# Fine-grained Access Control to Web Databases

Alex Roichman
Department of Computer Science,
The Open University, Raanana, Israel
Alexaro1@012.net.il

Ehud Gudes
Department of Computer Science,
The Open University, Raanana, Israel, and
Department of Computer Science,
Ben-Gurion University, Beer-Sheva, Israel
Ehud@cs.bgu.ac.il

## ABSTRACT

Before the Web era, databases were well-protected by using the standard access control techniques such as Views and SQL authorization commands. But with the development of web systems, the number of attacks on databases increased and it has become clear that their access control mechanism is inadequate for web-based systems. In particular, the SQL Injection and other vulnerabilities have received considerable attention in recent years, and satisfactory solutions to these kinds of attacks are still lacking.

We present a new method for protecting web databases that is based on fine-grained access control mechanism. This method uses the databases' built-in access control mechanisms enhanced with *Parameterized Views* and adapts them to work with web applications. The proposed access control mechanism is applicable for any existing databases and is capable to prevent many kinds of attacks, thus significantly decreases the web databases' attack surface.

## Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access (e.g., hacking, phreaking)*;
H2.7 [**Database Administration**]: Security, integrity, and protection.

## General Terms

Security

## Keywords

Access control, web database security, database vulnerability, parameterized view, session key, rolling key

## 1. INTRODUCTION

Information is the most valuable asset for organizations. In our days information is stored in databases that become accessible from the Internet. The information disclosure from such databases may have very serious impact on organization business. So new access control approaches for databases and especially for web databases have become a dire necessity.

New web applications replace the old systems. This is an unstoppable process that started at the mid 1990's and has accelerated in recent years. Databases were well-protected from attacks from old applications. But web applications place databases in a new situation exposing them to many illegal accesses and attacks.

Old applications were run from the users' desktop. These users were fixed and known in advance and their number was limited. The applications usually opened a connection to the database and all the transactions were passed via the same connection. This two-tier architecture resulted in a model where the user is working with an application layer that interfaces directly with the database layer. Consequently, the database would directly identify the user working with it, the computer from which she connects to, and the transactions that she runs. Thus it is quite easy to supply the user with the proper authorizations and follow up single user transactions to seek signs of intrusion, as all the transactions of the same user are passed via the same connection.

On the other hand, web applications differ significantly from old applications in their mode of working with databases. These applications are run from the users' browser windows. The users of web applications can be casual users and their number is not limited. The browser does not directly connect to the database, but instead transfer a request to a web server. The server processes the request and if required it performs a transaction to the database. The result of this three (or more) -tier architecture, is that the database does not identify the user who accesses it. From the database point of view, the user accessing the database is the super-user of the web application server!

Not only that the database does not identify the user who accesses it, but it is also impossible to follow transactions of the same user. This is derived from using the technique of *connection pooling*. In this technique the web application does not open a connection for each request and does not close it after performing a request, but instead, uses a connection pool mechanism where connections to the database are stored. Prior to performing a request, the application pulls a vacant connection from the pool, runs a request on it and immediately returns it to the pool. This way the time to open and close the connection is saved per each request. A large number of users can also be satisfied with a small number of connections, so that the application works more efficiently for each database request.

But the situation described above has serious implications. First, no user-based access control can be applied since the only recognized user is the super-user of the web application server. Databases cannot differentiate anymore between transactions of different application users. The principle of minimal privilege is violated! It is impossible to authorize the web application user with appropriate privileges: all application users have an access to the same data. This situation results with horizontal (e.g. a student has access to the data of another student) and vertical (e.g. a student has access to a professor's data) privileges escalation. No more fine-grained access control to the database exists, and the only mean to prevent attack to web databases is in the application level. Although many advances have been made in developing secure applications, trusting applications which are developed under time constraints by developers which are not security experts, presents a large risk to the database and therefore databases are threatened by these applications.

In this paper we first demonstrate that preventing illegal accesses to the database by means of application layer is not effective, as it try to shield the vulnerable database by some middleware ad hoc methods. Instead, we suggest dealing with the root of the problem and searching for new generic methods for protection databases by means of proper access control mechanisms that will be implemented by the databases themselves. We'll suggest the P*arameter Method* as being capable of preventing attacks on databases by the existing native database protection mechanisms.

The parameter method allows transferring the identity of the user working with the database to the database and not only to the application, thus solving the first major problem of fine-grained authorization at the data level. The parameter method will also allow distinguishing between the requests of different users thus solving the second problem of user-session's traceability for web systems. Therefore, developers of intrusion detection system will be able to analyze the transactions of the intruder when they discover penetrations to databases, as these transactions are distinguished from transactions of legal users.

We'll present two design solutions for the parameter method. The first solution is based on the concept of application session key, and the second solution enhances the first one by the concept of a rolling key. Both methods will use the concept of parameterized views. As our experiments show, the web site based on parameterized views is much secure and this is accomplished without a significant performance overhead.

The importance of our solution is that, on the one hand, it proposes the novel fine-grained access control mechanism to the web database, and, on the other hand, it enables real-life development of secure Internet databases by using commercially available tools.

The next section discusses the background of web databases access control, the concept of parameterized views and gives examples for attacks such as SQL injections. Section 3 presents the Related work. Section 4 presents the main ideas of our approach. Section 5 discusses in detail the parameter method, and then describes the way the parameter method and the parameterized views are integrated to provide the required protection. The security and performance of our approach are evaluated in Section 6, and Section 7 discusses some implementation issues of parameterized views. Section 8 concludes and describes some future work.

