Protecting federated databases using a practical implementation of a formal RBAC policy.

Steve Barker
Department of Computer Science, King's College, London

Paul Douglas
Cavendish School of Computer Science, University of Westminster

Protecting Federated Databases Using A Practical Implementation of a Formal RBAC Policy

Steve Barker
King's College
London, UK
steve@dcs.kcl.ac.uk

Paul Douglas
University of Westminster
London, UK
P.Douglas@wmin.ac.uk

Abstract

This paper describes the use of formally specified RBAC policies for protecting federated relational database systems that are accessed over a wide area network. The method that is described combines a formally specified RBAC policy with both temporal and locational constraints. It does not depend on any security mechanism supported by a specific DBMS and is thus portable across platforms.

1. Introduction

Protecting databases from unauthorized access requests has been recognized as important for many years. Nonetheless, vendors of database products typically provide very limited means for protecting the data their DBMS will be used to manage. Relational databases (RDBMSs) still dominate the market, so discussion is limited here to RDBMSs and the security mechanisms provided by SQL.

In the SQL92 standard [10], only *grant* and *revoke* statements are provided with which to implement security. These allow a limited subset of necessary access control policies to be specified, even when combined with the use of views [6]. The SQL:1999 standard improves on this by including features for expressing RBAC policies. However, these in turn are limited, and many currently available commercial systems fail to implement them [14].

The need for multipolicy specification using high-level specification languages with well-defined formal semantics has recently been recognised and a number of candidate proposals have been described in the literature [2, 8, 7, 12]. These proposals are generally theoretical in nature and lack full implementations or any performance measures ([7] is exceptional in the latter respect). More recently, [5] and [4] have looked at methods of constructing practical implementations of such policies. However, [5] relied on using an RDBMS that supports some sophisticated *request modification* facilities [10] that RDBMSs do not offer as standard features; [4] limited its implementation to a centralized system and did not include temporal constraints. The contribution of this paper is to investigate the application of formally specified access control policies to non-centralized databases, and to use a more sophisticated RBAC model (i.e., one that does include temporal constraints); it is not our intention to compare the theory upon which our approach is based with other work on access policy formulation by logic programming.

Because earlier work has concentrated on centralized databases, it has assumed that data access requests will typically come from within the organization that owns the data, and that access requests will be made from local client applications (usually on the same subnet as the server, and often within the same physical building that houses the server). Common changes to organizational working practises mean this is now less likely. Federated databases, where logical databases are formed from physically separate databases located at different sites, and queried from different physical locations, are increasingly common, and this is the model considered by this paper. We have chosen to use an internet browser as the client paradigm: this fits well with the current trend towards increasingly integrated desktop applications.

The remainder of the paper is organized as follows. In Section 2, some basic notions in access control, RBAC, temporal RBAC (TRBAC) and logic programming are described. In Section 3, the representation of RBAC and TRBAC policies, by using stratified logic programs, is discussed. These policies are based on the $RBAC_{H2A}$ model that is informally defined in [17] and formally defined in [3]. Henceforth, we refer to the logic programs that implement $RBAC_{H2A}$ or $TRBAC_{H2A}$ policies as "$RBAC$ programs". In Section 4, we describe our implementation of $RBAC$ programs for protecting relational databases from unauthorized access requests. In Section 5, we briefly consider performance results for our approach. Finally, in Section 6, some conclusions are drawn and suggestions for fur-

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)
0-7695-2108-8/04 $ 20.00 © 2004 IEEE

ther work are made.

2. Formal Policy Representation

The $RBAC$ programs that are considered in this paper are represented by using a finite set of *normal clauses* [13].

Definition 1 *A normal clause is a formula of the form:*

$$C \leftarrow A_1, \ldots, A_m, not\ B_1, \ldots, not\ B_n.$$

The *head*, C, of the clause above is a single *atom*. The *body* of the clause (i.e., $A_1, \ldots, A_m, not\ B_1, \ldots, not\ B_n$) is a conjunction of literals. Each A_i literal ($i \in \{1, .., m\}$) is a *positive literal*; each $not\ B_j$ literal ($j \in \{1, .., n\}$) is a *negative literal*. In the case of a negative literal, the relevant type of negation is *negation as failure* [**?**]. Variables in clauses appear in the upper case; constants appear in the lower case.

An $RBAC$ program S is defined on a domain of discourse that includes:

1. a set $\mathcal{U}$ of *users*;
2. a set $\mathcal{O}$ of *database objects*;
3. a set $\mathcal{A}$ of *access privileges*; and
4. a set $\mathcal{R}$ of *roles*.

