Improving Database Security in Web-based Environments

Francesco Di Tria, Ezio Lefons and Filippo Tangorra
Dipartimento di Informatica, Università degli Studi di Bari Aldo Moro, Via Orabona 4, 70125 Bari, Italy

Keywords: SQL Injection, Authentication, Authorization, Web Application, Architecture.

Abstract: In web applications, databases are generally used as data repositories, where a server-side program interacts with a Database Management System (DBMS), retrieves content, and dynamically generates web pages. This is known as a three-layer architecture, that is widely exposed to database threats. The attacks are usually performed through the injection of SQL code in the forms of the web applications, exploiting the dynamic construction of SQL statements. So, the database security relies on the quality of the code and the controls done by the web developer in the application level. In this paper, we present a solution for the improvement of security of databases accessed by web applications. The security is based on a user modelling approach that completely relies on the authorization mechanism of DBMSs.

1 INTRODUCTION

In Web-based environments, a three-layer architecture is commonly adopted, composed of a back-end and a front-end. In the back-end, we find two main servers: the web server and a database server (or DBMS). Currently, the web server hosts static web pages and/or scripts, which are server-side programs interacting with a DBMS in order to access the database and to provide dynamic and on-demand content.

A server-side program (or web application) needs to store a plain-text account to make connections to the database. As a consequence, anyone who has granted a physical access to the server's file system is able to view the password used for the authentication to the DBMS. In this sense, the account of a database user (even the administrator, in some cases) is publicly accessible.

Furthermore, the account is usually formed of a couple of username and password. This account is used for all the connections to the database, independently from the user profile. For the sake of simplicity, it is the web application that accesses the database, not the user. So, it is not possible for the DBMS to know who is really using the web application and to apply its own authorization mechanism (Gertz and Jajodia, 2007).

For example, if two roles, namely *manager* and *employee*, are defined in the web application, with the policy that an employee cannot access the financial data, then the DBMS is not able to know if the current user that is querying the database (through the web application) is logged in as a manager or an employee. Therefore, the DBMS cannot adopts its security model in the case of attacks to the database coming from illegitimate users (Bertino and Sandhu, 2005).

Clearly, in these environments, the database security widely depends on the web developer's ability and the program code quality. The developer has to bear all the controls necessary to protect the database. If the application is not well-developed, then the database is exposed to a class of attacks known as SQL injection that exploits the possibility to dynamically construct dangerous SQL statements (Roy et al., 2011). Therefore, the threats to database security are often attributable to the web developer's negligence or inexperience. To this reason, a web application assessment is useful to detect vulnerabilities (Huang et al., 2003; Xiang Fu et al., 2007). Another possible solution is the adoption of systems able to analyze the traffic towards a DBMS and report anomalies (Pinzón et al., 2010; Rietta, 2006).

In this paper, we present an approach to improve the database security where the security model does not depend on the application level, but it is incorporated in the DBMS level. In fact, DBMSs always include strong authorization mechanisms that, in web-based environments, are *de facto* unutilized. The proposed security model is based on

Tria, F., Lefons, E. and Tangorra, F.
Improving Database Security in Web-based Environments.
DOI: 10.5220/0005646901930200
In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016)*, pages 193-200
ISBN: 978-989-758-167-0

a mapping of web applications users, who access resources like dynamic web pages, to database users, who must be granted privileges to execute the views that generates the contents of those web pages.

In the following, we refer to a typical WAMP (Windows, Apache, MySQL, PHP) or LAMP (Linux, Apache, MySQL, PHP) environment to illustrate examples.

The paper is structured as follows. Section 2 introduces an overview of the threats to the database security. Section 3 presents our approach based on a mapping of web application users to corresponding DBMS users. Then, Section 4 reports some real scenarios and practical examples. Finally, Section 5 concludes the paper with our remarks.

2 OVERVIEW OF DATABASE VULNERABILITY

The first issue is related to the user modelling. DBMSs are provided with both an authentication and an authorization mechanism. The authentication deals with the identification of the user (who is the user) and is usually based on passwords, that is, the so-called *something you know* paradigm. The other paradigms are *something you are*, such as biometrics parameters, and *something you have*, like electronic badges.

