A Defense Model against SQL Injection Based on Parameterized Queries

Kuan Song and Hua Zhang

State Key Laboratory of networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China.

Keywords: SQL injection, parameterized queries, defense model, web application security.

Abstract. The SQL injection attack is one of the topmost threats for web applications. Most previously proposed methods for detecting SQL injection attacks suffer from false positives and false negatives. This paper describes a defense model against SQL injection based on parameterized queries. Results show that our method has improvement on accuracy and efficiency.

1. Introduction

With the rapid development of Internet technology, SQL injection attack becomes a top issue. As the data shows in 2013, SQL injection attack is the most serious and topmost attack found among the top ten vulnerability list defined by OWASP [1].

The recent approaches on detecting and defensing SQL injection are mainly based on the following methods: user input filtering [2-4], static analysis methods [5-7], dynamic analysis methods [8, 9], randomization and complementary encoding [10]. In this paper, we design a defense model against SQL injection based on parameterized queries, combining user input filtering and dynamic analysis methods. We examine our method various kinds of SQL injection attacks. Results show that our method has improvement on accuracy and efficiency.

2. Related Work

2.1 User Input Filtering

This is a common way to defense SQL injection. Usually, a blacklist is set up to filter user input. Snort [2], Bro [3] and ModSec [1] analyze the signature of all kinds of SQL injection attacks in modern web applications to set up rulesets, but the rulesets are so large that updating takes lots of effort. Another method, pSigene [4], raises a solution by automatically creating generalized signatures represented as regular expressions using machine learning methods. But the accuracy depends on the training sets and training time.

2.2 Static Analysis Methods

Static analysis methods use pointer and taint analysis to find taint data flows from user input to database queries. These methods can verify that a sanitization routine is always called on tainted inputs, but not whether sanitization is performed correctly. Since incorrectly sanitized input may cause an injection attack, it is essential to precisely model the semantics of string operations performing sanitization [5-7]. Static dataflow analysis is conservative, thus static analysis methods suffer from false positives.

2.3 Dynamic Analysis Methods

Dynamic analysis methods aim to examine whether the SQL queries maintain the same structure when constructed with user input. SQLCHECK [8] and CANDID [9] perform a pretreatment in the program on user input, compare the resulting query with the actual query, and report a code injection attack if the queries differ syntactically. But processing the pretreatment needs the modification of web application.

2.4 Randomization and Complementary Encoding

To prevent SQL injection, SQLrand [10] remaps SQL keywords to secret values. Web applications must be modified to use the remapped keywords in the generated queries, and database middleware must be modified to decrypt them back to original keywords, thus it is difficult to deploy.

3. SQL Injection and Parameterized Queries

SQL injection attack occurs when a malicious user inject carefully constructed query strings into the web application to change the original logic of the SQL query. According to the definition described by Ray and Ligatti [11], SQL injection occurs if the user input in the SQL query contains *code* including reserved keywords, build-in function call, bound words (variables and function names) and white spaces. Only numeric, string literals and reserved values are *non-code*.

Parameterized queries regard user input as parameters when analyze SQL queries. The key thing about parameterized queries is the query optimizer creating a cached plan that can be re-used. Regardless of any kinds of user input, the SQL queries maintain the same logical structure [12].

4. Design and Implementation

According to the analysis above, we propose a defense model containing two portion: user input filtering model and dynamic analysis model, as shown in Fig. 1:

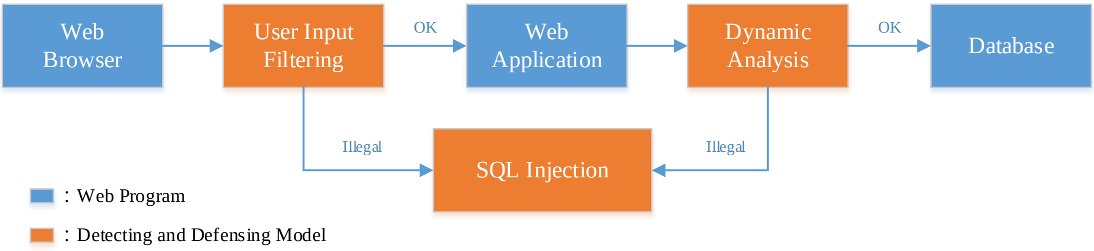


Fig. 1 Detecting and Defensing Model

4.1 User Input Filtering Model

Traditional methods filter all HTTP request with the same rulesets and does not allow SQL keywords in the request content [2]. But SQL keywords is allowed if the HTTP request is submit with *post* method, thus traditional methods suffer from false positives and false negatives. Our method separate *get* method and *post* method with different filtering strategies. We apply keywords filtering to requests in *get* method and rulesets filtering to those in *post* method as shown in Fig. 2:

