A Detection Method for SQL Injection Attacks in Web Applications

WITT YI WIN, HNIN HNIN HTUN

Abstract: SQL injection attack is a particularly dangerous threat that exploits application layer vulnerabilities inherent in web applications. Instead of attacking instances such as web servers or operating systems, the purpose of SQL injection is to attack RDBMSs, running as back-end systems to web servers, through web applications. SQL injection is a type of attack in which the attacker adds SQL code to an input box to gain access or make changes to data. This paper proposes a simple and effective detection method for SQL injection attacks. By using this method, we can detect input validation attacks against web applications in an efficient way. Experimental results show that the proposed detection method is efficient compared to other methods.

Keywords: World Wide Web security, SQL Injection, static analysis, runtime validation, Database Security.

I. INTRODUCTION

As more and more services are provided via the World Wide Web, efforts from both academia and industry are striving to create technologies and standards that meet the sophisticated requirements of today’s Web applications and users. With the aid of different Web applications, billions of transactions are done online. Though hundreds of people use these applications, the security level is weak in many cases, which makes them vulnerable to get compromised. A less secure Web application can cause lots of damages and unauthorized users theft of trusted users’ sensitive data. Sometimes, the attacker may gain full control over the Web application and can damage or destroy the system [1].

Currently, the SQL injection attacks (SQLIAs) is one of the most dangerous and common threats to databases and Web applications. Attackers can gain direct access to the underlying databases of a web application and the power to leak, modify, or even delete information that is stored on them by SQLIAs. Insufficient input validation is the root cause of SQLIAs. An attacker can submit input with embedded SQL commands directly to the database by leveraging invalid input validation. This kind of attack represents a serious threat to any web application that reads inputs from the user and makes SQL queries to an underlying database using these inputs. Most web applications of the corporations and government agencies work like this way and could be vulnerable to SQL injection attacks [2].

Defensive coding has been offered as a solution for preventing the SQLIAs but it is very difficult. Although developers try to put some controls in their source code, attackers continue to bring some new ways to bypass these controls. It is difficult to keep developers up to date, according the last and the best defensive coding practices. On the other hand, implementing of defensive coding is very difficult and need to special skills and also erring. These problems motivate the need for a solution to prevent and detect SQL injection attacks [3].

II. TYPES OF SQL INJECTION ATTACKS

SQL Injection Attacks refer to a class of code-injection attacks in which data provided by user is included in the SQL query and part of the user’s input is treated as SQL code. In this section, we present and discuss the different kinds of SQLIAs known to date. Depending on the specific goals of the attacker, the different types of attacks are generally not performed in isolation; many of them are used together or sequentially.

A. Tautologies

A SQL tautology is a statement that is always true. The most common usages are to bypass authentication pages and extract data such as query 1.

Query 1:
SELECT * FROM employee WHERE name = ‘ ’ or 1=1 --
AND password = ‘12345’;

The single quote (‘) symbol indicates the end of string and two dashes comment the following text of the query. Boolean expression 1=1 is always true. As a result, the user will be logged on as the first user stored in the table.

B. Logically Incorrect Queries

This attack lets an attacker gather important information about the type and structure of the back-end database of a Web application. The attack is considered a preliminary, information gathering step for other attacks. Query 2 shows an example.

Query 2:
SELECT * FROM employee WHERE name = ‘ ’ UNION
SELECT SUM(username) from users -- ’ and password= ’ “;
The query tries to execute the column username from users table and it tries to convert the username column into integer, which is not a valid type conversion. Hence, the database server returns an error message which contains name of the database and information of the column field.

C. Union Queries
This attack uses the “UNION” operator, which performs union between two or more SQL queries. As a result of this attack, database returns a dataset which is union of the results of original query and the injected query. An example is shown in query 3.

Query 3:
SELECT emp_id FROM employee WHERE name = ‘’
UNION SELECT cardNo FROM creditCard WHERE accNo = 10032 -- AND password = ‘’ ;

The original first query returns the null set, whereas the second query returns data from the “creditCard” table.

