Protecting Databases from Schema Disclosure
A CRUD-based Protection Model

Óscar Mortágua Pereira, Diogo Domingues Regateiro and Rui L. Aguiar
Instituto de Telecomunicações, DETI, University of Aveiro, Aveiro, Portugal

Keywords: Access Control, Information Security, Database Schema, CRUD, Software Architecture.

Abstract: Database schemas, in many organizations, are considered one of the critical assets to be protected. From database schemas, it is not only possible to infer the information being collected but also the way organizations manage their businesses and/or activities. One of the ways to disclose database schemas is through the Create, Read, Update and Delete (CRUD) expressions. In fact, their use can follow strict security rules or be unregulated by malicious users. In the first case, users are required to master database schemas. This can be critical when applications that access the database directly, which we call database interface applications (DIA), are developed by third party organizations via outsourcing. In the second case, users can disclose partially or totally database schemas following malicious algorithms based on CRUD expressions. To overcome this vulnerability, we propose a new technique where CRUD expressions cannot be directly manipulated by DIAs any more. Whenever a DIA starts-up, the associated database server generates a random codified token for each CRUD expression and sends it to the DIA that the database servers can use to execute the correspondent CRUD expression. In order to validate our proposal, we present a conceptual architectural model and a proof of concept.

1 INTRODUCTION

As our society becomes more and more dependent on information systems, there is an increased need to protect more efficiently our data stores from malicious users. Data stores keep all types and kinds of data, such as from persons, from processes and from businesses. Independently from the type and/or kind, in many organizations, business process models and data are their key assets. As an example, we can say very often that business process models are the base for designing the conceptual and logical models for the database responsible for storing the data to be protected.

Therefore, beyond protecting data which is traditionally the main concern, it is also crucial to protect the database schema. To demonstrate the importance of this problem, we can think of an organization that rates its quality control process according to some novel parameters and/or algorithms. These entities (parameters and/or algorithms) will be mapped in the database schema in one or more tables that, if disclosed, a competitor can benefit from. Thus, in such situations, it is crucial for those organizations to keep the knowledge about their database schemas as contained as possible. Unfortunately, when following traditional approaches, the knowledge about database schemas is weakly protected. In the next paragraph we briefly explain how it happens.

There are several ways to master and therefore to disclose database schemas. They can be organized into two main categories: trustworthy and malicious. The trustworthy category is based on authorization processes. They can occur when programmers are authorized to write Create, Read, Update and Delete (CRUD) expressions for business logic. In order to accomplish this goal, programmers must have access to database schemas, partially or totally, depending on the required extent. This mastering process assumes that programmers are reliable and, therefore, can be trusted. Nevertheless, serious security violations can still take place when software is developed by third party organizations, i.e. it is outsourced. The malicious category is based on malicious processes. There are several possibilities, but at this moment we emphasize situations where programmers intentionally use CRUD expressions to disclose sensitive and/or unauthorized parts of database schemas. For example, by trying CRUD expressions iteratively and successively until disclosing the required part of the database schema. While for the first category, there is no other possibility than mastering programmers,

Pereira, Ó., Regateiro, D. and Aguiar, R.
Protecting Databases from Schema Disclosure - A CRUD-Based Protection Model.
DOI: 10.5220/0005967402920301
In *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications (ICETE 2016) - Volume 4: SECRYPT*, pages 292-301
ISBN: 978-989-758-196-0

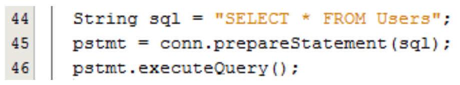
for the second category malicious users can resort to several techniques to achieve the same result. These aspects will be addressed in more detail in chapter 2 (Motivation).

A solution to avoid giving the programmers details about the server-side involves using a multi-tier architecture, also called n-tier architecture. In a multi-tier architecture, the presentation, application processing and data management are physically separated. In this architecture, a programmer does not need to know anything about the server-side expect for the set of interfaces that are provided. A related architecture is the Service-Oriented Architecture (SOA), where every feature that is implemented must be made available as a service. Other services that use them know nothing about their implementation, only their interface and functionality.

This, however, does not solve our problem which lies with the database applications, specifically what we call a database interface applications (DIA). These applications are applications that provide access to the database and may be a service provided in a SOA approach, for example, and not necessarily the client that uses it. If the DIA that provides access to the database is outsourced, the same problem with disclosing the database schema through the CRUD expressions remains. On the other hand, if the service is developed by the corporation that owns the database, the developers still need to master the database schema to write the required CRUD expressions without using any other tools.

The conclusion is that security violations can occur when CRUD expressions can be used by DIAs. In order to prevent their use, we propose a conceptual model where CRUD expressions are stored on the server-side. CRUD expressions are pushed, kept and managed by security servers and, from now on, DIAs use secure tokens instead. Basically, DIAs are provided with secure tokens and when they want to execute a CRUD expression, they send a token to the database server which, after a validation process, decides if the token is valid. If so, it evaluates if the correspondent CRUD expression can or cannot be executed. Details are provided in chapter 4. A proof of concept based on Java, JDBC and SQL Server is also presented to prove the feasibility of our architectural model. It is expected that the outcome of this research can contribute positively to the scientific community effort towards more security in database applications.

