Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 2008 Proceedings

Americas Conference on Information Systems (AMCIS)

2008

A Dynamic Query-Rewriting Mechanism for Role-Based Access Control in Databases

Jay Jarman
University of South Florida, jjarman@coba.usf.edu

James A, McCart
University of South Florida, jmccart@coba.usf.edu

Donald Berndt
University of South Florida, dberndt@coba.usf.edu

Jay Ligatti
University of South Florida, ligatti@cse.usf.edu

A DYNAMIC QUERY-REWRITING MECHANISM FOR ROLE-BASED ACCESS CONTROL IN DATABASES

Jay Jarman
University of South Florida
jjarman@coba.usf.edu

James A. McCart
University of South Florida
jmccart@coba.usf.edu

Donald Berndt
University of South Florida
dberndt@coba.usf.edu

Jay Ligatti
University of South Florida
ligatti@cse.usf.edu

ABSTRACT

Although Role-Based Access Control (RBAC) is a common security model currently, it has not been systematically applied in databases. In this paper, we propose a framework that enforces RBAC based on dynamic query rewriting. This framework grants privileges to data based on an intersection of roles, database structures, content, and privileges. All of this is implemented at the database level, which also offers a centralized location for administering security policies. We have implemented the framework within a healthcare setting.

Keywords (Required)

RBAC, Database, Security Policies, Security Mechanism, Dynamic Query Rewriting.

INTRODUCTION

Within the past decade laws have mandated mechanisms be put into place to secure corporate and health related data. A framework such as Role Based Access Control (RBAC) can provide the controls necessary to fulfill such needs. However, most RBAC security mechanisms have been implemented within client applications leaving data unsecured within databases. In this paper we propose a modified RBAC framework that allows fine-grained access of data through the use of policies and a dynamic query rewriting mechanism implemented at the database level. This framework provides a security mechanism at the same level as the data and centralizes security policies. In addition, the framework is designed to be vendor independent.

RBAC is a security model that restricts system access to authorized users and has been in existence in one form or another since 1992 (Ferraiolo and Kuhn, 1992). This model focuses on roles, permissions, users, and constraints. Roles are defined for various job functions and can also inherit the traits of a parent role in a hierarchical fashion. A permission is an approval of one or more operations on objects in the system. Users can be assigned one or many roles, and a role can be assigned to one or many users. During a session, users activate a subset of roles to which they belong. Constraints allow for the enforcement of separation of duties. A role can also have one or many permissions assigned to it, and a permission can be assigned to one or many roles.

Motivation

There are several factors motivating this paper. One factor is the need for better security mechanisms within databases. Most system security policies are implemented within the application and not on the backend database. If multiple applications access a single database, this requires security policies to be implemented several times and leaves security policy administration less centralized. Second, is the need for security policies that allow fine-grained access, which in most systems is implemented at the application level programmatically. Legislation is a third factor. Governments, especially in the USA since the corporate scandals with Enron, WorldCom, and others, have created legislation that requires security controls on information systems. There are two main acts of the US congress that direct owners of specific types of information to not only protect information but also to report on how that information is protected.

Security in the Database

There are several reasons for a security mechanism to be located within the database. Whether the data is accessed through an application or directly using a tool, if a security mechanism is correctly implemented within the database, the security policies will be executed. If a security mechanism is implemented only at the application level, a user can use a SQL editor

and bypass the application security to access data he or she is not supposed to have access to. Also, if there are multiple applications accessing the same database, the security policies have to be implemented within each of the different applications. By having security policies implemented at the database level they are implemented once and can be centrally managed.

Fine-Grained Security

The second motivating factor is the need for security policies that allow fine-grained access. Currently, some Database Management Systems (DBMSs) implement security policies using an RBAC framework. For example, a role can be created within a database that allows a user with that role to have read access to an entire column in a table and read/write access to another column in a table. However, many DBMSs lack the ability to define policies that allow fine-grained access, which allows a user access to a specific row column intersection based on a policy. For example, the nurse role is allowed access to the columns in the patient table but for only those patients admitted to the floor on which the nurse is working. Currently, this is mainly handled at the application level programmatically. Still other DBMSs have a proprietary solution to implement fine-grained security policies. This paper proposes a non-proprietary solution in which fine-grained security policies can be handled by a DBMS.

