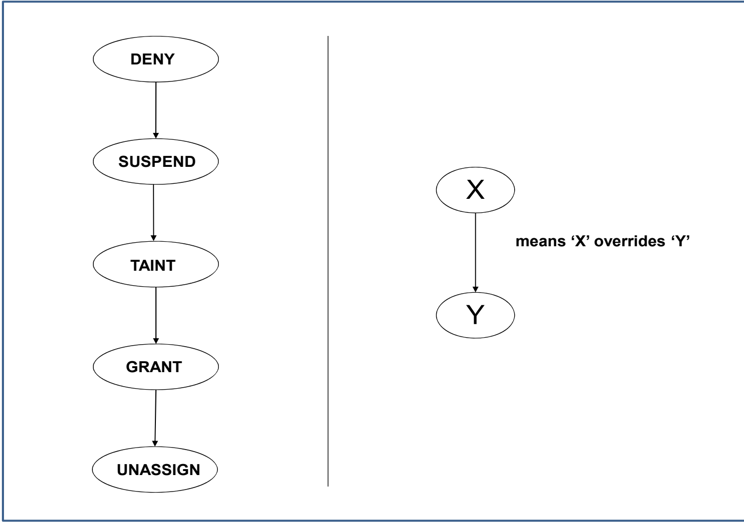


Fig. 2. Privilege States Dominance Relationship

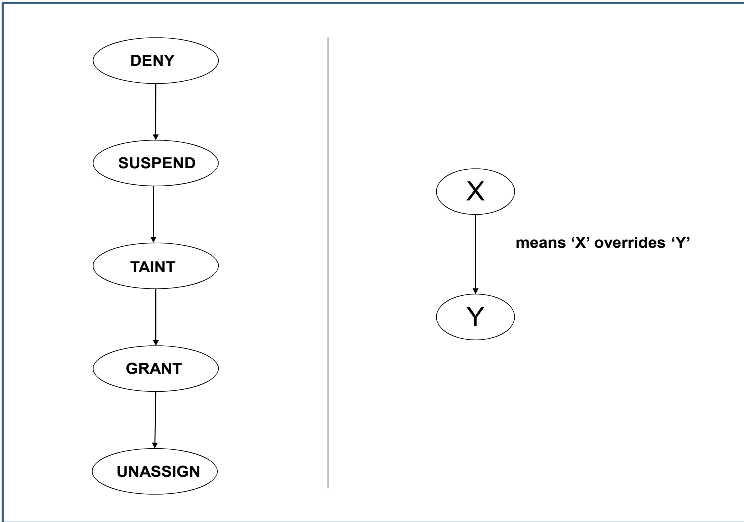


Fig. 3. Privilege State Transitions

the *unassign* state is the default (or initial) state of a privilege. The state transitions can be triggered as internal response actions by an AD system, or as ad-hoc administrative commands. In what follows, we discuss the various administrative commands available in PSAC to trigger privilege state transitions.

The GRANT command is used to assign a privilege to a principal in the *grant* state whereas the REVOKE command is used to assign a privilege to a principal in the *unassign* state. In this sense, these commands support similar functionality as the SQL-99 GRANT and REVOKE commands [2]. The DENY command assigns a privilege to a principal in the *deny* state. We introduce two new commands in PSAC namely, SUSPEND and TAINT, for assigning a privilege to a principal in the *suspend* and the *taint* states, respectively. The privilege state transitions are summarized in Figure 3. Note the constraint that a privilege assigned to a principal on a DBMS object can only exist in one state at any given point in time.

2.3 Formal Model

In this section, we formally define the privilege model for PSAC in the context of a DBMS. The model is based on the following relations and functions:

Relations

1. U, the set of all users in the DBMS.
2. R, the set of all roles in the DBMS.
3. $PR = U \cup R$, the set of principals (users/roles) in the DBMS.
4. OT, the set of all DBMS object types such as *server*, *database*, *schema*, *table*, and so forth.
5. O, the set of all DBMS objects of all object types.
6. OP, the set of all operations defined on the object types in OT, such as *select*, *insert*, *delete*, *drop*, *backup*, *disconnect*, and so forth.
7. $S = \{$*deny,suspend,taint,grant,unassign*$\}$, a totally ordered set of privilege states under the PSD relation (Definition 2.1).
8. $P \subset OP \times OT$, a many-to-many relation on operations and object types representing the set of all privileges. Note that not all operations are defined for all object types. For example, tuples of the form $(select, server)$ or $(drop, server)$ are not elements of P.
9. $URA \subseteq U \times R$, a many-to-many user to role assignment relation.
10. $PRUPOSA \subset PR \times U \times P \times O \times S$, a principal to user to privilege to object to state assignment relation. This relation captures the state of the access control mechanism in terms of the privileges, and their states, that are directly assigned to users (assignees) by other principals (assigners) on DBMS objects[3].
11. $PRRPOSA \subset PR \times R \times P \times O \times S$, a principals to role to privilege to object to state assignment relation. This relation captures the state of the access control mechanism in terms of the privileges, and their states, that are directly assigned to roles (assignees) by principals (assigners).