## 2. BACKGROUND AND TERMINOLOGY

### 2.1 Parameterized Views

Views are the basis for protection and access control in relational databases, as they enable to determine for a user the only part of the database that interests her. But because of the existence of the most powerful user, the web server super-user that is connected to the database on behalf of all actual users, traditional views cannot be used as means of access control. We checked many existing open source web applications and found that in most of these systems there is no use of views as means of access control.

Our goal is to revive the use of views as access control mechanisms in the context of web systems. We'll show that the replacement of a traditional view in the context of web applications is the parameterized view. The parameterized view will transfer the user's identity to the database and the view will display the relevant data to this user accordingly, thus providing fine-grained access control to web databases.

The subject of parameterized views was raised in academic discussions during the mid 1990's mainly in the context of Object Oriented databases. Eder in [4] presents the problem of relational databases in which the views are dependent on the table's name and not on the type of the table. Actually, the predicate (where clause) of each view is fixed and there is no way to modify it dynamically.

Jamil in [9] displays the syntax and semantics of parameterized views and show how parameters affect their predicate. For instance, a definition sentence (DDL) of a view that determines the grades of a specific student can appear like this:

```
CREATE VIEW Student_Marks_View
WITH   pStudent_No
SELECT *
FROM   Student_Marks_Table
WHERE  Student_No = :pStudent_No
```

**Figure 1: definition of parameterized view**

The content of this view depends on the value of the parameter `pStudent_No`. In Section 7 we'll discuss in detail the issue of implementing parameterized views, but for the rest of the paper we assume their existence.

### 2.2 SQL Injection and Other Attacks

As explained above, access control to web databases is currently implemented by applications and not by the database. Since the applications accessing the database can be very heterogeneous and their access control can be very inconsistent, this can results in a situation in which the database is exposed to attacks from web systems. The description of such attacks appears in many articles and the attack that received special attention of researchers is the SQL injection attack [18]. We will present a small example of such an attack: let's assume that the application displays a salary of an employee whose number is ***123*** for the period chosen by the employer:

```
strSQL= "SELECT  Salary
         FROM    Salary_Table
         WHERE   Employee_No = 123
         AND     Salary_Date = '" +
                      dateParam + "'"
```

**Figure 2: example of SQL with user input**

It's possible to see that the SQL sentence is structured as a string and the parameter is concatenated to this string. In a proper situation, the user keys in some date which is stored in the variable `dateParam` and concatenated to SQL. But if the user will type ***01.2007' or '1' = '1*** then we'll get:

```
SELECT Salary
FROM Salary_Table WHERE Employee_No = 123
AND  Salary_Date = '01/2007' OR '1' = '1'
```

**Figure 3: example of SQL injection**

As the condition ***'1' = '1'*** always holds, the application will return the salaries of all the workers for all the periods. Such attack is possible, as the application works with a strong DB user who has a retrieval authorization from the entire `Salary_Table`.

Another type of an attack is the Parameters Tampering attack. Like the SQL injection attack, this attack takes advantage of the fact that many programmers rely on parameters that come from the user. Attackers can easily modify these parameters to bypass the security mechanisms and attack the back-end database. But this attack is more difficult to detect than SQL injection: in SQL injection the structure of original SQL sentence is changed, but in parameter tampering the structure remains the same, only the parameter range is changed.

For example, in the Book-store application [22] the customer can view her ordering. For each order she can request its details. When this happens, the application sends a request to the web server with the parameter of customer's `Order_No`. The SQL sentence submitted by the application is as following:

```
strSQL = "SELECT * FROM Orders_Table
          WHERE Order No = " + orderParam
```

**Figure 4: SQL vulnerable to parameter tampering**

But if the intruder wants to view the details of another customer, she can change the value of `orderParam` to the value of a different order that does not belong to her (bypassing the application protection). In such a situation, she will be capable of retrieving the data of a different customer.
As we can see, all the attacks on web databases are caused by the inadequate access control mechanism of databases. If we will be able to authorize each application user to the part in the database relevant only to her, we can minimize the effect of these attacks as the attacker will be restricted to attack only her data and not the data of different users! What we need then is fine-grained access control to web databases.

## 3. RELATED WORKS

The conventional methods for providing protection in databases rely heavily on the identity of the entity accessing the database (User, Program-Id, etc) [2, 5]. Using such identity, the authorization provided by views and roles and the Grant/Revoke mechanism can be applied [6, 14]. Furthermore, in case of a failure in authentication, after the fact intrusion detection may still be used, since all transactions on behalf of a single user are properly identified and can be analyzed using the log. But as explained above, this situation does not exist anymore in the web environment, and databases are exposed to different attacks that are very hard to prevent and detect.

Several suggestions were published in the literature to prevent SQL injection attacks such as: precise checking of parameters that come from the user, prohibition of running SQL sentences directly from the application, but instead, running database stored procedures, use of prepared SQL statements etc [7, 11, 19].

