Preventing SQL injection attacks by automatic parameterizing of raw queries using lexical and semantic analysis methods

S. Dolatnezhad Samarin and M. Amini*

Department of Computer Engineering, Sharif University of Technology, Azadi Ave., Tehran, P. O. Box 11155-8639, Iran.

Received 22 May 2017; received in revised form 20 June 2018; accepted 7 January 2019

KEYWORDS
Database security; SQL Injection (SQLI); Intrusion detection and prevention; Parameterized query; Semantic analysis.

Abstract. SQL Injection (SQLI) is one of the most important security threats to web applications. Many techniques have been proposed for countenacting SQLI Attacks (SQLIAs); however, second-order attacks and the injection attacks that raise data-type mismatch errors have been ignored in most of them. In this paper, we propose a new anomaly-based method (deployed as a proxy between the application server and its database server) for detection and/or prevention of SQLIAs without requiring any modification to the source code of vulnerable applications. The majority of attacks, which lead to a change in the syntax of application queries, are identified in the detection phase by lexical analysis of the queries. The remaining types of attacks, such as second-order attacks and attacks generating data-type mismatch errors, are prevented in the prevention phase, where each query is automatically converted to a parameterized query (before submitting to the database) using a semantic analysis method.

© 2019 Sharif University of Technology. All rights reserved.

1. Introduction

Nowadays, most organizations use web applications to collect, store, process, and represent their required information using back-end databases. If an application does not preserve its security completely, intrusion to its back-end database can be performed easily. OWASP is a non-commercial organization that has been created to improve the security of applications. This organization reports the 10 most common vulnerabilities in web applications every three years. In its latest report [1], OWASP identified injection vulnerability as the most common security risk in web applications. One type of injection vulnerabilities in web applications is SQL injection (SQLI). The main reason for this vulnerability is the use of dynamic queries in applications. A dynamic query is a query that is created during the application execution time by concatenating some inputs (taken from a user) to the rest of the query. An attacker can append SQL codes to the dynamic query, create their desirable query, and run an SQL Attack (SQLIA).

SQLIAs can be classified from two different perspectives, namely the order of the attack and how the application responds to a malicious input. From the first perspective, we can classify attacks as the first-order and second-order. In the first-order attacks, the user’s input is concatenated to the query and when the database executes the query, the attack is completed. Second-order attacks occur when the application stores user’s input in the database and uses it in another query to complete the attack later.

Responses to a malicious input could be different with respect to the purpose of the attacker. The
application may do an action (such as user authentication), show some data to the attacker, show an error message, or do nothing. In many cases, an attacker uses error messages returned by the application and extracts some information from these messages. Database error messages could be sorted into three groups [2], namely logical errors, syntax errors, and data-type mismatch errors. Attacks with the purpose of raising the data-type mismatch errors cannot be simply detected and require the query to be executed or having information about the data type of the input fields in the query. In this paper, we focus more on the second-order attacks and data-type mismatch errors, which are not considered in most of the existing solutions to SQLIAs.

Many techniques and frameworks have been proposed for detecting and preventing SQLIAs in the application level, database level, and proxy level; however, some of them need to change the source code of applications and most of them ignore second-order attacks and injection attacks resulting in data-type mismatch errors. One of the best methods that can prevent SQLIAs completely is employing prepared statements (parameterized queries). Some methods use this feature for preventing SQLIAs automatically. Thomas et al. [3], Biht et al. [4], and Masri and Sleiman [5] stated that a method needed to access the application source code that automatically replaces queries with parameterized queries.

The proposed approach in this paper, similar to some other related work, utilizes parameterized queries to completely prevent SQLIAs. The main contribution of the proposed approach in contrast to other related approaches such as [3-5] is that the parameterized queries are created automatically (by semantic analysis of the queries) in the middle of the application and the database without interfering the functionality of the application or requiring any change in the application source code.

The proposed method is a kind of anomaly-based intrusion detection and prevention approach. This method consists of three phases: learning, detection, and prevention. In the learning phase, the profiles of legal queries are generated by lexical analysis and syntactic structure of them. These profiles are used to detect attacks in the detection phase and to create parameterized queries in the prevention phase. Executing parameterized queries (instead of the raw ones) prevents completing the attacks that do not directly change the structure of the queries, such as the second-order attacks. In preparing the parameterized queries, we can also easily detect the malicious queries resulting in data-type mismatch errors without needing to execute the queries.

Another advantage of the proposed method is its effect on the performance of the application. This method imposes negligible performance overhead in the initial execution of the queries due to the creation of parameterized queries, but overall, it also improves the performance of the application using the prepared parameterized (or pre-compiled) queries.

In summary, the main contributions of the paper are as follows:

- The proposed approach is a new general approach which is application-independent. It works as a proxy and does not need the source code of the under-protected applications (in fact, having the binary code is sufficient). It can be used for different applications written in different programming languages;
- The proposed approach can detect and prevent all known SQLIAs, especially those that use data-type mismatch errors and second-order attacks before delivering the query to the database server. To the best of our knowledge, the existing mechanisms are vulnerable to these two types of attacks;
- The proposed approach not only does not attenuate the performance of the under-protected applications, but also improves their performance after a while.