Legislation

In 2002, the United States Congress enacted the Public Company Accounting Reform and Investor Protection Act of 2002, or more commonly known as Sarbanes-Oxley. This act holds the CEO and CFO of a corporation responsible for the corporation's financial reports. Specifically, Section 404, Management Assessment of Internal Controls, requires that a corporation's annual report contain an internal control report, which both states the responsibility of management for maintaining an internal control structure for financial reporting procedures and contain an assessment of the structure and procedures (PCAOB, 2002).

In 1996, the United States Congress enacted the Health Insurance Portability and Accountability Act (HIPAA). Among other things, it requires the Department of Health and Human Services to draft rules, which create standards for the use and dissemination of health care information. These rules establish that administrative, physical, and technical security safeguards are required for Protected Health Information (PHI) (104th US Congress, 1996).

Related Work

Kuhn and Ferraiolo have done a significant amount of research into information security at the Computer Security Division of the National Institute of Standards and Technology (NIST) where they developed the concept of RBAC (Ferraiolo et al. 1995; Ferraiolo and Kuhn, 1992). The NIST submitted the original RBAC model for the American National Standards Institute (ANSI) RBAC model (ANSI-INCITIS, 2004). In the process of having the standard approved, several researchers gave input as to what the final model should look like (Li, Byun, and Bertino, 2007; Jaeger and Tidswell, 2000). The remainder of this paper references the most widely known RBAC model, ANSI/INCITS 359-2004 RBAC. The NIST model for RBAC that has been accepted as the ANSI standard is shown in Figure 1 (Sandhu, Ferraiolo, and Kuhn, 2000).

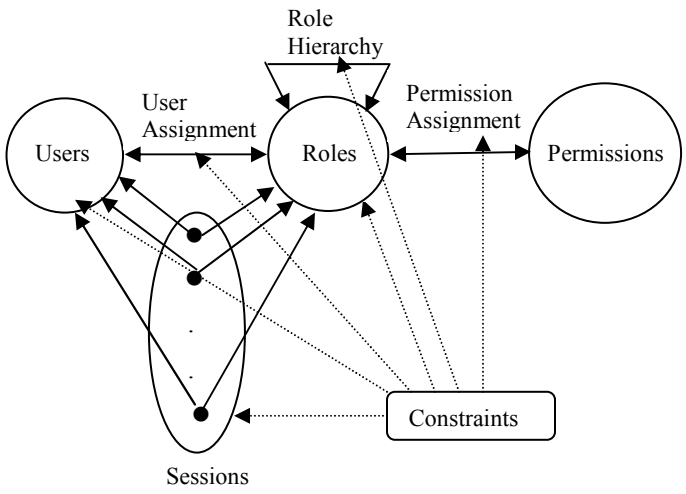

Figure 1 – NIST RBAC Model

RBAC or a modified version of RBAC is used in many systems to prevent global access to information and to grant access to only the information needed to those who need it. Many have researched either modifying RBAC or a specific implementation of RBAC (Chandramouli, 2001; Fernandez, 2006; Crampton, 2003; Chou, 2005; Hoffman, 1997; Sandhu, Coyne, Feinstein, and Youman, 1996). Gone are the days of super users having global access to all of the system's data. Business analysts define operations and roles and security specialists administer the security policies for systems by physically creating the roles and granting permissions to those roles and then assigning users roles. In addition, system security policies are audited not only by an auditor internal to the company but also by third party auditing firms.

Most researched RBAC solutions are application-level frameworks (Chou, 2004; Chandramouli, 2001; Chou, 2005; Hoffman, 1997). Some research has been done concerning RBAC within a database context. Chandramouli (Chandramouli, 2001) found three major Relational Database Management System (RDBMS) had implemented RBAC. However, most of the access controls where not fine-grained and focused on table-level rights, such as creating or modifying a table.

A common approach to limiting users' access to information in databases is through the use of views. One disadvantage to doing this is that many views have to be created, tested, and deployed in order to implement the security policies for a system. The work of administering these policies would also be transferred from a security administrator to a database programmer who would have to build all of the views.

