Improving the Identification of Actual Input Manipulation Vulnerabilities

Yonghee Shin
North Carolina State University
890 Oval Drive
Raleigh, NC, 27695-8260
919-946-0781
yonghee.shin@ncsu.edu

ABSTRACT
This paper proposes an automated, white-box security testing framework to identify true input manipulation vulnerabilities that can reduce warnings generated by static analysis tools or automated black-box testing tools.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing tools.*

General Terms
Reliability, Security

Keywords
Software Security Testing, Test case generation, Test input generation

1. INTRODUCTION
More than half of all of the cyber vulnerabilities reported in 2002-6 were input manipulation vulnerabilities, where attackers use unexpected input values to alter the expected system behavior [1]. Examples of input manipulation vulnerabilities include SQL injection attacks, cross site scripting (XSS), and buffer overflows. The number of input manipulation vulnerabilities is still increasing. Attackers can access or modify critical information in a system by entering well-devised malicious input in the user input fields via user interface such as web pages.

One approach to prevent input manipulation vulnerabilities is to fortify applications using input filters. Input filters check if user input includes malicious characters (*black-list*) or if the input consists of only non-malicious characters (*white-list*). Static analysis tools or testing are used, to help proper implementation of input filters. However, static analysis tools often cannot detect the presence or the effectiveness of input filters. As a result, static analysis tools may have a high false positive rate when reporting input manipulation vulnerabilities in applications with effective filters. Automated black-box testing techniques also might not generate enough test cases due to the lack of information on internal structure of the application under test. As a complementary approach to static analysis and automated black-box testing, automated white-box testing might help to identify actual input manipulation vulnerabilities because white-box testing uses more information about the internal structure of the applications compared to black-box approach. Additionally, testing shows feasible execution paths with concrete test inputs whereas static analysis only provides possibly-vulnerable locations or paths in source code.

The objective of the proposed research is to facilitate the efficient and accurate identification of actual input manipulation vulnerabilities in the development phase by automated test input generation using a white-box testing approach. Our approach generates well-devised test inputs that show the existence of genuine vulnerabilities in applications by analyzing the flow of string type user input and input filters in the applications. As with all other testing approaches, our approach cannot guarantee the identification of *all* vulnerabilities. We expect that our approach can demonstrate the existence of vulnerabilities more precisely than static analysis tools alone or automated black-box testing.

The rest of this paper is organized as follows. Section 2 provides related work. Section 3 describes the proposed approach. Section 4 describes an initial implementation of our approach called SQLUnitGen and its evaluation results. Section 5 concludes.

2. RELATED WORK
There are existing techniques that can be used to detect or prevent input manipulation vulnerabilities. This section discusses those techniques and compares them with our approach.

2.1 Manual Approaches
This section describes manual approaches to detect and prevent input manipulation vulnerabilities.

2.1.1 Defensive Programming
Programmers can implement their own input filters or use existing safe APIs that prevent malicious input or that convert malicious input into safer input. However, some of these approaches require programmers to learn the usage of APIs and therefore,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference'04, Month 1–2, 2004, City, State, Country.
Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

programmers may not be willing to use them. Furthermore, improper usage of safe APIs still can allow attacks. Our approach improves confidence on the applications using these defensive programming practices by automated test input generation.

2.1.2 Code Review

Code review is a time consuming task compared to automated static analysis and may be skipped by software development teams rushing to ship an application. Additionally, the reviewer must have deep knowledge about how input manipulation attacks work. Our approach helps to fortify applications with less effort and by non-security experts.

2.1.3 Manual Penetration Testing

Penetration testing is black-box testing that simulates security attacks based on knowledge about the system. Penetration testing is usually performed at the end of development life cycle within a limited amount of time. Therefore, the cost of removing the vulnerabilities found during penetration testing can be very expensive. Additionally, the effectiveness of penetration testing depends on the security expertise of the tester. Our approach helps ease testing by implementing the collective knowledge from reported vulnerabilities into a tool and by providing more thorough testing with the help of static and dynamic analysis.