The *users* in the case of our implementation will be external programs attempting to access the database, and the *database objects* will be tables. The *access privileges* are *read* and *write*.

Where these programs are extended to include temporal constraints, we also employ a set $\mathcal{T}$ of *time points*. We view time as a linearly ordered, discrete set of time points that are isomorphic to the natural numbers. Since we are also interested in the origin of the access request we additionally require a set $\mathcal{I}$ of *IP addresses*.

We thus formally define access privileges using the following definitions:

Definition 2 *An $authorization$ is a 5-tuple (u, a, o, t, i) that denotes that a user u ($u \in \mathcal{U}$) has the a access privilege ($a \in \mathcal{A}$) on the object o ($o \in \mathcal{O}$) at time t ($t \in \mathcal{T}$) from IP address i ($i \in \mathcal{I}$).*

Definition 3 *If a is an access privilege and o is an object then a $permission$ is a pair (a, o) that denotes that the a access privilege may be exercised on o.*

Definition 4 *A $permission$-$role\ assignment$ is a 4-tuple (a, o, r, t) that denotes that the permission (a, o) is assigned to the role r at time t.*

Definition 5 *A $user$-$role\ assignment$ is a triple (u, r, t) that denotes that the user u is assigned to the role r at time t.*

It is assumed in this paper that a *closed policy* [9] is being used to protect the database. However, the implementation of various open or hybrid policies require only minor modifications to the approach that we describe (see [7]). Such policies are particularly likely to apply in the area of databases accessed via the internet, where it is more likely that we would wish to deny access permission to queries originating from a specific group of IP addresses than only to permit those from sites regarded as "safe".

3. Representing $RBAC$ Programs

The $RBAC$ programs that we describe in this section are based on the specification of $RBAC$ as a normal clause program from [2]. A user U is assigned to a role R at time T by defining a 3-place $ura(U, R, T)$ predicate in an $RBAC$ program. For example, $ura(bob, r1, t)$ is used to record the assignment of the user Bob to the role $r1$ at time t. To record that the A access privilege on an object O is assigned to a role R at time T, clause form definitions of a 4-place $pra(A, O, R, T)$ predicate are used. For example, $pra(read, o, r1, t)$ expresses that the role $r1$ is assigned the $read$ privilege on the database object o at time t.

An $RBAC_{H2A}$ role hierarchy is expressed in an $RBAC$ program by a set of clauses that define a 2-place $senior\_to$ predicate as the reflexive-transitive closure of an irreflexive-intransitive 2-place ds predicate 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 (i.e., r_i is senior to r_j and there is no role $r_k \in \mathcal{R}$ such that r_i is senior to r_k and r_k is senior to r_i).

In clause form logic, $senior\_to$ is defined in terms of ds thus (where '_' is an anonymous variable):

$$senior\_to(R1, R1) \leftarrow ds(R1, \_).$$
$$senior\_to(R1, R1) \leftarrow ds(\_, R1).$$
$$senior\_to(R1, R2) \leftarrow ds(R1, R2).$$
$$senior\_to(R1, R2) \leftarrow ds(R1, R3), senior\_to(R3, R2).$$

To represent that a user u is active in a role r at a time t, an $active(u, r, t)$ fact is appended to a $RBAC$ program whenever u requests to be active in r at time t and this request is allowed. The set of $active$ facts in an $RBAC$ program at an instance of time t is the set of roles that users have active at time t.

A set of allowed IP addresses is also stored in simple 1-place $ip(I)$ predicates.

The set of authorizations by an $RBAC$ program is defined by the clause:

$$permitted(U, A, O, T, I) \leftarrow time(T), ura(U, R1, T), active(U, R1, T), senior\_to(R1, R2), pra(A, O, R2, T), ip(I).$$

This expresses that a user U has the A access privilege on an object O at time T (extracted from the system clock using $time(T)$) requesting it from location I (extracted from web server) if U is assigned to a role $R1$ at T, U is active in $R1$ at T, $R1$ is senior to the role $R2$ in an $RBAC_{H2A}$ role hierarchy defined by the $RBAC$ program, $R2$ is assigned the A access privilege on O at T, and I is an allowed address.

4. The Practical Implementation of $RBAC$ Programs

This section describes the practical implementation of $RBAC$ programs for protecting databases from unauthorized access.

A modular approach to developing the software that implements our proposal has been adopted. There are three principal components in our implementation:

- The Main Access Control Program.
- The Authorization Program.
- The Database System.