Oracle has developed Virtual Private Database (VPD), which is an instrument for controlling database access at the row and column level. VPD provides this control by allowing administrators to create functions, which implement security policies, and assign these functions to database objects such as tables or views. When a SQL statement accesses a database object having a function assigned to it, the function is executed and it returns a string. This string is then dynamically added to the original SQL statement's WHERE clause and the new rewritten SQL statement is then executed by the DBMS (Oracle, 2007). For example, a policy could exist that restricts nurses from viewing patient data to only patients on a nurse's floor. An original SQL statement might be

```
SELECT lname, fname
FROM patient;
```

After the function is executed the rewritten SQL statement would be

```
SELECT lname, fname
FROM patient
WHERE floor = 'pediatrics';
```

This would restrict the nurse to seeing only patient data for the patients on the pediatric floor rather than all of the patients registered in the hospital.

Oracle's VPD is similar to what we are proposing in our framework with two distinct differences. First, VPD only dynamically adds a WHERE clause (Oracle, 2007). This becomes problematic if the security policy restricts data based on a data value that exists in a table that does not exist in the original SQL statement. Suppose the previous policy exists. Nurses are restricted to viewing patient data only for patients that are admitted to the same floor as the nurse only in this case, the patient's admission data resides in a separate table. The original SQL statement would be

```
SELECT patient.lname, patient.fname
FROM patient;
```

The VPD function would execute and return the WHERE clause string which would be dynamically added to original SQL statement and result in

```
SELECT patient.lname, patient.fname
FROM patient
WHERE patient_visit.floor = 'pediatrics';
```

The patient_visit table does not exist in the original SQL statement and since VPD functions only return WHERE clauses, the execution of this SQL statement would result in an error. If VPD could also dynamically add tables to the FROM and add joins to the WHERE clause, this would solve the problem. It is not impossible to write a function that would enforce such a policy through the WHERE clause but it is not intuitively obvious and would require a database programmer rather than a security administrator. The framework we propose dynamically adds tables to the FROM and adds the appropriate joins to the WHERE clause.

The second distinct difference is that VPD is not RBAC based which causes more work in the maintenance of users and their applicable security policies. Reporting users' security policies is difficult because the policies are stored inside functions. For example, a user may have function f_data_entry_write but without knowing what that actual function does, it will be difficult to determine a user's security. Our framework is RBAC based. It allows policies to be defined for roles, which eases some maintenance overhead.

Hippocratic databases are another available solution to controlling database access. These databases were designed in order to take responsibility for the privacy of the data they contain (Agrawal, Kiernan, Srikant, and Xu, 2002) and are applicable to data-sensitive fields such as healthcare (Agrawal, Kini, LeFevre, Wang, Xu, and Zhou, 2004) and finance (Agrawal, Asonov, Bayardo, Grandison, and Johnson, 2005). IBM's Hippocratic Database (HDB) is one implementation that provides database security by enforcing security policies, auditing usage, and allowing data to be shared amongst databases (Agrawal et al., 2005). The Active Enforcement (AE) component of the HDB processes security policies by re-writing queries, similar to the mechanism outlined in this paper. However, the HDB is a middleware layer positioned between the database and any applications that may access it (Agrawal et al., 2005). Therefore, the ability to maintain secure control of data requires all access to the database take place through the middleware. Direct access to the database or through an application not using the middleware would completely bypass the security measures and render them totally ineffective. The query-rewriter outlined in this research is housed within the database and thus is not susceptible to such a security flaw.

Contributions

We propose a framework for enforcing RBAC at the database level rather than the application level using dynamic query rewriting. This framework takes a submitted SQL statement and dynamically rewrites it according to security policies. The query rewriter adds rules in the form of WHERE clauses to create a new SQL statement that is then submitted to the DBMS. With this method, security policies are enforced regardless as to whether the database is accessed through an application or directly using a tool. This framework is DBMS vendor independent and allows for attribute-level granularity, where an attribute is the intersection of a row and column.

In implementing this framework, we have created a set of meta-tables in which to store the data that make up the security policies. Also stored in the meta-tables are data about tables making up the database to be secured. This allows for joins, necessary for policies, to be created dynamically.

