ROW-LEVEL SECURITY MODEL

Maria Dobreva, Nikolay Pavlov, Asen Rahnev, Simeon Monov

Abstract. *This paper presents the design and implementation of a new row-level security model which supports propagation of security policies over database entities following the relations between them. We step upon an existing model for a database dictionary to abstract our model from the underlying data store. We present algorithms for policy propagation and generating necessary commands to restrict access to data. A proof of concept is developed and used in a real-world application.*

Key words: row-level security, database dictionary, entity relationship.

Introduction

Every company is unique and has different network infrastructures and security needs. There are 10 policies that need to be included in every network-security policy, no matter the company's size, scope or focus: root security, PC/Workstation, Server, Email, Web-Access, Remote-Access, Mobility, Wireless Device, Internet Gateway Configuration, and Incident-Response Strategy [1]. According to the classification of Edwards, at least four of the policies are directly related to protecting against unauthorized access to data. It is a well-known practice that companies restrict the access their employees have to data based on the function and the level in the hierarchy users have within the company.

A majority of enterprises across many industries store their sensitive information in relational databases. The confidential information includes data on supply chains, clients, finance, and personnel [5, 6, 7]. In this paper we put the focus on security control on the data itself through SQL commands. Securing the network, the database server, and the databases themselves are out of the scope of this study.

This paper describes the design and implementation of support for row-level security in a software framework for distributed business applications [2].

Database Securable Objects

Within the scope of our research are the schema-related securable objects: database tables, views, procedures, functions, aggregates [4]. Relational databases commonly provide security on tables and views. A user that can read from a table can normally read all the data from it. At the same time, it is a common case that enterprises will want to restrict their employees (users) from accessing all the data stored in one entity [6]. For example, a company can restrict its dealers to access data only for clients in their geographic region. A company can decide to limit users to access to sales and payments which are performed only by their office or department. A health institution can allow the health specialists to access the data of their own patients only.

Row-level security, also known as data permission security, is a well-known security feature to restrict users from seeing or modifying some data rows. While most popular relational database management and reporting systems already support some form of row-level security, it became available in more recent versions of these systems [3].

Software systems that need to implement row-level security can rely on the existing features of their chosen relational database management and / or reporting systems. However, administration of row-level security must be performed in the database or in the reporting tool, or both. This is a task with technical complexity for database administrators or software developers, but not for end-users or application administrators.

Within relational databases, row-level security operates using a predicate function, which becomes an integral part of the database schema. Dynamic adjustments of these policies increase the complexity of maintaining the database schema through software updates.

The relational model naturally increases the attack surface. It is common for data of domain entities to spread over multiple related database tables and views. The surface is further extended by the available functions and procedures, which can expose data. Proper implementation of data-row security requires complete understanding of the schema of the database. Missing a relation can expose sensitive data.

Our Row-Level Security Model

General

Our model for row-level security aims to:

1. Improve security by propagation of security policies to related tables.
2. Remove the row-level security definitions from the database schema.
3. Improve the user-experience for definition security policies.
4. Integrate seamlessly with database layer access in the application model.

Our row-level security model also relies on predicate functions to determine the access on each level. Each predicate function is a list of Boolean conditions in the following form:

```
{column} {expression} {column | value | configuration key}
```

`{column}` is the name of a field from the database table, or a related database table. The format of the column name is defined by [2] and is:

```
FieldName [ \ FieldName [\  ... ] ]
```

It is parsed from left to right, and every name is analyzed:

1. The algorithm discovers the corresponding table field for the current name.
2. If the field is not a lookup field, the actual field is returned. No further parsing is performed.
3. If the field is a lookup field, the algorithm uses the metadata to identify the referred table. Then, it continues with the next field on the path. If the path ends with a lookup field, the algorithm automatically uses the default user identifier field for the referred table.

Examples:

`OrderNumber`: the number of the order

- `Client\Name`: the name of the client of the order
- `Client\Country`: the country primary key of the client of the order
- `Client\Country\Code`: the country code of the client of the order

`{expression}` is a conditional operator. We support:

- Comparison: `=`, `<`, `>`, `<>`, `<=`, `>=`
- Is empty
- Is not empty
- In list

`{value}` is a user-defined constant, which must correspond with the type of the data table field.

`{configuration key}` is the name of an item of the current application configuration or execution state. For example, it can be the unique identifier of the current user.

Propagation of Security Policies

A key feature of our model is the propagation of security policies. Propagation means that security policies, defined for one table, must be applied to all tables which have one-to-many relationship with the former both directly and indirectly via other such tables.

Let's consider the following data model:

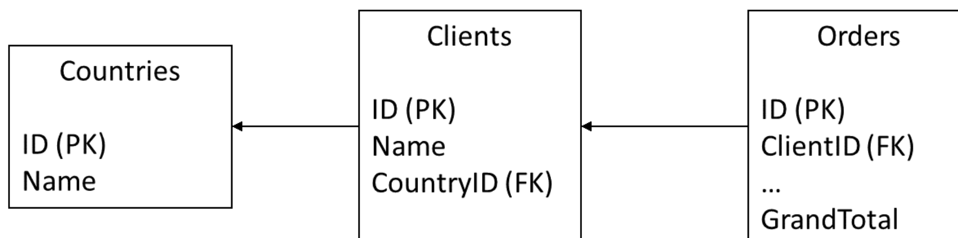

Figure 1. Propagation example

We define a security policy like this:

Table	**Rule**
`Countries`	`Name = ''Bulgaria''`

Table 1. Define security policy

Obviously, the policy will allow only access to the record from table Countries where the name is equal to "Bulgaria". Logically, though, it is expected that users with this policy must not see clients that are not from Bulgaria, and consequently, orders of clients that are not from Bulgaria. Propagation provides exactly this behavior: applying a security policy to all related entities in the chain of relationship. Thus, we guarantee that security policies cover all interrelated tables and views, and we narrow the attack surface.