Experiments trying to create signatures for known attacks such as SQL injections were also carried out. Mookhey and Burghate in [13] present a way to structure regular expressions for known instances of SQL injections, but the research displayed by Maor and Shulman in [12] shows that such signatures are not effective. For example, instead of injecting ***1 = 1*** the intruder may inject ***1 < 2*** or any other predicate that always holds. Some suggestions are very sophisticated methods, such as the use of SQLRand technique proposed by Boyd and Keromytis in [3]. The last technique applies the concept of instruction randomization to each SQL sentence. In SQL injection attack, the intruder injects some reserved SQL word, but with randomization this word is unpredictable by the intruder.

The recent tendency in the architecture of web applications is to build distributed multi-layered platforms (.NET, Java EE) where there exists some tier that offers database access control service which is supposed to shield the database from attacks. Oracle offers n-tier authentication to the database when the application can have multiple user sessions within a single database server session [21]. This mechanism is called "lightweight sessions" and it can preserve the identity of the real user through the middle tier. But in order to support the fine-grained authorization, Oracle must maintain all the application users and attach security policies to each one. Since the number of web users may be tremendous and they are created/dropped dynamically by the application (and not by DBA), this solution is not exactly practical for a typical web application.

The main disadvantage of the above proposed methods is clear: these methods are not native for databases and cannot be implemented by the database built-in mechanisms. We call these methods the *Virtual Patching* methods (this terminology is proposed by Chris Klaus, see [20]) as they try to shield databases from attacks instead of making databases resistant to them.
Another example for application based protection appeared in [19]. They use stored procedures or parameterized queries at the database, but calling them with parameters from the application. This can prevent SQL injections, but if there exists some place in the application that can access the database without using this technique, then the database will be threatened. Furthermore, stored procedures or parameterized queries can only prevent one specific kind of attack: the SQL injection. But we need new approaches, such that the database will be properly protected no matter how the application is written and what kind of an attack the intruder will run. We'll show that our method will make the database resistant to general kind of attacks instead of trying to shield vulnerable databases by outside ad hoc methods as the applicative approaches do.

A companion approach is intrusion detection. Much effort was invested in developing methods for detecting intrusions to databases in recent years. The main idea is based on analyzing transactions that arrive to the database with the purpose of searching for signs of intrusions. Valeur, Mutz and Vigna in [17] suggest creating fingerprints for each sentence that the application can run. Low, Lee and Teoh in [10] and Srivastava and Reddy in [15] suggest not only to structure fingerprints for single SQL sentences, but also to refer to the order of sentences within the transaction. One of the difficulties of this method is that the web applications have a tendency to use very short transactions and even use the implicit transaction; namely, each SQL sentence constitutes of a separate transaction. So there is no point in searching for order of sentences in a transaction that, in general, contains only one sentence.

Hu and Panda in [8] suggest a different strategy: it's possible to look for dependencies between the different items in SQL sentences. And these dependencies can be found by data mining algorithms. But the main disadvantage of all the above methods for detecting intrusions to databases is that they are not suitable for web systems. These methods presume that it's possible to analyze the log of databases when this log contains SQL sentences and indication to which user's session each such sentence belongs to. But this is impossible for web systems as was explained above. For example, look for the following sentences:

```
SELECT C1 FROM T2
UPDATE T3 SET C4 = 5
```

**Figure 5: SQL statements and their dependencies**

If we apply the approach proposed in [8], we can find dependency between C1 and C4. But for the web application these sentences can be submitted by different users so they can be completely independent!

In this paper we present the parameter method as a method that transfers the identity of the user to the database. Such indication will be reliable and difficult to fake, so the database can rely on it. We'll use the built-in access control mechanisms of databases that much resembles the use of the classical protection mechanism of views. All the existing methods of detecting intrusions to databases will also become relevant for web systems with the new parameter method.

## 4. OUR APPROACH

Our concept is based on the use of parameterized views as the means to supervise the accesses to the database from web systems. In the parameterized view, the parameter will contain the identity of the user. The main requirement is that the parameter will be difficult to fake. For example in a University system, the natural parameter is the student identifier. Then, the sentence that the application can run for a student whose id is ***1*** is:

```
SELECT *
FROM   Student_Marks_View(1)
WHERE  Course_No = 12345
```

**Figure 6: example of parameterized view**

But this method is not safe, because it does not prevent the SQL injection attack. It's possible, for example, to insert a whole sentence instead of the course number: ***12345 UNION Select * From Student_Marks_View(2)***. This way the original sentence will become:

```
SELECT *
FROM   Student_Marks_View(1)
WHERE  Course_No = 12345
UNION
SELECT *
FROM   Student_Marks_View(2)
```

**Figure 7: unsafe parameterized view with SQL injection**

So although it may appear that the parameter limits the access, SQL injection allows to retrieve every row from the table; namely, access indirectly the entire list of grades.

The solution to this problem as we shall see below is not to use the explicit user identity in the SQL statement but instead use a run-time generated identifier which will be very difficult to fake. We will present two such solutions to the parameter method: the first solution is based on a key of the application session and the second is based on the technique of a rolling code.

## 5. THE PARAMETER METHOD

The parameter method is supposed to distinguish between the accesses of different users. There are two types of web applications: applications that do not demand identification and applications which demand user's identification. For the applications that demand identification, the parameter method transfers this identification to the database and the access control of the database is based on this identification. The aim of most attackers is to attack the data of other users. So this method enables prevention of these attacks.

In case that the application does not request the user's identification, the goal of our method is to prevent unauthorized accesses and to distinguish between the sessions of the different users. But this will be done without violating the privacy of the anonymous user. In the next two paragraphs we only refer to applications that demand a process of identification. We will detail our method for applications that do not demand user's identification in Section 5.3.