The remaining of this paper is organized as follows. Chapter 2 presents the motivation for the work detailed in this paper, chapter 3 presents the related work, chapter 4 presents the architectural model of the concept, chapter 5 presents a proof a concept of the architectural model and chapter 7 concludes this paper and details the future work.



```
String sql = "SELECT * FROM Users";
pstmt = conn.prepareStatement(sql);
pstmt.executeQuery();
```

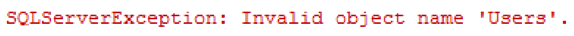
Figure 1: Select on a non-existent table.

2 MOTIVATION

The motivation for protecting the database schema comes from: the observed practice of providing an anonymized version of the database schema, along with the data, when some business was required to share it with consultants; and the fact that most DBMS expose the database schema when carefully crafted queries are issued at the DIA level. DIAs can be organized into two layers: business layers and application layers. Business layers are responsible for interacting with databases (eventually through CRUD expressions) and application layers sit on top of business layers. We resort to these layers to organize this chapter in two main sections: one for business layers and another for application layers. There is an additional section which slightly unveils the proposed solution. The presented examples are based on SQL Server and Java Database Connectivity (JDBC) for SQL Server.

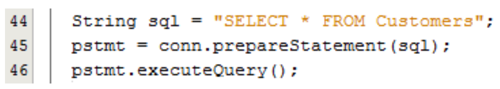
2.1 Business Logic Layer

During the development of the business layer, the developers need to master the database schema in order to write the CRUD expressions required by the application. This can pose security problems if the database schema cannot be disclosed, particularly when the development of the business layer is outsourced. Furthermore, even if the developers do not have access to the whole schema, they are able to get information about it by trying to execute other CRUD expressions. Notwithstanding the possibility that the access control rejects the execution of said CRUD expressions, the error messages generated by the Database Management System (DBMS) can disclose information about database schema. To exemplify this problem, fig. 1 shows a Select statement that is trying to select all the information of the table Users. Even if not allowed to execute Select statements (due to access control constraints) SQL Server returns the error message presented in Fig. 2. This error message states that the table Users does not exist, this way disclosing information about the database schema.



```
SQLServerException: Invalid object name 'Users'.
```

Figure 2: Raised error message for Select of Fig. 1.



```
String sql = "SELECT * FROM Customers";
pstmt = conn.prepareStatement(sql);
pstmt.executeQuery();
```

Figure 3: Select on an existing table.

The same user can continue to issue other Select statements as the one shown in Fig. 3. In this case, the table exists but due to access control constraints, an error message is raised, as shown in Fig. 4.

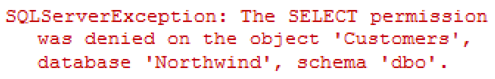
This error message also discloses information about the database schema. Basically, the user became aware that the database contains a table named as Customers.

If the database is not effectively protected by access control policies, the same user can try to disclose sensitive schemas and data. In the example shown in Fig. 5, the user (after becoming aware that table Customers exists) is trying to find out if table Customers has columns ContactName, Address and City. Similarly to previous examples, if some error message is raised, the user can modify the query and retry its execution. Iteratively, the user will end up disclosing the table schema.

Finally, if an API such as JDBC is being used, then users can have access to schemas through metadata after issuing a Select statement. The metadata allows users to obtain information such as table names, column names, column data types and other critical information, see Fig. 6. In this case we can disclose: database schemas at line 41 and the database tables at line 47, among others.

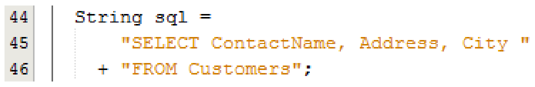
2.2 Application Layer

We can think that by decoupling the development process of business and application layers and assigning them to programmers playing different roles, it is possible to hide (by encapsulation, wrapping or some other technique) the CRUD expressions being used by business layers from programmers of application layers. Unfortunately, this is not possible. We emphasize the usage of reflection mechanisms of programming languages, which can be used to expose the CRUD expressions when the business layer is collocated with the application layer in the same application and, therefore, expose the database schema. Moreover, resorting to reflection it is also possible to implement additional security violations, such as replacing and modifying the underlying CRUD expressions. One of the most well-known technique is SQL Injection(Anley, 2002; Halfond et al., 2006).



```
SQLServerException: The SELECT permission
   was denied on the object 'Customers',
   database 'Northwind', schema 'dbo'.
```

Figure 4: Raised error massage for the select in Fig. 3.



```
String sql =
    "SELECT ContactName, Address, City "
  + "FROM Customers";
```

Figure 5: CRUD to disclose a table schema.