[3] In PSAC, a role can also be an assigner of privileges. Consider a situation when a user u gets a privilege p (with grant option) through assignment of role r. If u grants p to some other user u', PSAC records p as being granted to u' by r even though the actual GRANT command was executed by u.

These relations capture the state of the access control system in terms of the privilege and the role assignments. The functions defined below determine the state of a privilege assigned to a user/role on a DBMS object.

Functions

1. $assigned_roles(u) : U \rightarrow 2^R$, a function mapping a user u to its assigned roles such that $assigned_roles(u) = \{r \in R \mid (u, r) \in URA\}$. This function returns the set of roles that are assigned to a user.
2. $priv_states(pr, r', p, o) : PR \times R \times P \times O \rightarrow 2^S$, a function mapping a principal pr (privilege assigner), a role r', a privilege p, and an object o to a set of privilege states such that $priv_states(pr, r', p, o) = \{s \in S \mid (pr, r', p, o, s) \in PRRPOSA\}$. This function returns the set of states for a privilege p, that is directly assigned to the role r' by the principal pr, on an object o.
3. $priv_states(pr, u', p, o) : PR \times U \times P \times O \rightarrow 2^S$, a function mapping a principal pr (privilege assigner), a user u', a privilege p, and an object o to a set of privilege states such that $priv_states(pr, u', p, o) = \{s \in S \mid (pr, u', p, o, s) \in PRUPOSA\} \cup_{r \in assigned_roles(u')} priv_states(pr, r, p, o)$. The set of states returned by this function is the union of the privilege state directly assigned to the user u' by the principal pr, and the privilege states (also assigned by pr) obtained through the roles assigned to u'.
4. $priv_states(r, p, o) : R \times P \times O \rightarrow 2^S$, a function mapping a role r, a privilege p, and an object o to a set of privilege states such that $priv_states(r, p, o) = \cup_{pr \in PR}$ $priv_states(pr, r, p, o)$. This function returns the set of states for a privilege p, that is directly assigned to the role r by any principal in the DBMS, on an object o.
5. $priv_states(u', p, o) : U \times P \times O \rightarrow 2^S$, a function mapping a user u', a privilege p, and an object o to a set of privilege states such that $priv_states(u', p, o)$ $= \cup_{pr \in PR}$ $priv_states(pr, u', p, o)$. This function returns the set of states for a privilege p, that is directly assigned to the user u' by any principal in the DBMS, on an object o.
6. $PSD_state(2^S) : 2^S \rightarrow S$, a function mapping a set of states 2^S to a state $\in S$ such that $PSD_state(2^S) = s' \in 2^S \mid \forall_{s \in 2^S | s \neq s'}\ s' \preceq s$. This function returns the final state of a privilege using the PSD relation.

2.4 Role Hierarchy

Traditionally, roles can be arranged in a conceptual hierarchy using the role-to-role assignment relation. For example, if a role r_2 is assigned to a role r_1, then r_1 becomes a parent of r_2 in the conceptual role hierarchy. Such hierarchy signifies that the role r_1 inherits the privileges of the role r_2 and thus, is a more *privileged* role then r_2. However, in PSAC such privilege inheritance semantics may create a problem because of a $deny/suspend/taint$ state attached to a privilege. The problem is as follows. Suppose a privilege p is assigned to the role r_2 in the *deny* state. The role r_1 will also have such privilege in the *deny* state since it inherits it from the role r_2. Thus, denying

a privilege to a lower level role has the affect of denying that privilege to all roles that inherit from that role. This defeats the purpose of maintaining a role hierarchy in which roles higher up the hierarchy are supposed to be more privileged than the descendant roles. To address this issue, we introduce the concept of *privilege orientation*. We define three privilege orientation modes namely, *up, down*, and *neutral*. A privilege assigned to a role in the *up* orientation mode means that the privilege is also assigned to its parent roles. On the other hand, a privilege assigned to a role in the *down* orientation mode means that the privilege is also assigned to its children roles; while the *neutral* orientation mode implies that the privilege is neither assigned to the parent roles nor to the children roles. We put the following two constraints on the assignment of orientation modes on the privileges.