D. PiggyBacked Queries
In this case, attackers are not trying to modify the original intended query; instead, they are trying to include new and distinct queries that “piggy-back” on the original query. This type of attack can be extremely harmful such as query 4.

Query 4:
SELECT * FROM employee WHERE name = ‘guest’ and password = ‘123’; DROP TABLE employee; --;

After completing the first query, the database would recognize the query delimiter (“;”) and execute the injected second query. The result of executing the second query would be to drop the employee table.

E. Stored Procedures
DBMS has provided stored procedures method with which a user can store his own function that can be used as needed. To use the function, a collection of SQL queries is included. Query 5 shows an example.

Query 5:
CREATE PROCEDURE DBO @userName varchar2, @pass varchar2, AS EXEC (“SELECT * FROM user WHERE id= ‘”
“+@userName+’’ and password= ‘”+@pass+’’”); GO

This scheme is also vulnerable to attacks such as piggy-backed queries.

III. RELATED WORK
Over the past decade, a lot of work has been accomplished by the research community in providing new techniques to detect and prevent SQLIAs. In this section, we discuss state-of-the-art in SQLIA detection and prevention techniques. Dharam et al. in [4] proposed a Runtime Monitoring framework that is used in the development of runtime monitors. The framework uses two pre-deployment testing techniques, such as basis-path and data-flow to identify a minimal set of all legal/valid execution paths of the application. Runtime monitors are then developed and integrated to perform runtime monitoring of the application, during its post-deployment for the identified valid/legal execution paths. The results of their study show that runtime monitor developed for the application was successfully able to detect all the tautology based attacks without generating any false positives. The important limitation of the proposed technique is that it can detect only tautology based SQLIAs.

Frankl et al. in [5] introduced a tool named ASSIST (Automatic and Static SQL Injection Sanitization Tool) for protecting Java-based web applications which could come from applications developed as JSPs or Servlets. This paper presents the technique of automatic query sanitization to automatically remove SQL injection vulnerabilities in code. By using a combination of static analysis and program transformation, this technique automatically identifies the locations of SQL injection vulnerabilities in code and instruments these areas with calls to sanitization functions. This automated technique can be used to relieve developers from the error-prone process of manual inspection and sanitization of code. But, there are several sources of imprecision which may lead to false positives and false negatives in this technique.

Kadirvelu et al. in [6] introduced 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 in order to detect possible attacks by unethical users automatically. This work has been fully automated by implementing intelligent validation techniques in order to minimize user intervention. The main advantage of this system is that it applies the decision tree classification algorithm which is enhanced with temporal rules to find the unethical users intelligently at the query execution points where the database manager of the system can be informed of the new possible query execution points with an intent for attacks, and thereby preventing the SQL injection attacks.

Jeong et al. in [7] proposed a very simple and effective detection method for SQL injection attacks. This method uses combined static and dynamic analysis. This method removes the value of an SQL query attribute of web pages when parameters are submitted and then compares it with a predetermined one. The comparison is performed with only one pass exclusively OR operation for each character. The operation of this system is very simple and the complexity time is a constant.

Hidhaya et al. in [8] proposed an Intrusion Protection method to detect the SQL injection. This method uses a Reverse proxy and MD5 algorithm to check out SQL injection in user input. Using grammar expressions rules to check for SQL injection in URL’s. It does not do any changes in the source code of the application. The detection and mitigation of the attack is fully automated. The sanitizing application is placed in the Reverse proxy server between the client and the server. By increasing the number of proxy servers the web application can handle any number of
A Detection Method for SQL Injection Attacks in Web Applications

Sharma et al. in [9] introduced an effective detection method RDUD for SQL injection attack which is an enhanced version of DUD [10]. The method comprises a supervised machine learning approach using a Support Vector Machine (SVM) to learn and to classify a query at runtime. Legitimate web profile and attack web profile are generated for each of the web-application software which consists of a set of production rules extracted from the dynamic SQL queries. Both the web profiles are generated during training phase. At runtime a dynamic SQL query is matched with each of the web profile and accordingly it classify based on the matching distance. RDUD is independent of the developer’s initialization of syntax rules, valid trusted string database, static or pre-generated program code checking, etc.

