CHAPTER 5

INTELLIGENT TECHNIQUES TO PREVENT
SQL INJECTION ATTACKS

5.1 INTRODUCTION

In this research work, two new techniques have been proposed for addressing the problem of SQL injection attacks, one to identify the SQL injection attacks and the other to prevent such attacks. In detection, this work attempts to identify the attacks at query execution points by classifying the queries either at the query execution points or at call to query function calls where the past query function calls are compared. In prevention, the user profiles are classified for based on their past history for preventing SQL injection attacks by denying the access credentials for those who are classified as the un-ethical and are reported by the decision manager. The users are classified into three categories namely naïve users, privileged users and application programmers. Different access levels are given to these users based on the user type.

5.2 SQL INJECTION ATTACKS

In many web applications, SQL injection attacks are becoming serious threat to the integrity and security of web database systems. These types of attacks are vulnerable since the queries provided by the users are modified by the unethical users and hence they make the systems to execute their own query structures. Therefore, it often requires careful and vigilant
coding by the application programmers while writing SQL queries since the application programmer has to validate the user input queries. However, this type of validation becomes tedious for large projects where many users write the program code and relevant queries for the same application. In order to detect SQL injection attacks, this research work proposes an intelligent dynamic query intent evaluation technique to learn and predict the intent of the SQL queries provided by users and to compare the identified query structure with the query structure which has been generated with user input so that it can automatically detect possible attacks which are attempted by unethical users.

This type of evaluation is helpful in reducing the need for the user to have more consciousness when SQL queries are written. This work provides automatic access control techniques by implementing intelligent validation techniques in order to minimize user intervention. The main advantage of this system is that uses a newly proposed modified decision tree classification algorithm which has been developed by enhancing the decision rules with temporal rules to find the unethical users who try to provide malicious queries intelligently at the query execution points. In such a scenario, the decision manager informs the database manager of the system about the new possible query execution points with intent for attacks, and thereby preventing the SQL injection attacks. For example, if the user types the user name as ‘hanuman’ and if there is a condition checking construct which checks the queries of the form 1=1…, then the query pattern would remain the same but the user intended query is not executed and hence this clearly is an attack. This type of attacks have been detected and prevented effectively in this work using rule chaining where the execution of one rule triggers another rule and so on until a complete decision is obtained.
In this proposed model, besides looking for direct query execution points, it also looks for the user function call query points by providing explanation based learning for predicting intentions. Thus the web pages need not be recoded to suit the method. The identification of attack is carried out by applying intelligent dynamic intent evaluation technique which compares the evaluated intent with the query structure after user input and classification. Thus any changes in query structures are classified as abnormal and the attack is prevented. For the purpose of prevention, the users are classified into ethical and un-ethical users using their past queries and profiles by applying the modified decision tree classification algorithm where the patterns are stored in the knowledgebase.

After obtaining the classification results, only the ethical users are allowed to provide the next query and their queries are validated further by using syntax and semantic analysis. On the other hand, unethical users are monitored at query execution points as well as at the function calls and they are prevented if they are classified as abnormal. After classification, the user queries are compared with a threshold for checking his trust and the same user’s past queries are also analyzed in order to perform effective prevention. The main advantage of this proposed work is that it allows only ethical user queries directly and prevents SQL injection attacks attempted by the un-ethical users.

5.3 ARCHITECTURE FOR DETECTING AND PREVENTING SQL INJECTION ATTACKS

The architecture for the proposed subsystem that detects and prevents SQL injection attacks is shown in Figure 5.1. This subsystem consists of twelve of major components namely Application Interface, SQL Interface, Classification Module, Static Query Transform, Dynamic Query Transform, Decision Manager, Knowledgebase, Intention Detection and Validation, Intelligent Prevention Module, Data Base Management System and Physical Database.
Moreover, this subsystem has been implemented as a two phases. In phase one called static query transform, the web application’s query generation points are identified and detection statements are introduced for monitoring and capturing the attacks based on decision tree classification. In phase two called dynamic query transform, the intention of the queries are identified using prediction techniques based on temporal rules and by analyzing the past and current usage by comparing them with trusted intents using intelligent rules. If any mismatch is identified in the structure, the injections on such queries are reported.

