Test SQL Injection Vulnerabilities in Web Applications Based on Structure Matching

Haiyan Wu
Computer & Information Center
Tsinghua University
Beijing, China
wuhy@cic.tsinghua.edu.cn

Guozhu Gao  chunyu Miao
Computer & Information Center
Tsinghua University
Beijing, China
ggz@cic.tsinghua.edu.cn
mcy@tsinghua.edu.cn

Abstract—SQL injection, known as a popular attack against web applications, has become a serious security risk. However, traditional penetration test methods are insufficient to test SQL injection vulnerabilities (SQLIVs) in web applications. This paper presents a new test method called SMART, which automatically detects SQLIVs in web applications. SMART analyzes the SQL queries generated by web applications and uses a structure matching validation mechanism to determine whether SQLIVs exist. Comprehensive experiments show that SMART is effective in finding SQLIVs. Testing the web applications with SMART, the security against SQL injection can be greatly improved.

Keywords- SQL injection; web application; network security

I. INTRODUCTION

With the rapid development of Internet, web applications involving database component become more and more popular. Among these web applications, Structured Query Language (SQL) is the major language to interact with database systems, such as MS SQL Server, Oracle, Access, MySQL, etc. However, many web applications use unrestricted user input data when composing SQL queries, which potentially facilitates SQL injection attacks (SQLIA). According to the report of OWASP [1](Open Web Application Security Project) top ten web application security risks in 2010, injection has become the most serious security risk.

Here is a simple example of how SQL injection occurs. Suppose that a specific web application uses the following code for user authentication:

```
String query = "SELECT accounts FROM users WHERE login='" + login + "' AND pass='" + pass + "'";
```

The user name and password fields from web forms are parsed to generate a query which gets the user information from database. If the corresponding fields are filled with "admin" and " OR 1=1", the generated SQL query would be:

```
SELECT accounts FROM users WHERE login='admin' AND pass='" OR 1=1
```

In this situation, the WHERE clause always has a true value, and contributes to a query result of all the accounts in table "users". If the web application takes the first record to be the authenticated user, the user authentication mechanism would be broken by SQL injection.

In this paper, we present a new method SMART to test SQL injection vulnerabilities (SQLIVs) in web applications. SMART checks all the input parameters for every web page automatically, matches the structure of generated SQL queries, and reports the suspicious vulnerabilities.

II. RELATED WORK

In this section, we present some related test methods for SQLIVs.

Paros [2] is an open-source security scanner for testing web application vulnerabilities. It is a traditional penetration test which automatically scans web applications with injected HTTP request. By analyzing the response page, Paros determines whether SQLIVs exist or not. Paros is not effective enough in testing SQLIVs and has a high rate of false positives.

WAVES [3] implements a machine learning method in building attack requests. It analyzes the response page to verify SQLIVs and modify the attack request for deep injection. It is better than traditional penetration test methods by improving the attack methodology, but it cannot test all the vulnerable spots.

JDBC checker [4, 5] checks the type correctness of generated SQL queries, in order to find SQLIVs caused by improper type checking. It can guarantee the completeness of such kind of SQLIVs but may not be powerful when applied to other kinds of SQLIVs.

Sania [6] proposes syntactic and semantic analysis of the parse tree of intended SQL queries. After filling attack codes in leaf nodes, Sania checks the differences between initially parse tree and modified parse tree, and reports SQLIVs according to the differences. Its methodology is more effective than traditional test methods, but Sania needs a lot of configuration to describe vulnerable spots so it is not convenient enough for automatically test.

MUSIC [7] defines nine mutation operations to inject faults into where condition of SQL queries or database API method calls. By analyzing the data in database, MUSIC determines
whether SQL injection attack is successful or not. MUSIC is also effective than traditional test methods, but it requires sufficient data in database, and needs to change the source code of web applications.