IV. PROPOSED METHOD

A new method is proposed in this paper which deals with SQL Injection Attacks, caused via Vulnerable Web Applications. The proposed system has two modules: Input Checker and Query Checker. HTTP request coming from the client side instead of going to the web server is feed to both modules one by one. Any malicious input is found in the HTTP request by either of the module, the request is considered as an attack and its execution is prevented on the web server. The block diagram of the proposed system is shown in figure 1. Firstly, Input Checker checks the request and if the malicious input is found in the input query, it is rejected and not forwarded to the Query Checker. Only input query which contains valid input is forwarded to the next module. Query Checker detects the input query by comparing with the legitimate queries before forwarding to the server. The legitimate query goes to both modules before getting executed on the web server.

Fig.1. Block Diagram of the Proposed System

A. Input Checker

Input Checker checks the user input in the runtime query (user input embedded). It has an attack repository consisting of some malicious characters which are often used in writing malicious code for SQL Injection attacks. Input Checker first separates the user input from the runtime query. Then, it compares the user supplied data in the runtime query with the characters stored in the attack repository. If the user input contains any malicious characters which are present in the attack repository, it is marked as invalid input and the request is prevented to be executed on the web server. If the user input does not contain any special characters then the runtime query is forwarded to the Query Checker.

B. Query Checker

Query Checker compares the runtime query pattern with the legitimate query patterns to detect SQL Injection attacks. It has a Query Table consisting of legitimate query patterns. The system finds the legitimate queries using a tool, Java String Analyzer (JSA) at offline. The tool takes the .class file to analyse, which we call the application classes. In the application classes, one or more string expressions are selected as hotspots. A hotspot is defined as a point in the application code that issues SQL queries to the underlying database. In our analysis, we identify all the “execute” methods of the Statement class in Java as hotspots. The result of the analysis is NDFA that expresses, at the character level, all the possible values the considered string can assume at the hotspot. We perform a depth first traversal of each hotspot’s NDFA and group characters to produce legitimate query (LQ). Then the system converts the obtained legitimate queries (LQs) into legitimate query patterns (LQPs) using Token Table. SQL keywords, Logical Operators and Relational Operators are initially stored in Token Table. The string that is not included in the Token Table is marked as new Token. The following examples show the legitimate queries and their patterns.

LQ1: SELECT * FROM employee WHERE name = ‘var’ and password = ‘var’;
LQP1: S K1 K2 A1 K3 A2 R1 V L1 A3 R1 V
LQ2: SELECT name, title, department , email FROM employee WHERE name like ‘%var%’;
LQP2: S A1 A2 A3 A4 K5 A1 K3 A2 K4 V
LQ3: UPDATE employee SET password = ‘var’ where name = ‘var’;
LQP3: U A1 K3 A1 R1 V K1 A2 R1 V

The obtained legitimate query patterns (LQPs) are stored in Query Table. The query operation type of the legitimate query is the first character of its query pattern. For example, the query operation of the legitimate query 1 is ‘SELECT’ operation, so the first character of its query pattern is ‘S’. We extract the first char of the legitimate query pattern and store it in Query Table as a field shown in Table 1.