The rest of this paper is organized as follows. Section 2 surveys the existing detection and prevention methods. We explain our proposed solution in Section 3 and evaluate its accuracy and overhead in Section 4. Finally, Section 5 concludes the paper.

2. Related work

Various defense techniques and frameworks have been proposed for preventing SQLIAs in dynamic queries.

These techniques can be considered in three different levels [6]:

- **Application-level techniques**, which are called defense coding.
- **Database-level techniques**, such as access control and stored procedures.
- **Proxy-level techniques**, such as intrusion detection and filtering tools.

Recently, some methods have been introduced for using machine learning for huge amounts of data in learning and detecting SQLIAs [7,8].

Employing some of these techniques is not a simple task; it is often done during the application development process and needs to change the application source code, which is not always possible. Therefore, researchers try to propose methods to automate these techniques. For example, the proposed method by Hafiz and Johnson [9] automatically converts the input to a secure input in the application.
Overall, automated methods for detecting and preventing SQLIs are based on two different types of analysis, namely static and dynamic. By using static analysis methods, such as the ones proposed in [10-13], we can find and fix errors and SQL vulnerabilities of the application, before deploying the application in its operational environment. These methods need to change the source code of applications, while dynamic analysis methods, such as the ones proposed in [14-16], detect SQLIs at runtime and in the test phase of the application development process. These methods do not need to modify the application source code, and the vulnerabilities found using these methods should be fixed manually. Most of the existing methods, such as [17-20], use a combination of these two methods for detecting or preventing SQLIs at runtime when the application is deployed in its operational environment. These methods use static analysis to analyze the source code or equip the code (by making some changes) to detect or prevent SQLIs at runtime.

Our proposed method in this paper detects and prevents SQLIs at runtime. For this reason, in this paper, we focus more on the techniques that could be used at runtime and are closely related to our approach. These techniques can be classified into three categories: grammar-based, query skeleton-based, and anomaly-based methods.

2.1. Grammar-based methods

In this type of methods, detection or prevention of attacks is performed according to the syntax of SQL statements. The methods that use the parse tree data structure or models (which are created based on the SQL grammar) can be placed in this category. The parse tree is a suitable data structure for detecting SQLIs. Because a minimal change in the structure of the query leads to a different parse tree of the query, Su and Wasserman [21] added some meta-characters to the query to specify the beginning and end of the input values, and used an augmented grammar to create the parse tree of the query. At runtime, if the augmented query was parsed correctly, it was benign; otherwise, it was malicious. Buehrer et al. [22] and Bandhalavi et al. [23] used similar approaches in their systems. They created a benign sample of each query in the application. At runtime, the parse tree of the generated query and the sample query were compared; if they matched, no attack was diagnosed and the query was sent to the database. Halfford and Orso [19] used a finite state machine to model queries in an application. The change from one state to another was done with respect to the SQL operators, keywords, and delimiters. At runtime, the dynamic query traversed throughout the model and if it violated the model, it was deleted.

Grammar-based methods can detect all attacks leading to change in the syntax of the query, but the second-order attacks and attacks raising the data-type mismatch errors cannot be detected. Furthermore, this type of methods needs an analyzer (such as a parser) to analyze the SQL queries, which have various grammar in different DBMSs.

2.2. Query skeleton-based methods

In query skeleton-based methods, a template or a skeleton is generated for each query in an application that is used for detecting attacks. If the template of the query changes at runtime, the attack is detected. The template of the query is a query string in which all constant values are labeled or replaced by a placeholder. Lee et al. [24] used a function for removing attribute values from the query string. This function removed values based on the single quotes existing in the query. As we know, numeric values do not need to have single quotes, so their function does not work properly for numerical entries. Kar and Panigrahi [25] created a template of the query according to a conversion table. Their approach computed the hash of the template as a digest of the query and used it for detecting attacks. The main problem of this approach is its weakness in detecting and preventing the second-order attacks.

2.3. Anomaly-based methods

Anomaly-based intrusion detection systems create a profile of the normal behavior of users or applications. Anything outside the profile is considered as an attack. Valeur et al. [26], in 2005, proposed an anomaly-based intrusion detection system for databases. Their approach has been the basis of many studies in this field, such as [27-29]. Their system uses several models to characterize the profile of normal access to the database. This approach can detect all attacks with a low rate of false-positive errors; however, in this method, it is still possible to do some of the SQLIs. For example, attacks that raise data-type mismatch error cannot be identified using this method. Prabakar et al. [30], in 2013, proposed an anomaly-based prevention method that created profiles for anomaly patterns and checked the received query with them. If the query matched an anomaly pattern, an attack was detected; otherwise, an anomaly score was calculated for it. If the score was more than the predefined threshold, the query was reported to the administrator.

3. Overview of the proposed approach

Our proposed solution is a kind of anomaly-based intrusion detection and prevention methods. In this approach, the normal behavior is defined by collection of information about the raw (benign) queries of the application. The information is extracted from each raw query of the application in the learning phase and named as the profile of the query. The profile of a query
consists of the query identifier and other information about the structure of the query such as the number and type of its parameters.

