International Journal of Security and Its Applications
Vol.8, No.2 (2014), pp.67-76
http://dx.doi.org/10.14257/ijsia.2014.8.2.07

# A Smart-driver Based Method for Preventing SQL Injection Attacks

Zhongding Dong[1], Yun Liu[1,*], Guixun Luo[1] and Sumeng Diao[1]

*School of Electronic and Information Engineering,*
*Beijing Jiaotong University, Beijing, 100044)*

***Abstract***

*SQL injection is one of the most destructive network attacks that can lead to information leakage from the database including username, password, addresses, phone number and credit card statement and so on. This information may lead to huge loss to commercial vendor, and even threaten to the national security. In this paper we put forward a novel approach in which we define a new role called smart-driver that located between the web application and the back-end database. The smart-driver will only give normal users the information belonging to them by distributing a random number to the users as their identifier or reject masquerade behavior of invalid users. By analyses, we prove that our method is more safety and can effectively protect our web application.*

***Keywords:*** *smart-driver; SQL injection; identifier*

## 1. Introduction

According to OWASP (Open Web Application Security Project), SQL Injection is categorized as one of the top-10 2010 and top-10 2007 Most Critical Web Application Security Risks [1].

SQL Injection is a type of code injection in a web application, where attackers make malicious code into a normal user input field of a web application to acquire unauthorized and unlimited source. With this kind of attack, a malicious attacker can get protected information, modify or delete crucial data, or even destroy the whole web application [2].

Many Web applications do not confirm the information in the types, allowing an attacker to input malicious SQL codes that are implemented on a back-end database. Then, the attacker can apply automated tools to collect information form tables and columns in the database.[3] Furthermore, the attacker can compromise the database by injecting more malicious codes, potentially making a database server to download other programs from a higher database server that the attacker can obtain higher priority control ability.

As a result, the application can suffer severe loss in providing normal services to its users or it may face total destruction. Such kind of crash of application can lead to bankrupt of a company or a bank, even an industry.[4]Sometimes attacker can use such kind of attack to acquire confidential information that may be relevant to the security of a country. Consequently, SQL injection is very dangerous in many cases where the attacker make use of the unchecked assumptions into the web application to obtain the unauthorized information [5].

For instance, a user of forgetting password is intended to login a website. Under such of condition he can sent his e-mail address to database server for getting back his password. The code that is probably executed has the following steps:

SELECT * FORM passwords-list WHERE email = ‘tom@gmail.com’;

ISSN: 1738-9976 IJSIA

If an attacker inject the code "OR 'a'=a" behind the e-mail address, he can obtain the whole table named passwords-list. Analogously the attacker could insert or delete the contents of the table.

## 2. Related Works

For countering SQL injection attacks, we review some classical approaches including static and dynamic countermeasures.

**New APIs**- Cook and Rai [6] suggest a Safe Query Objects, a method for expressing type-safe queries that can be executed in a database server. Queries are ruled using object-oriented classes and methods, which are converted into code that invokes standard database APIs. The method supports complicated queries with joins, parameters, existential, and dynamic criteria. This method excludes the confused relationship between untyped Java strings and SQL statements, but do not address legacy code, while also demanding programmers to study a thoroughly new API.

**AMNESIA**- Halfond and Orso [7] present AMNESIA, a fully automated tool for securing Web applications against SQLIAs. This technique uses static analysis to build a model of the legitimate queries which can be acquired from the corresponding application, and at runtime monitors the sql statement form the application to ensure generated queries accord with the statically-generated model.

**PQL**-Martin and Livshits [8] present a Program Query Language called PQL that allows programmers to express error-detection questions easily in an application-specific context. This method uses both static and dynamic techniques to solve the PQL queries question. On the one hand the Static analysis can find all potential errors in some of the cases and it can prove that the pattern will never match in other cases, on the other hand dynamic analysis can guarantee that applications can trap any instance of a certain class of errors.

**Black Box Testing**- Huang [9] propose WAVES, a method adopting black-box to check Web applications for SQL injection. The technique uses a Web crawler to find all the leaks in a Web application that can be attacked be SQLIA. Then it builds attacks that target such vulnerabilities based on a specified list of patterns and attack methods. WAVES then monitors the application's reactions to the attacks and uses machine learning techniques to promote its attack methodology. This method improves over most penetration-testing techniques by using machine learning technique.