Due to the interest in HIPAA and security in the healthcare sector, we chose to implement this framework in a healthcare setting. A small sample database was created and populated with data to demonstrate healthcare related policies. Several example SQL statements were also created to demonstrate how those queries would be rewritten in the presence of security policies. The meta-tables were then populated based on the sample database. For each example, this paper will report on the SQL statement before and after the rewrite, as well as the results from the execution of the rewritten statement.

In summary, we contribute the following:

- A novel technique for enforcing RBAC based on dynamic query rewriting
- A proof-of-concept implementation of the technique
- A small case study in a healthcare setting

We proceed as follows. The next section contains the description of our modified RBAC framework. Following that, examples illustrating the model within the healthcare domain are presented. The fourth section describes the implementation, and the final section concludes.

DYNAMIC REWRITING OF QUERIES (DRQ)-RBAC FRAMEWORK

We propose a security framework as depicted in Figure 2. It combines portions of RBAC with what we define as a policy. A policy is the conjunction of a rule with a privilege. A rule contains a structure and content. The structure is the database structure or more specifically, the column within a specific table. The content is the column value for a particular row. A privilege is a structure that maps traditional data access levels (CRUD – Create, Read, Update, and Delete) to a specific structure. For example, for the role nurse, the "name" column of the diagnosis table may be assigned the access level read. An intersection of these components will allow users to access the appropriate data. Figure 2 shows these new components added to the ANSI/INCITS 359-2004 RBAC model. The separation-of-duties constraints of RBAC have been removed from this diagram for ease of reading. The user-submitted SQL statement is sent to the Dynamic Rewriting of Queries Engine (DRQE), which uses the policy to dynamically rewrite the query to ensure it does not violate applicable security policies.

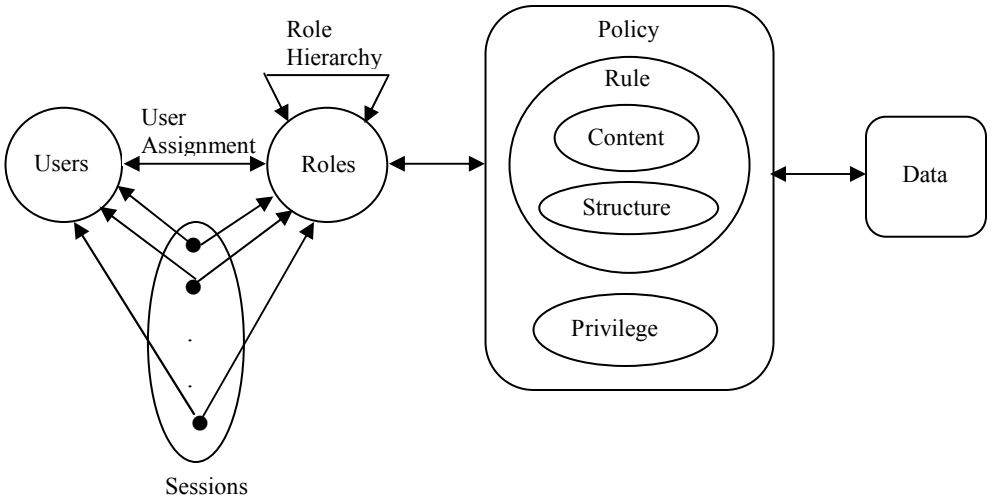

Figure 2 DRQ-RBAC Model

Query Rewriting Process

The DRQ-RBAC framework relies on a series of steps to ensure that only authorized users are capable of accessing the data they are authorized to access. Figure 3 outlines the process the DRQE follows. A user is defined as anyone with authorized access to the database including Database Administrators (DBAs). The entire process assumes that a user has already been successfully logged into the database and is currently operating within a user session. Therefore, the DRQE has knowledge of which user is attempting to access the data and can use that information on which to base access decisions.