2.2 Automated Approaches

This section describes automated approaches to detect and prevent input manipulation vulnerabilities.

2.2.1 Static Analysis

FindBugs [2] and LAPSE [3] are static analysis tools that can detect input manipulation vulnerabilities in Java programs. However, they ignore control flow or perform weak control flow analysis and therefore do not recognize the existence of user input filters in applications precisely. As a result, they may generate many false positives. Our approach can reduce false positives from static analysis tools by providing concrete test input with a high probability to reveal vulnerabilities.

2.2.2 Web Vulnerability Scanning

Web vulnerability scanners crawl and scan for web vulnerabilities by using software agents. These tools perform attacks against web applications, usually in a black-box fashion, and detect vulnerabilities by observing the applications' response to the attacks [4]. However, without exact knowledge about the internal structure of applications, a black-box approach might not have enough test cases to reveal existing vulnerabilities and also have false positives. On the other hand, our approach provides more precise attack inputs using a white-box approach.

2.2.3 Runtime Detection

SQLCheck [5], learning-based detection of SQL injection attacks [6] and AMNESIA [7] detect SQL injection at runtime. When used in production software, the runtime detection must be extremely accurate since any false positives may prevent legitimate user input from being processed. Additionally, the runtime detection of vulnerable locations in applications is not straightforward, and the timing gap between vulnerability detection and source code modification can take much longer than in other approaches. Since one application can be used in various environments and can also be reused for other applications, fortifying the application itself during development is important in addition to detecting vulnerabilities at runtime.

3. PROPOSED APPROACH

The hypothesis of this research is that *automated white-box testing is more effective in finding actual input manipulation vulnerabilities than static analysis alone or automated black-box testing approaches*. We propose a framework to facilitate the efficient and accurate identification of genuine input manipulation vulnerabilities in an application by generating test input based on the static and dynamic analysis of user input flow and input validation routines. Our approach generates test input that can reach vulnerable location in applications without being blocked by input filters based on the input validation analysis while the test input reveals vulnerabilities in the applications with a high probability.

3.1. Challenges in Input Validation Testing for Security

The goal of input validation testing for security is to prevent malicious input from entering applications. Challenges in generating test input for security are as follows. First, attackers are focused on particular critical resources or assets, such as a database. In many case, these assets can be accessed through a subset of available APIs. Therefore, test input must be generated in a way that as many paths as possible are executed between user input APIs and those APIs that give access to critical assets (*target APIs*). Test coverage can be measured for only those paths rather than for the whole application to measure the quality of security testing. Second, many traditional path-oriented or goal-oriented test input generation techniques can only support integer, array, or pointer types. Even though recent test input generation techniques, including jCUTE[1], can generate complex data types, they do not generate concrete string type input. However, most input manipulation attacks are caused from string type input. The proposed framework will deal with these challenges.

3.2 Proposed Framework

The framework consists of four steps: 1) input flow analysis; 2) input validation analysis; 3) test input generation; and 4) test case generation. Our approach requires the existence of predefined attack patterns and initial test cases with *normal test input* (non-attack, non-test-failing input). We do not assume particular static or dynamic analyses techniques. Therefore, the framework can be implemented with various combinations of techniques. The goal of each step and possible techniques to accomplish the goal are described below.

First, we perform *input flow analysis*. We identify input flow paths from fields in the user interface (the *source*) to target APIs (the *sink*). To illustrate the idea, suppose that a normal user enters login id to a user input field (Figure1). In an SQL injection attack, an attacker enters malicious input (Figure 2) that will be used in a SQL query (Figure 3). We identify the execution path for the input by executing the existing normal test cases. We can trace

[1] http://www-osl.cs.uiuc.edu/~ksen/cute/

the execution path by inserting trace information to application source code automatically. From input flow analysis, we also obtain the concrete string output at the sink obtained from normal test input. For example, the string output at the sink in Figure 3 is "SELECT userinfo FROM users WHERE id = 'Jon' AND password = 'Doe' when the user input from Figure 1 is used. Thus, from input flow analysis, we can generate strings that are candidates for input to security tests.