After learning and in the deployment phase, if the received query is a legal query of the application, the corresponding profile is found and other information in the profile is checked. In fact, in this approach, any deviation from the normal behavior is considered as an attack.

The structures of most of the malicious queries differ from all the benign queries of the application. Therefore, by extracting the structures of benign queries in the learning phase and storing them in a collection of normal query profiles, malicious queries could be detected.

After detecting a query as a benign one, the information in its profile is used to convert the query to a parameterized format. Then, the parameterized query is sent to the database instead of the main raw query. The purpose of creating the parameterized query, after the process of detection, is preventing attacks that do not change the structure of the queries. These attacks include second-order attacks and attacks that try to create data-type mismatch errors to infer some information about the schema of the database.

3.1. Proposed architecture
The architecture of the proposed solution is illustrated in Figure 1 and contains the following components:

- **Key generator module (lexical analyzer module):** This module creates a key (a query identifier) for each incoming query. The key is created using the lexical analysis of the query. In the learning phase, the generated key is stored in the knowledge base at runtime, and the key is used to detect attacks and extract the required information from the knowledge base for possibly normal queries;

- **Detection module:** In the detection stage, this module searches the generated key for each incoming query in the knowledge base. If it does not find the key, the query is considered as a malicious one and blocked. Otherwise, the query is transmitted to the prevention module;

- **Prevention module:** This module is to prevent attacks that are not diagnosed in the detection phase and consists of two sub-modules:

  1. **Semantic analyzer module:** In the learning stage, semantic analysis of each query is performed in order to extract the required information in creating a parameterized query in the prevention stage;

  2. **Parameterized query generator:** In the prevention stage, according to the information obtained in the learning phase, this module converts the query to a parameterized one. At this stage, we can detect data-type mismatch errors too.

- **Knowledge base:** The required information to detect and prevent attacks is stored in the knowledge base. The information consists of the keys of the normal queries and a sequence of required information to create parameterized queries at runtime.

According to the described architecture, our proposed process for detection and prevention of attacks is shown in Figure 2. As it is shown, this solution consists of three phases: the learning phase, the detection phase, and the prevention phase. Each of these phases is described in more details in the rest of this paper.

3.2. Learning phase
The power of an anomaly-based intrusion detection system depends on the accuracy and completeness of the information obtained in the learning phase.

![Figure 1. Architecture of the proposed system.](image_url)
Hence, the learning phase is the most important one in the process of detection and prevention. At this stage, all queries in the application that contain benign input values are sent to the database in a protected environment. The queries are captured in the middle of the way (from the application server to the database) in order to extract the required information and store it in the knowledge base. The gathered information is used during the process of detection and prevention to reduce the processing overhead at runtime. This approach is applicable to the applications that have queries with predefined structures. For example, consider an application allows its users to enter SQL codes in an input field. In this application, users can create any SQL commands. Hence, the learning phase cannot create a unique set of profiles and this approach cannot be leveraged in such applications.

The information obtained in this stage for a query \( q_i \) of an application contains the following items:

- \( key(q_i) \): The key of the query (a query identifier);
- \( I(q_i) \): The number of the constant tokens in the query that may contain user input;
- \( DT_j(q_i) \): The data type of the \( j \)'th constant token.

The data type is needed for setting the type of the parameters (or arguments) in the template of the parameterized query.

- \( Q\theta(q_i) \): The template of the query. The template of the query is a query string in which the probable input values are replaced by a placeholder.

To calculate the key, lexical analysis of the query and to obtain other information, semantic analysis of the query are required.

3.2.1. Lexical analysis

The process of converting a query into a sequence of tokens is called lexical analysis of the query [31]. The proposed detection method detects attacks by using the key of the query created in the lexical analysis process. The key is constructed from the string of the query with empty input fields. In the learning phase, the keys are created for all queries in the application. SQL As mainly lead to a change in the string of the query. At runtime, the query with filled input is received in the proxy and the key of the received query is created using the lexical analysis process. Then, the key is searched
among the keys created in the learning phase. If the key is found, the attack is detected.

Consider the following query as a sample query in an application. In this query, user input can determine the name and age columns of a record in the accounts table.

query = SELECT 2 FROM accounts WHERE name = 'goodUser' AND age BETWEEN 10 AND 50;

Key generation process
For generating a key for the above query, the following steps should be taken:
1. Perform the lexical analysis of the given query and obtain its tokens. The tokens of the above query are shown in Figure 3. Constant tokens in this query are highlighted;
2. Replace the constant tokens with question mark (?);
3. Uppercase the characters of the tokens and concatenate them together to create the new string;
4. Remove all white spaces in the query string;
5. Compute the hash of the key and store it in the knowledge base.

Based on the mentioned steps, the key for the above sample is shown in Box I.