On the other hand, the authorization is devoted to check whether an authenticated user is enabled to access a given resource (what the user can do). The authorization is usually done with privileges. So, DBMSs have a profile of each registered user (see Fig. 1) such that they are able to know the user identity and the database objects (such as procedures, views or tables) which he/she is allowed to access.

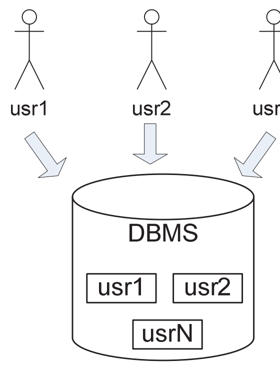

Figure 1: DBMS user modelling.

For these reasons, when users interact with a Web Application, the authentication and the authorization mechanism of the DBMS might not be applied.

In fact, such applications are configured with a unique account that is used to access the system database (see Fig. 2), which stores both data and user profiles.

As a consequence, the Web Application knows the user identity, but the DBMS does not (Ben Natan, 2005). From the DBMS's point of view, the database user is always the web application itself, *usrWA* in this example.

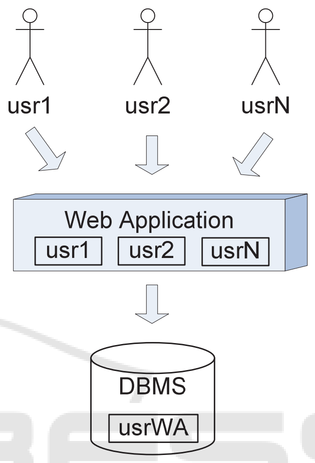

Figure 2: Web Application user modelling.

The second issue is related to the configuration file. The account (that always includes a username and a password of a database user) is used by the application and it is stored as a plain text, that is, it is not encrypted. Anyone, who has a physical (or a remote) access to the computer where the application resides, is able to read and copy the configuration file. Storing a plain-text password is the worst threat for any system.

The third important issue is related to dynamic and embedded SQL. Web applications include and create SQL instructions by concatenating text strings and user inputs. This leads to several security issue, known as SQL-injection attacks. According to this kind of attack, malicious users can insert specific sequences of characters when filling a form of a web page, determining the generation of dangerous SQL instructions (Boyd and Keromytis, 2004).

In (Halfond et al., 2006), the authors report a complete list of the possible goals that can be reached by malicious users performing SQL-injection attacks.

Bypassing Authentication. Users performing this kind of attack aim at bypassing the application authentication mechanism, by obtaining access to the system without providing a valid password.

This may happen when the web application executes no user input validation or uses a weak query like the following in order to authenticate the users:

```
select * from users where
     username='$username' and
     password='$password'
```

where `$username` and `$password` are the variables containing the user login input. In fact, if `$username` contains a malicious string like `"' or 1=1 #"`, then the query that is dynamically generated by the program is the following

```
select * from users where
     username='' or 1=1 #and
     password='$password'.
```

(The # symbol is used to comment code). The last SQL statement is a tautology and the user is authenticated without any password.

Furthermore, a threat to the authentication may be preceded by a port-scanning, a process devoted to discover and exploit potentially vulnerable services on a network host (Vasserman et al., 2009).

Extracting Data. This is the most common and the most dangerous attack, as it allows the visualization of (possible sensitive) data. Also this threat is based on alterations of legitimate SQL statements, for example adding *union* clause. As an example, if the application generates a query like

```
select projectName, projectDescr
     from projects where
     projectResp='$projectManager'
```

to extract the projects assigned to a given manager and the manager's name is a search parameter, then a user who accesses the search page can insert the string

```
union select username, password
     from user
```

to view all the projects and the user accounts. This threat can be mitigated using encrypted password and, even better, using privileges: the user used to query the database must not be granted of *read* privilege on the *users* table (*i.e.*, the table storing user profiles).

To summarize, the web applications should be configured with different database users having different privileges. So, when querying the database, the web application should connect to the DBMS using the user with the lowest privileges. On the other hand, when performing system maintenance, the application may connect to the DBMS using a user with higher privileges.