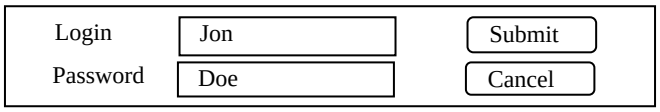

Figure 1. Web interface with normal user input

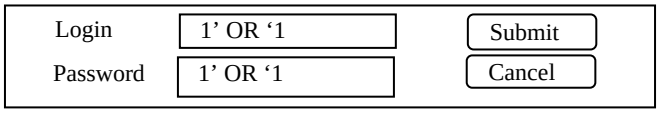

Figure 2. Web interface with malicious user input

```
public class Register extends HttpServlet {


public boolean isRegistered(String id,
                        String password) {
    …
   // input filter
   If(id.contains("1' OR '1) ||
         password.contains("1' OR '1))
               return false;
   String sqlQuery = "SELECT userinfo FROM users
               WHERE id = '" + id + "' AND
               password = '" + password + "'";
   Statement stmt = dbConn.createStatement();
   // sink
   ResultSet rs = stmt.executeQuery(sqlQuery);
   …
}
}
```

Figure 3. An example of SQL query in the server side application

Input validation analysis is performed to identify any existing black-list in the execution path from source to sink. Input strings are often checked against attacks defined in black-lists. Therefore, during input flow analysis, we record the predicates of if-statements that use string data types for each execution path from the source to the sink. From the recorded predicates, we identify black-lists used to validate for the given input. The reason we are doing this is that generating test oracles automatically is very difficult without annotating source code with preconditions and post-conditions for each method. Therefore, when there is no test oracle, the test results must be verified by programmers to check if the test input reveals true vulnerabilities. However, if too many test cases do not actually reveal vulnerabilities because the test input was already blocked by a black-list in the application, then programmers would not find much value in the test input generation tool. To cope with this challenge, our approach generates only test input that is not blocked by input filters, that is, test inputs that may reveal actual vulnerabilities. For example, the user input "1' OR '1" can cause a SQL injection attack. However, for the example in Figure 3, our approach does not generate test input that includes the pattern "1' OR '1" because the application checks if the user input contains the pattern "1' OR '1".

The *test input* is generated based on the vulnerability type identified by static analysis, the output value at the sink obtained from input flow analysis, predefined attack patterns, and the input validation information identified during input validation analysis. We will have to use some heuristics for each type of vulnerabilities to generate precise attack input. For example, if the attack pattern is "SELECT *columns* FROM *tables* WHERE *column* = '1' OR '1'", we need to automatically replace the italicized parts with the corresponding values identified from the output values at the sink. In the example in Figure 3, 'columns', 'tables', and 'column' in the attack pattern will be replaced with 'userinfo', 'users', and 'id', respectively. If a single quotation mark is blocked or sanitized in the application, the input generator can change the single quotation mark in the test input to the Unicode representation to make the attack successful. We plan to generate new attack patterns by creating rules from predefined attack patterns and known knowledge.

At the *test case generation* step, test cases are generated by replacing the test input of existing test cases with the test input generated from the previous steps.

The expected contribution of our research is as follows:

- We propose a framework to reveal actual vulnerabilities with a high probability using white-box approach.
- We implement the proposed framework as a tool that can be used during the development phase.
- We perform an evaluation to show the effectiveness of the proposed framework.

3.3 Future Plan and Evaluation

Various static and dynamic analysis techniques can be applied for each step described in Section 3.2. We will adopt existing static and dynamic analysis techniques to deal with the challenges described in Section 3.1. For example, for input validation analysis, we plan to use existing dynamic data flow analysis techniques, but we will modify those techniques to analyze only relevant data types including string types. Our plan is as follows:

- We will compare existing techniques and how those techniques can deal with challenges described in Section 3.1. We plan to write a survey paper for this topic.
- We will improve existing techniques or create new techniques to deal with challenges described in Section 3.1.
- Then we will implement the techniques into a tool. Improving our initial implementation described in Section 4 is one of the options.