<table>
<thead>
<tr>
<th>Query Operation</th>
<th>Legitimate Query Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S K1 K2 A1 K3 A2 R1 V L1 A3 R1 V</td>
</tr>
<tr>
<td>S</td>
<td>S A1 A2 A3 A4 K5 A1 K3 A2 K4 V</td>
</tr>
<tr>
<td>U</td>
<td>U A1 K3 A1 R1 V K1 A2 R1 V</td>
</tr>
</tbody>
</table>

At runtime, Query Checker tokenizes the runtime query (RQ) and then converts into runtime query pattern (RQP) using Token Table. It compares the runtime query pattern with the legitimate query patterns according to the query operation. For example, if the runtime query’s operation is update (U), Query Checker uses only legitimate update (U)
queries for comparison. If the runtime query pattern does not match with one of the legitimate query patterns, the runtime query is an attack query. Otherwise, the runtime query is the legitimate query. Once the attack query is detected, the system blocks the query and alerts to the user. If the runtime query is the legitimate query, the system allows the query to be executed on the database.

The following are the examples applying proposed method.

**RQ1:** SELECT name, title, department, and email FROM employee WHERE name like ‘%Smith%’;

**RQP1:** S A₁ A₂ A₃ A₄ K₁ K₂ A₁ A₃ A₅ K₄ K₂ K₅ V

**RQ2:** SELECT name, title, department, email FROM employee WHERE name like ‘%Smith%’ order by 1, 2, 3, 4, 5 -- %’;

**RQP₂:** S A₁ A₂ A₃ A₄ A₅ K₁ K₂ A₁ K₃ A₂ K₄ V K₆ VV VV VV V V

LQP₁: S K₁ K₂ A₁ K₃ A₂ R₁ V L₁ A₁ R₁ V
LQP₂: S A₁ A₂ A₃ A₅ A₂ K₁ A₂ K₄ V
LQP₃: U A₁ K₅ A₃ R₁ V K₂ A₂ R₁ V

LQP₂ = RQP₁
So, the RQ₁ is the legitimate query.

LQP₁ ≠ RQP₂
LQP₂ ≠ RQP₃
So, the RQ₂ is the attack query.

V. RESULT ANALYSIS

The system is tested on four real world applications, namely Portal, Event Manager, Employee Directory and Classifieds. The applications are taken from gotocode.com. For each application, there are two sets of inputs: LEGIT, which consists of legitimate inputs for the application, and ATTACK, which consists of attempted SQLIAs. The applications are deployed on Tomcat server with MySQL database. The queries are tested from the list of queries containing sets of both legitimate and attack queries and the request is provided to application using the wget command. The proposed technique causes two types of overhead. The first overhead is for the static analysis of the application source code to construct legitimate query patterns and the second due to runtime validation. Figure 2 shows the execution time required for the static analysis to be executed on different applications having varying number of queries.

The second overhead is due to the runtime validation. The runtime validation inverts some overhead in terms of execution time at both Input Checker and Query Checker. We measure the times required to run all of the inputs in the LEGIT set against instrumented and un-instrumented versions of each application and compare these two times. To get average access time, we measure the time required to run the entire LEGIT set and then divide it by the number of test inputs. Table 2 shows the result of this study. In the table, all absolute times are described in milliseconds.

**Table 2. Results of Overhead Measurement**

<table>
<thead>
<tr>
<th>Web Application</th>
<th>#Inputs</th>
<th>Avg Access Time (ms)</th>
<th>Avg Overhead (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portal</td>
<td>1293</td>
<td>29</td>
<td>7</td>
</tr>
<tr>
<td>Event Manager</td>
<td>427</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>Employee Directory</td>
<td>773</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td>Classifieds</td>
<td>581</td>
<td>18</td>
<td>1</td>
</tr>
</tbody>
</table>

VI. CONCLUSION AND FUTURE WORK

In this paper, brief study of various SQL Injection attacks is described and different methods for Detection and Prevention of these attacks are also discussed. The main goal of SQL injection attack is to inject some malicious script to the database to gain an unauthorized access to the system. We have proposed a method for detecting SQL injection attacks. Our proposed system can detect the attacks very efficiently by restricting the number of queries to be scanned during runtime. Furthermore, we have evaluated the performance of the proposed method by experimenting on vulnerable web applications. This work may be extended to include detection against other attacks like Cross-Site Scripting.

VII. REFERENCES


A Detection Method for SQL Injection Attacks in Web Applications