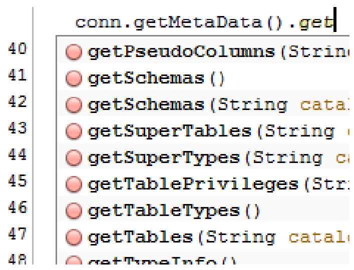
2.3 Solution Proposal

As previously explained, if CRUD expressions are allowed to be used on DIAs, there is no possibility to prevent database schemas from being disclosed. Therefore, to prevent attackers from accessing the used CRUD expressions on the database, we propose a technique that prevents their use by pushing them to the server-side, i.e. the database server. From now on, at the DIAs, CRUD expressions are replaced by secure tokens. These secure tokens are randomly and dynamically generated at runtime to avoid any security violation using other security breaching techniques, such as replay attacks. Replay attacks are attacks where a malicious user obtains valid token pairs by capturing the communication between a legitimate DIA and the server-side application, then using the same token pairs to request the execution of the associated CRUD expressions. To generate and manage the required tokens at runtime we have a supervisor application.

3 RELATED WORK

This section presents the different approaches used to secure the database schema. Regarding database schema, a lot of effort has been put into mapping the schema (the relational model) to the object oriented paradigm usually present in applications(Erhieyovwe et al., 2013; Pereira et al., 2011; Russell, 2008). For this, we have solutions such as Hibernate(Bauer and King, 2005) and Eclipse Link(Eclipselink, 2013), and even object-oriented databases(Bagui, 2003). They aim at freeing developers from the need to master the database schema, but they do not directly protect the schema used in the database. Basically, programmers can still write CRUD expressions and evaluate the results of their execution.

One possible reason for the lack of effort put into protecting the database schemas is because there is already a commonplace solution present in most DBMS, which are the stored procedures (Garcia-Molina, 2008; Sumathi and Esakkirajan, 2007; Rohilla and Mittal, 2013). Stored procedures do protect



```
conn.getMetaData().get
getPseudoColumns(Strin
getSchemas()
getSchemas(String cata
getSuperTables(String
getSuperTypes(String c
getTablePrivileges(Str
getTableTypes()
getTables(String catal
getTypeInfo()
```

Figure 6: Metadata provided by JDBC.

the database schema by encapsulating the CRUD expressions in the server-side. However they also have some problems. Among them, we emphasize that the use of stored procedures does not scale well because in complex database applications, the number of stored procedures would increase (in some degree) with the number of CRUD expressions. Another issue, eventually the most relevant, is that their names are static, meaning that they cannot be randomized. Due to this, users can try to execute them once they know their names.

There is also the possibility of using views to access the data on a database (Roichman and Gudes, 2007; Chaudhuri et al., 2007; Wilson, 1988). Views are defined with a select expression and users access the data provided by that expression, instead of accessing the tables directly. Nevertheless, the use of views does not scale well. Basically, their number also increases as the number of CRUD expressions increases.

As we have mentioned, using the multi-tier architecture allows the application processing and data management to be physically separated. Thus not allowing a client to connect directly to the database. However, this solution is not aimed to be used at the user level, but instead at the web server level that the users connect to. In many cases, these web servers do connect to the database directly, and all the assumptions we have made still hold. In fact, we can see that they are the definition of a DIA, not the user application/browser/etc.

Finally, in (Pereira et al., 2014; Pereira et al., 2012) is presented an architecture where business logic is dynamically built at runtime and in accordance with the established access control policies. Thus, CRUD expressions are deployed in each DIA but only at runtime. Nevertheless, malicious users can resort to reflection mechanisms to disclose the database schema.

4 ARCHITECTURAL MODEL

Our main objective with this work is to avoid the deployment of CRUD expressions in DIAs to prevent database schemas from being disclosed. We start by presenting a general conceptual model. Next, we present a possible implementation for the conceptual model.

4.1 Conceptual Model

As previously shown, to protect database schemas from being disclosed by CRUD expressions, CRUD expressions cannot be deployed in DIAs. However, the use of CRUD expressions cannot be avoided. CRUD expressions are unavoidable key entities to interact with database servers (those based on the SQL standard). This means that they need to be deployed somewhere and also that, to be executed, database servers need to know them. The first obvious attempt is to deploy them on database servers. However, this solution can only be feasible if the next two questions are positively answered. How can DIAs identify the CRUD expressions to be executed? 2) Is this solution effectively secure?

To answer the first question we need a way to provide DIAs with a substitute for each CRUD expression. We call this substitute a token. Basically, a unique token is associated with each CRUD expression. From now on, DIAs send tokens to database servers instead of CRUD expressions. However, this solution is not secure. For example, if each CRUD expression is assigned a fixed token, then malicious users can resort to reflection mechanisms to disclose tokens. Thereafter, being in possession of the tokens, the users can execute unauthorized CRUD expressions. As previously stated, even if the execution of some CRUD expression is not authorized by any established access control policy, the returned error message always discloses some information about the database schema. This security problem is independent from the policy to deploy tokens (statically or dynamically in DIAs).