To evaluate our methods, we will perform experiments on the known subjects in literatures and industrial applications and compare the results with other available static analysis and automated black-box testing tools. The currently available tools are FindBugs [2], LAPSE [3], and SecuBut [4]. To measure the effectiveness of our approach, we will measure the number of vulnerabilities identified, the ratio of the number of test cases that identify actual vulnerabilities to the number of test cases generated, the ratio of the number of test cases that identify vulnerabilities to the number of warnings generated by other tools. To measure the efficiency, we will compare the performance in terms of time taken to generate test input with our developed tool and other tools.

4. INITIAL IMPLEMENTATION AND EVALUATION

We implemented an initial version of our framework, SQLUnitGen v0.5, that focuses on SQL injection detection [8]. SQLUnitGen performs input flow analysis statically. SQLUnitGen also provides test oracles. Therefore, input validation analysis is not crucial as discussed in Section 3.

For SQLUnitGen, we modified existing tools, AMNESIA [7] and JCrasher [9]. AMNESIA [7] detects SQL injection at runtime by comparing dynamically-generated SQL queries with statically-generated SQL models. We modified AMNESIA [7] to include user input flow information in the statically-generated SQL model. From this model, we can identify the method arguments that hold user input and that are used for a SQL query. JCrasher [9] generates random test cases to reveal runtime exceptions. We modified JCrasher so that JCrasher generates only test cases that reach SQL processing APIs and change the test input to attack input. We use predefined attack patterns gathered from various sources including literatures. The attack input is assigned to the method variables for user input identified by static analysis of modified AMNESIA. Figure 4 shows an initial test case generated by JCrasher for the example in Figure 3. Figure 5 shows an attack test case generated by SQLUnitGen. The test case in Figure 5 tests if the variable `id` in the example in Figure 3 is properly validated or not.

```
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 4. 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 5. An attack test case

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. We used only the login modules of these applications after some modification of the source code due to the limitations of current implementation. We made three versions of each application so that the different versions have different levels of input filters: no input filters, partial input filters, and complete input filters. Thus, we have six versions from the two applications.

For the evaluation, the results were compared with the results of a static analysis tool, FindBugs [2]. FindBugs gives a warning when a SQL query is constructed from variables, not purely from constant values. 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, higher rate of false negatives can happen for other applications. On the other hand, **FindBugs generated ten false positives with no false negatives.** The detailed information about the implementation and evaluation can be obtained from our technical report [8]. We are currently investigating the possibility of removing the current limitations of SQLUnitGen.

Although our approach is useful to test SQL injection vulnerabilities, the current implementation has some limitations including false negatives, low scalability, and manual source code modification for test case generation [8].

5. CONCLUSIONS

The proposed research starts from the hypothesis that an automated white-box testing approach is effective in finding actual input manipulation vulnerabilities. To investigate this hypothesis, we established a framework to generate test input to detect input manipulation vulnerabilities based on white-box approach. We explained our framework that consists of four steps, discussed the expected contribution, evaluation method and our future plan. We also presented the initial implementation and its evaluation results.

6. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under CAREER Grant No. 0346903. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Alessandro Orso and William Halfond kindly provided us with an implementation of their AMNESIA tool used for our initial implementation.

7. REFERENCES

[1] National Vulnerability Database, http://nvd.nist.gov/.

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

[3] LAPSE: Web Application Security Scanner for Java, http://suif.stanford.edu/~livshits/work/lapse/.

[4] Kals, S., Kirda, E., Kruegel, C., and Jovanovic, N., "SecuBat: A Web Vulnerability Scanner," in *In Proceedings of the 15th International World Wide Web Conference*, 2006.

[5] Su, Z. and Wassermann, G., "The Essence of Command Injection Attacks in Web Applications," in *In Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'06)*, Charleston, South Carolina, USA, 2006, pp. 372 - 382.

[6] Valeur, F., Mutz, D., and Vigna, G., "A Learning-Based Approach to the Detection of SQL Attacks," in *In Proceedings of the Conference on Detection of Intrusions and Malware Vulnerability Assessment(DIMVA)*, 2005.

[7] Halfond, W. G. J. and Orso, A., "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.

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

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