### 5.1 The Application Session Key Based Parameter Method

This type of solution is similar to the challenge-response protocol. It uses a random number which is generated at the beginning of the protocol for each application session, and is sent thereafter with each SQL statement. It also assumes that the database stores in an internal table the Ids of the active users. The protocol works in the following way:

- A user requests to perform a process of identification when she provides a username and a password to the application.
- The application runs a database stored procedure that accepts the user's username and password and returns a random number (AS_key).
- The database stores the random number in a table of active users, for example, as follows:

**Table 1: active users for AS key**

| Student_No | User | Pass | AS_key |
|---|---|---|---|
| 1 | Jona | $#Hj#45 | 01100… |
| 2 | Mikes | *&SD12qF | 10010… |

- The application knows the user who works with it, and stores the AS_key as well. Each SQL sentence that will be run on behalf of the user will be run with a parameter of user's corresponding AS_key.
- The AS_key is cleared when a user disconnects from the application.

This way, when the SQL sentence arrives, the view returns only the data which belongs to the user with the given identifier. For example, the following view definition can be used:

```
CREATE VIEW Student_Marks_View
WITH   pAS_key
SELECT * FROM Student_Marks_Table
WHERE  Student_No IN
  (SELECT  Student_No
   FROM    Users_Table
   WHERE   Users_Table.AS_key=:pAS_key)
```

**Figure 8: parameterized view for AS key**

This approach significantly decreases the range of possible attacks on databases. The intruder still can execute her attacks, but with parameterized views she cannot affect the data of different users. Let us return to figure 3 that represented the SQL injection into select statement from Salary table. With parameterized view, the select will look as follows:

```
SELECT Salary
FROM   Salary_View(11011…)
WHERE  Salary_Date = '01/2007'
OR     '1' = '1'
```

**Figure 9: parameterized view resistant to SQL injection**

Because `Salary_View` returns only the data of the specific employee with AS_key ***11011..*** and this must be the key of an attacker (if not, the attacker must guess the parameter and probability of this is very small), this attack affects only the attacker's data. Namely, the attacker may access information about her salary from different months, but not the salary of different employees.

## 5.2 The Rolling Key Based Parameter Method

This solution uses a rolling key or a rolling code. In order to understand the concept, we will examine the alarm systems of vehicles which use a rolling key. Old alarms used remote controls that sent a signal each time that the driver locked or opened the vehicle. So the burglars knew to record the signal sent from the remote control and later to open the car by playing the previous recording. More sophisticated alarms use a rolling key in a way that the signal is changed each time that the remote control is activated, so recording the signal has no meaning. Similar techniques are used in rolling Secure-Ids tokens provided for example by RSA. Our protocol works in the following way:

- The first three steps are similar to the previous parameter method based on the application session key, except that now AS_key also serves as the seed of the rolling key.
- The database and the application agree on a common encryption key (Enc_key), for example using the Diffie-Hellman protocol [16]. This encryption key is used to generate the next rolling key from the current one as will be explained below. (Note that in most cases, the web server and the database server are placed on separated network segment so the man-in-the-middle attack against the Diffie-Hellman is not possible. In other cases we can prevent such attack by encrypting the channel between the web server and the database server.)
- Now the active users table may look like in Table 2:

**Table 2: active users for rolling key**

| Stdnt No | User | Pass | Enc key | AS key | Roll key |
|---|---|---|---|---|---|
| 1 | Jona | $#45 | 001… | 011… | 011… |
| 2 | Mikes | *&qF | 111… | 001… | 001… |

- Now, when the SQL sentence arrives, it contains a request for a parameterized view with 2 parameters: the AS_key and a rolling key. As a result, two things occur:
  1. The view returns the filtered data that belongs only to the user that the session and rolling keys belong to.
  2. The rolling key is advanced to the next number both in the application and in the database.

  The database view can use a stored function to perform the above actions as illustrated in Figures 10 and 11:

```
CREATE FUNCTION Authorize(pAS_key,
                          pRoll_key) AS
Begin
   SELECT    Enc_key,Roll_key,Student_No
   INTO      :enc, :roll, :stdnt_no
   FROM      Users_Table
   WHERE     AS_key = :pAS_key
   If roll == pRoll_key Then
      UPDATE Users_Table SET Roll_key =
             Compute_Next_Key(:enc,:roll)
      WHERE  AS_key = :pAS_key
      COMMIT
      RETURN :stdnt_no
    Else
       RETURN NULL
    End If
End
```

**Figure 10: Authorize function definition**

```
CREATE VIEW Student_Marks_View
WITH   pAS_key, pRoll_key
SELECT * FROM Student_Marks_Table
WHERE  STUDENT_NO =
         Authorize(:pAS_key,:pRoll_key)
```

**Figure 11: definition of Student_Marks view**

The auxiliary function `Compute_Next_Key` receives the encryption key and the last rolling key and computes the next rolling key. We suggest one of symmetric encryption protocols as a pseudo random number generator: The next code can be calculated according to the following formula: $Roll\_key_{m+1} = E_{Enc\_key}(Roll\_key_m)$, where $E_{Enc\_key}$

may be the DES algorithm and the Enc_key is the key which was determined by using the Diffie-Helman protocol.