- A privilege assigned to a role in the *grant* or in the *unassign* state is always in the *up* orientation mode thereby maintaining the traditional privilege inheritance semantics in a role hierarchy.
- A privilege assigned to a role in the *deny, taint*, or *suspend* state may only be in the *down* or in the *neutral* orientation mode. Assigning such privilege states to a role in the *down* or *neutral* mode ensures that the role still remains more privileged than its children roles. In addition, the *neutral* mode is particularly useful when a privilege needs to be assigned to a role without affecting the rest of the role hierarchy (when responding to an anomaly, for example).

We formalize the privilege model of PSAC in the presence of a role hierarchy as follows:

1. $RRA \subset R \times R$, a many-to-many role to role assignment relation. A tuple of the form $(r_1, r_2) \in R \times R$ means that the role r_2 is assigned to the role r_1. Thus, role r_1 is a parent of role r_2 in the conceptual role hierarchy.
2. $OR = \{up, down, neutral\}$, the set of privilege orientation modes.
3. $PRRPOSORA \subset PR \times R \times P \times O \times S \times OR$, a principal to role to privilege to object to state to orientation mode assignment relation. This relation captures the state of the access control system in terms of the privileges, their states, and their orientation modes that are directly assigned to roles by principals.
4. $assigned_roles(r') : R \rightarrow 2^R$, a function mapping a role r' to its assigned roles such that $assigned_roles(r') = \{r \in R \mid (r', r) \in RRA\} \cup assigned_roles(r)$. This function returns the set of the roles that are directly and indirectly (through the role hierarchy) assigned to a role; in other words, the set of descendant roles of a role in the role hierarchy.
5. $assigned_roles(u) : U \rightarrow 2^R$, a function mapping a user u to its assigned roles such that $assigned_roles(u) = \{r \in R \mid (u, r) \in URA\} \cup assigned_roles(r)$. This function returns the set of roles that are directly and indirectly (through the role hierarchy) assigned to a user.
6. $assigned_to_roles(r') : R \rightarrow 2^R$, a function mapping a role r' to a set of roles such that $assigned_to_roles(r') = \{r \in R \mid (r, r') \in RRA\} \cup assigned_to_roles(r)$. This function returns the set of roles that a role is directly and indirectly (through

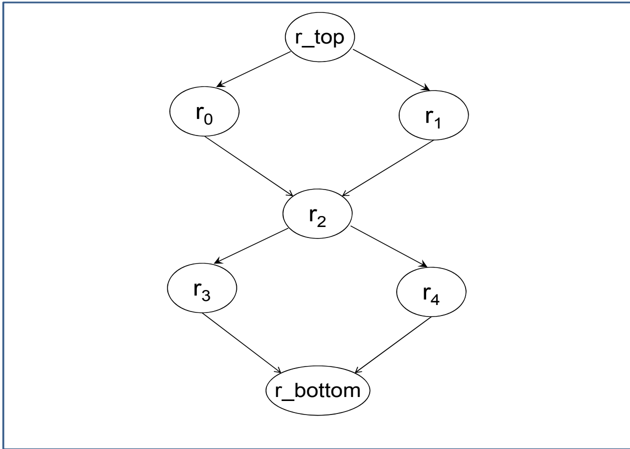


Fig. 4. A Sample Role Hierarchy

the role hierarchy) assigned to; in other words, the set of ancestor roles of a role in the role hierarchy.

We redefine the $priv_states(pr, r', p, o)$ function in the presence of a role hierarchy taking into account the privilege orientation constraints as follows:

7. $priv_states(pr, r', p, o)$: $PR \times R \times P \times O \rightarrow 2^S$, a function mapping a principal pr, a role r', a privilege s, and an object o to a set of privilege states such that $priv_states(pr, r', p, o) = \{s \in S \mid \forall\ or \in OR,\ (pr, r', p, o, s, or) \in PRRPOSORA\ \} \cup \{s \in \{grant, unassign\} \mid \forall\ r \in assigned_roles(r'),\ (pr, r, p, o, s, `up') \in PRRPOSORA\ \} \cup \{s \in \{deny, suspend, taint\} \mid \forall\ r \in assigned_to_roles(r'),\ (pr, r, p, o, s, `down') \in PRRPOSORA\ \}$. The set of privilege states returned by this function is the union of the privilege states directly assigned to the role r' by the principal pr, the privilege states in the *grant* or the *unassign* states (also assigned by pr) obtained through the descendant roles of r', and the privilege states in the *deny*, *suspend*, and *taint* states (also assigned by pr) obtained through the roles that are the ancestor roles of r', and that are in the *down* orientation mode.

We now present a comprehensive example of the above introduced relations and functions in PSAC. Consider a sample role hierarchy in Figure 4. Table 2 shows the state of a sample $PRRPOSORA$ relation.