To overcome this security gap, tokens need to be generated such way that their values cannot be reused outside the context in which they were generated. By context we mean the running instances of the DIAs and not the particular application deployed on a particular device. Thus, tokens need to be generated at runtime and under the request of DIAs. No system can be 100% secure, nevertheless, we can now ensure that tokens are more secure, and the level of security that can be reached with this approach depends on the implementation process only. The level of security, as we will see in the proof of concept, can be very high. Finally, to close the argument, tokens being sent by DIAs need to be validated before the execution of any

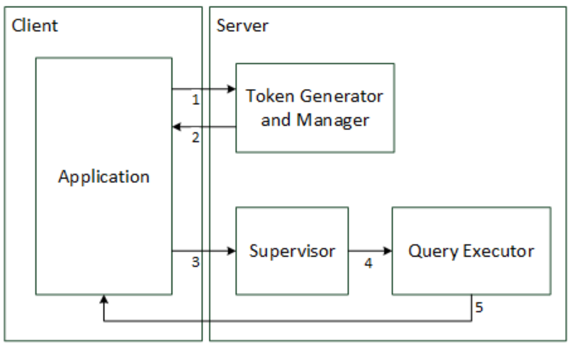


Figure 7: Conceptual model's block diagram.

CRUD expression. This task can also be done in the database server.

Fig. 7 shows a block diagram of the conceptual model, where the client application is a DIA. The general functionality is as follows:

1. The conceptual model starts with the application (an instance of a DIA) connecting to the Token Generator and Manager.
2. The Token Generator and Manager generates and provides the necessary set of unique tokens to the DIA.
3. The DIA uses the tokens to request the execution of CRUD expressions.
4. Each token pair is verified and validated by the Supervisor.
5. If the tokens are considered as valid, the correspondent CRUD expression is executed by the Query Executor and returns its output back to the DIA.

If an invalid token is detected by the Supervisor, that particular DIA should be prevented from sending further tokens to avoid brute force attacks.

4.2 Conceptual Model Implementation

In this section we propose basic lines for an implementation of the conceptual model. This is only one possibility among many others. With this example, we expect to provide a better understanding of some details that were not clear enough through the conceptual model.

Before presenting the general operation, we have to provide some additional information. As previously mentioned, tokens need to be randomly generated every time a DIA is instantiated. In practice, this means that the generated set of tokens needs to be aggregated in a higher level entity. This higher level entity uniquely identifies the running instance. The concept of Session can be used and we identify it as SessionID. Basically, every time a DIA establishes a valid connection, a SessionID is created. Then, the set of tokens is randomly generated for the authorized CRUD expressions for that SessionID. Each CRUD expression is identified by a CrudSRID (CRUD Session Remote ID).

This approach using two tokens decreases the probability of a malicious user executing a CRUD expression that he was not meant to execute. Let S be the number of active sessions, N_i the number of CrudSRID tokens associated to session i, B the number of random bits in the SessionID token and b the number of random bits in the CrudSRID token. The probability γ, at any given moment, of a user randomly guessing a valid pair of tokens is given by formula 1.

$$\gamma = \sum_{n=1}^{S} \frac{N_i}{2^{B+b}} \tag{1}$$

The universe of possible tokens that can be used in the system is given by 2^{B+b}, i.e. each bit has two possible values and there are a total of B+b bits in each token pair. The guessing probability γ is the number of valid token pairs, i.e. the sum of the number of CrudSRID tokens in each active session, divided by the universe of possible token pairs.

The formula shows that the guessing probability increases with the number of valid tokens (i.e. S and N_i), but it decreases with the increase in the number of random bits in the tokens. Hence, a security expert can accommodate the system to any number of active sessions and CRUD expressions by manipulating the number of bits used in each token. Additionally, if N is the total number of CRUD expressions in the system, then each session cannot have more than N associated CrudSRID tokens. This implies that the probability of randomly selecting a valid token pair is majored by formula 2, so it is possible to measure the maximum guessing probability Γ and lower it to be within acceptable limits.

$$\Gamma = \frac{S * N}{2^{B+b}} \tag{2}$$

Consider then a system where the number of active DIAs peaks at 1000 and the number of CRUD expressions configured in the system is 100. In this scenario we can say that the maximum number of valid token pairs will not exceed 100 000. Thus, following formula 2, we can calculate the chance of a malicious user guessing a valid token pair by chance given the number of random bits in the tokens.

Fig. 8 shows the number of attempts required to find one of the 100000 valid token pairs given the total number of random bits in the tokens. Note that the Y axis uses a logarithmic scale, so we can see that

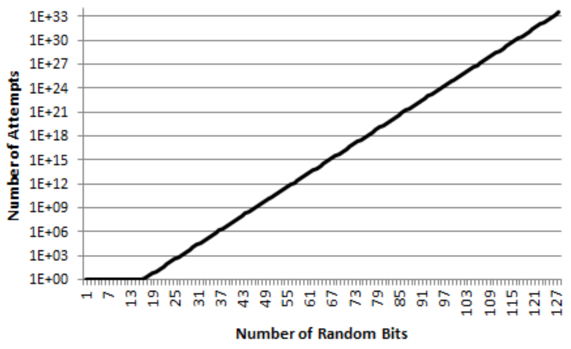


Figure 8: Number of attempts required to guarantee to find one of the 100.000 valid token pairs given the total number of random bits in the tokens.

the number of attempts required to guarantee that a malicious user is able to find a valid token pair grows exponentially with the number of random bits used on the tokens. This shows that a security expert can decrease significantly the risk of a malicious user executing a CRUD expression that he was not meant to execute, with small increases in the token's number of random bits.