In the following, we focus on these two kinds of attack, because are strictly related to dynamic and embedded SQL statements.

To complete the overview about threats to web applications, other security issues due to network protocol or DBMS vulnerability are:

Performing Denial of Service. This attack is devoted to destroy a service, preventing other legitimate users to access the database (Jiangtao et al., 2009).

Performing Privilege Escalation. Users performing this kind of attack aim at reaching database administrator privileges (the so-called *root* privileges) by exploiting bugs in the DBMS.

3 USER MODELLING

Our approach is based on a mapping of web application users to database users, as shown in Fig. 3. So, when a new user is registered in the web application, we do what follows.

- A new record in the *users* table is inserted. This record stores the username chosen by the user. A default role, the one with the minimum privilege, is assigned to him/her. This information is managed at application level. At this level no password is necessary anymore.
- A new password-protected user is created in the database, with the same username chosen for the application level.

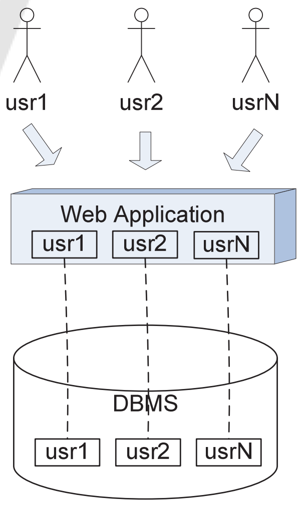

Figure 3: Web Application vs. DBMS users.

The security model at application level

establishes the resources accessible for each user on the basis of his/her role. So, the username is used by the application only to show the correct web pages to a logged user.

When a user requires a resource, by accessing a web page, the application executes a call to a procedure stored in the database. To make the connection to the database, the application must use the account provided by the user (and not the predefined one, stored in the configuration file). Of course, the user must provide username and password only the first time he/she accesses a resource, as the account can be temporarily stored in session variables.

At this point, the procedure executes the query, if the user has the correct privileges, and returns a well-formed HTML object, as a table, for example.

So, this approach is also based on a mapping of resources (web pages) at application level and corresponding resources (procedures) at DBMS level. The procedures dynamically create one or more SQL statements and return HTML tables. The main strategy is that a user who accesses a web page showing data must be granted of the necessary privileges by the database to read and/or modify those data.

In Fig. 4, there is depicted an example of a web application whose authorization model aims only at visualizing the right resources for each user. However, the database security is entrusted to the DBMS's authorization mechanism that is based on privileges.

In the example, John and Mary, whose usernames are, respectively, *john* and *mary*, are web application users. John has the *student* role, while Mary has the *teacher* role. On the basis of his/her role, a user can access a predefined set of resources. A student can access the courses list, while a teacher can access both the courses and teachers lists. So, the web application will show (the link for) the courses list to John, and it will show both (the links for) the courses and teachers lists to Mary.

In order to display the necessary data, the resource (that is, the web page) needs to invoke a procedure to retrieve the content. To do so, the courses procedure has to generate and execute the courses view. At this point, if John does not have the *read* (R) privilege on the courses view, he will not able to visualize the content. For the same reason, Mary needs the *read* privilege on both the courses and teachers views in order to display the data.

So, if a user, fraudulently or due to a web developer's error, succeeds in accessing a denied resource (at application level), he/she will not able to access the related data anyway, because he/she does not have the necessary privileges on that view and/or table.

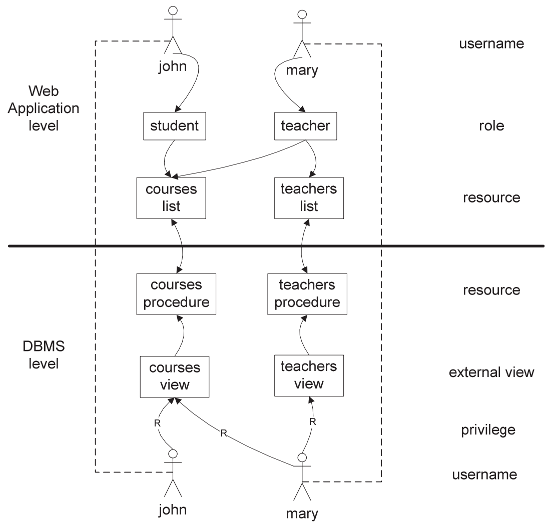

Figure 4: DBMS resources *vs.* application resources.