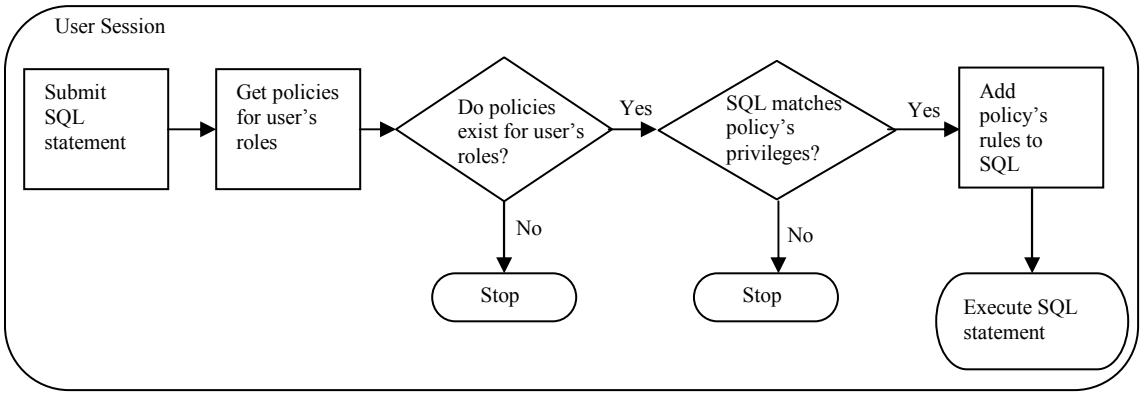

Figure 3 – DRQE Process

SQL, Roles, and Policies

The first step of the process occurs when the user submits a SQL statement to the database. The DRQE determines all the roles that have been assigned to the user and retrieves any polices that have been assigned to those roles. The DRQ-RBAC framework centers on the use of policies to grant users access to the underlying data. When a role has not been assigned to any policies, that role has no access rights within the database. Therefore, if there are no policies that have been assigned to any of the user's roles then the DRQE stops the process and returns an error informing the user "No policies exist for this user's role(s)."

Policy Privileges

If there are policies assigned to any of the user's roles then the DRQE continues onto the next step. The DRQE goes through each applicable policy and determines if the submitted SQL statement is attempting to access any structural aspects of the database that the policy does not allow. The submitted SQL statement is parsed to determine the type of SQL statement submitted (i.e., SELECT, INSERT, UPDATE, DELETE) along with the structural components the SQL statement is trying to access (i.e., the columns). The parsed information is then compared against the allowable privileges of the policy. If there is any attempt to access structures not explicitly allowed by any of the applicable policies then the DRQE stops the process and returns an error. This was a design decision in order to ensure the principle of atomicity when it comes to manipulating data.

If a policy does explicitly permit access to what the submitted SQL statement is attempting to access, the process continues with the applicable policy. In many situations a user's submitted SQL statement matches more than one policy's privileges. In those cases, all applicable policies will be used in the remainder of the process.

Policy Rules

In the next step, the DRQE appends any of the applicable policies' rules onto the submitted SQL statement. This action restricts the accessible data to only those records allowable by the pre-specified rules of the policy. In the case multiple policies explicitly permit access, the rules of both policies are appended onto the submitted SQL statement and separated via an OR statement.

SQL Execution

The final step in the process is that the transformed SQL statement is executed. The result of the transformed SQL statement is access only to data that has been explicitly allowed via a policy that the user's role has been assigned.

Dynamic Policy Change

As each policy is interpreted at query-time any changes made to the rules of a policy are automatically implemented the next time the policy is used.

SCR Meta-Tables

The data needed to support the DRQ-RBAC framework is stored in the DQRE meta-tables. An entity-relationship diagram (ERD) of the DRQE meta-tables is shown in Figure 4. The DRQE meta-tables were designed to support three goals. First, the meta-tables must be self-contained and hold all necessary information needed to enforce policies defined within the DRQ-RBAC framework. This removes the dependency of needing to rely on DBMS system tables for potentially crucial information (e.g. users and roles). Second, there must be a separation between the underlying database and the DRQE meta-tables. As many organizations currently have databases in place, it would be an unreasonable requirement that the underlying database be changed in such a way to work within the DRQ-RBAC framework. Third, the meta-tables must be generalizable regardless of database design or problem domain.