**SQL Guard**- Buehrer and Weide [10] put forward a method that check sql statement at runtime to see if they match the model consisted of expected queries. In these methods, the model is expressed as a device that only accepts legal queries. The model is deduced at runtime by checking the structure of the query before and after the addition of user-input. Additionally, the approaches requires the programmer to either rewrite code by using a special intermediate library or manually embed special notations into the code.

**WebSSARI**- Huan and Yu [11] describe a sound and holistic method to ensuring Web application security. Viewing Web application vulnerabilities as a secure information flow problem, the authors propose a lattice-based static analysis algorithm derived from type systems and type state, and addressed its soundness. During the analysis period, sections of code considered vulnerable are instrumented with runtime guards, thus securing Web applications without user intervention. The WebSSARI tool runs by considering as filtered input that has passed through a predefined set of filters. In their evaluation, the authors can find security vulnerabilities in a range of existing applications. The elementary drawbacks of this technique are that it assumes that sufficient preconditions for sensitive functions can be precisely expressed adopting their typing system. For many types of applications, this assumption is too strong.

**Proxy Filters**- David Scott and Richard Sharp [12] put forward a proxy filtering system that enhances input verification rules on the data flowing to the Web application.

 

Using their Security Policy Descriptor Language (SPDL), programmers provide constraints and specify transformations to be applied to application parameters as they flow from the Web application to the application server. Because SPDL is highly expressive, it allows programmers to develop freely in expressing their thoughts. However, this method is human-based and, like defensive programming, commands programmers to be aware of not only which data needs to be filtered, but also what patterns and filters to apply to the data.

In this paper we put forward a novel approach that will effectively prevent the SQL injection. In our approach we define a new role called smart-driver that located between the web application and the back-end database. The smart-driver will give the normal users only the information belonging to them by distributing a random number to the users as their identifier reject masquerade behavior of invalid users. The entire processing will only run in the smart –driver and from the web application point of view it operates as normal as well. If an attacker is intended to acquire other user's information, he won't get any information about others with being unaware of other user identifier. Beside every time when a user obtain data from database, he will be distributed again a new random as his identifier. Although the attacker decode occasionally the identifier one time, he will not succeed for a new random in the next stage connecting to database. In the most case of the SQL injection, the aim of attacker is to steal the other user's data from database. So such an approach can effectively prevent the SQL injection [13].

## 3. Smart-driver

Traditionally, the structure of the web application consists of at least application interface that interact directly with users and a database running at background.[14] Users can browse all kinds of information, purchase and pay the goods they selected through application interface, while the relevant data such as personal information and the goods records stored in the database. (see the Figure 1)

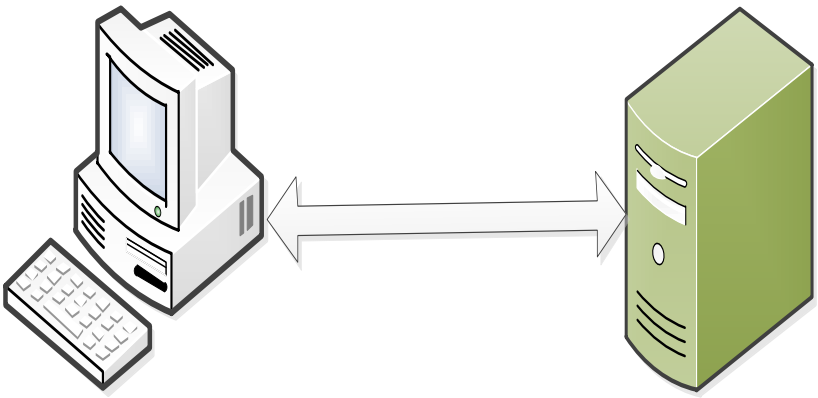

**Figure 1.**

Our smart-driver located between the application interface and the back-end database (see figure2). As an independent device, it is transparent that its function is only to prevent SQL injection and it relies neither on web application, nor on the back-end database. Additional in the most cases there is a database connectivity driver based on protocols like ODBC and JDBC. The aim of such a connectivity driver is to provide transitive function by delivering the SQL statement from the application interface to the database.

