Chapter 3

IMPLEMENTING RBAC POLICIES

Steve Barker, Paul Douglas and Terry Fanning

Abstract We show how role-based access control (RBAC) and temporal RBAC (TRBAC) policies may be formally represented and translated into PL/SQL code to implement practical access control policies to protect Oracle SQL databases. Performance results for an implementation of a TRBAC policy in PL/SQL are presented.

Keywords: Role-based access control, logic programming, PL/SQL

1. Introduction

In the SQL2 standard [8], the security-specific language features are restricted to simple GRANT and REVOKE statements (albeit views may also be used to help to provide security). The GRANT-REVOKE model enables a *security administrator (SA)* to represent only a small subset of the access control policies that are needed in practice [3]. SQL does not, for instance, enable limitations to be imposed on the use of a granted privilege [12].

To augment the expressive power of access control languages, application programs written in an imperative language (e.g., C or Java) may be used. However, the implementation of an access policy in these relatively low-level, procedural languages complicates the maintenance of the policy, and makes it difficult for SAs to reason about the consequences and effects of the policy specification.

In this paper, we describe an approach which involves using a logic-based language for formally specifying a variety of access control policies. A specification of an access policy expressed in this logic language can be naturally translated into PL/SQL code; the implementation of an access policy in PL/SQL can be used to protect Oracle databases.

The main contribution of our work is to demonstrate how formal specifications of access policies may be translated into a practical language that can be used to protect the information in real databases. The importance of our pro-

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35697-6_26](https://doi.org/10.1007/978-0-387-35697-6_26)

E. Gudes et al. (eds.), *Research Directions in Data and Applications Security*

© IFIP International Federation for Information Processing 2003

posal is that it may be used by database practitioners to enable them to specify and implement practical, application-specific access policies for protecting Oracle databases. The high-level formal specification of policies makes it easy for SAs to understand what is required to implement access control policies to protect Oracle databases. Moreover, the specification of an access policy may be used to simplify the SA's task of maintaining a PL/SQL implementation of an existing access control program.

Our approach is based on the formal specification of *role-based access control (RBAC)* policies [14]. In particular, our concern is with the representation and implementation of the $RBAC_{C3A}$ model from [14], and its extension to permit *temporal RBAC (TRBAC)* policies to be represented [4]. RBAC has a number of well documented attractions [9], and has been shown to be a popular choice of access control policy for many types of organization to adopt [10]. Temporal authorizations are also useful in practice [5]. Temporal authorizations enable a SA to specify that a user's permission to access a data item is to hold for a restricted interval of time, and automatically expires as soon as the maximum time point in the interval is reached. Temporal authorizations provide SAs with considerable flexibility when it comes to expressing access policies, they provide a SA with a fine level of control on permission assignments, and they can limit the damage that unauthorized users may wreak if they do gain access to a database system [13]. Moreover, temporal authorizations enable a SA to closely relate a user's authorizations to the tasks the user performs in an organization.

In the SQL3 standard, language features which permit RBAC policies to be represented are proposed [8]. What is more, several commercial RDBMSs now provide language facilities which enable RBAC policies to be defined to protect SQL databases [11]. However, the range of RBAC policies that can be defined by using SQL3 and existing RDBMSs is limited (e.g., temporal authorizations are not supported). The proposal we describe in the sequel shows how these limitations may be avoided.

The rest of this paper is organized thus. In Section 2, we describe the formal specification of RBAC and TRABC policies in a logic language. In Section 3, we show how the specification of an RBAC or TRBAC policy may be translated into PL/SQL code, and how this code may be used to control access to an Oracle SQL database. In Section 4, we briefly consider performance issues relating to a PL/SQL implementation of a TRBAC policy. Finally, in Section 5, some conclusions are drawn and suggestions for further work are made.

2. Formally Specifying (T)RBAC Policies

The access control programs that we describe in this section are based on a specification of $RBAC_{H2A}$ policies using normal clause programs [1], and

the specification of $TRBAC_{C3A}$ policies as constraint logic programs [4]. The $TRBAC_{C3A}$ programs subsume the $RBAC_{H2A}$ programs.

In the formal specification of $RBAC_{H2A}$ policies described in [1], a user U is recorded as being assigned to a role R by a SA defining a $ura(U, R)$ predicate in an $RBAC_{H2A}$ program. For example, $ura(bob, r1) \leftarrow$ records the assignment of the user Bob to the role $r1$. To record that a P privilege on an object O is assigned to a role R , definitions of an $rpa(R, P, O)$ predicate are used. For example, $rpa(r1, write, o1) \leftarrow$ expresses that the role $r1$ is assigned write privilege on the object $o1$.

