Reverse engineering of database security policies

Salvador Martínez[1], Valerio Cosentino[1], Jordi Cabot[1] and Frédéric Cuppens[2]

[1] ATLANMOD, & École des Mines de Nantes, INRIA, LINA, Nantes, France
{salvador.martinez_perez,valerio.cosentino,jordi.cabot}@inria.fr
[2] Télécom Bretagne; Université Européenne de Bretagne Cesson Sévigné, France
frederic.cuppens@telecom-bretagne.eu

Abstract. Security is a critical concern for any database. Therefore, database systems provide a wide range of mechanisms to enforce security constraints. These mechanisms can be used to implement part of the security policies requested of an organization. Nevertheless, security requirements are not static, and thus, implemented policies must be changed and reviewed. As a first step, this requires to discover the actual security constraints being enforced by the database and to represent them at an appropriate abstraction level to enable their understanding and reenginering by security experts. Unfortunately, despite the existence of a number of techniques for database reverse engineering, security aspects are ignored during the process. This paper aims to cover this gap by presenting a security metamodel and reverse engineering process that helps security experts to visualize and manipulate security policies in a vendor-independent manner.

1 Introduction

Relational databases are at the core of most companies information systems, hosting critical information for the day to day operation of the company. Securing this information is therefore a critical concern. For this purpose, both, researchers and tool vendors have proposed and developed security mechanisms to ensure the data within the database system is safe from possible threats. Due to its relative conceptual simplicity, one of the most used mechanisms within DataBase Management Systems (DBMSs) are *access control (AC)* policies where Role-based access control (RBAC) [10] is the current trend.

However, and despite the few methods that attempt to automatically derive these policy implementations from high-level security specifications[8,3,2], the task of implementing an access control security policy remains in the vast majority of cases a manual process which is time-consuming and error-prone. Besides, several database mechanisms may be needed in combination to implement a policy, e.g., triggers can be used to add fine-grained control on privileges, scattering the policy and increasing the complexity of the definition process. Furthermore, as security requirements are rarely static, frequent modifications of the security policy implementation are required, what increases the chances of introducing new errors and inconsistencies.

In this context, discovering and understanding which security policies are actually being enforced by the DBMS comes out as a critical requirement for the reenginering of

the current policies, to adapt them to evolving needs of the company and also to detect inconsistencies between the enforced and the desired ones. The main challenge for this discovery process is bridging the gap between the vendor-dependent policy representation and a more logical model that 1 - express these policies in a way that abstracts them from the specificities of particular database systems and 2 - that can be understood by security experts with no deep knowledge of the particularities of each vendor. Representing and processing the security policies at this logical level is much easier than a direct exploration of the database dictionary whose structure is unfortunately not standardised. Additionally, this logical model would also allow us to implement all analysis/evolution/refactoring operations on the security policies in a vendor-independent and reusable way.

The goal of this paper is then two-fold. First we provide a means to represent such logical models for security concerns in relational DBMSs. Secondly, we describe a reverse engineering approach that can automatically create this logical model out of an existing database. Reverse engineering, as a process aimed to represent running systems at higher abstraction level, has been proved useful in many domains, including database systems [7], [4]. However, these works have focused on the database structure and ignored the security aspects. We intend to cover this gap.

Moreover, we discuss possible applications and benefits of using a model-based representation given this enables to reuse in the security domain the large number of model-driven techniques and tools. Model manipulation techniques can then be applied to visualize, analyze, evolve, etc the model. Then, a forward engineering process could be launched in order to generate the new security policy implementation ready to import in the target database system.

The rest of the paper is organized as follow. In Section 2 a metamodel able to represent models of database AC policies is provided whereas in 3 we show the needed steps to obtain such models from an existing database. In 4 we discuss the related work. We finish the paper with section 5 presenting the conclusions and future works.

2 Access Control Metamodel

This section describes our relational database-specific access control metamodel (RBAC-inspired). Next section describes how to automatically populate it based on the actual policies enforced in a particular database.