- Since the database knows the user who works with it, it can store the Enc_key, the AS_key and the Roll_key for each active user. For each next SQL sentence it will generate the next rolling key as the database does. The three numbers are cleared when the user disconnects from the application.

However, one problem with using the rolling code is known as the problem of synchronization. The problem of synchronization can arise when the application will advance the code and send the SQL statement, but the database will not accept it (for example, because of a disconnection in communication between a web server and a database server). In such case the application already advanced the code, but the database remained with the old code, as it did not receive the sentence.

It is similar to using of a rolling code in a car alarm. Both the car's remote control and the alarm system need to be synchronized. The solution is for the database (the car) to check ***n*** forward rolling keys and not only the last one. So we need to adapt the function that checks if the next rolling code is legal and check ***n*** next codes instead of one.

The advantages of the AS key method compared to the method of rolling key are clear: The calculation of the next key in the rolling key method is expensive regarding the system's resources, especially when the quantity of sentences sent is large. This method does not use such calculation, so it is more efficient. The performance of the Update following each calculation is also saved in this method. So the AS key method is preferable from the efficiency point of view to the parameter method which is based on a rolling key.

But the advantages of the rolling key method compared to AS key method are mainly that it's more secure! The method is resistant to replay attacks and a correct guess of a code is only valid for running one sentence. If the attacker tries to guess the code and comes to the conclusion that a particular code is not valid, she has no confidence that this code will not be valid in a few seconds, while the user under attack will run the next sentence. For the AS key, if the attacker rules out a particular code, then she knows that this code will not be valid throughout the entire session of the user, so the attacker can reduce the number of possibilities of valid codes according to the AS key method. Therefore, the rolling key method is much safer.

## 5.3 The Parameter Method for Anonymous Users

Although the application does not authenticate users, it is aware of each user session, so it can attribute each session with AS key or rolling key. The table of anonymous users will appear as follows:

**Table 3: anonymous users for AS key**

| User | AS_key |
|---|---|
| Anonymous1 | 01100… |
| Anonymous2 | 10010… |

In the case the intruder wants to bypass the application logic and submit SQL injection or other type of attack, she must know the random key. So our method can also prevent illegal accesses of anonymous users. For example, in the University system all the students can retrieve a course list for a chosen semester. The select statement can look like this:

```
SELECT *
FROM   Course_View(01100…)
WHERE  Semester = 'Fall 2006'
```

**Figure 12: course list retrieval**

Now if the intruder wants to change this SQL by the SQL injection attack, she may change her input of **`Fall 2006`** into some more complicated union form and submit the following statement:

```
SELECT *
FROM   Course_View(01100…)
WHERE  Semester = 'Fall 2006'
UNION
SELECT *
FROM   Student_Mark_View()--'
```

**Figure 13: course list retrieval resistant to SQL injection**

But her attack will be unsuccessful as she does not know the random parameter that must be supplied to the `Student_Mark_View`!

Another thing that can be useful is the ability to differentiate between sessions of different anonymous users. The idea is to use a separate session key for each application session and after that to partition the SQL log that can be created by the database. Now we can employ many existing intrusion detection methods when they are applied to the appropriate SQL log slice and thus we can look for intrusions on a session layer instead of a single SQL statement layer as it happens with traditional approaches.

One important thing for the applications that do not require the user authentication is privacy preserving. Since our method transfers a random key that tell the database nothing about the actual user, our method also preserve privacy of the anonymous user. In the next section we will describe how to define views for tables that do not contain identifier in one of its fields- these are the views required by us for the applications without identification process.

## 5.4 DB Schema with Parameterized Views

We presented the use of parameterized views for the control of the access to databases from web systems. Now we will show how to structure a database schema, which allows the use of parameterized views.

For each table in the database the view must be defined, the access to this view must be granted to the application and the access to the table must be revoked. Let's look at an example of a table with many different access roles to it: in the School system there is the `Students_Table` and a student can access her detailed record or names and e-mails of other students in her class. For such a case the following definition can be used:

```
CREATE VIEW   Students_View WITH pAS_key
SELECT Name, E_Mail, Other_fields...
FROM   Students_Table
WHERE  Student_No IN
   (
       SELECT Student_No FROM Users_Table
       WHERE  Users_Table.AS_key=:pAS_key
   )
UNION
SELECT Name, E_Mail, 'dummy'...
FROM   Students_Table
WHERE  Class_No IN
   (
       SELECT Class_No
       FROM   Classes_Table, Users_Table
       WHERE  Users_Table.AS_key=:pAS_key
       AND    Classes_Table.Student_No =
              Users_Table.Student_No
   )
AND Student_No NOT IN
   (
       SELECT Student_No FROM Users_Table
       WHERE  Users_Table.AS_key=:pAS_key
   )
```

**Figure 14: students parameterized view definition**

This definition is polymorphic in the sense that it allows a single definition to be used with different types of access roles. With this kind of polymorphic definitions, the number of views expected to be maintained is exactly the number of tables in the database schema.

The structuring of views needs to be based on the parameter which either provides the identity of the user, or the session identity. In general, a database schema (in Relational systems) may contain tables of two types:

- Tables which include the user identifier in one of the fields. In general, this identifier is included in the primary key of the table.
- Tables which do not contain a user identifier.