In addition to user-role and permission-role assignments, role hierarchies are the other key element of $RBAC_{H2A}$ [14]. An $RBAC_{H2A}$ role hierarchy is represented by a partial order $(R, \geq)$ defining a seniority ordering $\geq$ on a set of roles R . Role hierarchies in $RBAC_{H2A}$ are used to represent that, unless constraints are imposed, “senior roles” inherit the (positive) permissions assigned to roles that are “junior” to them in an $RBAC_{H2A}$ hierarchy (but not conversely). Hence, if role r_i is senior to role r_j in an $RBAC_{H2A}$ role hierarchy ($r_i \geq r_j$) then r_i inherits the (positive) permissions on objects which are assigned to r_j .

In [1], an $RBAC_{H2A}$ role hierarchy is represented by a set of clauses that define a *senior_to* relation (i.e., $\geq$) as the reflexive-transitive closure of an irreflexive-intransitive *ds* relation that defines the set of pairs of roles (r_i, r_j) such that r_i is directly senior to role r_j in an $RBAC_{H2A}$ role hierarchy. The definition of *ds* in terms of $\geq$ may be expressed thus:

$$\begin{aligned} \forall r_i, r_j [ds(r_i, r_j) \leftrightarrow r_i \geq r_j \wedge r_i \neq r_j \wedge \\ \neg \exists r_k [r_i \geq r_k \wedge r_k \geq r_j \wedge r_i \neq r_k \wedge r_j \neq r_k]]. \end{aligned}$$

In clause form logic, the *senior_to* relation may be defined in terms of *ds* thus (where ‘_’ is an anonymous variable):

$$\begin{aligned} senior_to(R1, R1) &\leftarrow ds(R1, _). \\ senior_to(R1, R1) &\leftarrow ds(_, R1). \\ senior_to(R1, R2) &\leftarrow ds(R1, R2). \\ senior_to(R1, R2) &\leftarrow ds(R1, R3), senior_to(R3, R2). \end{aligned}$$

The clauses defining the *senior_to* relation are included in every instance of an $RBAC_{H2A}$ program; the application-specific *ds*, *ura* and *rpa* clauses define a particular instance of an $RBAC_{H2A}$ program.

The *senior_to* predicate is used in the definition of *permitted* that follows:

$$permitted(U, P, O) \leftarrow ura(U, R1), senior_to(R1, R2), rpa(R2, P, O).$$

The *permitted* clause expresses that a user U is authorized to exercise the P access privilege on an object O if U is assigned to a role $R1$ that is senior to the role $R2$ in an $RBAC_{H2A}$ role hierarchy, associated with the $RBAC_{H2A}$ program, and $R2$ has been assigned P access on O . That is, U has P access on O if U is assigned to a role that inherits the P privilege on O .

To incorporate the session concept in an $RBAC_{H2A}$ program [14], a ground instance of an *active*($U, R1$) fact may be appended to the current instance of a $RBAC_{H2A}$ program to record that a user U is active in role $R1$ [4].

When *active* facts may be appended to an $RBAC_{H2A}$ program, the *permitted* rule is extended to include an *active*($U, R1$) condition, viz.:

$$\begin{aligned} permitted(U, P, O) \leftarrow & \text{ura}(U, R1), \text{active}(U, R1), \\ & \text{senior_to}(R1, R2), \text{rpa}(R2, P, O). \end{aligned}$$

In $TRBAC_{H2A}$ programs [4], a temporal argument is added to the *ura* and *rpa* predicates, and a SA uses specifications of *ura* and *rpa* to, respectively, represent the times at which a user is assigned to a role, and the times at which a permission on an object is to hold. Hence, the set of authorizations defined by a $TRBAC_{H2A}$ program may be specified thus:

$$\begin{aligned} permitted(U, P, O) \leftarrow & \text{date}(T), \text{ura}(U, R1, T), \text{active}(U, R1), \\ & \text{senior_to}(R1, R2), \text{rpa}(R2, P, O, T). \end{aligned}$$

In a $TRBAC_{H2A}$ program, the *permitted* clause above is used to record that a user U has P access on object O if at the time T (taken from the system clock) at which U requests P access on O , U is assigned to role $R1$, U is currently active in $R1$, $R1$ is senior to a role $R2$, and $R2$ has P access on O at time T .

