SQLUnitGen: SQL Injection Testing Using Static and Dynamic Analysis

Yonghee Shin Laurie Williams Tao Xie
Department of Computer Science, North Carolina State University
yonghee.shin@ncsu.edu williams@csc.ncsu.edu xie@csc.ncsu.edu

Abstract

This paper proposes an approach to facilitate the identification of true input manipulation vulnerabilities via automated testing based on static analysis. We implemented a prototype of SQL injection vulnerability detection tool, SQLUnitGen. Our case study shows that SQLUnitGen had no false positives, but had a small number of false negatives while a static analysis tool called FindBugs had a false positive for every vulnerability that was actually protected by a white or black list input filter.

1. Introduction

More than half of all of the vulnerabilities reported in 2003-4 were input manipulation vulnerabilities, such as SQL injection, cross site scripting (XSS), and buffer overflows. Among these vulnerabilities, a SQL injection vulnerability allows attackers to access or modify critical information in a database. SQL injection attack is caused by the fact that a SQL query can be constructed from user input in a way that the user input can change the intended function of a SQL command in an application.

Traditional approaches to deal with SQL injection attacks are to fortify applications using black or white list input filters or using special APIs, to detect SQL injection vulnerabilities by using static analysis tools, or to detect SQL injection attacks at runtime. To ensure that the filters are properly implemented, testing is required. However, manual test case generation takes time and requires developers to understand ever-evolving attack patterns. Static analysis tools can detect vulnerabilities at an early development phase. However, these tools cannot detect the presence or the effectiveness of input filters. As a result, static analysis tools may have a high false positive rate. Runtime detection does not provide information that can be used to fix the vulnerable code in the early development phase.

Our research objective is to facilitate the identification of true input manipulation vulnerabilities via automatic testing based on static analysis. We implemented a prototype tool SQLUnitGen v0.5 that can be used to identify SQL injection vulnerabilities. As we refine SQLUnitGen, we expect that our approach can be expanded for the identification of other types of input manipulation vulnerabilities.

2. Approach

Our approach uses static analysis to trace the flow of user input values and to obtain concrete attack input for the relevant method arguments. Our approach uses an existing automatic test case generation tool, JCrasher [2], with slight modifications for two purposes. The first purpose is to obtain the initial test cases whose execution reaches SQL query processing APIs. The second purpose is to obtain the attack test cases with test input modified from the initial test cases. The test input is modified with the pre-defined attack patterns. Attack patterns are assigned to the method arguments that are used to construct a SQL query through a chain of method calls.

Static analysis is performed by using AMANESIA [3], a SQL query model builder, and by extending the query model to include input flow information. Test case generation is performed by using JCrasher [2]. We modified JCrasher slightly for our purpose. To help programmers to easily identify vulnerable locations in the program, our approach generates a colored call graph indicating secure and vulnerable methods. The detailed information about our approach can be obtained from our technical report [5]. Figure 1 shows an example application. Figure 2 shows an initial test case generated by JCrasher for the application. Figure 3 shows an attack test case generated by SQLUnitGen. The test case in Figure 3 tests if the variable `id` in the example in Figure 1 is properly validated or not.

Although our approach is useful to test SQL injection vulnerabilities, the current implementation has some limitations. First, false negatives can happen when the predefined attack patterns are not effective to detect all the possible attacks. Second, false negatives can be generated when the initial test cases generated by JCrasher do not include all the possible paths to SQL processing APIs in the application. Finally, our approach inherits the limitations of underlying static analysis tool, AMNESIA, in our current implementation. The limitations include scalability and the limitation of JSA

[1] (an underlying tool of AMNESIA), in analyzing non-local variables (fields in a class).

```
public boolean isRegistered(String id,
                    String password) {
   …
   String sqlQuery = "SELECT userinfo FROM users
             WHERE id = '" + id + "' AND
             password = '" + password + "'";
   Statement stmt = dbConn.createStatement();
   ResultSet rs = stmt.executeQuery(sqlQuery);
   …
}
```