A parameterized view for the first type was presented in Sections 5.1 and 5.2. There are three possibilities for access control on tables of the second type:

- The access to the table is not relevant to the system, so no authorization should be provided to this table for the DB user of the application.
- The access to all the rows of a table is relevant to all users. An example of such table can be the table of courses offered during the semester, as all the users of the system are allowed to access this table. So a parameterized view needs to be structured. But the role of the parameter is not to filter rows, but to return a full table in case the parameter is valid; otherwise, it will not return anything. For example, the view defined in Figure 15 may be used.

```
CREATE VIEW   Courses_View
WITH   pAS_key, pRoll_key
SELECT *
FROM   Courses_Table
WHERE  Authorize(:pAS_key, :pRoll_key)
              IS NOT NULL
```

**Figure 15: course parameterized view definition**

The idea of this definition is to allow only the application's users to access the table. In other words, it is not enough to connect to the database with the DB user, in order to access the table, but you also need to be one of the legal application users, so that you can access data. Another reason for the use of such view is to enable intrusion detection. By having the database storing the AS_key, it can later identify the user session precisely and thus enable the process of post-mortem intrusion detection by analyzing the database log.

- Tables without identifiers can also be a subject to individual user authorizations, but in this case the parameterized view must join them to some identifier-based table. For example, the Course Stuff table can contain confidential information so the natural policy may be that the student can access only the stuff details of the courses she is enrolled in. In this case the parameterized view will be constructed on joining the Student Courses table with the Course Stuff table.

## 6. ANALYSIS AND EVALUATION

The advantages of the parameter method described above are clear. Since the parameter enables to identify the user, each SQL sentence that arrives to the database is attributed to a specific user. Consequently, the database provides access only to the data relevant to a particular user, while being assisted by the mechanism of parameterized views. Furthermore, the log of SQL sentences which arrive to the database includes the indication of the user who ran it. This is very important for developers of intrusion detection systems of databases, because now they can distinguish between the transactions of different users. Next we present analytic and experimental evaluation of the parameter method.

### 6.1 Analysis of Parameter Method

The safety of the parameter method rests with the difficulties of guessing the various keys. For the application session key method, there is a need to guess a session key of some currently active user. If we assume a key of 64 bits then the probability of guessing this number is $\frac{1}{2^{64}}$. If we assume that the maximum of active users connected to the system at the time of an attack is X, the probability for the successful attack will be $\frac{X}{2^{64}}$. Note that this is an unselective attack since the attacker cannot control the attacked user identity when she tries to guess some key.

For the rolling key method, there are two items to guess, the session key and the rolling key. If we assume a session key of only 8 bit, rolling key of 64 bit and 256 possible correct codes (because of dealing with the synchronization problem) then probability for the correct guess will be $\frac{X*2^8}{2^{64}*2^8}=\frac{X}{2^{64}}$.

Obviously, it's possible to use longer codes, if we want to increase the level of security; for example, 128 bits and then the probability for a successful attack will decrease respectively. But it can be a burden on the system:

1. As the key is longer, the computing of the next key need more processor's time.
2. As the key is longer, the length of the SQL sentence increases and it can be a burden on the communication between the web server and databases server. The databases will also need to cope with longer SQL sentences

Note that the method of a rolling parameter is resistant against replay attacks, as the repetition on the same sentence with the same parameter does not bring any result because of the rolling key. Even if the intruder succeeded to obtain the code from the parameter of another user, it does not help him at all, as without knowing the encryption key, he does not have the ability to compute the next valid code.

## 6.2 Performance Evaluation

The performance evaluation was split into two stages. In the first stage our target was to compare the performance of a single regular SQL statement with our two methods: AS key and rolling key methods. We used a stand alone computer with the SQL Server with approximately 1,000,000 tuples in relevant tables and created views like the view from Fig 8. Next we run a transaction that included Select/Update statement directly on the table and on the view with AS key and rolling key. To average the processing time we repeated this for 100, 1000 and 10000 serial transactions:

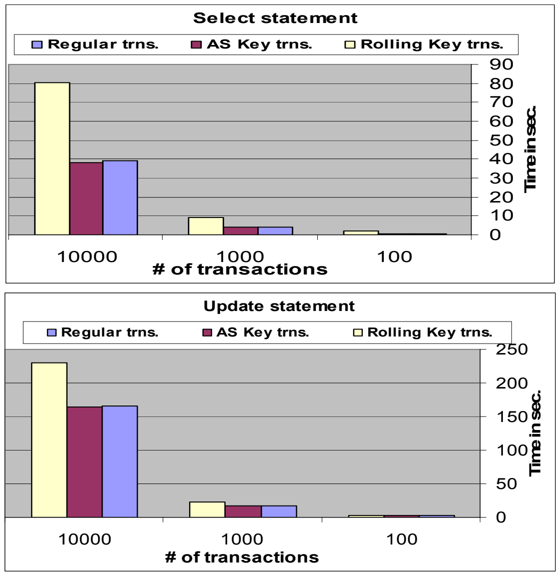

**Figure 16: performance evaluation chart**

Our experiment shows that the performance of the AS key method is the same or even better than the performance of a regular SQL statement both for select and update statements. We can explain this by the fact that the AS key method uses function that is stored in the database in a compiled form so its performance is better than the regular statement. On the other hand the performance of the rolling key method is slower than the regular statement or AS key method. This can be explained by the fact that the rolling key includes additional computation, so it must be relatively slower. We can also see that for the select statement the AS key method is approximately twice as fast as the rolling key method, but for the update statement the proportion is less than twice.