HOSPITAL EXAMPLE

To illustrate the manner in which the DRQ-RBAC framework operates, we present two examples. While this framework is meant to be general enough to be applicable to all domains, our examples focus on the healthcare industry due to the sensitive nature of the data and the robust set of regulations surrounding it, such as HIPAA. Figure 5 is an ERD of a simple hospital database that will be used throughout the remainder of this paper. This hospital database allows personal information regarding a patient to be stored, along with their date of stay, diagnosis, and assigned physicians. The data tables have been excluded from this document due to space constraints.

Figure 6 provides sample data from the hospital database and is used throughout the examples. This data has been denormalized.

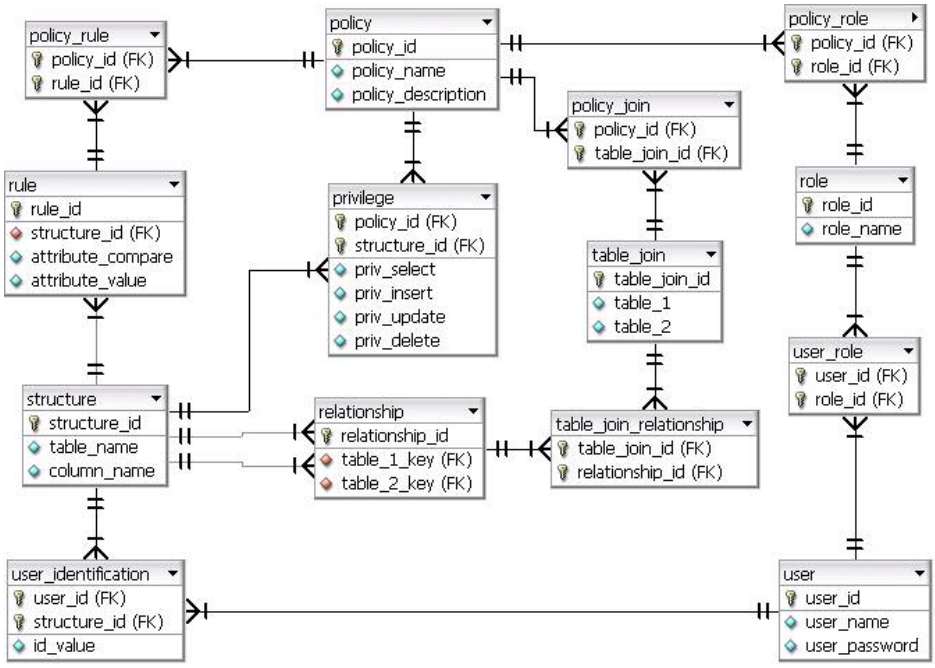

Figure 4 – DRQE Meta-Schema

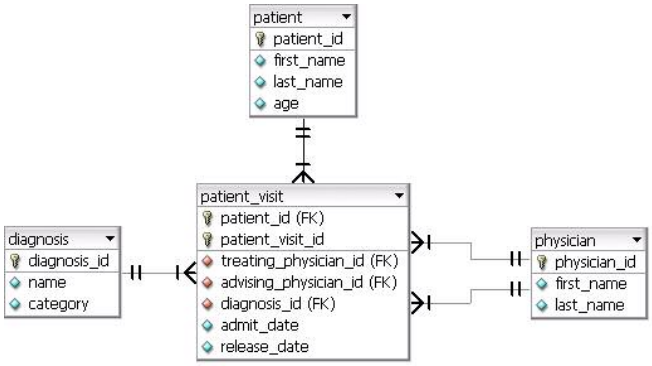

Figure 5 – Hospital ERD

Patient ID	First Name	Last Name	Age	Name	Category	Treating Physician ID	Advising Physician ID	Admit Date
1	John	Doe	87	West Nile Virus	B	1	2	03/30/07
2	Sally	Jones	1	Broken Leg	C	2		03/09/07
3	Tonya	Smith	23	Broken Arm	C	2	3	03/01/07
4	George	Adams	4	Cancer	A	1		03/31/07
5	Stan	West	3	Broken Leg	C	1		03/18/07