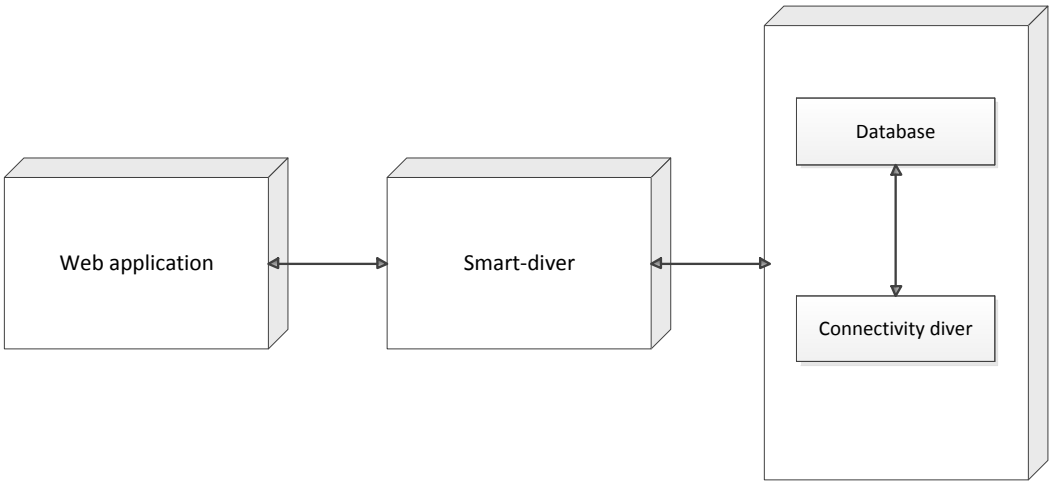


**Figure 2.**

Our smart-driver works as a connectivity driver, like a proxy lying between the application and the database. The difference of above-mentioned drivers is the function where our smart-driver enables prevention of these SQL attacks operating as a firewall between the application and the database. However, other most of connectivity drivers just simply forward the SQL statement to the database.

### 3.1. Structure

The structure of our smart-driver consists of two parts: one part is control module including a reference clock, a number generator and a calculation unit, the other part is memory module, storing user tables from the database and logged in users from web application.(see Figure 3). The processing procedure of smart-driver divides into two phases, *i.e.*, prepared stage and protected stage respectively. More detailed information about our smart-driver will be presented in the next section [15].

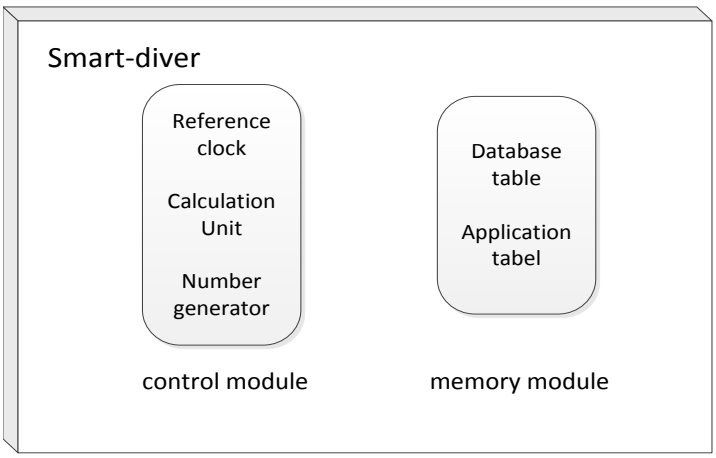


**Figure 3.**

### 3.2. Prepared Stage

As an independent compartment, the smart-driver should connect to the web application and the back-end database. Then the smart-driver needs to acquire user table and the relevant tables from the database, as well as active users who already register on the web application. For example, as Table 1 and 2 below:

 

**Table 1. User Table from a Bank Database**

| Bank clients | Credit number | Password | Other relevant information |
|---|---|---|---|
| Alex | 85435100 | 885235 | …… |
| Bob | 85636121 | 556312 | …… |