Additional experiment has been carried out to evaluate the performance overhead of the view-based web site with many concurrent sessions. The open source Book-store application (the source code can be found in [22]) was taken and modified to work with parameterized views instead of tables. For each table of the original application a parameterized view was defined, the access to this view was given to the application and the direct access to the table was revoked. The from clause of each SQL statement in the application was modified to include the As key. No additional modification of the application code or the database schema was made. The stress test was run on the original table-based application and on the modified view-based one. The results appear in the table 4 and compare the average time (in msec.) for the page response for 1, 10 and 100 concurrent users:

**Table 4: stress test evaluation**

| # of concurrent users / Application Type | 1 user | 10 users | 100 users |
|---|---|---|---|
| Original Table-based app | 10.06 | 136.99 | 1708.26 |
| Modified View-based app | 9.27 | 135.03 | 1598.20 |

Of course, additional tests must be accomplished, but it seems that there is no performance degradation when the number of concurrent users grows. It also seems that with comparison to the traditional table-based application, the parameterized view-based web application does not require additional definition of indexes or query normalizations/optimizations.

## 7. IMPLEMENTATION OF PARAMETERIZED VIEWS

Despite the fact that at the time of writing this article the parameterized views are not yet the part of the SQL standard, the need for it is great and one of the frequent questions in forums of manufacturers of databases is why there is no support for parameterized views. Because of existence of many users' requests who ask for support for these views, we may assume that the manufacturers of databases will support them in a short period of time. And this article can only emphasize the need of parameterized views in the context of database protection and access control. As we have seen, parameterized views can serve as the natural replacement of traditional views in the web era. So we want to turn to the developers of access control standards and emphasize the importance of parameterized views in the context of access control.

But our method is generic enough and can rely on any existing database's entity that receives parameters and returns data according to the received parameters. For example, we can implement our method with existing functions stored in the database (stored DB functions). These functions are supported in most databases and unlike stored procedures they can be used in standard DML sentences. The returned value of such function can be the table type. For instance, the function that returns grades of a student will be defined as following:

```
CREATE FUNCTION Student_Marks_Func
     (pAS_key) RETURNS Table
Begin
  RETURN
    SELECT   *
    FROM     Student_Marks_Table
    WHERE    Student_No IN
       (
             SELECT Student_No
             FROM   Users_Table
             WHERE  AS_key = :pAS_key
       )
End
```

**Figure 17: definition of table function**

This definition is very similar to the definition of `Sudent_Marks_View` from the figure 8 and its functionality is exactly the same in the context of database access control. The retrieval from this function will be exactly the same as from the parameterized view:

```
SELECT * FROM Student_Marks_Func(10101…)
```

**Figure 18: select from table function**

Baron and Chipman in [1] show the use of these functions for the SQL Server. They show that the table functions can be more flexible than the views as they work with parameters. The article emphasizes a very important thing: these functions can be updatable; namely, not only that it is possible to retrieve from them, while the retrieval is limited by the parameter, it is also possible to perform other operations of DML, including updating and deleting. And these operations will be limited to the respective rows according to the transferred parameter. One of the disadvantages of regular views is the difficulty of actions such as addition, deleting and updating. But as displayed in [1], the advantage of table functions is their ability to support these actions, while limiting the activity of delete and update to the return value.

For example, the next figure shows the update submitted by the customer with the key ***10101...*** that wants to change the quantity of her order:

```
UPDATE        Orders_Func(10101…)
SET           Quantity = 2
WHERE         Product_Id = 1
```

**Figure 19: update of table function**

As SQL does not update directly the `Oreders_Table`, but the `Orders_Func` instead, and the function restricts the access only to the data of the current customer, this customer cannot submit any kind of attacks that updates the data of different customers.

## 8. DISCUSSSION AND FUTURE WORK

One of the goals of organizations is to share their data and at the same time to enforce their policies. Heterogeneous web systems succeeded in sharing data to the Internet consumers. But they failed in enforcing the most important organization policy - preventing unauthorized accesses and, especially, preventing accesses of one customer to the data of another customer. The rate of attacks on web databases growths exponentially and this indicates that the existing access control mechanism of databases is inadequate for web applications.

The lack of fine-grained access control at the database level results in a situation when the database cannot explicitly authorize users accessing it. The principle of minimal privileges is violated and auditing and monitoring of user's transactions is impossible as the only user accessing the database is the super-user of a web application. Furthermore, the traditional views cannot fulfill anymore their role of the access control mechanism: if databases cannot distinguish between different users, they are not capable of defining a view that is relevant to a specific user.

Our method tries to solve the lack of authentication and fine-grained authorization at the database layer of n-tier architecture of web applications. We propose to supervise the access to the database not by the application or other external to the database tools as it happens with the existing web systems, but by the built-in database access control mechanism, that is enhanced with parameterized views.

From the standpoint of access control, the parameterized view is the natural substitution of traditional views in the web era. The use of parameterized views for controlling the accesses to the database enables to significantly reduce the range of attacks on databases. The proposed approach is an attempt to minimize the attack surface of web databases by means of native database access control mechanism that is tailored to web databases and not by the applicative means as all other methods do. Thus we move the web database protection and access control mechanism from the application layer to the database layer!