Other techniques can be used in tandem with the randomized tokens to decrease this risk, such as preventing a user from requesting the execution of a CRUD expression after invalid tokens are used, but there are ways to go around them, potentially only increasing the time between execution requests. We will now detail the proposed implementation of our solution.

A block diagram of our proposed implementation of the conceptual model is shown in Fig. 9. The general operation is as follows:

1. The DIA starts up and connects to the server-side application.
2. The server-side application assigns a SessionID to the DIA.
3. A random CrudSRID token is assigned to each CRUD expression the DIA is authorized to use. All CrudSRID are also assigned to the SessionID.
4. The SessionID and all the associated CrudSRIDs are delivered to the DIA.
5. The DIA uses the SessionID and a CrudSRID tokens to request the execution of CRUD expressions, passing also the parameters to the server-side, if any.
6. The server validates the tokens, executes the correspondent CRUD expression and outputs its results.
7. If any token is invalid, the DIA will not be allowed to continue on using its tokens and: i) the SessionID and all CrudSRID tokens are revoked; ii) an error is sent to the application; iii) the application terminates.
8. The application requests the execution of more CRUD as needed.
9. When the DIA has no more CRUDs to execute, it terminates. The server, upon receiving the DIA's request to terminate, revokes its tokens.

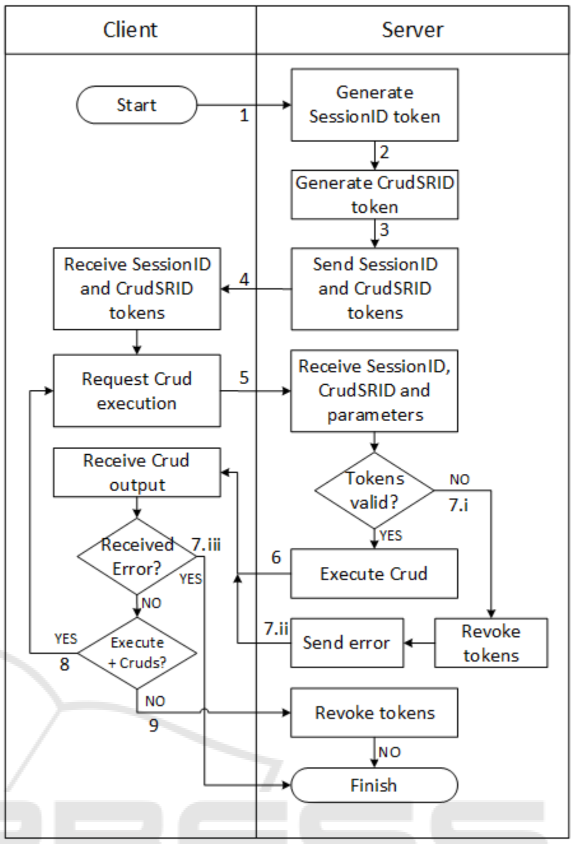


Figure 9: Proposed conceptual model implementation.

4.3 Other Security Concerns

While this solution is aimed to solve a very specific security problem, in practice its implementation may introduce other security concerns and, thus, should not be disregarded.

When the DIA connects to the server-side application, it will need to authenticate not only to safely associate a SessionID token to the session, but also to determine which CRUD expressions can be used by the DIA. Any authentication mechanism can be used, as our solution is not dependent on any in particular, but to provide a secure solution to this problem the SSL/TLS protocol[17] can be used to not only provide a secure communication channel on which the

DIA's credentials can be sent safely, but also to prevent attacks such as a man-in-the-middle.

Another security concern that should be considered when implementing this solution are replay attacks. In a replay attack scenario, a malicious user obtains valid token pairs by capturing the communication between a legitimate DIA and the server-side application and then using the same token pairs to request the execution of the associated CRUD expressions. Other attack that could use the SessionID token is a denial of service attack, where a malicious user could purposely use another client's SessionID with invalid CrudSRID tokens to revoke its tokens. These problems can be avoided by, once again, using the SSL/TLS protocol between the DIA and the server-side application, since it encrypts the communication, preventing the tokens' disclosure at the network level.

5 PROOF OF CONCEPT

In this proof of concept, to identify the outcome of the CRUD expression associated to each CrudSRID token that a DIA receives, we opted to send a generic name with the CrudSRID token, e.g. 'GET_ALL_CLIENTS' would mean that the associated CrudSRID refers to a query that selects all clients. This way, programmers know which information they are requesting the database for. There are, however, other methods for doing so. A more sophisticated method that can handle this transparently for the programmer could be an extension of the work presented in (Pereira et al., 2014).