Figure 6 – Sample hospital data

Example 1 – Basic Policy

This first example illustrates a policy that has been assigned to the role of a CDC Heath Official and has only one rule, to restrict the viewing of patient data to only patients diagnosed with West Nile Virus. Assume a user is logged into the hospital database, has been assigned the role of a CDC Health Official, and submits the following SQL statement.

```
SELECT p.first_name, p.last_name, pv.admit_date
FROM patient p, patient_visit pv
WHERE p.patient_id = pv.patient_id;
```

The DRQE locates one policy applicable to this user and determines that the policy allows SELECT privileges on the columns specified in the submitted SQL. The rule restricting access to only patients diagnosed with West Nile Virus is appended to the SQL statement and a dynamic join is created between the rule and the original SQL statement. The result of the DRQE is the following SQL statement, which is executed and returns patient John Doe as having West Nile Virus and being admitted on 3/30/07. This also demonstrates an important feature of the DRQE. This example's policy uses tables not in the original SQL statement, specifically, the diagnosis table. It adds this necessary table for the policy to the FROM and adds the necessary statement to WHERE for the joins.

```
SELECT p.first_name, p.last_name, pv.admit_date
FROM patient p, patient_visit pv, diagnosis d
WHERE p.patient_id = pv.patient_id
AND d.diagnosis_id = pv.diagnosis_id
AND d.name = 'West Nile Virus';
```

Example 2 – Multiple Rule Policy

The next example illustrates a policy that has been assigned to the role of a Child Services Official and has two rules. The first rule restricts the viewing of patient data to only patients diagnosed within a particular category of diagnoses, with the actual diagnoses being determined dynamically via a sub-query. The second rule requires the patients' ages to be four years old or younger. Assume a user is logged into the hospital database, assigned the role of a Child Services Official, and submits the following SQL statement.

```
SELECT first_name, last_name, age FROM patient;
```

Both rules of the policy are appended to the SQL statement via an AND operation to ensure only data meeting both rules are seen by the user. The resulting SQL statement below is executed and returns patients Sally Jones and Stan West ages 1 and 3, respectively.

```
SELECT p.first_name, p.last_name, p.age
FROM patient p, patient_visit pv, diagnosis d
WHERE p.patient_id = pv.patient_id
AND d.diagnosis_id = pv.diagnosis_id
AND d.diagnosis_id IN (SELECT diagnosis_id FROM diagnosis WHERE category = 'C')
AND p.age <= 4;
```

IMPLEMENTATION

A proof-of-concept implementation was created in order to verify that the DRQE meta-schema and the DRQE provide the functionality specified in this paper. The DRQE meta-schema and the example hospital schema were loaded into a MySQL 5.0 database along with the corresponding data. The DRQE was written as a front-end application in Java 1.6 that consisted of 4 classes and 654 logical SLOC. Each example provided in the previous section was confirmed to work as specified.

Through the process of implementing the DRQE a number of issues arose which helped strengthen the design of the DRQE meta-schema and the logic of the DRQE. Earlier designs of the DRQE meta-schema failed to capture relationships between tables in the underlying database that may need to be joined dynamically. Additionally, the logic of the DRQE failed to take into account the manner in which joins would be determined when linking the submitted SQL statement to the SQL added by the rules.

Although the DRQE is meant to reside as a part of the database, for this initial implementation our focus was on verifying the correctness and ability of the DRQE meta-schema and the DRQE to work as envisioned. Future research will transfer the DRQE from the application tier to the database tier.

CONCLUSIONS

In this paper we have proposed a new technique for enforcing RBAC policies in DBMSs by dynamically rewriting queries. As far as we are aware, this is the first such framework, which is capable of fine-grained access control within a database. We have also provided a proof-of-concept implementation of the DRQE and finally, we have presented a small case study of the DRQE within a healthcare setting.

We believe the use of a security mechanism at the database-level is advantageous to mechanisms employed at the application-level. For instance, regardless of the manner in which the database is being accessed, a layer of security is always present. This is in stark contrast to applications that require the use of the application in order to constrain access according to security policies. The DRQ-RBAC framework also provides a centralized repository in which to store and execute all security policies, independent of the applications accessing the database. This provides less coupling between the application and the security policies so organizations are free to utilize multiple applications or change applications without needing to administer security policies separately for each application. Our security framework also has a large benefit over current DBMS security mechanisms by providing fine-grained cell-level access to data.