$RBAC_{C3A}$ ($TRBAC_{C3A}$) programs extend $RBAC_{H2A}$ ($TRBAC_{H2A}$) programs by including a *separation of duties* constraint [4,14]. To ensure that a user is not assigned to two roles that are required to be statically separated at a common point in time, the specifications of $TRBAC_{C3A}$ programs in [4] include the following rule:

$$\text{inconsistent_ssd}(U, R1) \leftarrow \text{ura}(U, R1, T), \text{ssd}(R1, R2), \text{ura}(U, R2, T).$$

This rule is used by a SA posing an *inconsistent_ssd*($U, R1$) query with respect to a policy whenever a new *ura* fact is inserted into the access policy program that protects a database. If a user U is to be assigned to a role $R1$ and a SA's *inconsistent_ssd* query succeeds on the policy specification to which the new *ura* fact is appended then the insertion is rejected; otherwise the new fact is included in the current instance of the policy. Ground instances of a binary

ssd relation are used to record pairs of roles (r_i, r_j) such that r_i is statically separated from r_j (the pair (r_j, r_i) is also recorded since *ssd* is a symmetric relation).

In [4], a *dynamic separation of duties* constraint on $TRBAC_{C3A}$ programs is expressed thus:

$$active(U, R1) \leftarrow not\ inconsistent_dsd(U, R1), activate(U, R1).$$

The definition of the auxiliary *inconsistent_dsd*($U, R1$) predicate is:

$$inconsistent_dsd(U, R1) \leftarrow dsd(R1, R2), activate(U, R2).$$

These clauses are used to express that a user U can only be active in a role $R1$ if U has requested to be active in $R1$, and U has not also requested to be active in a dynamically separated role $R2$.

Ground instances of a binary *dsd* relation are used to record pairs of roles (r_i, r_j) such that r_i is dynamically separated from the role r_j (the pair (r_j, r_i) is also recorded since *dsd* is a symmetric relation).

In the ensuing discussion, we describe, in outline, the implementation, in PL/SQL, of a $TRBAC_{C3A}$ policy. The implementation has been used in practice at British Telecom to protect the information in some Oracle SQL databases. Note that the PL/SQL implementation of an $RBAC_{C3A}$ policy is a simple case of the $TRBAC_{C3A}$ implementation which we describe in the sequel, and all of the PL/SQL code discussed within this paper may be found at our website [2].

3. Mapping $TRBAC_{C3A}$ Clauses to PL/SQL

In our implementation of $TRBAC_{C3A}$ in PL/SQL, the *ds*, *ssd*, and *dsd* relations are represented as Oracle 8 tables. The tables are called *ROLE_HIER*, *STATIC_DUTY_SEPARATION* and *DYNAMIC_DUTY_SEPARATION*, respectively.

To express user-role and permission-role assignments, the *ura* and *rpa* relations are represented as sets of facts in tables which include additional attributes to capture temporal constraint information. The tables we use to represent *ura* and *rpa* are *USER_ROLE_ASSIGN* and *ROLE_PERMISSION_ASSIGN*, respectively.

3.1 Representing Sessions

To represent the *active* relation, a table, *ROLE_SESSION*, may be used with the attributes $\langle USERNAME, ROLE, SESSION_ID \rangle$ to record the session in

which the role was activated by a user. A row is inserted into *ROLE_SESSION* whenever a user *U* activates a role *R1*. This is possible if: no row exists in the *ROLE_SESSION* table which records that *U* is currently active in a role *R2* ($R1 \neq R2$), and *R2* is represented as being dynamically separated from *R1* by a row in the *DYNAMIC_DUTY_SEPARATION* table.

The PL/SQL code for *inconsistent_dsd*, implemented by the *conflict* cursor [2], implies that a violation of the dynamic separation of duties constraint requires the existence of at least one row in the *DYNAMIC_DUTY_SEPARATION* table. If a row, relating to role r_i , exists in *DYNAMIC_DUTY_SEPARATION* then a user *U* cannot be active in any role r_j which is recorded as being dynamically separated from r_i . In the case where the *dsd* constraint is satisfied when a request to activate role r_i is made by *U*, the **INSERT INTO *ROLE_SESSION*** statement is used to add r_i to the current set of roles activated by *U* [2]. In the **INSERT** statement, *l_session* is a variable initialized to the value of the current Oracle session by using the following declaration:

```
l_session_number := USERENV('SESSIONID');
```

A PL/SQL procedure *deactivate_role* may be used to remove active roles from the table *ROLE_SESSION*. For that, the following SQL statement may be used:

```
DELETE FROM role_session sess
WHERE sess.username = user
AND role = upper(P_ROLE)
AND session_id = l_session;
```

3.2 Implementing Static Separation of Duties

In our PL/SQL implementation of *inconsistent_ssd*, the required logic is to check for the existence of a row in the table *STATIC_DUTY_SEPARATION* with a value for the role *R2* which is defined as being statically separated from the new role being assigned (i.e., *R1*), and a check for a row in the *USER_ROLE_ASSIGN* table which records the user as being currently assigned to role *R2*. This logic may be implemented by using database triggers in Oracle 8.