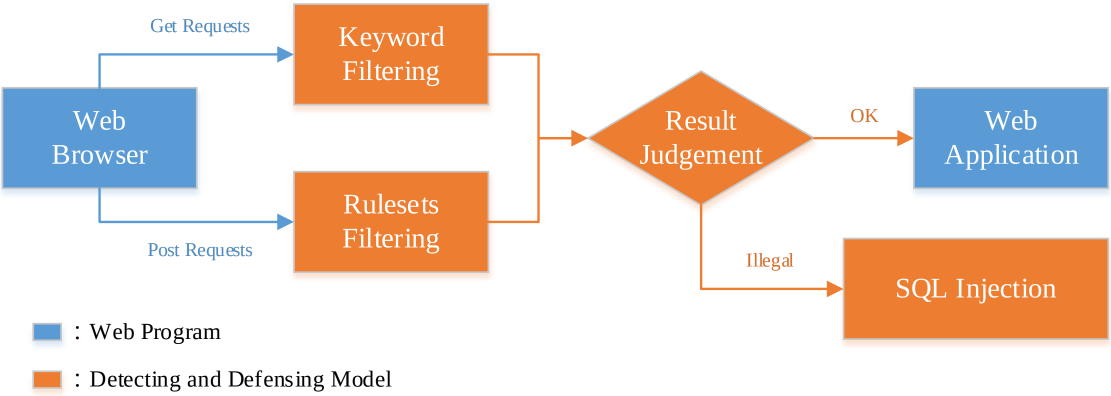
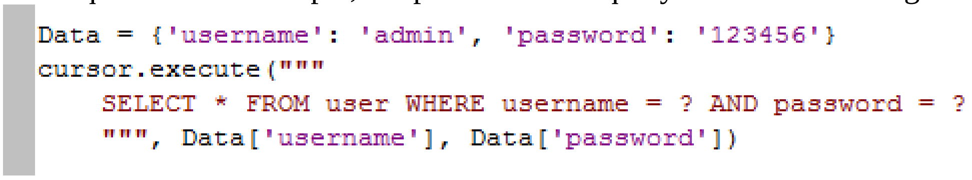


Fig. 2 User input filtering model

4.2 Dynamic Analysis Model

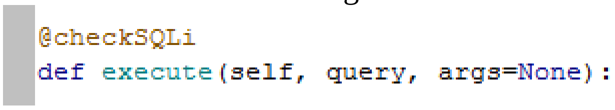
Most dynamic methods perform a pretreatment in the program on user input and compares the syntax tree structure between the resulting query and the original query. But processing the pretreatment requires modification of web application, thus it is difficult to deploy. Our method fixes the problem towards those web applications that are programmed in Python and are based on parameterized queries. For example, the parameterized query is described in Fig. 3:



```
Data = {'username': 'admin', 'password': '123456'}
cursor.execute("""
    SELECT * FROM user WHERE username = ? AND password = ?
    """, Data['username'], Data['password'])
```

Fig. 3 Parameterized SQL query

In the figure above, “Data” contains parameter values match with two placeholder “?” in the SQL query which represent the username and password. Our method only modifies the definition of the function “execute” in the third-party library “pyodbc” and remains the web application code unchanged. Details are shown in Fig. 4:



```
@checkSQLi
def execute(self, query, args=None):
```

Fig. 4 Modification in the third-paity library

“@checkSQLi” is a decorator in Python which will always call another function, which named “checkSQLi”, before processing the function decorated, which named “execute”. The function “checkSQLi” takes function “execute” and its parameters together as parameters to get the original SQL queries and user inputs. The syntax tree structure of original SQL queries and that filled with user inputs are analyzed and compared. An exception will be raised to inform the web application of a SQL injection if the syntax tree structure are different. Otherwise, the decorator will be silence and leave the web application process as usual. Fig. 5 shows the details of dynamic analysis model:

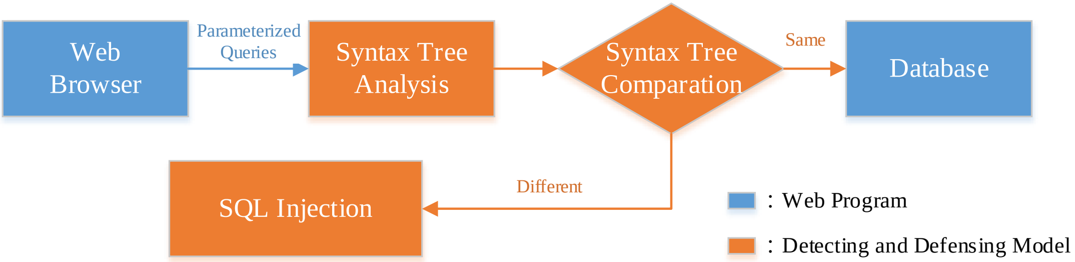


Fig. 5 Dynamic analysis model

5. Evaluation

In this section, we design an experiment to test our defense model. We get SQL queries automatically generated by sqlmap [13] and modify them into two groups. The first group contains normal input and various types of SQL injection to examine the accuracy. The second group contains complicated input to examine the efficiency. We compare our method with three method we discussed above: Snort [2], SQLCHECK [8] and CANDID [9].

Table 1 Results toward first group SQL queries

	Total Queries	Correct Detected	False Positives	False Negatives	Accuracy Rate
Snort [2]	1000	921	18	61	92.1%
SQLCHECK [8]	1000	982	12	6	98.2%
CANDID [9]	1000	977	14	9	97.7%
Our Method	1000	986	10	4	98.6%

Results in Table 1 show that user input filtering method has more false negatives rate because of the limitation of the rulesets. Dynamic methods have high accuracy rate by comparing the syntax tree structure of SQL queries. Our method combines these two methods so that it can detect attacks that with the same syntax tree structure by input filtering, thus is more accurate.

Table 2 Results toward second group SQL queries

	Shortest Response Time (ms)	Longest Response Time (ms)	Average Response Time (ms)
Snort [2]	837	4160	2017
SQLCHECK [8]	1326	2771	1773
CANDID [9]	1084	2403	1525
Our Method	242	3255	834

Results in Table 2 show that user input filtering method has the shortest and the longest response time which is dependent on the process of ruleset matching, thus it has longer average response time. Dynamic methods have stable response time but the process of syntax tree structure on every SQL query takes time. Our method has shorter average response time because of the combination of classified user input filtering method and dynamic analysis method.

6. Summary

Traditional detecting and defensing methods against SQL injection attack suffer from false positives and false negatives. We propose a defense model combining user input filtering method and dynamic analysis method against SQL injection attack for web applications based on parameterized queries. Results show that our method has higher accuracy and efficiency rate.

Acknowledgments

This work is supported by NSFC (Grant Nos. 61300181, 61202434), the Fundamental Research Funds for the Central Universities (Grant No. 2015RC23).

References

[1]. Information on: http://www.owasp.org

[2]. Roesch M. Snort: Lightweight intrusion detection for networks. Lisa (1999), p. 229-238.

[3]. Paxson V. Bro: A system for detecting network intruders in real-time. Computer Networks. Vol. 31 (1999), p. 2435-2463.

[4]. Howard G M, Gutierrez C N, Arshad F, et al. pSigene: Webcrawling to generalize SQL injection signatures. Dependable Systems and Networks (DSN). 2014 44th Annual IEEE/IFIP International Conference on IEEE, 2014, p. 45-56.

[5]. Son S, Shmatikov V. SAFERPHP: Finding semantic vulnerabilities in PHP applications. PLAS (2011) No. 8.

[6]. Nguyen-Tuong A, Guarnieri S, Green D, et al. Automatically hardening web applications using precise tainting. 20th IFIP International Information Security Conference. 2005, p. 372-382.

[7]. Xu W, Bhatkar S, Sekar R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. Proc of Usenix Security Symp. Vol. 15 (2006) No. 9.

[8]. Su Z, Wassermann G. The essence of command injection attacks in web applications. Acm Sigplan Notices. Vol. 41 (2006), p. 372-382.

[9]. Bandhakavi S, Bisht P, Madhusudan P, et al. CANDID: Preventing SQL injection attacks using dynamic candidate evaluations. 14th ACM conference on Computer and communications security. 2007, p. 12-24.

[10]. Boyd S W, Keromytis A D. SQLrand: Preventing SQL injection attacks. Lecture Notes in Computer Science. Vol. 3089 (2004), p. 292-302.

[11]. Ray D, Ligatti J. Defining code-injection attacks. Acm Sigplan Notices. Vol. 47 (2012), p. 179-190.

[12]. Information on: http://msdn.microsoft.com/en-us/library/ms172984(SQL.100).aspx

[13]. Information on: http://sqlmap.org/