The SQL standard predefines a set of privileges and object types that can be used in the definition of a relational database. Our metamodel provides a direct representation of these concepts to facilitate the understanding of the security policies linking the security elements with the schema objects constrainted by them as depicted in figure 1. In the following, we describe the main elements of the metamodel.

Database objects: The main objects users can be granted privileges on are: tables, columns, views, procedures and triggers. Each one of these elements is represented in the metamodel by a metaclass inheriting from the *SchemaObject* metaclass. Views (metaclass *View*) can include derived columns. In fact, views can be used as a fine-grained access control mechanism. A view filters table rows and columns with respect

to a desired criteria so when a subject is granted privileges over a view, is actually being granted privileges over a table/s but with certain constraints. This actually represents *column-level* and *content-based (row level)* access control.

Privileges: We can divide the privileges that can be granted in a DBMSs in five categories: *Database-level* privileges, for those privileges that imply creation of database objects including users and roles. *Table-level* privileges for those that implies table, columns and index access. *Permission-delegation* privileges to delegate permission administration to users and roles. *Execute* privileges for the executable elements (stored procedures and functions) and *Session* privileges. These categorization is depicted in the bottom part of Figure 1. If needed, an extension of this metamodel (by inheriting from the *Operation* metaclass) could be provided to deal with vendor-specific privileges.

Subjects: Subjects executing actions on the DBMS are users and roles and they are, as such, represented in the metamodel with the corresponding metaclasses *User* and *Role*, possibly part of role hierarchies. In the metamodel, the metaclass *Role* inherits from the metaclass *Subject* which enables it to become grantee. The *User* metaclass also inherits from *Subject* what means that, privileges, in contrast to pure RBAC, can be granted to both users and roles.

Other elements and constraints: There are several other aspects that have to be taken into account. The execution of privileges may need to be constrained. In DBMSs this is normally achieved by using triggers and procedural code. The metamodel should permit the representation of the existence of such constraints. The metaclass *Constraint* meets that purpose. Typically it points to a *Trigger* linked to the object and the operation on it that is constrained by the trigger. The *Trigger* metaclass also includes the attributes the trigger body and the condition and error message to be displayed when the trigger execution fails due to any exception. Another aspect to be considered is object ownership as it is the basis for permission delegation. An association between objects and subjects records this relationship. Finally, global permission, i.e., permissions that are granted on all the elements of the same type are represented by allowing in the metamodel permissions to be granted on any object, including the metaclasses *Schema* and *Database*. A select permission granted on *Schema* or *Database* represents that the grantee can select all the tables and views contained in those objects.

3 Reverse Engineering process

The previous metamodel allows to represent database access control policies at the logical level. Models conforming to this metamodel express the security policies in place in a given database in a vendor-independent manner. These models can be manually created but ideally they should be automatically created as part of an automatic reverse engineering process that instantiates the model elements based on the information extracted out of the DBMSs. Note that, the extracted models are vendor-independent but the extraction process is not since each vendor uses different internal structures (i.e. a different set of tables/columns in the data dictionary to express this information). In fact, we could regard this “injection” process as the one that abstracts out the specific product details.