4.1. The Main Access Control Program

Java is used to implement the Access Control Program (ACP). This is not the most efficient implementation (in terms of performance overheads) that we could have employed. Because the authorization program (see below) is accessed via a C library API, using C for the ACP would have provided a more seamless interface and made the use of an extra level of indirection unnecessary. The program could then have been easily adapted into a CGI executable, thus providing our web interface mechanism. However, there are a number of reasons why this is not the best method. There are several well-known security problems associated with the use of CGI programs [18], and these make CGI unsuitable as a vehicle for the implementation of a security control program. Java is now widely used for a wide variety of Internet-based applications. It offers comprehensive server programming support (using *Java Servlets*) and excellent DBMS interfacing using JDB. In addition, it is easy to access applications written in a variety of other languages (through the *Java Native Interface (JNI)* mechanism). Java's support for distributed processing means that it will also be well-suited for future developments that we are considering for access control in a distributed DBMS environment.

The ACP acts as a server program for all database access requests: no other method of accessing the database is allowed. Access is via a client application that will handle all input of queries and output of data. It is assumed that this client will operate within a browser window. A user, whether on a local or remote host, will be presented with an identical user interface and the physical location of the server will thus be immaterial (and could change from time to time as dictated by operational requirements).

Typically, a good quality interface will be achieved by using a scripting language like Javascript, but for the purpose of testing we used HTML. As a web server, we used Apache's Tomcat; Tomcat is the standard container for Java servlet implementations [1].

The ACP is invoked by receiving a data access request from a client. The validity of this data request is determined by calling an authorization program, and passing it the following data:

- the user id and password (entered by the user);
- the IP address of the client (obtained from Tomcat);
- the current time (obtained from the system clock);
- the type of data access requested; and
- the database object the access request refers to.

The password would of course normally be encrypted, but for testing purposes we omitted this step. The type of data access requested is either read or write, where read corresponds to an SQL "select" statement and write corresponds to any of "insert", "update" or "delete". In our implementation we determine which is required by parsing the query for the relevant operation. The database object is a table; which one is determined once again by parsing the query to find the table names referred to. Object granularity is limited to tables in this implementation.

The authorization module returns a code indicating either a granting or a denial of the requested data access.

If the request is granted, the query from the client is passed to the DBMS by using a JDBC object. For all operations (i.e., select, insert, update or delete) a status code will be returned by the DBMS. Where the status code indicates that an error has occurred, an error message will be the output from the APC to the client. Where no error has occurred, a message indicating success will be the output; if the client's query was a "select" statement, there will also be the set of tuples returned by the DBMS.

If the request is denied, this information is returned to the client and the transaction is terminated. No query is passed to the DBMS and no database activity occurs.

Each database transaction is authorized individually, so the duration of a user session is precisely one transaction. This does involve a certain processing overhead, but does give a high level of security.

4.2. The Authorization Module

XSB [16] is used to implement the logic program that defines the access control applicable to the database being protected. XSB offers excellent performance that has been demonstrated to be far superior to that of traditional Prolog-based systems [15]. The actual calls are handled by the YAJXB [11] package. YAJXB makes use of Java's JNI mechanism to invoke methods in the C interface library package supplied by XSB. It also handles all of the data type conversions that are needed when passing data between C and Prolog-based applications. YAJXB effectively provides all the functionality of the C package within a Java environment. Although, as described above, this does involve some additional overhead, the method is flexible and straightforward and, for these reasons, preferable to using either a low-level sockets-based approach or a Java/Prolog hybrid.

Once invoked, XSB loads a program that contains the Prolog expression of the $RBAC$ policy described in Section 3 above. This is used to determine whether the access request that has been made is to be permitted or not. XSB's interface library provides a number of mechanisms for passing Prolog-style goal clauses from a calling program. Our implementation constructed the goal clause in a Java String and passed it to XSB using the $xsb\_command\_string$ function via a YAJXB interface object. The actual call is:

```
i=core.xsb_command_string(command.toString());
```

where the assignment, as one would expect, handles the returned value that indicates whether or not the request has been granted. Because the authorization module is used solely for the purpose of determining the validity of the access request, no other returned data is required.

4.3. Database System

One of our objectives in designing this implementation was that it should be as flexible as possible. The only assumption we make is that the query will be expressed in SQL,[1] and any database that accepts SQL queries and that has a JDBC driver could be used with this system. Because we do all query authorization before the database is accessed, the specific security provision of the DBMS is immaterial. We have successfully tested the system using both Oracle and MySQL with no modification other than to the JDBC driver information.