There are many areas we wish to explore in the future. One area we consider a priority is defining a language in which the policies and rules of the DRQ-RBAC framework can be fully expressed, along with a tool to help administer policies. We are also interested in implementing the DRQ-RBAC framework on a large-scale example database, such as the one outlined in this paper. After that we are interested in testing the framework within an actual organization to further refine the framework. Lastly, the performance of a system implementing the DRQ-RBAC framework needs to be examined to determine if justifications can be made regarding any overhead incurred.

REFERENCES

1. 104th US Congress. (1996) Public Law 104-191, Health Insurance Portability and Accountability Act of 1996.
2. Agrawal, R., Kiernan, J., Srikant, R., & Xu, Y. (2002) Hippocratic Databases *Proceedings of the 28th Very Large Database Conference*, August, 2002, Hong Kong, China.

3. Agrawal, R., Kini, A., LeFevre, K., Wang, A., Xu, Y., & Zhou, D. (2004) Managing Healthcare Data Hippocratically, *ACM SIGMOD International Conference on Management of Data*, June 2004, Paris, France.
4. Agrawal, R., Asonov, D., Bayardo, R., Grandison, T., & Johnson, C. (2005) Managing Disclosure of Private Financial Data with Hippocratic Databases, IBM Almaden Research Center, January, 2005.
5. ANSI-INCITIS. (2004) American national standard for institute - role-based access control, ANSI-INCITIS 359-2004, Feb 2004.
6. Chandramouli, R. (2001) A framework for multiple authorization types in a healthcare application system, *Proceedings of the 17th Annual Computer Security Applications Conference*, 2001, 137-148.
7. Chou, S. (2005) An RBAC-based access control model for object-oriented systems offering dynamic aspect features, *IEICE Transactions on Information and Systems,* E88-9, 9, 2005, 2143-2147.
8. Chou, S. (2004) Embedding role-based access control model in object-oriented systems to protect privacy, *Journal of Systems and Software*, 71, 1-2, Apr 2004, 143-161.
9. Clark, D. D. and Wilson, D. R. (1987) A comparison of commercial and military computer security policies, *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, May 1987, 184-194.
10. Crampton, J. (2003) Specifying and enforcing constraints in role-based access control, *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies* Jun 2003, 43-50.
11. Fernandez, R. (2006) Enterprise dynamic access control, version 2, *US Navy, COMPACFLT*, Jan 1, 2006, 1-42.
12. Ferraiolo, D. F., Cugini, J., and Kuhn, D. R. (1995) Role based access control: Features and motivations, *Proceedings of the 11th Annual Conference on Computer Security Applications,* 1995.
13. Ferraiolo, D. F., and Kuhn, D. R. (1992) Role based access control, *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, 1992.
14. Hoffman, J. (1997) Implementing RBAC on a type enforced system, *Proceedings of the 13th Annual Computer Security Applications Conference*, Dec 1997.
15. Jaeger, T., and Tidswell, J. E. (2000) Rebuttal to the NIST RBAC model proposal, *The Fifth ACM Workshop on Role-Based Access Contro,* 2000, 65-66.
16. Li, N., Byun, J., and Bertino, E. (2007) A critique of the ANSI standard on role based access control, *IEEE Security and Privacy*, 5, 6, 1-30.
17. Oracle. (2007) Oracle Database Security Guide 11g, Release 1, Oct 2007.
18. Public Company Accounting Oversight Board, PCAOB. (2002) PCAOB standards and related rules - auditing standards no. 2 http://www.pcaobus.org/Standards/Standards_and_Related_Rules/Auditing_Standard_No.2.aspx), Apr 29, 2007.
19. Sandhu, R. S., Ferraiolo, D.F., and Kuhn, R. (2000) The NIST model for role based access control: Towards a unified standard, *Proceedings, 5th ACM Workshop on Role Based Access Control*, July 26-27.
20. Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996) Role-based access control models, *Computer*, 29, 2, 1996, 38-47.