3.2.2. Semantic analysis
To automatically create a parameterized query in the prevention phase, we need additional information about the query. This information is derived from the semantic analysis of the query in the learning phase. The semantic analysis is performed using the parse tree structure of the query. The main parts of the SQL statements, which may contain the inputs, are SQL expressions. In the proposed solution, expressions in different parts of the query are obtained with the pre-order traversal of the parse tree. Then, the required information is extracted from the expressions. The information extracted for each input (in the output of the semantic analysis process) has the same order as the input in the query.

For example, the pre-order traversal of the following expression is shown in Figure 4.

\[ \text{expression E: `text'|| (1+2) } \]
\[ \text{DT1 (E) = String} \]
\[ \text{DT2 (E) = Numeric} \]
\[ \text{DT3 (E) = Numeric} \]
\[ \text{template (E) = `? ` || (?*?)} \]

3. Comparative expressions: The comparative
expressions contain comparison operators (such as >, = or Like). In some comparative expressions, in which one of the operands has a constant value and another is a column of a table, we obtain the data type of the constant value with querying the database. For example, information obtained from the expression `username = ‘gooduser’` is as follows.

expression E: `username = ‘gooduser’`

DT1(E) = String

template (E): `username = ?`

4. **Logical expressions:** The expressions containing logical operators (such as AND OR ) are logical expressions. Each operand in these expressions can be a relational, computational, or comparative expression; thus, we need to analyze them separately based on their type.

The semantic analysis in different SQL commands varies with the grammar of the queries. In this paper, we explain the semantic analysis of the SELECT, UPDATE, INSERT, and DELETE statements that are commonly utilized in web applications.

**Semantic analysis of the SELECT statement**

The SELECT statement is typically used in most of the applications. We can use this statement in different parts of the SELECT (such as the SELECT clause, FROM clause, or WHERE clause) and other commands. For semantic analysis of the SELECT query, all expressions in different parts of the query must be extracted and processed in order. The grammar of this command is shown in Figure 5.

In this figure, the parts which are enclosed in a square may contain expressions. Thus, they must be extracted and analyzed in the semantic analysis process. Algorithm 1 describes how the semantic analysis of the SELECT statement is done. This algorithm first checks whether the command is a relational expression or not. If it is a relational expression, each of the queries on the left and right operands must be extracted and analyzed separately. Otherwise, all expressions in the query are extracted and analyzed respectively.

For example, consider the following query:

```
query = SELECT name||‘;’||lastname FROM
(SELECT * FROM accounts.users WHERE
username = ‘user_id’ and age BETWEEN 20 AND 50) 
WHERE name LIKE ‘ali%’;
```

After the semantic analysis of this query, the information extracted from the SELECT clause will be as follows:

```
SELECT_clause: name||‘;’||lastname
```

```
DT1(SELECT_clause)= String
Template(SELECT_clause)= name||‘;’||lastname
```

The following information is extracted from the FROM clause:

```
FROM_clause: (SELECT * FROM accounts.
users WHERE username = ‘user_id’
AND age BETWEEN 20 AND 50)
```

```
DT1(FROM_clause)= String
DT2(FROM_clause)= Number
DT3(FROM_clause)= Number
```

```
template(FROM_clause): (SELECT * FROM accounts.
users WHERE username=? 
AND age BETWEEN ? AND ?)
```

The following information is extracted from the conditional WHERE clause:

```
WHERE_clause: name LIKE ‘ali%’
```

```
DT1(WHERE_clause)= String
```

![Figure 5. Grammar of the SELECT statement.](image-url)
1: procedure AnalyzeSelectStmt(SelectStmt)
2:   if SelectStmt is relational expression then
3:     AnalyzeSelectStmt (left_operand)
4:   else
5:     for each clause in the SelectStmt do
6:       if the clause contains expression then
7:         left_operand = expression → left_operand
8:         right_operand = expression → right_operand
9:       if left_operand is sub_query then
10:      AnalyzeSelectStmt (left_operand)
11:     else
12:       add the information of the
13:       left_operand to the inputs.info
14:     end if
15:     if right_operand is sub_query then
16:      AnalyzeSelectStmt (right_operand)
17:     else
18:       add the information of
19:       the right_operand to the inputs.info
20:     end if
21:   end for
22: end procedure

Algorithm 1. Semantic analysis of the SELECT statement.

![Figure 6. Grammar of the UPDATE statement.](image)

template(WHERE_clause): name LIKE ?

Finally, the information obtained for the above query is as follows:

I(query) = 5
DT1(query) = String
DT2(query) = String
DT3(query) = Number
DT4(query) = Number
DT5(query) = String
Qt(query): SELECT name||\?||last_name FROM (SELECT * FROM accounts, users WHERE username=?
     and age BETWEEN ? AND ?)
WHERE name LIKE ?;

Semantic analysis of the UPDATE statement

The UPDATE statement is used for assigning a new value to a column of a table in a database. The syntax of this command is shown in Figure 6. In this command, the input can be entered in the SET clause and WHERE conditional clause; therefore, we need to extract these two clauses and analyze them in order. In a SET clause, the left operand is a column name or a list of column names and the right operand is only a constant value or a list of constant values. Hence, the data type of the constant value of the right operand should be same as the corresponding column in the left operand and it is obtained by querying the database. The conditional WHERE clause is analyzed similarly to the analyzing WHERE clause in the SELECT statement.