Table 2. $PRRPOSORA$ relation

PR	R	P	O	S	OR
SU_1	r_top	$select$	t_1	$deny$	$neutral$
SU_1	r_0	$select$	t_1	$taint$	$down$
SU_1	r_bottom	$select$	t_1	$grant$	up
SU_2	r_top	$select$	t_1	$suspend$	$down$

Let the role r_2 be assigned to the user u_1. To determine the final state of the $select$ privilege on the table t_1 for the user u_1, we evaluate $priv_states(u_1, select, t_1)$ as follows:

$$
\begin{aligned}
& priv_states(u_1, select, t_1) \\
& = priv_states(SU_1, u_1, select, t_1) \cup \\
& \quad priv_states(SU_2, u_1, select, t_1) \\
& = priv_states(SU_1, r_2, select, t_1) \cup \\
& \quad priv_states(SU_2, r_2, select, t_1) \\
& = \{taint\} \cup \\
& \quad \{grant\} \cup \{suspend\} \\
& = \{taint, grant, suspend\}
\end{aligned}
$$

The final state is determined using the $PSD_state()$ function as follows:

$$PSD_state(taint, grant, suspend) = suspend$$

3 Implementation and Experiments

In this section, we present the details on how to extend a real-world DBMS with PSAC. We choose to implement PSAC in the PostgreSQL 8.3 open-source object-relational DBMS [5]. In the rest of the section, we use the term PSAC:PostgreSQL to indicate PostgreSQL extended with PSAC, and BASE:PostgreSQL to indicate the official PostgreSQL 8.3 release. The implementation of PSAC:PostgreSQL has to meet two design requirements. The first requirement is to maintain backward compatibility of PSAC:PostgreSQL with BASE:PostgreSQL. We intend to release PSAC:PostgreSQL for general public use in the near future; therefore it is important to take into account the backward compatibility issues in our design. The second requirement is to minimize the overhead for maintaining privilege states in the access control mechanism. In particular, we show that the time taken for the access control enforcement code in the presence of privilege states is not much higher than the time required by the access control mechanism of BASE:PostgreSQL. In what follows, we first present the design details of PSAC:PostgreSQL, and then we report experimental results showing the efficiency of our design.

3.1 PSAC:PostgreSQL

Access control in BASE:PostgreSQL is enforced using access control lists (ACLs). Every DBMS object has an ACL associated with it. An ACL in BASE:PostgreSQL is a one-dimensional array; the elements of such an array have values of the *ACLItem* data type. An ACLItem is the basic unit for managing privileges of an object. An ACLItem is implemented as a structure with the following fields: *granter*, the user/role granting the privileges; *grantee*, the user/role to which the privileges are granted; and *privs*, a 32 bit integer (on 32 bit machines) managed as a bit-vector to indicate the privileges granted

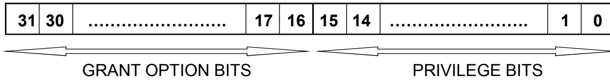


Fig. 5. ACLItem privs field

to the grantee. A new ACLItem is created for every unique pair of granter and grantee. There are 11 pre-defined privileges in BASE:PostgreSQL with a bit-mask associated with each privilege [6]. As shown in Figure 5, the lower 16 bits of the privs field are used to represent the granted privileges, while the upper 16 are used to indicate the $grant$ option[4]. If the kth bit is set to 1 ($0 \leq k < 15$), privilege p_k is granted to the user/role. If the $(k + 16)^{th}$ bit is also set to 1, then the user/role has the grant option on privilege p_k.

Design Details. There are two design options to extend BASE:PostgreSQL to support privilege states. The first option is to extend the ACLItem structure to accommodate privilege states. The second option is to maintain the privilege states in a separate data structure. We chose the latter option. The main reason is that we want to maintain backward compatibility with BASE:PostgreSQL. Extending the existing data structures can introduce potential bugs at other places in the code base that we want to avoid. In BASE:PostgreSQL, the pg_class system catalog is used to store the metadata information for database objects such as tables, views, indexes and sequences. This catalog also stores the ACL for an object in the *acl* column that is an array of ACLItems. We extend the pg_class system catalog to maintain privilege states by adding four new columns namely: the *acltaint* column to maintain the tainted privileges; the *aclsuspend* column to maintain the suspended privileges; the *acldeny* column to maintain the denied privileges; and the *aclneut* column to indicate if the privilege is in the *neutral* orientation mode. Those state columns and the *aclneut* column are of the same data type as the *acl* column, that is, an array of ACLItems. The lower 16 bits of the privs field in those state and *aclneut* columns are used to indicate the privilege states and the orientation mode respectively. This strategy allows us to use the existing privilege bit-masks for retrieving the privilege state and orientation mode from these columns. The upper 16 bits are kept unused. Table 3 is the truth table capturing the semantics of the privs field bit-vector in PSAC:PostgreSQL.