[1] We assume an SQL conforming to an SQL standard, but it would be relatively simple to add an additional module to this software to convert queries into other dialects of SQL, which would increase flexibility in the case of heterogeneous federated systems.

5. Performance Measures

Our authorization model does not introduce any additional network traffic, and this is always the largest time overhead in any distributed system. The performance capabilites of XSB (the additional component that our system introduces to a traditional server architecture) have been well documented ([15]). It could therefore be expected that the system would work efficiently, and the results we have obtained when testing the program bear this out. We used a modified version of an $RBAC$ program that we have used for testing purposes before (see [5] and [4]). It includes a definition of a 53 role $RBAC_{H2A}$ role hierarchy that has been represented by using a set of facts to represent all pairs of roles in the $senior\_to$ relation (a total of 312 $senior\_to$ facts). There is one user, one ura rule, 8 database objects (tables), 720 pra rules and 15 ip rules. It is sufficient for test purposes to use one user to demonstrate a worst-case use of the access control information in an $RBAC$ program. This worst case test involves assigning a user u to the unique top element in the $RBAC_{H2A}$ role hierarchy, such that u has complete access to all of the tables used in the test queries. The permissions are assigned to the unique bottom element in the $RBAC_{H2A}$ role hierarchy. Hence, our testing involves the maximum amount of multiple upward inheritance of permissions.

The experiments were performed using XSB Version 2.5 on a Sun Ultra 60 server (2 450MHz CPUs and 1GB RAM) running Solaris. In line with previous findings [4], the time taken to evaluate an authorization request is typically less than a hundredth of a second. This is a very small overhead to introduce. In the case where the request is denied, this time is *instead* of the DBMS access time, since no access takes place. It follows that for unauthorized queries there is actually a performance improvement gained from using this approach.

6. Conclusions and Further Work

We have shown how the information in federataed relational databases may be protected from unauthorized access requests by using $RBAC$ programs. The high-level formulation of an access policy as a logic program makes it relatively easy for a Database Administrator to express an access policy, to reason about its effects and to maintain it. We have demonstrated that the access policies that we use for protecting relational databases may be efficiently implemented.

In future work, we intend to investigate the extension of the approach that we have described here to Distributed DBMS environments with more open access policies.

References

[1] The apache jakarta project, 1999. http://jakarta.apache.org/tomcat/.

[2] S. Barker. Data protection by logic programming. pages 1300–1313. 1st International Conference on Computational Logic, Springer, 2000.

[3] S. Barker. Secure deductive databases. pages 123–137. 3rd Inat. Symp. on Practical Applications of Declarative Languages (PADL'01), Springer, 2001.

[4] S. Barker and P. Douglas. Practical rbac policy implementation for sql databases. IFIP WG 13, 2003, Kluwer, to appear.

[5] S. Barker, P. Douglas, and T. Fanning. Implementing rbac policies in pl/sql. pages 27–36. IFIP WG 13, 2002, Kluwer, 2003.

[6] S. Barker and A. Rosenthal. Flexible security policies in sql. pages 187–199. DBSec 2001, 2001.

[7] S. Barker and P. Stuckey. Flexible access control policy specification with constraint logic programming. ACM Transations on Information and System Security, 2003.

[8] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A system to specify and manage multipolicy access control models. pages 116–127. IEEE 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002), 2002.

[9] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison-Wesley, 1995.

[10] C. Date. *An Introduction to Database Systems*. Addison-Wesley, 2000.

[11] S. Decker. Yajxb. http://www-db.stanford.edu/~stefan/rdf/yajxb.

[12] S. Jajodia, P. Samarati, M. Sapino, and V. Subrahmanian. Flexible support for multiple access control policies. In *ACM TODS*, volume 26(2), pages 214–260, 2001.

[13] J. LLoyd. *Foundations of Logic Programming*. Springer, 1987.

[14] C. Ramaswarmy and R. Sandhu. Role-based access control features in commercial database management systems. pages 503–511. Proc. 21st National Information Systems Security Conference, 1998.

[15] K. Sagonas, T. Swift, and D. Warren. Xsb as an efficient deductive database engine. page 512. ACM SIGMOD Proceedings, 1994.

[16] K. Sagonas, T. Swift, D. Warren, J. Freire, and P. Rao. *The XSB System Version 2.0, Programmer's Manual*, 1999.

[17] R. Sandhu, D. Ferraiolo, and R. Kuhn. The nist model for role-based access control: Towards a unified standard. pages 47–61. Proc. 4th ACM Workshop on Role-Based Access Control, 2000.

[18] W3C. The world wide web security faq, 2002. http://www.w3.org/Security/Faq/.