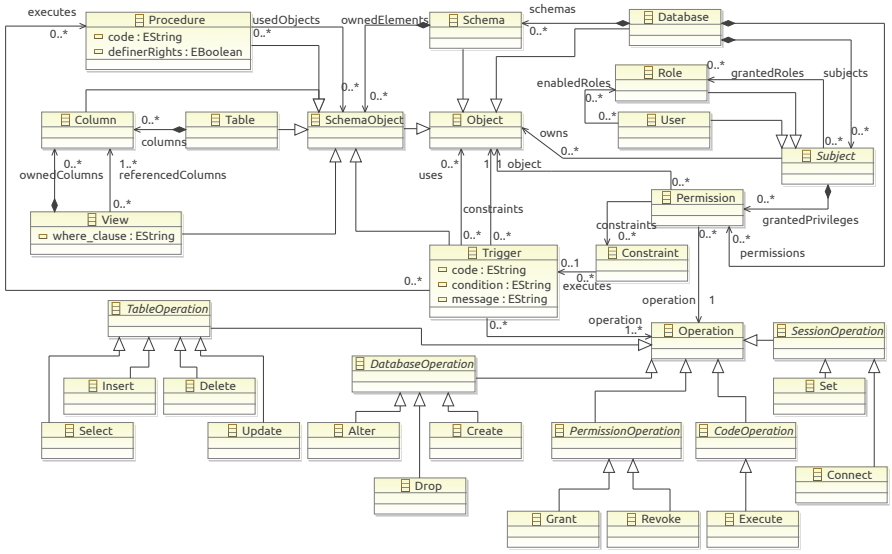

Fig. 1. DBMS Security Metamodel

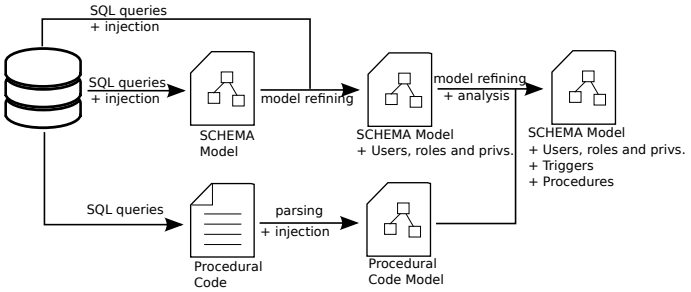

Fig. 2. Extraction process

Our reverse-engineering approach is depicted in Figure 2. It starts by populating our security model with the basic schema information (tables, views, etc). Then this model is refined to add user, roles and privileges information whereas in a last step, the bodies of triggers and stored procedures are analyzed to complement the access control policies in the model.

In the following we will detail the process for the reverse engineering of security policies over an Oracle 10g DBMS. This same process could be reused for other DBMSs changing the references to the specific data dictionary tables with the corresponding ones in that system.

3.1 Extracting general schema information

This first step populates the part of the model describing the structure of the database itself e.g., schemas, tables (and columns), views, procedures, etc. An injector (in our terminology, an injector is a software component that bridges technical spaces, in this case moving information from the database technical space to the modeling technical space) is needed. This injector connects to the database and queries the dictionary tables to retrieve all the necessary information. The selected objects are then inserted as model elements in the security model. In Oracle, our injector has to query tables like *DBA_TABLES*,*DBA_TAB_COLS* and *DBA_ROLES*.
View extraction is more challenging since we need to parse the view definition to get the list of tables (or other views), columns and conditions the view selects. The result of the parsing is a set of links between the view object and the other objects filtered by the view. The *where* clause will be copied as it is into the *condition* attribute of the view metaclass. Moreover, a view can contain derived columns. Derived columns are created as view columns in the model.

3.2 Extracting users, roles and privileges

As before, the injector queries the database dictionary for user and roles information and populate the model with the results. After this step, the general access control information of the database, i.e., subjects, objects and permissions are already present in the security model.

3.3 Extracting stored procedures and triggers information

Triggers. Complex security checks can be implemented by means of triggers that fire to prevent certain actions when performed in a certain context. However, triggers are used for a huge variety of tasks beyond security purposes. In our approach, all triggers are retrieved and parsed, then, they are analyzed with respect to a number of heuristic conditions in order to select which of them are implementing security checks and discard the rest.

The heuristic conditions analyze the following aspects:

-*Trigger kind:* Triggers are fired before/after a certain statement/s is invoked. *BEFORE STATEMENT* triggers are executed before the statement is completed, enabling the possibility of evaluating security concerns (e.g., the possibility to make inserts in certain tables could be enabled only to working days). Conversely, *AFTER STATEMENT* triggers are executed once the action of the statement is performed, so, when involved in security, they are normally used for logging purposes. Clearly, our focus should be in the *BEFORE STATEMENT* triggers. -*Trigger contents:* Although the kind of the trigger is an important hint, it is not enough. We analyze the trigger actions and conditions to find operations that are likely to be used when performing security checks like system context information checks, user information checks, exception raising's, etc.