Authorization Commands. We have modified the BASE:PostgreSQL GRANT and REVOKE authorization commands to implement the privilege state transitions. In addition, we have defined and implemented in PSAC:PostgreSQL three new authorization commands, that is, the DENY, the SUSPEND, and the TAINT commands. As discussed in the Section 2, the DENY command moves a privilege to the *deny* state, the SUSPEND command moves a privilege to the *suspend* state, and the TAINT command moves a privilege to the *taint* state. The default privilege orientation mode for these

[4] Recall that the grant option is used to indicate that the granted privilege may be granted by the grantee to other users/roles.

Table 3. Privilege States/Orientation Mode Truth Table for the privs Field in PSAC:PostgreSQL

acl k^{th} bit	**acl taint** k^{th} bit	**acl suspend** k^{th} bit	**acl deny** k^{th} bit	**acl neut** k^{th} bit	p_k **state**
0	0	0	0	0	unassign/up
1	0	0	0	0	grant/up
0	1	0	0	0	taint/down
0	0	1	0	0	suspend/down
0	0	0	1	0	deny/down
0	1	0	0	1	taint/neutral
0	0	1	0	1	suspend/neutral
0	0	0	1	1	deny/neutral
Rest all other combinations are not allowed by the system.					

Table 4. New Authorization Commands in PSAC:PostgreSQL

TAINT {privilege name(s) | ALL} ON {object name(s)} TO {user/role name(s) | PUBLIC} [NEUT ORNT]
SUSPEND {privilege name(s) | ALL} ON {object name(s)} TO {user/role name(s) | PUBLIC} [NEUT ORNT]
DENY {privilege name(s) | ALL} ON {object name(s)} TO {user/role name(s) | PUBLIC} [NEUT ORNT]

commands is the *down* mode with the option to override that by specifying the *neutral* orientation mode. The administrative model for these commands is similar to that of the SQL-99 GRANT command, that is, a DENY/SUSPEND/TAINT command can be executed on privilege p for object o by a user u iff u has the grant option set on p for o or u is the owner of o. The syntax for the commands is reported in Table 4. Note that in the current version of PSAC:PostgreSQL, the new commands are applicable on the database objects whose metadata are stored in the pg_class system catalog.

Access Control Enforcement. We have instrumented the access control enforcement code in BASE:PostgreSQL with the logic for maintaining the privilege states and orientation modes. The core access control function in BASE:PostgreSQL returns a true/false output depending on whether the privilege under check is granted to the user or not. In contrast, the core access control function in PSAC:PostgreSQL returns the final state of the privilege to the calling function. The calling function then executes a pre-configured action depending upon the state of the privilege. As a proof of concept, we have implemented a re-authentication procedure in PSAC:PostgreSQL when a privilege is in the *suspend* state. The re-authentication procedure is as follows:

Re-authentication Procedure. Recall that when a privilege is in the $suspend$ state, further negotiation with the end-user must be satisfied before the user-request is executed by the DBMS. In the current version of PSAC, we implement a procedure that re-authenticates the user if a privilege, after applying the PSD relationship, is found in

the *suspend* state. The re-authentication scheme is as follows. In BASE:PostgreSQL, an authentication protocol is carried out with the user whenever a new session is established between a client program and the PostgreSQL server. In PSAC:Postgresql, the same authentication protocol is replayed in the middle of a transaction execution when access control enforcement is in progress, and a privilege is found in the *suspend* state. We have modified the client library functions of BASE:PostgreSQL to implement such protocol in the middle of a transaction execution. If the re-authentication protocol fails, the user request is dropped. If it succeeds, the request proceeds as usual, and no changes are made to the state of the privilege. Note that such re-authentication procedure scheme is implemented as a proof-of-concept in PSAC:Postgresql. More advanced forms of actions such as a second-factor of authentication can also be implemented.

Access Control Enforcement Algorithm. The pseudo-code for the access control enforcement algorithm in PSAC:PostgreSQL is presented in the Listing 1. The function *aclcheck()* takes as input a privilege in_priv - whose state needs to be determined, a database object in_object - that is the target of a request, and a user in_user - the user exercising the usage of in_priv. The output of the algorithm is the state of the in_priv. The algorithm proceeds as follows. Since we define a total order on the privilege states, it is sufficient to check each state in the order of its rank in the PSD relation (cfr. Section 2). Thus, we first check for the existence of in_priv in the *deny* state, followed by the *suspend* state, the *taint* state, and then the *grant* state. The function for checking the state of in_priv (function *check_priv()*) in an Acl is designed to take into account all the roles that are directly and indirectly (through a role hierarchy) assigned to the in_user. Note that most expensive operation in the *check_priv()* function is the run-time inheritance check of roles, that is, to check whether the *user_role* is an ancestor or descendant of the *acl_role* (lines 58 and 62). We make such check a constant time operation in our implementation by maintaining a cache of the assigned roles for every user/role in the DBMS. Thus, the running time of the access control enforcement algorithm is primarily dependent upon the sizes of various Acls.