Propagation in our model works only from parent to child tables. Rules are not propagated "upwards" the relationship, i.e., from child to a parent.

We construct a cyclic oriented graph of the database, where each table is a node, and each relation is a directed link. For every possible source of data in the query, which are the tables and views in the FROM clause and all JOIN-ed such, we check if there exists a policy for that table and view or any table and view that is directly or indirectly connected to it.

Implementation

We have implemented a component which integrates with the data access layer of the application. Discovered applicable security policies are used to modify the SQL statement generated by the data access layer.

The algorithm works in the following steps:

1. Analyzes the generated SQL query and breaks it down into primary parts: `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `ORDER BY`, `HAVING`.
2. For each security policy we traverse the data model graph to determine whether the policy is applicable. We use Dijkstra's algorithm [8]. Our algorithm detects cyclic relations: if we encounter a direct or indirect cycling relation, we break the traversal.
3. We add the necessary tables to the FROM clause as join operators using the metadata about relations from the database itself. We detect and eliminate redundant joins.
4. We add the necessary constraints in the WHERE clause.

For example, let's consider the following model:

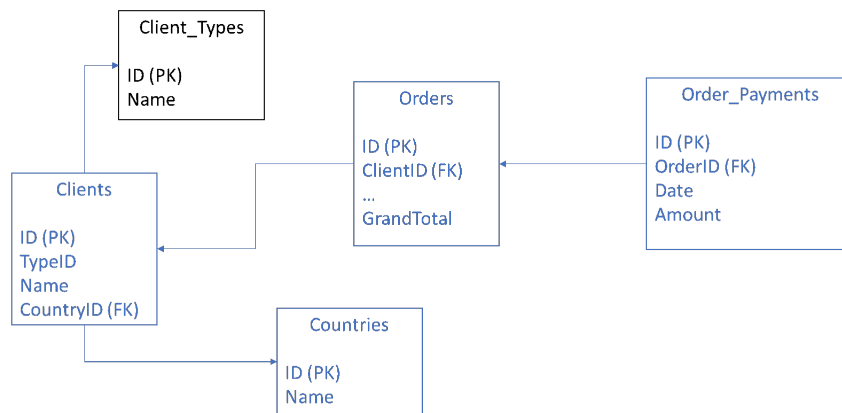

Figure 2. Data Model

And a policy:

```
Countries\Name = ''Bulgaria''
```

The original query is to retrieve the sum of payments of all customers on a given date, sorted by customer name.

```
SELECT c.Name, SUM (op.Amount)
FROM order_payments op
   INNER JOIN clients c ON c.Id = op.ClientID
WHERE op.Date = GetDate()
GROUP BY c.Name
ORDER BY c.Name
```

Original query

Despite the fact that there exists no policy for tables order payments and clients, our model will discover that the policy on table "Countries" must be applied because table "Clients" is linked to table "Countries". Our implementation will modify the query as follows:

```
SELECT c.Name, SUM (op.Amount)
FROM order_payments op
   INNER JOIN clients c ON c.Id = op.ClientID
   INNER JOIN countries c1 ON c1.id = c.CountryId
WHERE op.Date = GetDate()
   AND c1.Name = ''Bulgaria''
GROUP BY c.Name
ORDER BY c.Name
```

Query with applied policy

We cache security policies and generated SQL statements to improve performance.

As a part of our implementation, we have created a graphical user interface for authoring security policies. With the help of the database dictionary [2] we provide well-known, user-friendly names of fields and tables for the application administrators.

Conclusion

We presented a new model for row-access data security for relational databases, which extends its application to related tables and views. An implementation is created for an application for distributed business applications, which is used for the development of several software products for the insurance industry in the Netherlands. There our model is tested successfully in practice.

References

[1] J. Edwards, Company Security Policy 101, 2007, `http://www.networksecurityjournal.com/features/security-policy-101-102307/`.

[2] N. Pavlov, Rich Metadata Model for Business Applications with Database Dictionary, *International Journal of Applied Science and Technology (IJAST)*, Vol. 4, No. 2, 2014, 58–66, ISSN 2221-0997.

[3] SQL Server Row Level Security, 2022, `https://learn.microsoft.com/en-us/sql/relational-databases/security/row-level-security?view=sql-server-ver16`.

[4] SQL Server Securables, 2022, `https://learn.microsoft.com/en-us/sql/relationaldatabases/security/securables?view=sql-server-ver16`.

[5] N. Bhatnagar, Security in Relational Databases, *Handbook of Information and Communication Security*, Springer, Berlin, Heidelberg, 2010, 257–272, `https://doi.org/10.1007/978-3-642-04117-4_14`.

[6] M. Mateev, INDUSTRY 4.0 AND THE DIGITAL TWIN FOR BUILDING INDUSTRY, *Industry 4.0*, 5 (1), 2020, 29–32, `https://stumejournals.com/journals/i4/2020/1/29`.

[7] M. Mateev, Creating Modern Data Lake Automated Workloads for Big Environmental Projects, *18th Annual International Conference on Information Technology & Computer Science*, Athens, 2022.

[8] E. Dijkstra, A note on two problems in connexion with graphs, *Numerische Mathematik*, 1959, 1, 269–271, https://doi.org/10.1007/BF01386390.

Maria Dobreva[1,*], Nikolay Pavlov[2], Asen Rahnev[3], Simeon Monov[4]
[1,2,3,4] "Paisii Hilendarski" University of Plovdiv,
Faculty of Mathematics and Informatics,
236 Bulgaria Blvd., 4003 Plovdiv, Bulgaria
Corresponding author: m-dobreva@uni-plovdiv.bg