Note that we also raise error messages when the execution is aborted for some reason, e.g. because invalid tokens were used. However, we now have complete control over the message generated, so we make sure that it does not disclose any information about the underlying database schema when returned to the user. With this technique, we effectively push the queries from the DIA to the server-side, i.e. the database server, protecting the database schema from being disclosed unintentionally.

We will now present our proof of concept. It is composed of the tables we had to create in the database to contain the tokens and the CRUD expressions, as well as the method used by the DIA to execute them, a stored procedure we called RemoteCall. The RemoteCall stored procedure can be seen as a mix of the supervisor and the query executor modules shown in Fig. 7, since it first validates the tokens it receives and then executes the relevant CRUD expression. The token generator and manager is a simple server-side application that creates and revokes tokens as needed when sessions are created and terminated, so we will not detail it.

5.1 Database Tables

For our implementation to support the tokens and store the CRUD expressions required by the DIAs, we required two main tables: the Queries table and the SessionQueries table. We also appended the SessionID parameter to an existing table that already stored the connected DIAs, so that it can be validated. The Queries table stores the actual CRUD expressions to be used. The SessionQueries table maps a generated SessionID and the generated CrudSRID's to the fixed database identifier, i.e. the primary key of each CRUD expression which is an incrementing CrudRID value that remains constant for each CRUD.

We also wanted to support parameters for the CRUD expressions, so we added an Operands table, which stores metadata on the list of operands required by each CRUD expression, such as the parameter's names and their data types, information that is required by the RemoteCall stored procedure. The usage of parametrized CRUD expression is explained further in section 5.2 where we detail the RemoteCall stored procedure.

To demonstrate how these tables are used to achieve the desired functionality, consider the two CRUD expressions *A* as SELECT * FROM Customers and *B* as SELECT * FROM Orders WHERE CustomerId = @CustomerId and ShipCountry = @ShipCountry that exist in the Queries table and the operands in the Operands table, shown on Table. 1 and Table. 2, respectively.

Table 1: Example of two CRUD expressions in the Queries table.

CrudID	CRUD Reference	CRUD Expression
1	S_Customers_all	A
2	S_Orders_byShipCountry	B

Table. 1 shows an example of the Queries table with two CRUD expressions. The 'CrudRID' column is the primary key, which is independent from the sessions and is always the same for each CRUD expression. The 'CRUD Reference' column is a generic name related to the operation associated with the CRUD expression and the 'CRUD Expression' column is the associated SQL statement to be executed.

The CRUD expression has its parameters defined in the format '@<parameter_name>', which facilitates its execution by the RemoteCall stored procedure and will be explained in section 5.2.

Table. 2 shows an example of the Operands table. The 'CrudRID' column references a query in

Table 2: Example of information in the Operands table for the queries in Table. 1.

CrudID	Position	Name	Type
2	1	@CustomerId	nchar(5)
2	2	@ShipCountry	nvarchar(15)

the Queries table and the 'Position' column denotes the parameter position in the CRUD Expression that the operand refers to and is used for performance optimization purposes. The 'Name' column indicates the name of the parameter used in the CRUD expression and the 'Type' column indicates the operand type, which follows the database management system's supported data types.

When a DIA initiates a new session with permission to use both CRUD expressions shown in Table. 1, it is assigned a random SessionID (12345678 in the example) and the following information is appended to the SessionQueries table:

Table 3: Example of the generated identifiers for a DIA session in the SessionQueries table.

SessionID	CrudSRID	CrudRID
12345678	13572468	1
12345678	24681357	2

The information in Table. 3 shows us that the session with the SessionID 12345678 can execute two queries (i.e. the CRUD expressions with the CrudRID 1 and 2). Additionally, the DIA using that session has to use the CrudSRIDs 13572468 and 24681357 together with its SessionID to execute each respective query successfully.

5.2 RemoteCall Stored Procedure

Having the database tables created, we now require a way to execute the queries using the SessionID and the CrudSRID tokens. To achieve this we chose to use a stored procedure, which we called RemoteCall. Stored procedures are able to take parameters as inputs and output the result of queries back to the DIA, which is precisely the functionality we require. Since we have the queries stored in a table, we can easily obtain the CRUD expression too, but the question is how to get the DBMS to execute it. The process varies from DBMS to DBMS and it must be done carefully. Since the CRUD expression is a simple string, appending the parameters received and executing it without any parameter validation is dangerous because of attacks such as SQL Injection.

The DBMS used, i.e. SQL Server 2010, offered two options to execute queries defined in strings: the EXEC and the sp_executesql commands(IETF, 2008). The EXEC command allows executing CRUD expressions stored in strings, but does not support parametrization, which leaves it vulnerable to SQL Injection attacks if used with parametrized queries. On the other hand, the sp_executesql command takes a CRUD expression and a list of parameters. Not only it is resilient against SQL Injection attacks by treating the parameters differently from the CRUD expression, even if they are strings themselves, but it also enables the DBMS to perform optimizations.