If the privilege is not found to be in the above mentioned states, the *unassign* state is returned as the output of the access check algorithm.

```
1 ----------------------------------------------------------------
2 Input
3 in_user   : The user executing the command
4 in_object : Target database object
5 in_priv   : Privilege to check
6
7 Output
8 The privilege state
9 ----------------------------------------------------------------
10 function aclcheck(in_user, in_object, in_priv) returns state
11 {
12   //Get the neutral orientation ACL for in_object
13   NeutACL = get_neut_ornt(in_object);
```

```
14
15   //Deny if in_user has in_priv in DENY state
16   DenyACL = get_deny_state_acl(in_object);
17   if (check_priv(in_priv,DenyACL,in_user,NeutACL,DENY) == true)
18       return DENY;
19
20   //Suspend if in_user has in_priv in SUSPEND state
21   SuspendACL = get_suspend_state_acl(in_object);
22   if (check_priv(in_priv,SuspendACL,in_user,NeutACL,SUSPEND) ==
        true)
23       return SUSPEND;
24
25   //Taint if in_user has in_priv in TAINT state
26   TaintACL = get_taint_state_acl(in_object);
27   if (check_priv(in_priv,TaintACL,in_user,NeutACL,TAINT) == true
        )
28      return TAINT;
29
30   //Grant if in_user has in_priv in GRANT state
31   GrantACL = get_grant_state_acl(in_object);
32   if (check_priv(in_priv,GrantACL,in_user,NeutACL,GRANT) == true
        )
33      return GRANT;
34
35   //Else return UNASSIGN state
36   return UNASSIGN;
37 }
38 ----------------------------------------------------------------
39 function check_priv(in_priv,AclToCheck,in_user,NeutACL,
      state_to_check)
40 returns boolean
41 {
42   //First, perform the inexpensive step of checking the
        privileges directly assigned to the in_user
43   if (in_user has in_priv in AclToCheck)
44     return true;
45
46   //Get all the roles directly assigned to in_user
47   user_role_list = get_roles(in_user);
48
49   //Do the following for every role directly assigned to in_user
50   for each user_role in user_role_list
51   {
52     //Do the following for every role entry in AclToCheck
53     for each acl_role in AclToCheck
54     {
55       if (state_to_check == GRANT)
56       {
```

```
57        //Orientation of privileges in GRANT state is UP
58        if ((user_role == acl_role OR user_role is an ANCESTOR
              of acl_role) AND acl_role has in_priv)
59
60          return true;
61      }
62      else if ((user_role == acl_role OR user_role is a
            DESCENDANT of acl_role) AND acl_role has in_priv)
63      {
64        if (acl_role has in_priv in NeutACL)
65          continue looping through AclToCheck;
66        else
67          return true;
68      }
69    }
70  }
71
72  return false;
73 }
```

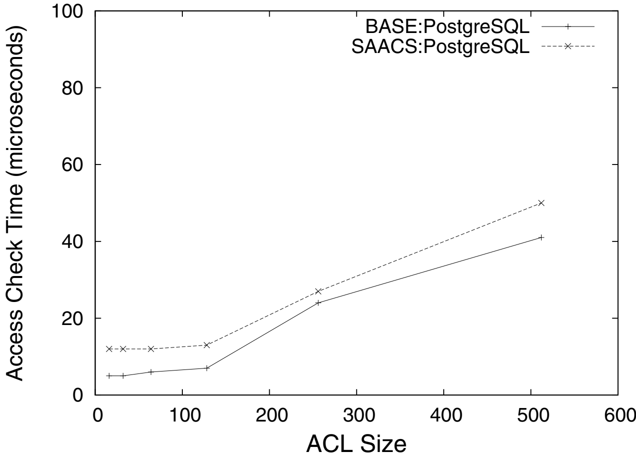
Lising 1. Access Control Enforcement Algorithm in PSAC:PostgreSQL

3.2 Experimental Results

In this section, we report the experimental results comparing the performance of the access control enforcement mechanism in BASE:PostgreSQL and PSAC:PostgreSQL. Specifically, we measure the time required by the access control enforcement mechanism to check the state of a privilege, $test_priv$, for a user, $test_user$, on a database table, $test_table$. We vary the *ACL Size* parameter in our experiments. For BASE:ProstgreSQL, the *ACL Size* is the number of entries in the acl column of the pg_class catalog. For PSAC:PostgreSQL, the ACL size is the combined number of entries in the acl, the $acldeny$, the $aclsuspend$, and the $acltaint$ columns. Note that for the purpose of these experiments we do not maintain any privileges in the *neutral* orientation mode.