Figure 5.1 Architecture for Detecting and Preventing SQL Injection Attacks

The application programmer can send query requests either by running the application program using the application interface which consists of a function list containing the program components and query components or can send SQL queries using the SQL interface. The queries obtained from these interfaces are sent to the decision manager where it uses a knowledge base that consists of previous queries for all registered users and the necessary rules for decision making.
The classification module performs classification using decision tree algorithm and the queries are sent to query transformation module based on the classification results. The classification directs the queries to undergo either a static query transform or a dynamic query transform. The results obtained from the query transformation modules are then sent to the intention detection and validation a module which uses temporal rules to predict the intentions. These queries with intentions are sent to the intelligent prevention module which prevents the SQL injection attacks and allows the legitimate queries to the database manager. The data base manager executes only the legitimate queries.

5.4 ALGORITHM USED TO DETECT ATTACKS

The algorithm used for detecting SQL injection attacks uses Modified Decision Tree and validation checks. The steps of the algorithm are given below.

**Input:** SQL Queries  
**Output:** No of SQL injection attacks captured  
Algorithm detect_sql_injattack (String SQL query, Int no. of attacks captured)

1. **Step 1:** Get Query from user  
2. **Step 2:** Compute the decision parameters.  
3. **Step 3:** Train the system using the modified decision tree  
4. **Step 4:** Call tokenizer and tokenize the query  
5. **Step 5:** Perform SQL validation using validation rules  
6. **Step 6:** Classify external function calls and query execution points  
7. **Step 7:** Classify the query as normal or abnormal and execute the legitimate queries  
8. **Step 8:** Prevent abnormal user queries and send normal user queries to the database engine.
5.5 ALGORITHM USED TO PREVENT ATTACKS

The algorithm used for preventing SQL injection attacks and the steps of the algorithm are given below.

**Input:** User Queries

**Output:** Classified as normal and abnormal

**Step 1:** Collect the user queries with attack intentions

**Step 2:** Collect the past queries of such users

**Step 3:** Find the trust of past queries (attack intended or normal) using trust rules.

**Step 4:** Apply rules to compare these queries with normal queries

**Step 5:** If variation exceeds above the level of threshold, prevent them.

5.6 IMPLEMENTATION DETAILS

This architectural framework introduces a decision manager which uses a knowledge based approach to classify the intents of the queries and compares this identified query structure with the query structure the query execution that has been generated by the user input. This is helpful to detect attacks by unethical users automatically since the classification of the query into static and dynamic provides an effective decision methodology. In this process, first rules are used to identify the query execution points or the call to the query execution function.

The intention detector and validator uses inference rules to identify the intent of queries dynamically by dynamic query transform techniques. The dynamic query transform transforms queries containing user text (string or character) into an equivalent string of a’s of same length and user defined
numeric’s to positive integers. After transforming, it prevents all possible injections using query rewrite technique. However, it maintains the query intent in the knowledge base by adding new rules. This dynamically discovered intent are used to evaluate the query with user input. If the intents vary, then the attack is identified and thus can be prevented.

5.6.1 Classification Module

The classification module has two main components namely the static query transformer and dynamic query transformer. The static query transformer scans the file to be protected against SQL injection attacks and finds the query execution points or the calls to the functions containing them and performs feature selection for evaluating the intended query pattern. This basically finds the query strings or the features containing query strings and adds a new transformed query string before them.