The benefits of these approach are the following:

- the web application does not need to store any plain-text password to access the database;
- for each connection to the database, the personal database user account is used instead of a common database account assigned to the application;
- the web application must deal only with the correct creation of the user interface (leading only to the resources the users is allowed to access according his/her role);
- no SQL injection is possible, as the query are created at DBMS level by the procedures using parameters and not at application level on the basis of string concatenation;
- assuming to correctly assign privileges on databases objects, users will never be able to view sensitive data, even by performing union-based SQL injection attacks or exploiting bugs in the application;
- the web application does not need to know anything about the logical schema of the underlying database; so, a complete independence of the application layer from the data layer is ensured; and
- the web application invokes a database resource (not a query) to retrieve the contents.

4 SCENARIO EXAMPLES

In this Section, we show some real scenarios according to the data that are exchanged between the two layers. Figure 5 illustrates the data that are currently exchanged in web-based environments.

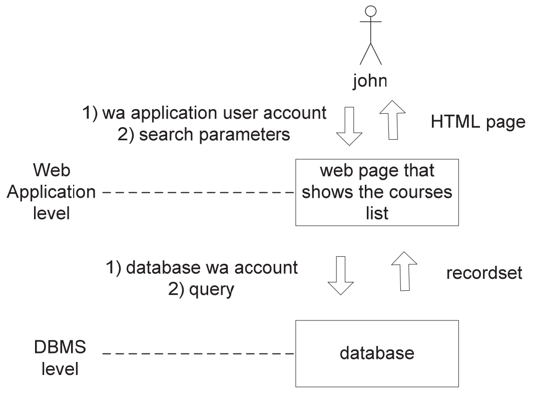

Figure 5: Traditional data flow.

Users performs authentication in the web application (WA) providing both username and password managed at application level. (The authentication is usually performed in the login page solely on the basis of the user profile stored in the database, then the logged user identifier is stored in session variables and used to access private pages).

In order to extract data, the web application uses its own database account (*wa account*, in the figure) to make a connection to the DBMS. When connected, the application generates a dynamic query, using the search parameters, and executes it on the database. The database returns a recordset that is fetched by the application and, as an example, transformed in HTML tables.

In our approach, the two layers exchange data in a different way, as shown in Fig. 6.

The user provides his/her account in the login page of the web application. Nonetheless, the application ignores the password and uses only the username to display the right web pages to the user, whereas each username is associated to a role and each role can access one or more resources. The complete account is stored in session variables, as this must be passed to the DBMS when connecting to execute queries.

To read data, the application calls a procedure with a well-defined interface (specifying the input and output parameters). The procedure eventually generates a dynamic query (or uses predefined views) and, if the provided user account has the correct privileges on views and/or tables, it returns an HTML table. To summarize, the application uses the personal account provided by the user in order to interact with the DBMS.

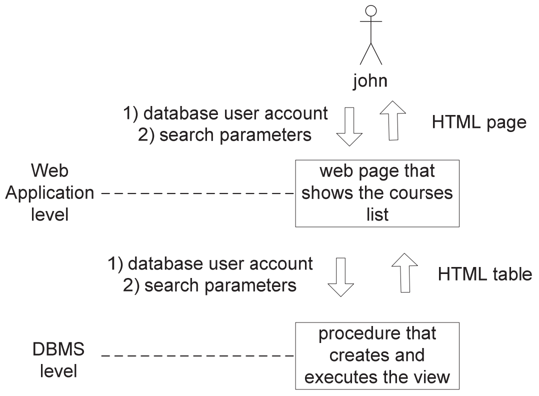

Figure 6: Proposed data flow.

4.1 Accessing Resources

In this subsection, we show an example of the login phase performed using our approach. As an example, this can be used to generate the home page of the user, showing the correct resources he/she is allowed to access.