Semantic analysis of the INSERT statement

The INSERT command is used for inserting one or more rows in a table of a database. This command can select some columns of a table and insert the values in them. In Figure 7, the syntax of this command is illustrated. As it is observed, the input can just appear in the VALUES clause. The VALUES clause can be a SELECT query, a set of constant values, or a set of expressions. The data type of the values is determined based on the data type of the corresponding column in the query. If the names of the columns are not specified in the query, they should be obtained by querying the database.
above query, the generated key becomes as follows, which differs from the main key; thus, the attack is detected and the query is blocked.

```
SELECT * FROM accounts WHERE username=? AND password=?;
```

When the attacker’s input does not change the syntax of the query (e.g., in second-order attacks), this module cannot detect the attack. Therefore, we transmit the query to the prevention module. The prevention module converts the raw query to a parameterized query that results in preventing such attacks and other unknown attacks.

3.4. Prevention phase

As mentioned before, in the prevention phase, the query is converted to a parameterized query and sent to the database. Parameterizing the query or using a prepared statement is a method that can prevent SQLiAs, including second-order attacks and attacks raising data-type mismatch errors. Creating the parameterized query from the profile of a query (which is formed in the learning phase) consists of two stages:

- **Extracting input values**: Input values are derived by comparing the template of the query (which is obtained from the knowledge base) and the main query. The process of extracting inputs for a sample query is shown in Figure 9.
- **Creating parameterized query**: After extracting all the inputs, the parameterized query is created by using the information existing in the profile of the query (obtained from the knowledge base). The process of creating a parameterized query is represented in Algorithm 2. In this algorithm, \( Qt(q) \) is the template of the given query \( q \), \( I(q) \) is the number of inputs in \( q \), \( DT(q) \) is a set containing all data types

```
SELECT * FROM accounts WHERE name = ? AND age BETWEEN ? AND ?
```

```
SELECT * FROM accounts WHERE name = 'Good User' AND age BETWEEN 10 AND 50
```

Figure 9. Extracting input values of a sample query.
of inputs, and Inputs_array is the set of all input values. Each input value is set in the template based on the corresponding data type, so we can also detect the attacks raising data-type mismatch errors.

The following example shows how the parameterized query is created for the query shown in Figure 9.

\[
\text{Qt} = \text{SELECT * FROM accounts WHERE name=?} \\
\text{AND age BETWEEN ? AND ?;}
\]

\[
\text{ps=prepareStatement(Qt);} \\
\text{ps.setString(1, 'goodUser');} \\
\text{ps.setNumberValue(2,10);} \\
\text{ps.setNumberValue(3,30);} \\
\text{ps.execute();}
\]

4. Evaluation and experimental results

In order to evaluate our approach, we developed a prototype of the proposed system (which was called APQS - Automatic Parameterizing Query System). We implemented the system in Java and ran it on a machine with 4 GB RAM and 2.3 GHz CPU processor. Moreover, we used PostgreSQL 9.3.2 as a back-end database server. The developed system is somewhat dependent on the DBMS, because it needs a parser which varies in different DBMSs. However, it can be easily improved to support other databases.

The main features of the proposed method in detection and prevention of SQLiAs in comparison with other methods are presented in Table 1. Our method is a combination of grammar-based, skeleton-based, and anomaly-based methods. The proposed method creates a profile for each query based on the grammar of the query and the key of the profile is actually the template of the query. However, the main mechanism used in its detection engine is based on the profile of the legal queries, so we consider it as an anomaly-based method.

The techniques proposed to detect and prevent SQLiAs are usually evaluated from two aspects: ac-

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Defense mechanism</th>
<th>Detection approach</th>
<th>Detection location changing</th>
<th>Changing source code</th>
<th>Preventing data type mismatch error</th>
<th>Preventing second-order attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bucher et al. [22]</td>
<td>Prevention</td>
<td>Grammar-based</td>
<td>Application</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Bandkhavari et al. [23]</td>
<td>Prevention</td>
<td>Grammar-based</td>
<td>Application</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Halfond and Orso [10]</td>
<td>Detection prevention</td>
<td>Grammar-based</td>
<td>Application</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Lee et al. [24]</td>
<td>Detection prevention</td>
<td>Query skeleton-based</td>
<td>Application/proxy</td>
<td>✓ / ×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Kar and Panigrahi [25]</td>
<td>Detection prevention</td>
<td>Query skeleton-based</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Valeur et al. [26]</td>
<td>Detection prevention</td>
<td>Anomaly-based</td>
<td>Proxy</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Proposed method</td>
<td>Detection prevention</td>
<td>Anomaly-based</td>
<td>Proxy</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
curacy rate of the detection and prevention, and the performance overhead that the approach imposes on the applications.

4.1. Accuracy

For measuring the accuracy of APQs in detection and prevention of SQLIs, we considered a vulnerable web-based application written in PHP and extracted all the normal queries (140 queries) employed in the application. The application did not validate any inputs; hence, all the inputs were vulnerable to SQLIs. We first trained the system with these 140 normal queries, created a profile for each normal query, and stored them in a hash table.