**Table 2. Active USER TABLE from WEB APPLICATION**

| Number | User name | Password |
|---|---|---|
| 1 | Alex | 885235 |
| 2 | Bob | 556312 |

### 3.3. Protected Stage

The aim of our smart-driver is to differentiate the accesses of users. [16]There are two kinds of web application: demand user's identification or not demand user's identification. For the web application which demands identification, our smart-driver adopts this identification to control access of the database. In the most cases the attacker is intended to acquire others data. Thus our smart-driver could protect the web application effectively and efficiently. For the application which doesn't need user's identification, the aim of our smart-driver is to prevent unauthorized accesses. In the following paragraph, we only consider the situation that the web application demand user's identification. We will give the detailed information about the web application that doesn't need user's identification in Section 3.3.2.

#### 3.3.1. For Registered User

In order to secure the application from SQL injection, we adopt a method that is similar to the challenge-response protocol [17]. We use a random number as the identifier following each SQL query which generates from the web application. The smart-driver works in the following ways:

1. When a user logs in the web application, he needs to provide username and password as his identification. Then the web application forwards username and password to the smart-driver.
2. The smart-driver receives and stores the data in the Active user table (see Table 2). Then it finds the corresponding user in User table (see Table 1).
3. The generator provides a random number for logged in users as their identifier. Then the smart-driver stores it in the Active user table and the User table. For example, as follow:

**Table 3. User Table from a Bank Database**

| Bank clients | Credit number | Password | Identifier (a random number) |
|---|---|---|---|
| Alex | 85435100 | 885235 | 0111001000…… |
| Bob | 85636121 | 556312 | 1011010000…… |

**Table 4. Active User Table from Web Application**

| Number | User name | Password | Identifier(random number) |
|---|---|---|---|
| 1 | Alex | 885235 | 0111001000…… |
| 2 | Bob | 556312 | 1011010000…… |

4. Next the smart-driver gives the active user's identifier back to web application. Each SQL query forwarding to database from web application will represent the corresponding user with the identifier.
5. Through the identifier followed the SQL query, the smart-driver finds the corresponding user according to User table. Then the smart-driver puts the correct SQL query to the database and gets the relevant data back.
6. The identifier will be cleared when the corresponding user departs from the web application.

By this way, when a SQL query comes, the database only provides the data which belongs to the corresponding user with the given identifier. This method can reduce the kind of attack where the attacker is intended to acquire other's information. For example we introduced in the Section 1.

SELECT * FORM passwords-list WHERE email = 'tom@gmail.com 'OR 'a'='a';

As the condition 'a '= 'a' always holds, the database will return the passwords of all the users. But with our smart-driver, such a condition can't happen. For with a random number as the identifier, the user only can obtain the specific data. Namely, the attacker may get data about his password and other information, but not the data of others.

To be easy to explain the procedure of our smart-driver, we omit to describe the function of reference clock. Under an unfortunate circumstance, the attacker may guess the identifier. In that case, the attacker can successfully attack the web application. So in order to secure our smart-driver, we put the reference clock in it. Every once in a while we update the random number in table from both web application and database. The frequency we update the random number depends on specific condition of the corresponding web application and the database. In such case, even if an attacker guesses a random number once, he will not successfully attack the system, because the random number will be updated in a few seconds.

Considering some worst situation that the attacker lucky guessed the random several times or has the strong computing power, we put forward the second relatively complicated way to prevent SQL injection. [18] The second way is similar to the above method, but needs more calculation in smart-driver. Specific operations are described as follows:

1. The smart-driver takes the user's identity $ID_i$ and password $PW_i$ from the web application when the user $U_i$ logs in the web application, and stores the data in the Active user table.
2. The generator provides a random number $b$ and two secret numbers x and r for the logged in users. Then the smart driver computers($T_i$ ,$V_i$ , $A_i$ , $R_i$ , $H_i$ ) in accordance with the following equations:

 

$$T_i = h(ID_i \,||\, x)$$

$$V_i = T_i \oplus h(ID_i \,||\, h(\oplus b PW_i))$$

$$A_i = h(h(\oplus b PW_i) \,||\, \oplus h) \oplus r($$

$$B_i = A_i \oplus h(b \oplus PW_i)$$

$$R_i = h(h(\oplus b PW_i) \,||\, r)$$

$$H_i = h(T_i)$$

3. Then the smart-driver stores $(V_i, B_i, R_i, H_i, b)$ in the Active user table. For example, as follow:

**Table 5. Active User Table from Web Application**

| User name | User's identity($ID_i$) | Password($PW_i$) | Identifier |
|---|---|---|---|
| Alex | 1211001 | 885235 | $(V_i, B_i, R_i, H_i, b)$ |
| Bob | 1211002 | 556312 | $(V_i, B_i, R_i, H_i, b)$ |

4. This procedure is same as the above.
5. When the user $U_i$ wants to acquire the information from the database, he puts forward the SQL query with the Identifier to smart-driver. Then the smart-driver performs the following computations:

$$T_i = V_i \oplus h(ID_i \,||\, \oplus(b \quad P_i$$

$$H_i^* = h(T_i)$$

6. Next checks whether the $H_i^*$ and $H_i$ is equal or nor, if yes, the legitimacy of the user can be assure and puts the correct SQL query to the database and gets the relevant data back; otherwise, rejects the request.

The attacker cannot masquerade as a legitimate user to get the confidential data table the database, even if the attacker acquire the Identifier $(V_i, B_i, R_i, H_i, b)$, because he cannot breach secret numbers x and r. Therefore this method can prevent SQL injection effectively. However, it also has disadvantage. It needs more calculation and response time. For the first approach it looks simple, but it is convenient and saves computing resource. Depending on diverse situations, we adopt corresponding method.

### 3.3.2. For Anonymous Users

For the web application which doesn't authenticate users, we can also use the smart-driver to prevent the SQL injection. The procedure is similar to the previous method. First when an anonymous use logs in the web application, the smart-driver gives a random number as his identifier.

**Table 4. Active Anonymous User Table**

| User | Identifier |
|---|---|
| Anonymous 1 | 1011110001…… |
| Anonymous 2 | 0100101000…… |

At the same time another table records the relation between the identifier and the priority of the database. Namely, the anonymous users are assigned the limited access right for the different table in the database. For example as follow:

**Table 5. Relation between Identifier and Right**

| Identifier | The limited right | Table name |
|---|---|---|
| 1011110001…… | View | Scores |
| 0100101000…… | Write | Comment |

In such a case, if an intruder wants to launch the SQL injection attack, he will not be successful to obtain the unauthorized data, because his access permission for the table in the database is limited. For example in the shopping website, every consumer can retrieve the all goods information. The select statement can look like this:

SELECT * FROM goods-list WHERE goods-id = '……'

Now if the attacker wants to tamper the goods information by SQL injection, he will fail due to limited right that he only has read permission to goods-list table. So our smart-driver can also prevent the SQL injection attack by the anonymous user effectively.

## 4. Analyses

The advantage of the smart-diver describe above are obvious, because it can distribute the specific information to the users. Every SQL statement form the web application carries with the specific identifier that stands for the corresponding users, so the database only provides access to the data relevant to the particular user. Next, we will present the analysis of smart-driver from cryptology point of view.

The safety of our smart-driver relies on the difficulty of guessing the special identifier. [19] If a malicious user is intended to acquire other information, he needs to get the corresponding identifier. We assume the identifier as a 32 bits random number, so that the probability of guessing this number correctly is $1/2^{32}$. If we assume the maximum of the users logged in the web application at the same time is $x$, the probability of successful attack is $x/2^{32}$. Note that this is an unselective attack for the attacker cannot control the attacked user's identifier. Additionally our smart-driver will update the user's identifier from both web application and the database, so the probability of the successful attack is very low. If we want to promote the security of the smart-driver, we can increase the length of identifier. However doing that will add the burden on the communication between the web application server and the database server.

## 5. Conclusions

Recent years the rate of code injection attack on web database growths exponentially, [20] especially the SQL injection, as a common and relative facile way and this demonstrate that the existing access control mechanism of the database is deficient for the web application. The lack of access control at the database can lead to a condition where the database cannot definitely authorize users accessing it and auditing and monitoring of user's operation is impossible.