For example,

```java
String query = "SELECT * FROM admin WHERE username = "" + userdetails + " " AND password = "" + password + "";
```

The static query transformer creates a new query string called `query_t` as follows:

```java
String query_t = "SELECT * FROM admin WHERE username = "" + Transformer.transform(userdetails) + " " AND password = "" + Transformer.transform(password) + "";
```

The dynamic query transform uses the methods of transformer class that converts the string to a string of ‘a’s of same length and numbers to 1.
The SQL Intention Detector and Validator checks the discovered intents with the intent of the query after user inputs are given. If both are matching, then the query is allowed to execute otherwise exception is thrown.

This subsystem has been tested with Apache server, MySQL database and Java. The system was deployed and implemented with five real world applications namely Portal, Book Store, Classifieds, Employee Directory and Event Manager and was tested with all possible types of SQL injection attacks. All these applications were considered for testing since they are known to be SQL attack prone.

5.6.2 SQL Intention Detector and Validator Module

The SQL Intention Detector and Validator module identifies the attacks by performing classification using decision tree algorithm and then by comparing the dynamically discovered intent with the intent of the query on user input to check whether the query falls within the given threshold. This matching ensures that the user input is same as intent discovery or not.

The Intention Detector and Validator defines the attacks as follows:

- **SQLKEYWORDS** \(\rightarrow\) The keywords such as SELECT, INSERT, FROM, WHERE that are used to form the SQL queries along with some user data.
- **SQLCOMMENT** \(\rightarrow\) Anything that follows the -- including the -- and it can be NULL.
- **SQLINTENT** \(\rightarrow\) Made of one or more SQLKEYWORDS and one SQLCOMMENT
- **SQLATTACK** \(\rightarrow\) SQLINTENT user query is compared with Transformed SQLINTENT user query.
5.6.3 Intelligent Intent Prevention on External Function Calls

In this work, an external function called LoadQueryObj which takes a query and loads the members of the class with the query results has been proposed and implemented. The structure of this function is as follows:

```java
Class QueryObj {
    Int attribute1;
    float attribute2;
    ResultSet rs;
    ....
    .
    .
    Void LoadQueryObj(String query) {
        .
        Validate(query)
        rs = st.executeQuery(query)
        // load the members with tuple values
    }
    Void loadNexttuple() {
        // load members with next tuple
    }
}
```

This sub system first validates the input by calling the validation functions and then detects the query execution call points by executing the statement st.executeQuery() in the LoadQueryObj method and detects the intents dynamically from the parameters sent in the query to the called
methods. Hence, this subsystem detects and prevents the SQL injection attacks.

To illustrate, how this attack is prevented in external function calls is explained below:

Consider the query string

```java
String query = "SELECT * FROM employee WHERE username =" + userinput1 + " AND password =" + userinput2 + "";
```

and the query execution function

```java
TupleObjO.LoadtupleObj(query);
```

In this process, when the method is called, the whole query is passed as parameter to this function. This system tracks the user input inside the external function by executing intelligent rules.

### 5.6.4 Lexical Analyser

In this work, a new lexical analyser has been used for performing lexical analysis. The lexical analyser carries out simple regular expression string matching to identify sinks and others. There are some other techniques that are capable of identifying the sinks in the literature, all the existing works directly pass tainted data (un-trusted) as parameters to sinks. On the other hand, this proposed method combines both string matching and trust comparison and hence this proposed method has the ability to identify sink sources.

There are many limitations that can be found on the existing systems. All these existing techniques rely on some strategies. For example, they are relying heavily on grep like syntax searches and regular expressions
to locate the use of dangerous functions and in some cases which is highlighting the code that incorporates dynamic SQL-string building techniques. In some cases, these string matching techniques are incapable of accurately mapping the data flows of next execution paths which leads to false positives, and some other methods are not capable of making distinctions between comments in code and actual sinks.

Therefore, it is necessary to distinguish between queries with and without objects. In this work, the lexical analyser has been implemented by avoiding the above limitations. This lexical analyser converts a sequence of characters into a sequence of tokens using rules. Moreover, this lexical analyser is capable of interpreting normal SQL statements and other SQL statements. Therefore, for a given SQL statement, this SQL lexical analyser generates a set of tokens with the corresponding token types and provides suitable annotations which are used later to detect attackers easily.