III. ALGORITHM OVERVIEW

In this section, we present a new test method SMART for testing SQLIVs in web applications. SMART includes four steps: structure definition, structure extraction, structure matching and validation. Structure definition defines the structure features, which represent the characteristics of SQL language. Structure extraction method extracts the structure features from given SQL queries. Structure matching method matches the structure of the original SQL and injected SQL. Finally, we have a validation mechanism to determine whether SQL injection succeeds.

A. Structure definition

According to the ANSI SQL standard, the SQL elements can be divided into seven categories:

1. Blank, includes space characters, tabs, carriage returns, line feeds, etc.

2. Single-line comment, often lead by comment symbol “--”.

3. Multiple-line comment, often cited by a pair of comment symbol “/*” and “*/”.

4. Keyword, pre-defined by SQL standard, which makes the SQL query meaningful, such as “SELECT”, “INSERT”, “GRANT”, etc.

5. Punctuation, often used to separate SQL queries, or used in some mathematical operations, such as “=”, “(”, “)”, “,”, etc.

6. Identifier, often used to specify database name, table name, variable name, etc.

7. Data, includes all kinds of data used in SQL standard, such as integers, real numbers, strings, dates, times, etc.

According to the study of successful SQLIVs, “Keyword” and “Punctuation” elements are usually changed in generated SQL queries. In the previous example, when SQL injection attack succeeds, the new SQL query has a new keyword “OR” and a new punctuation “=”. To describe the structure of SQL queries, SMART defines the SQL structure features of SQL query Q as a string array $S = \langle a_1, a_2, \ldots, a_m \rangle$, where $a_i(1 \leq i \leq m)$ is the $i$th keyword or punctuation in query $Q$, and $m$ is the total number of keywords and punctuations.

For example, the structure features of query “SELECT accounts FROM users WHERE login = 'admin' AND pass = 'admin'” is: ‘SELECT FROM WHERE = AND =’.

B. Structure extraction

For a given SQL query, we use the regular expressions analysis technology to extract its structure features. First, we specify the regular expressions for all the seven type of SQL elements. For example, the regular expressions of blank is “\s+”, and the regular expressions of single-line comment is “--[^\n]*\n?”

Base on the representation of the regular expressions, we then analyze the SQL query from its beginning: If the longest substring of the query matches with one regular expression, we remove it from the query and record it and its type. Then we continue to iterate the analysis on the remaining query until it becomes an empty string. After regular expressions analysis, SMART generates the structure with recorded keywords and punctuations. If the analysis fails, SMART returns an error message for further validation.

Suppose the query is “SELECT accounts FROM users WHERE login = 'admin'”, “SELECT” is currently the longest substring that matches with the keyword regular expression. So we record “SELECT” as a keyword, modify the query to be “accounts FROM users WHERE login = 'admin'”, and repeat the analysis. Finally, we have 13 records: “SELECT”, “”,”-, “accounts”, “,”,”FROM”, “,”,”users”, “,”,”WHERE”, “,”,”login”, “=”, “admin”. According to the type records and the definition of structure features, we have 4 strings left to generate the structure features: “SELECT”, “FROM”, “WHERE” and “=”. For a given structure features array S, we define three kinds of operation to modify it: “add” an element into it, “delete” an element from it, and “change” one of its elements into another.

Given SQL structure features $S_1 = A[1..m]$, $S_2 = B[1..n]$, we define $d(i, j) = \delta(A[1..i], B[1..j])$ as the least number of the modification operations to transfer $A[1..i]$ into $B[1..j]$. For any $a \in S_1$ and $b \in S_2$, $\delta(a, b) = 0$ when $a = b$, and $\delta(a, b) = 1$ when $a \neq b$. Then we define the matching value:

$matching\_value(S_1, S_2) = d(m, n) = \delta(S_1, S_2)$.

Suppose $E = e_1, e_2, \ldots, e_k$, is a shortest operation sequence to transfer $A[1..i]$ into $B[1..j]$. It is also called an optimal solution for $d(i, j)$. Then $e_k$ must be one of the three following operations:

1. Change $A[i]$ to be $B[j]$. If $A[i] = B[j]$, $e_k$ is an empty operation. So $E'' = e_1, \ldots, e_{k-1}$ is an optimal solution for $d(i-1, j)$.