As each SQL sentence which arrives to the database from the application that requests the user's identification, contains indication of the user who ran it, the log of the database can contain SQL sentences beside the users who ran them. So it is also easy to follow the transactions of the same user, easy to distinguish between the users' different sessions and easy to discover the intruder's footprints, if needed. If the application did not request for user's identifier, it is still possible to distinguish between the sessions of different users. So the developers of intrusion detection systems to a web database can now analyze these logs and use their algorithms which were not relevant before. Thus we solved the SQL session traceability problem of the web applications!

As our method only requires definitions of a proper database schema and authorization and it can be implemented for any existing databases and enables to prevent and detect many kinds of attacks – it may become the preferable solution in comparison to any other corresponding web database access control solution from the view-point of cost and efficiency.

As we have seen, the problem of databases accessible from heterogeneous web applications is very important one. Nowadays such databases manage critical commercial and governmental information, but their access control mechanisms are inadequate. This problem is well known to the web application developers and security consultants, but it was almost not examined by academic researches. We hope that this article will expose the actual problem of web databases' access control mechanism to the research community and will motivate further researches.

Our future research will focus on the following aspects:

- extension of the proposed mechanism to the RBAC systems
- use of the above techniques as the basis for intrusion detection algorithms applied to web databases

## 9. REFERENCES

[1] Andy Baron, Mary Chipman. Creating and Optimizing Views in SQL Server. Article from: http://www.informit.com /articles/printerfriendly.asp?p=130855&rl=1 (2000)

[2] Elisa Bertino, Pierangela Samarati, Sushil Jajodia. An Extended Authorization Model for Relational Databases, In *Proceeding of IEEE Transactions on Knowledge and Data Engineering*, Volume 9, Issue 1, Pages: 85-101 (1997)

[3] Stephen W. Boyd, Angelos D. Keromytis. SQLrand: Preventing SQL Injection Attacks, In *Proceedings of the 2nd Applied Cryptography and Network Security Conf*, Pages: 292--302 (2004)

[4] Johann Eder. View Definitions with Parameters. Published in: *Advances in Databases and Information Systems (ADBIS'95)*, Pages: 170-184 (1995)

[5] Patricia P. Griffiths, Bradford W. Wade. An Authorization Mechanism for a Relational Database System. *ACM, Transactions on Database Systems* (1976)

[6] P. Gulutzan, T. Pelzer. SQL-99 Complete, Really An Example-Based Reference Manual of the New Standard. R&D Books Miller Freeman, Inc. (1999)

[7] Michael Howard, David LeBlanc. Writing Secure Code. Microsoft Press, ISBN 0-7356-1722-8 (2002)

[8] Yi Hu, Brajendra Panda. A Data Mining Approach for Database Intrusion Detection. In *Proceedings of the ACM symposium on Applied computing*, Nicosia, Cyprus. Pages: 711 – 716 (2004)

[9] Hasan M. Jamil. GQL: A Reasonable Complex SQL for Genomic Databases. In *Proceedings of International Symposium on Bio-Informatics and Biomedical Engineering*, IEEE, Pages: 50-59 (2000)

[10] Wai Lup Low, Joseph Lee, Peter Teoh. DIDAFIT: Detecting Intrusions in Databases Through Fingerprinting Transactions. In *Proceedings of the 4th International Conference on Enterprise Information Systems*, Ciudad Real, Spain, Pages: 121-128 (2002)

[11] Colin Angus Mackay. SQL Injection Attacks and Some Tips on How to Prevent Them. Article from: http://www.codeproject.com/cs/database /SqlInjectionAttacks.asp (2005)

[12] Ofer Maor, Amichai Shulman. SQL Injection Signatures Evasion. Article from: http://imperva.com/application_defense_center/white_papers /sql_injection_signatures_evasion.html (2004)

[13] K. K. Mookhey, Nilesh Burghate. Detection of SQL Injection and Cross-site Scripting Attacks, Article from: http://www.securityfocus.com/infocus/1768 (2004)

[14] Raghu Ramakrishnan, Johannes Gehrke. Database Management Systems, Chapter 17.1, Introduction to Database security. Second Edition (2001)

[15] Abhinav Srivastava, Sai Rahul Reddy. Intertransaction Data Dependency for Intrusion Detection in Database Systems, part of Information and System Security course, School of Information TEchnology, IIT Kharagpur (2005)

[16] William Stallings. Cryptography and Network Security, Third Edition, Prentice Hall International (2003)

[17] Fredrik Valeur, Darren Mutz, Giovanni Vigna. A Learning-based Approach to the Detection of SQL Attacks. DIMVAn Vienna, Austria, Pages: 123-140 (2005)

[18] SecuriTeam, SQL Injection Walkthrough, Article from: http://www.securiteam.com/ securityreviews/5DP0N1P76E.html (2002)

[19] Advanced Topics on SQL Injection Protection, OWASP. Article from: http://www.owasp.org /images/7/7d/Advanced_Topics_on_SQL_Injection_ Protection.ppt(2006)

[20] Information Security News: ISS hatches 'virtual patching' plan. Article from: http://seclists.org/isn/2003/May/0113.html (2003)

[21] Controlling Database Access, Oracle9i Database Concepts Release 2 (9.2). Article from: http://download-west.oracle.com/docs/cd/B10501_01/server.920/ a96524/c23acces.htm

[22] Online Book-Store application. The open source from the site: http://www.gotocode.com/apps.asp?app_id=3&