Then, we generated a complete set of test cases and measured the accuracy of the system with them. The generated test cases were categorized into 10 classes, which are shown in Table 2. For measuring FPR (detecting benign queries as malicious), we generated 140 benign queries with benign inputs (different from the queries generated in the training phase). For measuring FNR (detecting malicious queries as benign), we generated 1072 malicious queries.

Since there was no standard source for the test cases, we collected all attacks reported in the related articles, books, and websites. We also used two automated SQLI tools, namely Havij [32] and SQLMAP [33], to analyze our vulnerable Web-based application. We intercepted the URLs generated by these tools and extracted malicious inputs from them for reconstructing successful attacks driven by these tools. Overall, we generated 1072 malicious queries among which 300 were generated by employing the inputs extracted from successful attacks run by these tools.

After completing the process of learning, we sent back the benign and malicious queries in the test set to the database. The evaluation results are shown in Table 3. As it is observed, the system was able

| Table 2. Different types of SQLIs. |
| --- | --- | --- |
| **Attack type** | **Attacker’s input** | **Example** |
| Tautology | ' OR 'some thing'='some thing' -- | SELECT * FROM accounts WHERE username=' ' OR 'some thing'='something' -- ' |
| Attack using comment | admin('-- | SELECT * FROM accounts WHERE username='admin' --' AND password='123'; |
| Union attack | ' UNION SELECT 1,2,3 ; -- | SELECT * FROM accounts WHERE username=' ' UNION SELECT 1,2,3 ; --' AND password='123'; |
| Piggy-backed queries | ';DROP TABLE tablex; -- | SELECT * FROM accounts WHERE username='';DROP TABLE tablex; --' AND password='123'; |
| Attacks using groupby, order by, having clause | 'GROUP BY 1,2,3 -- | SELECT * FROM accounts WHERE username=' 'GROUP BY 1,2,3 --' AND password='123'; |
| Alternate encodings | ||chr(65)||chr(68)||chr(77) ||chr(73) ||chr(78)||chr(29) -- | SELECT * FROM accounts WHERE username=' '||chr(65)||chr(68)||chr(77)|| chr(73) ||chr(78)||chr(29) --' AND password='123'; |
| Inference attack | input1= ' OR 1 = 1 -- and input2= ' OR 1 = 0 -- | SELECT * FROM accounts WHERE username=' ' OR 1 = 1 --' AND password='123'; |
| Second-order attacks | Alice'' or username= '', admin | INSERT INTO accounts(username, password) VALUES('Alice'' or username=' ''admin', 'pass'); |
| Illegal/incorrect queries | some strings | INSERT INTO accounts(username,password) VALUES('ali','pass', 'some strings'); |
Table 3. Accuracy of detection of attacks by APQS.

<table>
<thead>
<tr>
<th>Attack type</th>
<th>No. of queries</th>
<th>Correct detection</th>
<th>FPR</th>
<th>FNR</th>
<th>Preventing attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign queries</td>
<td>140</td>
<td>140/140</td>
<td>%0</td>
<td>%0</td>
<td>✓</td>
</tr>
<tr>
<td>Tautologies</td>
<td>173</td>
<td>173/173</td>
<td>%0</td>
<td>%0</td>
<td>✓</td>
</tr>
<tr>
<td>Attack using comment</td>
<td>120</td>
<td>120/120</td>
<td>%0</td>
<td>%0</td>
<td>✓</td>
</tr>
<tr>
<td>Union attack</td>
<td>430</td>
<td>430/430</td>
<td>%0</td>
<td>%0</td>
<td>✓</td>
</tr>
<tr>
<td>Piggy-backed queries</td>
<td>30</td>
<td>30/30</td>
<td>%0</td>
<td>%0</td>
<td>✓</td>
</tr>
<tr>
<td>Attacks using groupby, orderby, having clause</td>
<td>20</td>
<td>20/20</td>
<td>%0</td>
<td>%0</td>
<td>✓</td>
</tr>
<tr>
<td>Alternate encodings</td>
<td>20</td>
<td>20/20</td>
<td>%0</td>
<td>%0</td>
<td>✓</td>
</tr>
<tr>
<td>Inference</td>
<td>199</td>
<td>199/199</td>
<td>%0</td>
<td>%0</td>
<td>✓</td>
</tr>
<tr>
<td>Second-order attacks</td>
<td>20</td>
<td>0/20</td>
<td>%0</td>
<td>%100</td>
<td>✓</td>
</tr>
<tr>
<td>Illegal/incorrect queries</td>
<td>60</td>
<td>60/60</td>
<td>%0</td>
<td>%0</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4. Detection accuracy of the method proposed by Buehrer et al. [22].