For example, for the SQL query

```
UPDATE Books
SET BOOK_NAME = 'SQLPRIMER' and PRICE = 1000
WHERE BOOK_ID = 1234,
```

the SQL lexical analyser generates the following set of tokens and the corresponding token types as shown in Table 5.1.
Table 5.1 Tokens obtained from Lexical Analysis

<table>
<thead>
<tr>
<th>No.</th>
<th>Token</th>
<th>Token Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>UPDATE</td>
<td>[IDENTIFIER]</td>
<td>SQL Keyword</td>
</tr>
<tr>
<td>2.</td>
<td>Books</td>
<td>[IDENTIFIER]</td>
<td>Table-Name</td>
</tr>
<tr>
<td>3.</td>
<td>SET</td>
<td>[IDENTIFIER]</td>
<td>SQL Keyword</td>
</tr>
<tr>
<td>4.</td>
<td>BOOK_NAME</td>
<td>[IDENTIFIER]</td>
<td>Attribute_Name (user defined)</td>
</tr>
<tr>
<td>5.</td>
<td>=</td>
<td>[OPERATOR - EQUALS]</td>
<td>Comparison operator</td>
</tr>
<tr>
<td>6.</td>
<td>'SQLPRIMER'</td>
<td>[LITERAL - STRING]</td>
<td>Value provided in the query for checking</td>
</tr>
<tr>
<td>7.</td>
<td>,</td>
<td>[COMMA]</td>
<td>Separator in SQL syntax</td>
</tr>
<tr>
<td>8.</td>
<td>PRICE</td>
<td>[IDENTIFIER]</td>
<td>Attribute name</td>
</tr>
<tr>
<td>9.</td>
<td>1000</td>
<td>[LITERAL - INTEGER]</td>
<td>User provided attribute value</td>
</tr>
<tr>
<td>10.</td>
<td>WHERE</td>
<td>[IDENTIFIER]</td>
<td>SQL Keyword</td>
</tr>
<tr>
<td>11.</td>
<td>BOOK_ID</td>
<td>[IDENTIFIER]</td>
<td>Attribute_name, Primary key</td>
</tr>
<tr>
<td>12.</td>
<td>1234</td>
<td>[LITERAL - INTEGER]</td>
<td>Attribute Value</td>
</tr>
</tbody>
</table>

The SQL lexical analyser is used by the Intention Detector and Validation module to find a set of literal types in the intercepted SQL statement, such as LITERAL - STRING in row 6 and LITERAL - INTEGER in rows 9 and 12 of Table 5.1.

In this way, the lexical analyser accepts the SQL commands given by the user in valid positions which are in the given string from left to right positions and forms the SQL structure required for the comparison of queries.
The tokenizer splits the query string into words and searches for the SQL commands from left to right in parts that are not part of SQL data.

The lexical analyser checks the data dictionary for validating the table_name and the user defined attributes present in the table for effective intent detection.

For Example, consider the query string,

String query = “SELECT details FROM useraccount WHERE username = ” + username + “” AND password =”” + password + “””

If user input for username is 'hanuman' and password is 'passwd' then in dynamic intent discovery, the transformed inputs are 'aaaaaaa' for user name and 'aaaaaa' for password and hence thus the query string for intent discovery would be

“SELECT details FROM useraccount WHERE username='aaaaaaa'
AND password='aaaaaa'

From this, the structure is identified as SELECT, FROM, WHERE with AND option. Here the comments are also taken care. Since the user input maintains only the intent structure, queries without comments are executed successfully. On the other hand, if the query contains a comment, the query execution fails.

For example, If user input for username is  'hanuman' AND 1=1 --, then the query becomes as follows when the comment is considered.

