SQLUnitGen: SQL Injection Testing Using Static and Dynamic Analysis

Yonghee Shin Laurie Williams Tao Xie
Department of Computer Science, North Carolina State University, Raleigh, NC 27695
yonghee.shin@ncsu.edu williams@csc.ncsu.edu xie@csc.ncsu.edu

Abstract

This paper proposes an approach to facilitate the identification of actual input manipulation vulnerabilities via automated testing based on static analysis. We implemented a prototype of a SQL injection vulnerability detection tool, SQLUnitGen, which we compared to a static analysis tool, FindBugs. 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.

1. Introduction

More than half of all of the cyber security vulnerabilities reported in 2002-6 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 attacks exploit SQL queries that can be constructed from user input in a way such that the user input can change the intended function of a SQL command in an application.

Traditional approaches to deal with SQL injection attacks include fortifying applications using black or white list input filters, using special APIs, detecting SQL injection vulnerabilities by using static analysis tools, or detecting SQL injection attacks at runtime. Testing is required to ensure that the filters are properly implemented. 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 implemented in the code. As a result, static analysis tools may have a high false positive rate. Also, 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 actual input manipulation vulnerabilities via automated testing based on static analysis and dynamic analysis. We have implemented a prototype tool, SQLUnitGen v0.5, that can be used to identify SQL injection vulnerabilities.

2. Approach

Our approach uses static analysis to trace the flow of user input values and to obtain concrete attack input for testing purposes. Our approach uses an existing automatic test case generation tool, JCrasher [1], with slight modifications. The modified JCrasher tool is used to obtain the initial test cases whose executions reach the SQL query processing APIs. The modified JCrasher is also used to create attack test cases with test input modified from the initial test cases. The test input is modified with pre-defined attack patterns. Attack patterns are assigned to the method arguments in the system under test that are used to construct a SQL query through a chain of method calls.

Static analysis is performed by using AMNESIA [2], a SQL query model builder, and extending the query model to include input flow information. Test case generation is performed by using JCrasher [1]. To help programmers easily identify vulnerable locations in the program, our approach generates a colored call graph indicating secure and vulnerable methods. 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 in testing for SQL injection vulnerabilities, the current implementation has some limitations. First, false negatives can happen when the predefined attack patterns are not sufficient 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 APIs in the application. Third, SQLUnitGen generates test cases only when user input is passed as a method argument after the user input is read from input methods to be used for a SQL query. Fourth, AMNESIA does not account for non-local variables (fields in a class). Finally, the current implementation of SQLUnitGen has scalability problem due to the inefficient modification of AMNESIA and JCrasher.

```
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 a 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 project, Bookstore, from http://www.gotocode.com. Because of the limitations described in Section 2, we examined only the login module of these applications after some modification of the source code. We modified the source code so that the string fields in a class are passed as string type arguments. We also modified the source code so that all the user input is passed as method arguments. 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. Thus, we used a total of six versions of the two applications.

For the evaluation, the results were compared with the results of a static analysis tool, FindBugs [3]. FindBugs detects various bug patterns in Java programs, including SQL injection vulnerabilities.

SQLUnitGen generated 483 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, a higher rate of false negatives may happen for other applications. On the other hand, FindBugs generated ten false positives with no false negatives. Table 1 shows the results of comparison between FindBugs and SQLUnitGen. More detailed information about implementation and evaluation can be obtained from our technical report [4] . Our future work will focus on dealing with the limitations revealed in the initial implementation and evaluation.

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

Reference

[1] C. Csallner and Y. Smaragdakis, "JCrasher: An Automatic Robustness Tester for Java," *Software -- Practice & Experience*, vol. 34, pp. 1025-1050, September 2004.

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

[3] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *SIGPLAN Notices*, vol. 39, 2004.

[4] 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.