<table>
<thead>
<tr>
<th>Attack type</th>
<th>No. of queries</th>
<th>Correct detection</th>
<th>FPR</th>
<th>FNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign queries</td>
<td>140</td>
<td>140/140</td>
<td>%0</td>
<td>%0</td>
</tr>
<tr>
<td>Tautologies</td>
<td>173</td>
<td>173/173</td>
<td>%0</td>
<td>%0</td>
</tr>
<tr>
<td>Attack using comment</td>
<td>120</td>
<td>120/120</td>
<td>%0</td>
<td>%0</td>
</tr>
<tr>
<td>Union queries</td>
<td>430</td>
<td>430/430</td>
<td>%0</td>
<td>%0</td>
</tr>
<tr>
<td>Piggy-backed queries</td>
<td>30</td>
<td>30/30</td>
<td>%0</td>
<td>%0</td>
</tr>
<tr>
<td>Attacks using groupby, orderby, having clause</td>
<td>20</td>
<td>20/20</td>
<td>%0</td>
<td>%0</td>
</tr>
<tr>
<td>Alternate encodings</td>
<td>20</td>
<td>20/20</td>
<td>%0</td>
<td>%0</td>
</tr>
<tr>
<td>Inference</td>
<td>199</td>
<td>199/199</td>
<td>%0</td>
<td>%0</td>
</tr>
<tr>
<td>Second-order attacks</td>
<td>20</td>
<td>0/20</td>
<td>%0</td>
<td>%100</td>
</tr>
<tr>
<td>Illegal/incorrect queries</td>
<td>60</td>
<td>46/60</td>
<td>%0</td>
<td>%23</td>
</tr>
</tbody>
</table>

to distinguish between benign queries and all other attacks leading to change in the structure of the query.

The second-order attack did not change the structure of the query. Thus, the detection component of APQS could not detect it. However, this attack was prevented completely in the prevention phase. In the set of illegal/logically incorrect queries, seven queries caused data-type mismatch errors. As previously mentioned, this type of attack did not change the structure of the query, but in the prevention phase, it was detected and prevented completely. The overall accuracy of the detection component, using Eq. (1), was equal to 98.3% and the remaining 1.7% of the attacks were prevented to have no negative side effect on the database (in the prevention phase), although they were not detected by the detection component.

\[
AC = \frac{TP + TN}{TP + TN + FP + FN}.
\]

(1)

In the previous section, we described the process of calculating the key of a query in detail. The key was computed based on the grammar of SQL statements and used for detecting attacks. To compare the detection accuracy of our method with methods using the grammar of SQL statements, we tested and compared our proposed method with the method proposed by Buehrer et al. [22] using the same train and test queries. Their method compares parse trees of each query before and after entering user inputs at runtime. If they match, no attack is detected. The results of this test are shown in Table 4. The results confirm that the method by Buehrer et al. cannot detect or prevent the second-order attacks and the attacks raising data-type mismatch errors. Also, its precision is 97.1%, which is less than the accuracy of our proposed method.

4.2. Performance overhead

Performance overhead, which was imposed by the proposed method on an application, could be measured in terms of different parameters in the detection and prevention stages. In fact, we evaluated the system using two different scenarios. The measures considered for evaluating the system in these two scenarios are as follows:

- \(d_{detect}\): Delay of detection stage;
- \(d_{prevent}\): Delay of prevention stage;
• \(d_{pexec}\): Parameterized query execution time;
• \(d_{exec}\): Normal query execution time.

To measure the performance overhead, we need to send a set of benign or malicious queries to the database. The execution time of an SQL command varies in different runs and depends on several factors such as the speed of the processor, the complexity of the command, the number of rows processed by the command, database load at the command execution time, and the plan selected by the optimizer. Thus, in some circumstances, we need to run a command multiple times and consider the average time as the query execution time. The query execution time also depends on the size of the value stored in the database. Thus, we created tables similar to those in our test application and inserted 1000 records in each of them.

4.2.1. First scenario: sending a query containing malicious input to the database

By sending a malicious query to the database, if the system can detect the attack, the time overhead (or \(\text{delay}\)) added to the application is shown as \(d_{detect}\). For detecting an attack, the system only needs to calculate the key and search it in the hash table; thus, the detection time is much less than the query execution time.

\[\text{delay} = d_{detect} \ll d_{exec}\]

To obtain the detection time, we sent 1000 randomly selected malicious queries to the database and computed the average detection time for each of them (each query was sent to the database 100 times). The average detection time for each query and the total average detection time are shown in Figure 10. As it is shown, the average detection time is trivial and equal to 0.667 milliseconds.

It seems that the detection time depends on the number of tokens existing in the query. To validate this hypothesis, we measured the detection time based on the number of tokens in the query. We sorted the queries based on the number of tokens in them (from 7 to 72 tokens) and calculated the average detection time for each of them. The detection time with respect to the number of the tokens existing in the query.

![Figure 10](image.png)

**Figure 10.** Attack detection time in different test queries.

\[d_{detect} = d_{pexec} + d_{prevent} + d_{pexec}\]

If the query is sent to the database for the first time (during a session), the parameterized query execution time becomes greater than the normal query execution time, i.e., \(d_{delay} > d_{exec}\). If the query is sent to the database for the second time or more, the value of \(d_{pexec}\) is usually much less than \(d_{exec}\). Since execution time of parameterized queries decreases significantly after building the parse tree and query plan (for the first time), and the overhead of detecting and preventing attacks in the proposed approach is very low, it is expected that the overhead added by the system be less than the query execution time \(d_{exec}\), i.e., \(d_{delay} < d_{exec}\) in this situation. In this condition, the system not only does not add any overhead to the application, but also has a positive impact on the application performance.