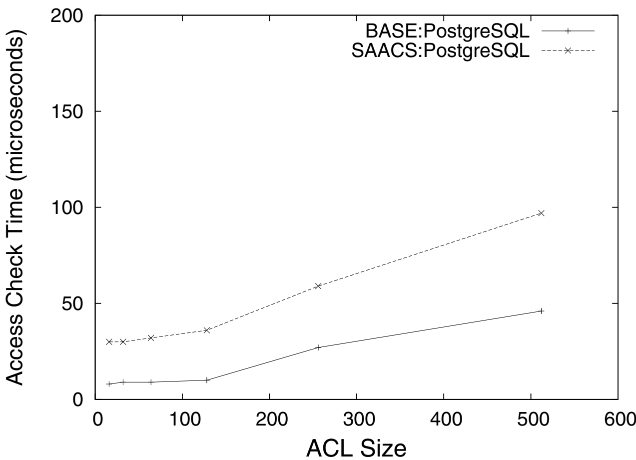
We perform two sets of experiments. The first experiment compares the access control overhead in the absence of a role hierarchy. The results are reported in Figure 2. As expected, the access control overhead for both BASE and PSAC PostgreSQL increases with the ACL Size. The key observation is that the access control overhead for PSAC:PostgreSQL is not much higher than that of BASE:PostgreSQL.

The second experiment compares the access control overhead in the presence of a hypothetically large role hierarchy. We use a role hierarchy of 781 roles with depth equal to 4. The edges and cross-links in the role hierarchy are randomly assigned. The rationale behind such set-up is that we want to observe a reasonable amount of overhead in the access control enforcement code. The role hierarchy is maintained in PSAC:PostgreSQL in a manner similar to that in BASE:PostgreSQL, that is, a role r_p is the parent of a role r_c if r_c is assigned to r_p using the GRANT ROLE command. A role and its assigned roles are stored in the $pg_auth_members$ catalog [5]. Next, in the experiment, we randomly assigned 10 roles to the $test_user$. In order to vary the size of the ACL



Lising 2. Exp 1: Access Control Enforcement Time in BASE and PSAC PostgreSQL in the absence of a role hierarchy

on the $test_table$, we developed a procedure to assign privileges on the $test_table$ to randomly chosen roles. Figure 3 reports the results of this experiment. First, observe that the access check time in the presence of a role hierarchy is not much higher than that in the absence of a role hierarchy. As mentioned before, this is mainly because we maintain a cache of the roles assigned to a user (directly or indirectly), thus preventing expensive role inheritance tests at the run-time. Second, the access control enforcement algorithm of PSAC:PostgreSQL reported in Section 3.1 is very efficient with a maximum time of approximately 97 microseconds for an ACL of size 512. Also, it is not



Lising 3. Exp 2: Access Control Enforcement Time in BASE and PSAC PostgreSQL in the presence of a role hierarchy

much higher than the maximum access control enforcement time in BASE:PostgreSQL which stands at approximately 46 microseconds.

Overall, the two experiments confirm the extremely low overhead associated with our design in PSAC:PostgreSQL.

4 Related Work

Access control models have been widely researched in the context of DBMSs [10]. To the best of our knowledge, ours is the first solution formally introducing the concept of privilege states in an access control model.

The implementation of the access control mechanism in the Windows operating system [1], and Network File System protocol V4.1 [3] is similar to the semantics of the *taint* privilege state. In such implementation, the security descriptor of a protected resource can contain two types of ACLs: a Discretionary Access Control List (DACL), and a System Access Control List (SACL). A DACL is similar to the traditional ACL in that it identifies the principals that are allowed or denied some actions on a protected resource. A SACL, on other hand, identifies the principals and the type of actions that cause the system to generate a record in the security log. In that sense, a SACL ACL entry is similar to a PSAC ACL entry with *taint* privilege state. Our concept of privilege states, however, is more general as reflected by the semantics of the other states introduced in our work.

The *up,down*, and *neutral* privilege orientations (in terms of privilege inheritance) have been introduced by Jason Crampton [12]. The main purpose for such privilege orientation in [12] is to show how such scheme can be used to derive a role-based model with multi-level secure policies. However, our main purpose for introducing the privilege orientation modes is to control the propagation of privilege states in a role hierarchy.