More specifically, a trigger will qualify as a *security trigger* if it fulfills all the following heuristic conditions:

Listing 1.1. Trigger's code

```
1  TRIGGER update_job_history
2    AFTER UPDATE OF job_id,
         department_id ON employees
3    FOR EACH ROW
4  BEGIN
5    add_job_history(:old.employee_id, :
         old.hire_date, sysdate,
6      :old.job_id, :old.department_id);
7  END;
8
9  TRIGGER secure_employees
10   BEFORE INSERT OR UPDATE OR DELETE
         ON employees
11 BEGIN
12   IF TO_CHAR (SYSDATE, 'HH24:MI') NOT
          BETWEEN '08:00' AND '18:00'
13         OR TO_CHAR (SYSDATE, 'DY') IN
                ('SAT', 'SUN') THEN
14     RAISE_APPLICATION_ERROR (-20205,
15     'You may only make changes during
            normal office hours');
16   END IF;
17 END secure_employees;
```

Listing 1.2. Procedure code

```
18 PROCEDURE add_job_history
19  (
20      p_emp_id job_history.employee_id
            %type
21    , p_start_date job_history.
          start_date%type
22    , p_end_date job_history.end_date%
          type
23    , p_job_id job_history.job_id%type
24    , p_department_id job_history.
          department_id%type
25    )
26
27 IS
28 BEGIN
29   INSERT INTO job_history
30     (employee_id, start_date,
          end_date,
31       job_id, department_id)
32     VALUES(p_emp_id, p_start_date,
33            p_end_date, p_job_id,
34            p_department_id);
35 END add_job_history;
```

1. The trigger is a before statement trigger.
2. The trigger contains an exception section that raises an exception.
3. The trigger evaluates conditions on the system (ip address, host, time) or user information (name, assigned privileges).

As an example, listing 1.1 shows two triggers from the well-know Human Resources (HR) [1] schema example privided by Oracle: *Secure_employees* and *Update_job_history*. The latter one is an *AFTER STATEMENT* trigger which directly disqualifies it as a security enforcement trigger. Conversely, the former fulfils all the heuristic conditions. It is a *BEFORE STATEMENT* trigger, it raises and exception and checks system information (the time) and thus it qualifies as a *security trigger*.

Once a trigger is identified as a *security trigger*, the constraints imposed by the trigger need to be extracted and added to the model.

Stored procedures. Stored procedures can be executed with invoker or definer's rights. In the latter, the invoker role gets transitive access to certain database objects. The source code of the procedure must be analysed to obtain such set of transitive permissions. Our approach parses the store procedures and extracts the accessed database objects. These are then linked to the procedure in the database security model in order to easily retrieve them later on during the analysis phase. As an example, the *add_job_history* procedure in listing 1.2 is declared to be invoked with definer rights. The table it accesses, *Job_history*, would appear linked to this procedure in the model.

[1] http://docs.oracle.com/cd/B13789_01/server.101/b10771/scripts003.htm

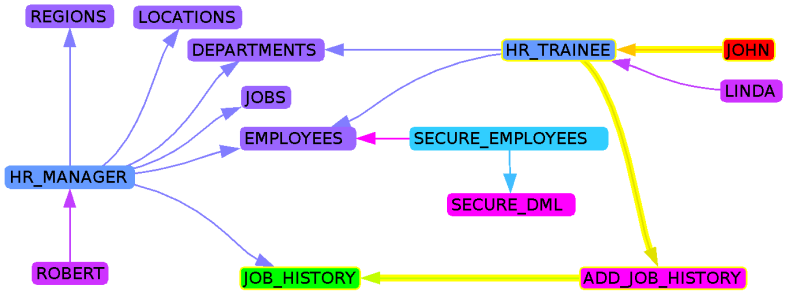

Fig. 3. Database security model visualization

3.4 Operations

The generated security model can be used in many different scenarios such as visualizations, queries, metrics, refactoring and reengineering operations. Note that working at the model level we can benefit from all existing model-driven techniques when implementing these scenarios and, furthermore, we can work at a vendor-independent level which means they can be used no matter the database system in place.