2. Delete $A[i]$ from $A[1..i]$. So $E'' = e_1, \ldots, e_{k-1}$ is an optimal solution for $d(i-1, j)$.

3. Add $B[j]$ into the end of $A[1..i]$. So $E'' = e_1, \ldots, e_{k-1}$ is an optimal solution for $d(i, j-1)$.

Then we summarize the recursive equation:

$$d(i, j) = \min\{d(i-1, j-1) + \delta(A[i], B[j]), d(i-1, j) + 1, d(i, j-1) + 1\}$$

And its boundary conditions are:

$$d(i, 0) = i, 0 \leq i \leq m, d(0, j) = j, 0 \leq j \leq n$$

Using the recursive equation, we can calculate $d(m, n)$ as $matching\_value(S_1, S_2)$.
D. Validation

Web applications are usually composed by many web pages where each web page has zero or more input parameters. For example, the following HTTP request shows that the “Login.jsp” page has two input parameters “login” and “pass”, and their default values are “admin” and “admin”:

HTTP://www.bookstore.com/Login.jsp?login=admin&pass=admin

In our method, we test the two input parameters in turn. If “login” parameter is the current parameter being tested, we first send the above original request over HTTP, and get all the SQL queries generated by it, denoted as $Q = Q_1 \cdots Q_m$. For each test case (such as “” OR 1=1”), we inject it into the current parameter:

HTTP://www.bookstore.com/Login.jsp?login=admin%27%20OR%201%3D1&pass=admin

Then we get all the SQL queries generated by the injected request, denoted as $Q' = Q_1' \cdots Q_m'$. If $m$ does not equal $n$, we cannot determine whether SQL injection succeeds, because the SQL queries are probably generated by different code branches. Otherwise, if $m$ equals $n$, then we examine each pair of $Q_i$ and $Q_i'$ ($1 \leq i \leq n$), and extract their structure features, denoted as $S_1$ and $S_2$.

If the extraction of $S_1$ and $S_2$ are both failed, we cannot determine whether SQLIV exists. If the extraction of $S_1$ is failed and $S_2$ is successful, we believe that the injection has broken some authentication mechanism, and alert it as a SQLIV. If the extraction of $S_1$ is successful and $S_2$ is failed, we believe that the injection has broken the structure of the generated SQL query, and also alert it as a SQLIV. If the extraction of $S_1$ and $S_2$ are both successful, we calculate the matching value of $S_1$ and $S_2$.

If $\text{matching\_value}(S_1, S_2) = 0$, we believe that the structure of the SQL query doesn’t change so the SQL injection doesn’t succeed. If $\text{matching\_value}(S_1, S_2)$ is larger than zero and not larger than a given upper bound specified with corresponding test case, we believe that the SQL injection appears and changes the structure of the SQL query, and alert it as a SQLIV. Otherwise, if $\text{matching\_value}(S_1, S_2)$ is larger than the upper bound, we believe that the change of the structure is caused by executing different code branches.

If we still cannot determine whether SQL injection succeeds when finishing all the test cases, we believe that the tested input parameter is probably safe against SQL injection. Then we continue to test next input parameter, until all the input parameters are tested.

IV. EXPERIMENTS

We implemented a SMART test system for experiments. It was written in Java and based on the platform of the open-source security scanner Paros. We used the browse history function of Paros to record the web page’s parameter information and used its alert function to send test report. In addition, we designed a configuration module to let user define their own test cases by rewriting an XML file. We also designed a log audit module to dynamically get SQL queries from database logs. Most important, we implemented our SMART test module to automatically generate test cases and test the web applications.

Comprehensive experiments are conducted in this section to show the performance of SMART. Paros and a popular business security scanner IBM Rational Appscan [8] were also used to compare with SMART.

A. Experimental Setup

Ten subjects of JSP open-source web applications from CodeCharge [9] were selected. The detailed information of these web applications are shown in Table I.