The sp_executesql command always receives a statement, which is the CRUD expression to execute. It also received two optional arguments, of which the first contains some metadata about the parameters of the statement to execute, i.e. their names and data types, and the other contains the parameters' values. A query that selects data from a table where a column value is less than 10 would be executed as follows:

EXEC sp_executesql N'SELECT * FROM table WHERE col > @Param', N'@Param int', N'@Param = 10';

In the example above, the query retrieved from the Queries table remains untouched, but the other parameters still need to be created by the RemoteCall stored procedure prior to the execution of the query per se. The result of the RemoteCall stored procedure is exactly the same as if the CRUD expression had been executed, therefore no changes are required in DIA's source code that handles it. Hence, the RemoteCall stored procedure requires three arguments: the SessionID, the CrudSRID, and a string called Params, which by default is empty and contains the values of the parameters needed to execute the CRUD expression.

Fig. 10 shows a block diagram of the RemoteCall stored procedure implementation. The RemoteCall stored procedure, upon being executed, retrieves the CrudRID associated to the SessionID and the CrudSRID from the SessionQueries table. This is part of the validation process of the tokens. If the SessionID or CrudSRID are not valid, i.e. if there is no CrudRID associated with the token pair received, the execution ends at this point, an error raised and the session terminated. Then it retrieves the CRUD expression to execute from the Queries table, using the CrudRID obtained initially. Next, the parameter definition string is built from the information stored in the Operands table. Finally, it executes the sp_executesql command with the CRUD expression retrieved, passing the parameter definition string and the parameter's values received from the DIA. Instead of the usual CRUD expressions, for the DIA to pass the tokens and the parameter values it executes the RemoteCall stored procedure as follows:

EXEC PolicyServer2._remote.RemoteCall @Ses-

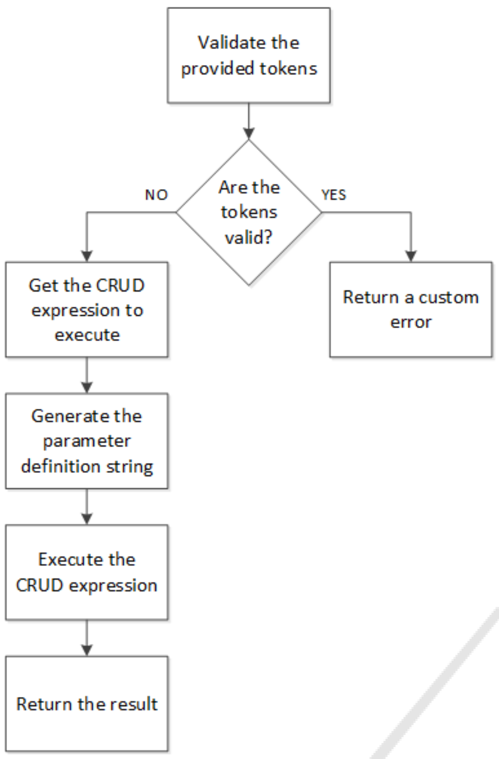

Validate the provided tokens
Are the tokens valid?
NO
YES
Get the CRUD expression to execute
Return a custom error
Generate the parameter definition string
Execute the CRUD expression
Return the result

Figure 10: RemoteCall stored procedure block diagram.

sionID = ?, @CrudSRID = ?, [@Params = '@Param1 = <value>, @Param2 = <value>'];

The first parameter would be the SessionID of the DIA and the second parameter the CrudSRID of the query to execute. The @Params is only needed if additional parameters are required by the CRUD expression. It can be seen that these SQL expressions do not disclose any information about the CRUD expression that will be executed, nor does it directly disclose any information about the database schema, assuming that the parameters and the columns' names obtained from executing them are aliases and they do not use the same name as defined in the database schema.

After the RemoteCall stored procedure was created, we only had to ensure that it was not possible for the DIAs to use any other method of executing CRUD expressions. Hence, we revoked all the user's permissions on the database except the ability to execute the RemoteCall stored procedure. This, however, limited its ability to execute the queries defined in the database due to the caller's lack of permissions. The solution found was to use the "execute as owner" modifier on the RemoteCall, effectively granting it the permissions it required regardless of which user called it. This also addresses the schema disclosure problem created by the SQL Server's JDBC driver metadata, which returns empty result sets when invoking the methods.

5.3 Performance Assessment

In order to evaluate the overhead induced by our solution, a performance assessment was carried out. Basically, we compared the response time between the traditional solution and the solution proposed in this paper. Since we want to assess the overhead that our solution imposes in a system, we aimed to maximize its contribution in the results obtained. To achieve it we used an environment where the execution time of a Select statement is as low as possible: we created a table with a single attribute of type Integer and only one row was inserted into the table. Then, a Select statement was written to select the inserted row. The two solutions were implemented and tested in a PC with Windows 7 Enterprise. All unnecessary processes and networking were shut down.

We tested both solutions by performing 100.000 requests to the database executing the select statement directly and using the RemoteCall stored procedure. The tool used to perform the tests was Apache JMeter, which provided the results are shown in Table. 4. The results include the average time per request, the 90th, 95th and 99th percentile and includes the throughput achieved with each solution, which is the number of requests served per second.