![Figure 11](image.png)

**Figure 11.** Attack detection time with respect to the number of the tokens existing in the query.

To evaluate the impact of the system on the application performance, we sent a legal query to the database 5 times and measured the query execution time with and without APQS. The query execution
times in these two cases are shown in Figure 12. As it is observed, when the query is sent to the database for the first time, the query execution time is more than the normal query execution time. By sending the query for the second time or more, the overhead becomes less than the normal query execution time and 34.7% improvement can be seen in this case.

To measure the average prevention time, we measured the average time for executing 100 legal queries. Table 5 represents the average times required to execute the query (for the first time) using APQS and without using it. The overhead of using our system is equal to 2.442 ms. Note that if we run the queries for the second time or more, the execution time would be less than the values represented in this table in case of using APQS (similar to the one depicted in Figure 12).

In the prevention phase, one of the main factors affecting the attack prevention time (in creating and executing the parameterized queries) is the number of input values in the query. To determine the variation of the query execution time with the number of inputs in the template of the query, we considered a simple query and then, increased the number of entries in it from 1 to 10. Each of these queries was sent to the database 100 times and the average execution time was calculated for each of them. The obtained results are shown in Figure 13. As it was expected, by increasing the number of inputs in the query, the execution time of the parameterized query increased too.

In order to compare the performance of our

method with similar methods, there is a trade-off between the performance and accuracy of the method in detecting attacks and using the opportunity of preventing unknown attacks. As shown in Table 1, and to the best of our knowledge, the existing methods can neither detect attacks raising data-type mismatch errors nor prevent the second-order attacks. In order to make these improvements, the overhead of the proposed method is somewhat higher than those of the other similar methods.

Nevertheless, as observed in Table 5, the performance overhead of the method is insignificant and in the prevention phase, in some conditions (when a query is executed more than one time), it helps to improve the overall performance of the system. In addition, it should be noted that our code has not been implemented in the most efficient mode. Certainly, efficient implementation will reduce this overhead.

\[ \text{Table 5. Overall query execution times with and without APQS.} \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average attack detection time using APQS</td>
<td>0.667 ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average query execution time using APQS</td>
<td>2.442 ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average query execution time without APQS</td>
<td>2.233 ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{Figure 12. Comparing the query execution times with and without APQS.} \]

\[ \text{Figure 13. Required time to execute the parameterized queries with respect to the number of inputs in them.} \]
5. Conclusion

Most of the existing methods for detection or prevention of SQLiAs are not able to counteract all types of SQLiAs (attacks described in Table 2), such as second-order attacks and the attacks that try to raise data-type mismatch errors.

In this paper, we have proposed a method for detecting and preventing SQLiAs. In fact, SQLiAs can be classified into the attacks that change the structure of normal queries and the attacks that do not change the structure (like second-order and mismatch-type error attacks). Our proposed method could detect all attacks changing the syntactical structure of the queries. For this purpose, in the learning phase (in which only the benign inputs were used in the application), the structure of normal queries and auxiliary information (such as the data type of the input) were extracted and profiled in the system. In the detection phase, the malicious queries with different structures were identified by comparing them with the existing profiles of normal queries. For the second category of SQLiAs, in which the malicious inputs did not modify the structure of the parse tree of a normal query, the described solution was not efficient. For this type of attacks, we proposed a prevention method without the need to detect the attack. To this aim, each query was automatically converted to a parameterized query using the information existing in the profile of the corresponding normal query. In this situation, the malicious inputs could not affect the normal behavior of the application, because in executing the parameterized queries, the parse tree of the query was constructed without considering the values of its parameters.

The main contribution of this paper is proposing an automated method to create parameterized queries dynamically by semantically analyzing the raw queries sent from a vulnerable application. By creating parameterized queries, we could prevent second-order and unknown attacks, and detect/prevent attacks causing data-type mismatch errors without the need to execute the query. In the proposed method, we did not need to change the application source code. Evaluation of the proposed approach in practice showed that the performance overhead imposed by this method was minimal and in some cases, it even led to improvement in the performance of the application by parameterizing the queries.

References


Biographies

Somayeh Dolatnezhad Samarin received her MSc degree in IT Engineering from Sharif University of Technology, Tehran, Iran. Currently, she is PhD candidate in the Department of Computer Engineering at Sharif University of Technology, Tehran, Iran, doing research on computational integrity of outsourced computations. Her research interests are database security, cloud computing security, secure delegation of computation, and cryptography.

Morteza Amini received his PhD in Software Engineering (with specialty in Data Security) from Sharif University of Technology. He is currently an Assistant Professor in the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. He is one of the leaders of Data & Network Security Lab (DNSL) in this department. His research interests include database security, access and information flow control, intrusion detection and alert correlation, and mobile devices security.