We can assume the following relational schema that manages users and the related resources.

- `resources(id, resource, description)`. The table storing the list of resources (web pages) managed at application level
- `roles(id, description)`. The list of roles defined for users.
- `users(username, role, registration_date)`. The table storing user profiles, without passwords.
- `access(role, resource)`. The table the associates a user role to the allowed resources.

We report in Code 1 the server-side program.

Code 1. Access phase.

Input:
`$username` username,
`$password` password,
`$db_host` database IP address,
`$db_name` database name.

Output:
Web page with the list of the resources the user can access

```
1.    $db = mysqli_connect($db_host, $username,
          $password, $db_name);
```

```
2.    $query="call resources('$username',@b)";
3.    $result = mysqli_query($db,$query);
4.    $row = mysqli_fetch_array($result);
5.    print $row['b'];
```

This program first reads the account provided by the user in the login form, then uses it to connect to the DBMS (lines 1), and invokes a procedure (lines 2-3) which returns a well-formed HTML tables. At last, the HTML table is extracted just in a single statement, *i.e.,* without a *do-while* cycle (line 4), and displayed (line 5).

Then, we report the procedure (see Procedure 1) stored in the DBMS that retrieves the resources to which a user can access on the basis of the role associated to his/her username.

This procedure generates a query (line 5) and uses a cursor to fetch the rows (lines 8-15). Each row is inserted into a string containing HTML tags (line 14). To this end, the MySQL *concat* function is used to create a well-formed HTML table. However, a different text format, such as XML or JSON, can be used.

It is worth noting that in line 5 a dynamic query is created without a string concatenation, but on the basis of the parameters, *username* in this example.

To summarize, the first benefit of this approach is that the authentication cannot be bypassed using SQL-injection. Indeed, the web application does not has to generate any SQL statement and it does not manage any password neither.

Procedure 1. Access phase.

Input:
`username` nickname of the user at application and database level,

Output:
`b` HTML table

```
      CREATE PROCEDURE
1.    `security`.`resources` (in username
      varchar(50), out b text)
2.    BEGIN
3.    DECLARE no_more_rows BOOLEAN;
4.    DECLARE a varchar(50);
      DECLARE cur1 CURSOR FOR
      select resources.resource
        from resources inner join access on
          resources.id=access.resource
5.          inner join roles on
              access.role=roles.id
                inner join users on
                roles.id=users.role
        where users.username=username;
      DECLARE CONTINUE HANDLER FOR
6.    NOT FOUND
      SET no_more_rows = TRUE;
7.    set b := '<table border=1>';
8.    OPEN cur1;
9.    r: LOOP
10.     FETCH cur1 INTO a;
11.     IF no_more_rows THEN
12.       LEAVE r;
13.     END IF;
        select
14.     concat(b,'<tr><td>',a,'</td></tr>')
          into b;
15.     END LOOP r;
16.   select concat(b,'</table>') into b;
17.   select b;
18.   CLOSE cur1;
19.   END
```

Of course, if the user provides a wrong username and/or password, then a DBMS connection error will be triggered.

Moreover, since the password is provided by the user and is stored in session variables, it cannot be stolen by anyone who may be able to access the web directory (for example, using a brute force attack to the FTP server).

4.2 Search Function

As another testing scenario, now we show that this approach is also resistant to union-based SQL injections executed by appending union-clauses in fields of traditional search forms used for generating reports.

The server-side code is similar to that illustrated in Code 1. So, it is not shown again. For the same reason, we do not report the whole procedure stored in the DBMS, but only the dynamically generated query.

If the user has the role necessary to access the teaching courses list, then he/she will be able to reach the web page that contains the search form, which represents an application-level resource.

The only search parameter is *year* and this will be sent to a procedure (see, Procedure 2) in order to generate a query. In detail, the query aims at retrieving the teaching courses available in a given academic year (line 2).

To access data, the user needs also the privileges on both *courses* and *courses_year* tables that are DBMS-level resources.

Procedure 2. Search.