As an example of a possible interesting scenario, we show in Figure 3 a visualization obtained from a model extracted out of a modification of the HR schema. where each kind of element and relation is shown is a different color. This visualization helps to quickly grasp important information. Moreover, we can easily check if a given user can access a given table or view by just checking existing paths as the one highlighted between the user *JOHN* and the table *JOB_HISTORY*.

4 Related Work

To the best of our knowledge, ours is the first security metamodel tailored to the security mechanisms available in relational databases. Our metamodel integrates RBAC[10] concepts but extends the standard RBAC model to cover the full spectrum of database security mechanisms useful to express access-control policies. Although several (extensions to) modelling languages able to model security concerns[5,6] have already been proposed, they are aimed to model the security aspects of the whole information system and thus lack of precision to define in detail database-specific access control policies.

Regarding the reverse-engineering of security aspects, in [11] the authors present an approach to discover and resolve anomalies in MySQL access-control policies by extracting the access-control rules and representing them in the form of Binary Decision Diagrams. They do not provide a higher-level, easier to understand and manipulate representation of the extracted policies nor take into account the contribution that several other elements like triggers, stored procedures and views provide to access-control.

Finally, there exist a plethora of reverse engineering efforts [7], [4], [9],[1] (among many others) focused in recovering a (conceptual or logical) schema from a database. Nevertheless, none of them covers security aspects and therefore, they could benefit from our approach to extract a richer model.

5 Conclusions and Future Work

We have presented a reverse engineering process to extract a logical model of the security constraints enforced by a database. This model is vendor-independent and, as such, facilitates the analysis of the implemented policies by security experts not necessarily familiar with the specific details of each database system. This also facilitates comparing the security policies across different units of the company to make sure their policies are consistent with each other.

As future work we plan to continue working on the applications mentioned in section 3 and investigate the benefits of raising further the abstraction level by generating XACML specifications from our logical model to benefit from existing general security algorithms during the analysis of the security policies. Moreover, we would like to exploit the information of database audits to check the actual usage of the implemented policies (e.g. which permissions are more frequently used, which roles are never employed,...) and/or attempted attacks.

References

1. I. Astrova. Towards the semantic web - an approach to reverse engineering of relational databases to ontologies. In *ADBIS Research Communications*, volume 152 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.
2. S. Barker and P. Douglas. RBAC policy implementation for SQL databases. In *DBSec*, pages 288–301, 2003.
3. D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15:39–91, January 2006.
4. R. H. L. Chiang, T. M. Barron, and V. C. Storey. Reverse engineering of relational databases: extraction of an EER model from a relational database. *Data Knowl. Eng.*, 12:107–142, March 1994.
5. J. Jürjens. UMLsec: Extending UML for secure systems development. UML '02, pages 412–425. Springer, 2002.
6. T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 426–441. Springer, 2002.
7. J. luc Hainaut, M. Chandelon, M. Ch, C. Tonneau, M. Joris, J. l. Hainaut, M. Ch, C. Tonneau, and M. Joris. Contribution to a theory of database reverse engineering. In *in Proc. of the IEEE Working Conf. on Reverse Engineering*, pages 161–170. IEEE Computer Society, 1993.
8. S. Oh and S. Park. Enterprise model as a basis of administration on role-based access control. In *CODAS'01*, pages 165–174, 2001.
9. J.-M. Petit, J. Kouloumdjian, J.-F. Boulicaut, and F. Toumani. Using queries to improve database reverse engineering. ER '94, pages 369–386. Springer, 1994.
10. R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: towards a unified standard. In *Proceedings of the fifth ACM workshop on Role-based access control*, RBAC '00, pages 47–63. ACM, 2000.
11. M. Shehab, S. Al-Haj, S. Bhagurkar, and E. Al-Shaer. Anomaly discovery and resolution in MySQL access control policies. volume 7447 of *LNCS*, pages 514–522. Springer, 2012.