Figure 1. An example of SQL query.

```
public void test0() throws Throwable {
     java.lang.String s4 = "normal";
     java.lang.String s5 = "normal";
     SampleApp s2 = new SampleApp();
     boolean result = s2.isRegistered(s4, s5);
}
```

Figure 2. An initial test case

```
public void test0() throws Throwable {
    java.lang.String s4 = "1' OR '1'='1";
    java.lang.String s5 = "normal";
    SampleApp s2 = new SampleApp();
    boolean result = s2.isRegistered(s4, s5);
}
```

Figure 3. An attack test case

3. Evaluation

To investigate the effectiveness of the proposed approach, we performed preliminary case studies with SQLUnitGen v0.5 on two small web applications, a class project, Cabinetstore, and an open source code, Bookstore, from http://www.gotocode.com. Because the limitations described in Section 3, we used only the login module of these applications after some modification of the source code. We made three versions of each application so that different versions have different levels of input filters: no input filters, partial input filters, and complete input filters. Therefore, we have six versions from the two applications.

For the evaluation, the results were compared with the results of a static analysis tool, FindBugs [4]. FindBugs is a tool that detects various bug patterns in Java programs, including SQL injection vulnerabilities. FindBugs gives a warning when a SQL query is constructed from variables, not purely from constant values.

SQLUnitGen generated 194 attack test cases. The evaluation results show that SQLUnitGen generated no false positives and two false negatives. However, due to the current limitations of SQLUnitGen, higher rate of false negatives can happen for other applications. On the other hand, FindBugs generated ten false positives with no false negatives. Table 1 shows the comparison results with FindBugs. The detailed information about the evaluation can be obtained from our technical report [5] .

Table 1: Comparison with static analysis tool.

App.	Tools	VH	VF	FP	FN
B 1	SQLUnitGen	1	1	0 (0%)	0
	FindBugs	1	1	0 (0%)	0
B 2	SQLUnitGen	1	1	0 (0%)	0
	FindBugs	1	1	0 (0%)	0
B 3	SQLUnitGen	0	0	0 (0%)	0
	FindBugs	0	1	1 (100%)	0
C 1	SQLUnitGen	5	3	0 (0%)	2
	FindBugs	5	5	0 (0%)	0
C 2	SQLUnitGen	1	1	0 (0%)	0
	FindBugs	1	5	4 (80%)	0
C 3	SQLUnitGen	0	0	0 (0%)	0
	FindBugs	0	5	5 (100%)	0

B *n*: Bookstore version *n* C *n*: Cabinet store version *n*
VH: Vulnerable hotspots VF: Vulnerabilities found
FP: False positives FN: False negatives

4. Conclusion

This paper presented an automated test case generation technique to identify SQL injection vulnerability based on static analysis. The evaluation results show that our approach can be used to locate precise vulnerable locations of source code and help to identify false positives that are caused by static analysis tools. Our future work will focus on dealing with the limitations revealed in the initial implementation and evaluation.

References

[1] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise Analysis of String Expressions," In Proceedings of the 10th International Static Analysis Symposium (SAS'03), pp. 1–18, June. 2003.

[2] C. Csallner and Y. Smaragdakis, "JCrasher: An Automatic Robustness Tester for Java," Software -- Practice & Experience, vol. 34, no. 11, pp. 1025-1050, 2004.

[3] W. G. J. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks," In Proceedings of 20th ACM International Conference on Automated Software Engineering (ASE'05), pp. 174 - 183, Long Beach, California, U.S.A., November 7-11. 2005.

[4] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," SIGPLAN Notices, vol. 39, no. 12, 2004.

[5] Y. Shin, L. Williams, and T. Xie, "SQLUnitGen: Test Case Generation for SQL Injection Detection," North Carolina State University, Raleigh Technical report, NCSU CSC TR 2006-21, 2006.