Because Oracle 8 triggers prohibit select statements on the table currently being updated, two triggers are used in our PL/SQL implementation (PL/SQL code for these triggers is given in [2]):

- a row level trigger *ura_row_trigger* which executes once for every row inserted or updated in the *USER_ROLE_ASSIGN* table and stores the new values in a global PL/SQL table.

- a statement level trigger *ura_stat_trigger* which executes once after all rows are updated and checks all rows in the PL/SQL table for a violation of the static separation of duties constraint.

A trigger is also used in our PL/SQL implementation to check for authorizations that would violate the static separation of duties constraint. The trigger is used on the *STATIC_DUTY_SEPARATION* table and executes when a new *ssd* constraint is defined [2].

The SQL code for *ssd_row_trigger* is used to check for:

- A row in the *STATIC_DUTY_SEPARATION* table with a value for role, *R2*, that is associated with the new role, *R1*, being assigned to a user.
- A row in the *USER_ROLE_ASSIGN* table where the user is assigned to role *R2* for a period that overlaps the user's assignment to a statically separated role *R1*.

3.3 Defining Authorizations

In the *TRBAC_{C3A}* model [4], the following rule is invoked when a request for *P* access on a data object *O* is made by a user *U*:

$$\text{permitted}(U, P, O) \leftarrow \text{date}(T), \text{ura}(U, R1, T), \text{active}(U, R1), \\ \text{senior_to}(R1, R2), \text{rpa}(R2, P, O, T).$$

To represent the temporal form of the *permitted* rule in our implementation of *TRBAC_{C3A}* in PL/SQL, the required logic is to only permit a user *U* to have *P* access on an object *O* at the time *T* at which an access request is made if:

- A row exists in the table *USER_ROLE_ASSIGN* recording that *U* is assigned to a role *R1* at *T*.
- A row exists in the table *ROLE_SESSION* that records that *U* has the role *R1* active at *T*.
- A row exists in the table *ROLE_PERMISSION_ASSIGN* which records that *P* access on *O* is assigned to *R1* at *T* or rows exist in the table *ROLE_HIER* such that the role *R1* is recorded as being senior to some role *R2*, and a row exists in the table *ROLE_PERMISSION_ASSIGN* which records that *P* access on *O* is assigned to *R2* at *T*.

The logic (above) must be executed whenever a user attempts to perform a read or write operation on an object to which access control constraints apply.

To implement the required logic, a PL/SQL function, *permitted*, has been created. The *permitted* function implements the *permitted(U,P,O)* clause from

$TRBAC_{C3A}$. The *permitted* function that we use in our PL/SQL implementation takes as parameters a user U requesting P access on an object O . The code to implement the *permitted* clause is given in [2].

3.4 Placing Objects Under TRBAC Control

To ensure that the necessary code is executed when a user issues a **SELECT** statement on an object stored in an Oracle database, a view of the object is created with a call to the *permitted* function. Permissions are then assigned to the user *public* for the view. A *public* synonym is created for the view to allow users to refer to the original object name. End users do not have any permissions on the original object; they are only able to access the view. Finally, a trigger is created on the original table that calls the *permitted* function when inserting, updating and deleting. It should be noted that this approach assumes that the views under TRBAC control are not updateable.

4. Performance Measures

Test results for our PL/SQL implementation of $TRBAC_{C3A}$ policies offer suggestive evidence that these implementations can be used in practice to protect a realistically sized Oracle database. Even in worst-case computations using our implementation, where the access control information only increases the number of tuples involved in answering a query, our tests reveal that incorporating access control information at run-time imposes a minimal extra overhead. For example, in our suite of tests, we consider a worst-case user query that involves joining 8 tables containing a total of 432,261 tuples (the total number of tuples generated at run-time is approximately 2 million). We compare the time taken to evaluate this join query (query $Q1$) on our test database without using the access control information to the equivalent query (query $Q2$) on our database protected using a $TRBAC_{C3A}$ policy for a user who has access to all required tables (i.e., the access control information does not constrain the search space of solutions). The $TRBAC_{C3A}$ policy specification defines a role hierarchy which includes 41 roles (dynamically generated at run-time) and 3974 *rpa* and 616 *ura* tuples. Our results reveal that $Q1$ takes 18.66 seconds whereas $Q2$ takes 19.74 seconds.