<table>
<thead>
<tr>
<th>No.</th>
<th>Subject</th>
<th>Line of Code</th>
<th>Parameters Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Online bookstore</td>
<td>9765</td>
<td>159</td>
</tr>
<tr>
<td>2</td>
<td>Bug tracker</td>
<td>3961</td>
<td>88</td>
</tr>
<tr>
<td>3</td>
<td>Classified Ad.</td>
<td>5701</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>Employee directory</td>
<td>3099</td>
<td>61</td>
</tr>
<tr>
<td>5</td>
<td>Scheduling management</td>
<td>3817</td>
<td>83</td>
</tr>
<tr>
<td>6</td>
<td>Online forum</td>
<td>2253</td>
<td>27</td>
</tr>
<tr>
<td>7</td>
<td>Online yellow pages</td>
<td>4327</td>
<td>109</td>
</tr>
<tr>
<td>8</td>
<td>Link directory</td>
<td>4631</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>Portal</td>
<td>8800</td>
<td>204</td>
</tr>
<tr>
<td>10</td>
<td>Task management</td>
<td>3980</td>
<td>75</td>
</tr>
</tbody>
</table>

B. Time Efficiency Analysis

Figure 1 shows the detailed time used to test the web applications both by Paros and SMART. We divided the test time by the total number of the parameters for each application. As we can see, the test time for each parameter used by SMART, which is 271.2ms in average, is less than the time used by Paros, which is 339.8ms in average. SMART used about 80% time of Paros, which has relatively higher time efficiency. Because Appscan used much more time for test, we did not record it in our experiment.

![Figure 1. Test time used by Paros and SMART](image-url)
C. Test Ability Analysis

Figure 2 shows the detailed number of SQLIVs found by Paros, Appscan and SMART. In our experiment, Paros found 13 SQLIVs, Appscan found 36 SQLIVs, and SMART found 155 SQLIVs in total. These SQLIVs found by SMART were consisted of many kinds of SQL queries, such as SELECT, INSERT, UPDATE and DELETE. Traditional scanners find SQLIVs by analyzing response pages and it is not an accuracy judgment basis. But our method treats the generated SQL queries as our judgment basis, which is much closer to the fundamental of SQL injection. According to the experiment, we verified the high ability of our structure matching method to find SQLIVs.

D. False Positive Analysis

In our experiment, Paros generated 21 false positives, Appscan generated 10 false positives and SMART generated 40 false positives. Their false positive rates were 61.76%, 21.70% and 20.50% respectively. SMART also generated a lot of false positives. But its false positive rate was within an acceptable range. There were three major kinds of false positives generated by SMART.

1. Inject into state parameters. There are many state parameters used in web applications which are not input by users but automatically generated by web applications, for example, the name of the form, the type of the action, etc. When we tried to inject these parameters, the web applications missed some important executive information, run through another code branch, and generated another SQL queries. If the new SQL queries are similar to the original SQL queries, false positive would be generated.

2. Injection leads to parameter filter. Many web applications have validation mechanisms to check the correctness of user input, for example, number validation. If the format of user input parameter is not correct, web applications are likely to filter it and cut the sub-query associated with it. This often changes the structure of the generated SQL queries and causes false positives.

3. Inject into encoded parameter. Some web applications try to use encoded input parameter to build SQL queries. The encoded content sometimes contains some keywords or punctuations, for example, the minus symbol “-”. This mechanism also generated some false positives.

V. CONCLUSION

In this paper we present SMART, a new method to automatically test SQL injection vulnerabilities in web applications. SMART tests each input parameter of web applications, matches the SQL queries generated by both original HTTP request and injected HTTP request, and determines whether it has SQL injection vulnerability. By comprehensive experiments and comparisons with traditional security scanners, SMART is proved to be effective. It use less time, finds much more SQLIVs, and has an acceptable false positives rate. It is also easy to deploy and use, which is more convenient than some recent test methods.

REFERENCES