SELECT details
FROM useraccount
WHERE username = 'hanuman' AND 1=1 -- ’ AND
password ="""".
The structure of the query would remain same even if the comment is not considered. Hence, the SQL keywords SELECT, FROM, WHERE and AND, are used for attack. However, they are missed when the structure is only considered without comment part. If the comment part is considered as a unit, the use of these keywords would disagree with the intent identified and hence the SQL attack will be prevented successfully.

5.7 EXPERIMENTAL RESULTS

In this work, five real world applications namely Portal, Bookstore, Classifieds, Employee Directory and Event Manger have been considered for carrying out the experiments. These applications are known to be SQL injection attack vulnerable. The applications have been deployed on glassfish server with MySQL as database. The queries are read from the list of queries containing sets of both legitimate and SQLIA queries and the request is provided to application using the ‘wget’ command. The result from the queries are classified either as success, attack caught, not caught, syntax error, error and false positives. The values obtained from the experiments have been tabulated in Table 5.2.

Table 5.2 Attacks Caught and SQL Exceptions Found

<table>
<thead>
<tr>
<th>Application</th>
<th>False  Positives</th>
<th>Attacks Caught</th>
<th>SQL Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portal</td>
<td>1</td>
<td>2751</td>
<td>624</td>
</tr>
<tr>
<td>Book Store</td>
<td>2</td>
<td>2841</td>
<td>216</td>
</tr>
<tr>
<td>Classifieds</td>
<td>1</td>
<td>3111</td>
<td>250</td>
</tr>
<tr>
<td>Employee Directory</td>
<td>2</td>
<td>3872</td>
<td>159</td>
</tr>
<tr>
<td>Events</td>
<td>1</td>
<td>3137</td>
<td>367</td>
</tr>
</tbody>
</table>
Figure 5.2 shows the graph for the number of SQL exceptions caught with respect to the number of queries sent for employee directory application that have been tested in this work. Moreover, when compared with previous work, this implemented method proves to be more effective in detecting and preventing SQL injection attacks and in catching SQL exceptions.

![Graph showing the number of SQL exceptions caught](image)

**Figure 5.2 Number of SQL Exceptions Captured for Employee Directory**

Figure 5.3 shows the graph for the detection and prevention of SQL injection attacks implemented in this work. From this graph, it can be seen that this proposed work proves to be more effective in detecting the SQL injections for the five applications tested in this research work.
Figure 5.3 Percentage of Attacks Detected in Various Applications

Figure 5.4 shows the graph for the SQL exceptions caught in portal application for the number queries that has been sent for testing is method.

Figure 5.4 Number of SQL Exceptions Captured for Portal
The transformed webpage was tested using the query set developed for Amnesia (William G.J. Halfond and Alessandro Orso 2005). The requests to the webpage were sent using the ‘wget’ command. The output was recorded and the following values were obtained. From these values, it has been observed that the proposed method has only negligible number of false positives. Moreover, this proposed method caught all the attacks with considerable performance.

Moreover, these attacks were evaluated using the queries already prepared for these applications on the tests of previous methods namely SQLIA, and CANDID. The queries have been prepared in such a way that they are exhaustive and cover all types of possible attacks. The evaluation was also based on the false positive detected which is almost null in all the cases for our method.

![Figure 5.5 Attacks Caught, SQL Exceptions, False Positive for Applications](image-url)
Therefore, this proposed method is successful in detecting all the SQL injection attacks and the SQL Exceptions which are generated by malformed attacks.

5.8 SUMMARY

In this work, an intelligent approach for capturing and handling SQL exceptions has been proposed and implemented. Five applications namely Portal, Book Store, Classifieds, Employee Directory and Event Manager have been considered in the experiments. The results obtained from these experiments show that this system significantly improves the detection accuracy in comparison with the earlier systems. Moreover, this system handles exceptions very well in order to perform effective prevention in addition to the reduction of false positives.