Input:
`year` year of the academic course,

Output:
`b` HTML table

```
1.  CREATE PROCEDURE
    `security`.`teaching` (in year varchar(100), out
        b text)
    …
2.  select courses.name from
      courses inner join courses_year on
        courses.id=courses_year.id
    where courses_year.year=year;
```

In case of an attempt of injecting malicious code in the search field by the following statement

```
2015 union select username from
     users
```

then the procedure will create a syntactically wrong SQL instruction equivalent to the following

```
select courses.name from courses
     inner join courses_year on
     courses.id = courses_year.id
     where courses_year.year =
     '2015 union select username
     from users'.
```

Notice that the data type of the input parameter is *varchar* in this *ad hoc* example (line 1). Had we used the correct data type, that is *year*, we would have obtained a data truncation error.

Moreover, assuming to create a syntactically correct SQL instruction, the user will not be able to view the data, because s/he does not have the necessary privileges on the *users* table.

5 CONCLUSIONS

In this paper, we have presented an approach to improve database security that is a current threat to web-based environments. In fact, the system security is always entrusted to web developers who must implement all the necessary input validation in order to ensure system invulnerability.

Our approach is based on a replication of user accounts at DBMS level such that the personal account of a given user is used to connect to the DBMS instead of a common password, stored as plain text. In this context, web resources, such as web pages, are mapped to database resources, such as procedures managing further resources that are external views.

The assumption is that a user may pass the application's controls, but that user will never be able to access data anyway, if he/she is not granted the necessary privileges.

Future work is devoted to a benchmark between the traditional data flow and our approach, as we believe that the creation of an HTML object is faster when using a cursor to scan a table at DBMS level than fetching rows at application level.

REFERENCES

Ben Natan, R., 2005. Implementing Database Security and Auditing. Elsevier Digital Press.

Bertino, E., and Sandhu, R., 2005. Database security - Concepts, Approaches, and Challenges, IEEE Transactions on Dependable and Secure Computing, vol. 2, issue 1, pp. 2-19.

Boyd, S. W., and Keromytis, A. D., 2004. SQLrand: Preventing SQL Injection Attacks, Applied Cryptography and Network Security, Lecture Notes in Computer Science, vol. 3089, pp. 292-302.

Gertz, M., and Jajodia, S., 2007. Handbook of Database Security: Applications and Trends, Springer, 1 edition.

Halfond, W. G. J., Viegas, J., and Orso, A., 2006. A Classification of SQL Injection Attacks and Countermeasures, Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, USA.

Huang, Y-W, Huang, S-K., Lin, T-P., and Tsai, C-H., 2003. Web Application Security Assessment by Fault Injection and Behavior Monitoring, Proceedings of the 12th International Conference on World Wide Web, Budapest, Hungary, pp. 148-159.

Jiangtao Li, Ninghui Li, XiaoFeng Wang, Ting Yu, 2009. Denial of Service Attacks and Defenses in Decentralized Trust Management, International Journal of Information Security vol. 8, issue 2, pp.89-101.

Pinzón, C., De Paz, J. F., Bajo, J., Herrero, A., and Corchado, E., 2010. AIIDA-SQL: An Adaptive Intelligent Intrusion Detector Agent for Detecting SQL Injection Attacks, 10th International Conference on Hybrid Intelligent Systems (HIS), pp. 73-78.

Rietta, F. S., 2006. Application Layer Intrusion Detection for SQL Injection, Proceedings of the 44th Annual Southeast Regional Conference, Melbourne, Florida, pp. 531-536.

Roy, S., Kumar Singh, A., and Singh Sairam, A., 2011. Analyzing SQL Meta Characters and Preventing SQL Injection Attacks Using Meta Filter, 2011 International Conference on Information and Electronics Engineering, IPCSIT vol. 6, pp. 167-170.

Vasserman, E. Y., Hopper, N., and Tyra, J., 2009. SilentKnock: Practical, Provably Undetectable

Authentication, International Journal of Information Security, vol. 8, pp. 121-135.
Xiang Fu, Xin Lu, Peltsverger, B., Shijun Chen, Kai Qian, Lixin Tao, 2007. A Static Analysis Framework for Detecting SQL Injection Vulnerabilities, 31st Annual International Computer Software and Applications Conference, Beijing, pp. 87-96.