Much research work has been carried out in the area of network and host based anomaly detection mechanisms [16]. Similarly, much work on intrusion response methods is also in the context of networks and hosts [19,20]. The fine-grained response actions that we support in this work are more suitable in the context of application level anomaly detection systems in which there is an end user interacting with the system. In that context, an approach to re-authenticate users based on their anomalous mouse movements has been proposed in [17]. In addition, many web applications may force a re-authentication (or a second factor of authentication) in cases when the original authenticator has gone stale (for example expired cookies) to prevent cross-site request forgery (CSRF) attacks.

Foo et. al. [13] have also presented a survey of intrusion response systems. However, the survey is specific to distributed systems. Since the focus of our work is on fine-grained response actions in the context of an application level anomaly detection system, most of the techniques described in [13] are not applicable our scenario.

5 Conclusion

In this paper, we have presented the design, formal model and implementation of a privilege state based access control (PSAC) system tailored for a DBMS. The fundamental

design change in PSAC is that a privilege, assigned to a principal on an object, has a state attached to it. We identify five states in which a privilege can exist namely, *unassign*, *grant*, *taint*, *suspend* and *deny*. A privilege state transition to either the *taint* or the *suspend* state acts as a fine-grained response to an anomalous request. We design PSAC to take into account a role hierarchy. We also introduce the concept of privilege orientation to control the propagation of privilege states in a role hierarchy. We have extended the PostgreSQL DBMS with PSAC describing various design issues. The low access control enforcement overhead in PostgreSQL extended with PSAC confirms that out design is very efficient.

References

1. Access control lists in win32 (June 7, 2009), `http://msdn.microsoft.com/en-us/library/aa374872VS.85.aspx`
2. Incits/iso/iec 9075. sql-99 standard (January 2, 2009), `http://webstore.ansi.org/`
3. Nfs version 4 minor version 1 (June 7, 2009), `http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-minorversion1-29.txt`
4. Oracle database security guide 11g release 1 (11.1) (January 2, 2009), `http://download.oracle.com/docs/cd/B28359_01/network.111/b28531/toc.htm`
5. The postgresql global development group. postgresql 8.3 (June 7, 2009), `http://www.postgresql.org/`
6. Postgresql global development group. postgresql 8.3 documentation (January 2, 2009), `http://www.postgresql.org/docs/8.3/static/sql-grant.html`
7. Sql server 2008 books online. identity and access control (database engine) (January 2, 2009), `http://msdn.microsoft.com/en-us/library/bb510418(SQL.100).aspx`
8. Bertino, E., Kamra, A., Terzi, E., Vakali, A.: Intrusion detection in rbac-administered databases. In: ACSAC, pp. 170–182. IEEE Computer Society, Los Alamitos (2005)
9. Bertino, E., Samarati, P., Jajodia, S.: An extended authorization model for relational databases. IEEE Transactions on Knowledge and Data Engineering 9(1), 85–101 (1997)
10. Bertino, E., Sandhu, R.: Database security-concepts, approaches, and challenges. IEEE Transactions on Dependable and Secure Computing 2(1), 2–19 (2005)
11. Chandramouli, R., Sandhu, R.: Role based access control features in commercial database management systems. In: National Information Systems Security Conference, pp. 503–511
12. Crampton, J.: Understanding and developing role-based administrative models. In: ACM Conference on Computer and Communications Security, pp. 158–167 (2005)
13. Foo, B., Glause, M., Modelo-Howard, G., Wu, Y.-S., Bagchi, S., Spafford, E.H.: Information Assurance: Dependability and Security in Networked Systems. Morgan Kaufmann, San Francisco (2007)
14. Kamra, A., Bertino, E.: Design and implementation of a intrusion response system for relational database. IEEE Transactions on Knowledge and Data Engineering, TKDE (to appear 2010)
15. Kamra, A., Bertino, E., Terzi, E.: Detecting anomalous access patterns in relational databases. The International Journal on Very Large Data Bases, VLDB (2008)
16. Patcha, A., Park, J.-M.: An overview of anomaly detection techniques: Existing solutions and latest technological trends. Computer Networks 51(12), 3448–3470 (2007)

17. Pusara, M., Brodley, C.E.: User re-authentication via mouse movements. In: ACM Workshop on Visualization and Data Mining for Computer Security (VizSEC/DMSEC), pp. 1–8. ACM, New York (2004)
18. Sandhu, R., Ferraiolo, D., Kuhn, R.: The nist model for role-based access control: Towards a unified standard. In: ACM Workshop on Role-based Access Control, pp. 47–63 (2000)
19. Somayaji, A., Forrest, S.: Automated response using system-call delays. In: Proceedings of the 9th USENIX Security Symposium, p. 185. USENIX Association, Berkeley (2000)
20. Toth, T., Krügel, C.: Evaluating the impact of automated intrusion response mechanisms, pp. 301–310. IEEE Computer Society, Los Alamitos (2002)