Table 4: Performance test results over 100.000 requests.

	Average	90%	95%	99%	Throughput
Select	2	1	1	48	392,6 req/s
RemoteCall	2	1	1	48	351,9 req/s

From these results we can see that the overhead introduced by the proposed solution slightly reduces the throughput by about 41 requests per second, from 392 to 351, in this particular test suit. It is worth pointing out that this test suit was aimed to maximize the contribution of the overhead by using a very simple query with a minimal result set. In a more realistic scenario, the database computation time will be greater with more complex queries and bigger data sets, making the overhead contribution of the database processing delay more significant and close the gap between the response time and throughput on both solutions.

Note that this overhead is constant. Having one CRUD expression configured or thousands of them does not impact the RemoteCall stored procedure execution plan in any way. The only aspect that can impact these results is if the CRUD expression, when it is being fetched using the tokens, does not reside in memory. This would require a disk access and a much larger delay time, but it can be easily solved by adapting the conceptual implementation. For example, the supervisor application that resides close to the database can store the CRUD expressions in memory and implement the RemoteCall stored procedure itself. The only downside to this approach is that the client would not be able to use JDBC, relying instead on a library that could provide a similar interface.

ACKNOWLEDGEMENTS

This work is funded by National Funds through FCT - Fundação para a Ciência e a Tecnologia under the project UID/EEA/50008/2013.

6 CONCLUSION AND FUTURE WORK

In this paper we have proposed a technique to prevent database schemas from being disclosed in the DIAs through the CRUD expressions being used. With this technique, the CRUD expressions are stored in the server-side instead of on the DIAs and are executed using a set of tokens given to the DIAs. Since these tokens are randomly generated, the DIAs are not able to disclose any information about the database schema used, and the potential attacks against this technique were discussed.

This technique requires every CRUD expression to be stored on the server-side, which can be inconvenient while developing a DIA and might require a security expert to manually insert them. The existence of an application that is capable of registering these CRUD expressions in the server-side, and a library that allows the programmer to retrieve the token of a query using its generic name, would facilitate the usage of this technique. Furthermore, the RemoteCall stored procedure can be simplified by removing the definition string generation entirely, having it stored directly in the Operands table instead of each parameter individually. However, this technique is more prone to errors due to the strict syntax required, which a CRUD registering application could address.

REFERENCES

Anley, C. (2002). Advanced SQL injection in SQL server applications. *White paper, Next Generation Security Software . . . .*

Bagui, S. (2003). Achievements and weaknesses of object-oriented databases. *Journal of Object Technology*, 2(4):29–41.

Bauer, C. and King, G. (2005). *Hibernate in action*. Manning Publications.

Chaudhuri, S., Dutta, T., and Sudarshan, S. (2007). Fine grained authorization through predicated grants. In *Proceedings - International Conference on Data Engineering*, pages 1174–1183, Istanbul.

Eclipselink, U. (2013). *Understanding EclipseLink 2.4*. Eclipse.

Erhieyovwe, E., Oghenekaro, P., and Oluwole, N. (2013). An Object Relational Mapping Technique for Java Framework. *International Journal of Engineering Science Invention*, 2(6):1–9.

Garcia-Molina, H. (2008). Stored Procedures. In *Database systems: the complete book*, chapter 9.4, pages 391–404. Pearson, 2nd e. edition.

Halfond, W., Viegas, J., and Orso, A. (2006). A classification of SQL-injection attacks and countermeasures. *Proceedings of the IEEE . . . .*

IETF (2008). RFC 5246: The Transport Layer Security (TLS) Protocol - Version 1.2.

Pereira, Ó. M., Aguiar, R. L., and Santos, M. Y. (2011). CRUD-DOM: a model for bridging the gap between the object-oriented and the relational paradigms: an enhanced performance assessment based on a case study. *International Journal On Advances in Software*, 4(1):158–180.

Pereira, Ó. M., Regateiro, D. D., and Aguiar, R. L. (2014). Role-Based Access Control Mechanisms. . . . *(ISCC), 2014 IEEE . . . .*

Pereira, Ó. Ó. M., Aguiar, R. R. L., and Santos, M. Y. M. (2012). ACADA: access control-driven architecture with dynamic adaptation. *SEKE'12 - 24th Intl. Conf. on Software Engineering and Knowledge Engineering*, pages 387–393.

Rohilla, S. and Mittal, P. K. (2013). Database Security by Preventing SQL Injection Attacks in Stored Procedures. *Software Engineering Conference, 2006. Australian*, 3(11):915–919.

Roichman, A. and Gudes, E. (2007). Fine-grained access control to web databases. *Proceedings of the 12th ACM symposium on Access control models and technologies - SACMAT '07*, page 31.

Russell, C. (2008). Bridging the Object-Relational Divide. *Queue*, 6(June):18.

Sumathi, S. and Esakkirajan, S. (2007). *Fundamentals of relational database management systems*. Springer.

Wilson, J. (1988). Views as the security objects in a multilevel secure relational database management system. *Proceedings. 1988 IEEE Symposium on Security and Privacy*.