For the other test cases we consider, we have found that the access control information which is used at run-time to evaluate queries and updates adds, on average, a few hundredths of a second to the time taken to evaluate the same access request on the same database when no access control information is involved. Moreover, in many cases the access control information used in query evaluation may be exploited to help to reduce the set of answers generated for the user's query to an authorized subset of answers the user is permitted to know to be true in the database. In cases where a user does not have sufficient

access permissions to perform a requested retrieval then, of course, the query will fail early and this saves wasted computation. Hence, in some cases, query evaluation performance may actually be increased by using access control information rather than being reduced by it.

5. Conclusions and Further Work

In this paper we have shown how RBAC and TRBAC policies may be formally specified in a high-level language, and how this type of formal specification may be straightforwardly translated into the equivalent PL/SQL code that is required to implement access policies to protect an Oracle database.

Whilst the access control policy implementation that we have described only supports positive authorizations, it is relatively straightforward to implement other types of policy by using the approach that we have discussed. Moreover, whilst the update and delete operations on an SQL database are effected by destructive assignment (i.e., only the current state or snapshot is retained), the implementation that we have described could easily be enhanced to maintain a history of authorizations. For that, shadow tables could be created, with identical attributes to the *USER_ROLE_ASSIGN* and *ROLE_PERMISSION_ASSIGN* tables, and additional attributes may be used to record the time at which a transaction occurred, and the type of operation (i.e., *INSERT*, *UPDATE* and *DELETE*). Providing facilities for recording and using historical information in our implementations is a matter for further work.

References

- [1] Barker, S., Data protection by logic programming, *Proceedings of the First International Conference on Computational Logic (LNAI 1861)*, Springer, pp. 1300-1313, 2000.
- [2] Barker, S., Douglas P. and Fanning T., Implementing RBAC policies in PL/SQL (www.cscs.wmin.ac.uk/~douglap/rbac-plsql.html).
- [3] Barker, S. and Rosenthal, A., Flexible security policies in SQL, *Proceedings of the Fifteenth IFIP WG 11.3 Conference on Database and Applications Security*; also published in M. Olivier and D. Spooner (eds.), *Data and Applications Security*, Kluwer, 2002.
- [4] Barker, S. and Stuckey, P., Flexible RBAC policy specification with constraint logic programming, submitted for publication.
- [5] Bertino, E., Bettini, C., Ferrari, E. and Samarati, P., An access control model supporting periodicity constraints and temporal reasoning, *ACM Transactions on Database Systems*, vol. 23(3), pp. 231-285, 1998.
- [6] Castano, S., Fugini, M., Martella, G. and Samarati, P., *Database Security*, Addison-Wesley, 1995.

- [7] Clark, K., Negation as failure, in H. Gallaire and J. Minker(eds), *Logic and Databases*, Plenum, pp. 293-322, 1978.
- [8] Date, C., *An Introduction to Database Systems*, Addison-Wesley, 2000.
- [9] Ferraiolo, D., Cugini, J. and Kuhn, R., Role-based access control (RBAC): Features and motivations, *Proceedings of the Eleventh Annual Computer Security Applications Confence*, IEEE Press, pp. 241-248, 1995.
- [10] Ferraiolo, D., Gilbert, D. and Lynch, N., An examination of federal and commercial access control policy needs, *Proceedings of the Sixteenth NIST-NSA National Computer Security Conference*, pp. 107-116, 1993.
- [11] Ramaswamy, C. and Sandhu, R., Role-based access control features in commercial database management systems, *Proceedings of the Twenty-First National Information Systems Security Conference*, pp. 503-511, 1998.
- [12] Rosenthal, A. and Sciore, E., Extending SQL's grant and revoke operations to limit and reactivate privileges, *Proceedings of the Fourteenth IFIP WG 11.3 Confence on Database and Applications Security*, 2000; also published in B. Thuraisingham et al., (eds.), *Data and Applications Security*, Kluwer, 2001.
- [13] Sandhu, R., Coyne, E., Feinstein, H. and Youman, C., Role-based access control: A multi-dimensional view, *Proceedings of the Tenth Annual Computer Security Applications Conference*, pp. 54-62, 1994.
- [14] Sandhu, R., Ferraiolo, D. and Kuhn, R., The NIST model for role-based access control: Towards a unified standard, *Proceedings of the Fourth ACM Workshop on Role-Based Access Control*, pp. 47-61, 2000.