 

Our approach tries to handle the problem of the lack of adequate authentication at the database of n-tier architecture of web application. We propose an independent device called smart-driver between web application server and back-end database. It is portable and relies neither on web application, nor on the back-end database. As each SQL query which arrives to the database from the application carries with specific identifier. By this identifier, the database gives the corresponding data to the user who has the corresponding access permission. Through the preceding part of the paper, our smart-driver is safety and can protect the web application from SQL injection effectively and efficiently.

## References

[1] http://www.owasp.org/index.php/Top_10_2010-A1-Injection, retrieve on 13/01/2010.
[2] J. Clarke, "SQL injection attacks and defense", Access Online via Elsevier, **(2012)**.
[3] S. M. S. Sajjadi and B. T. Pour, "Study of SQL Injection Attacks and Countermeasures", International Journal of Computer and Communication Engineering, vol. 2, no. 5.
[4] C. Anley, "Advanced SQL injection in SQL server applications", White paper, Next Generation Security Software Ltd., **(2002)**.
[5] U. Chandrasekhar and D. Singh, "Understanding Query Vulnerabilities for Various SQL Injection Techniques", In Intelligent Computing, Networking, and Informatics, Springer India, **(2014)**, pp. 1063-1075.
[6] W. R. Cook and S. Rai, "Safe query objects: statically typed objects as remotely executable queries", In Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference, IEEE, **(2005)** May, pp. 97-106.
[7] W. G. Halfond and A. Orso, "Preventing SQL injection attacks using AMNESIA", Proceedings of the 28th international conference on Software engineering, ACM, **(2006)** May, pp. 795-798.
[8] M. Martin, B. Livshits and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language", ACM SIGPLAN Notices, ACM, vol. 40, no. 10, **(2005)** October, pp. 365-383.
[9] Y. W. Huang, S. K. Huang, T. P. Lin and C. H. Tsai, "Web application security assessment by fault injection and behavior monitoring", Proceedings of the 12th international conference on World Wide Web, ACM, **(2003)** May, pp. 148-159.
[10] G. Buehrer, B. W. Weide and P. A. Sivilotti, "Using parse tree validation to prevent SQL injection attacks", Proceedings of the 5th international workshop on Software engineering and middleware, ACM, **(2005)** September, pp. 106-113.
[11] Y. W. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee and S. Y. Kuo, "Securing web application code by static analysis and runtime protection", Proceedings of the 13th international conference on World Wide Web, ACM, **(2004)** May, pp. 40-52.
[12] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, Y. W. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee and S. Y. Kuo, "Securing web application code by static analysis and runtime protection", Proceedings of the 13th international conference on World Wide Web, ACM, **(2004)** May, pp. 40-52.
[13] D. Scott and R. Sharp, "Abstracting application-level web security", Proceedings of the 11th international conference on World Wide Web, ACM, **(2002)** May, pp. 396-407.
[14] R. Kozik and M. Choraś, "Machine Learning Techniques for Cyber Attacks Detection", Image Processing and Communications Challenges, Springer International Publishing, vol. 5, **(2014)**, pp. 391-398.
[15] D. Mitropoulos and D. Spinellis, "SDriver: Location-specific signatures prevent SQL injection attacks", computers & security, vol. 28, no. 3-4, **(2009)**, pp. 121-129.
[16] W. Win and H. H. Htun, "A Simple and Efficient Framework for Detection of SQL Injection Attack", IJCCER, vol. 1, no. 2, **(2013)**, pp. 26-30.
[17] A. Roichman and E. Gudes, "Fine-grained access control to web databases", Proceedings of the 12th ACM symposium on Access control models and technologies, ACM, **(2007)** June, pp. 31-40.
[18] H. C. Hsiang and W. K. Shih, "Improvement of the secure dynamic ID based remote user authentication scheme for multi-server environment", Computer Standards & Interfaces, vol. 31, no. 6, **(2009)**, pp. 1118-1123.
[19] L. K. Shar, H. B. K. Tan and L. C. Briand, "Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis", Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, **(2013)** May, pp. 642-651.
[20] J. Minhas and R. Kumar, "Blocking of SQL Injection Attacks by Comparing Static and Dynamic Queries", International Journal of Computer Network and Information Security (IJCNIS), vol. 5, no. 2, **(2013)**, pp. 1.

 