Idea: Using System Level Testing for Revealing SQL Injection-Related Error Message Information Leaks

Ben Smith, Laurie Williams, and Andrew Austin

North Carolina State University, Computer Science Department
890 Oval Drive, Raleigh, NC, USA
{ben_smith,laurie_williams,andrew_austin}@ncsu.edu

Abstract. Completely handling SQL injection consists of two activities: properly protecting the system from malicious input, and preventing any resultant error messages caused by SQL injection from revealing sensitive information. The goal of this research is to assess the relative effectiveness of unit and system level testing of web applications to reveal both error message information leak and SQL injection vulnerabilities. To produce 100% test coverage of 176 SQL statements in four open source web applications, we augmented the original automated unit test cases with our own system level tests that use both normal input and 132 forms of malicious input. Although we discovered no SQL injection vulnerabilities, we exposed 17 error message information leak vulnerabilities associated with SQL statements using system level testing. Our results suggest that security testers who use an iterative, test-driven development process should compose system level rather than unit level tests.

Keywords: SQL, Exception, Tomcat, Java, web application, system level, unit testing, database, SQL injection attacks, coverage, error message.

1 Introduction

In this paper, we examine two input validation vulnerabilities that are in the CWE/SANS Top 25 Most Dangerous Programming Errors[1] due to their prevalence and potential damage: SQL injection vulnerabilities and error message information leak vulnerabilities. *SQL injection vulnerabilities* occur when a lack of input validation could allow a user to force unintended system behavior by altering the logical structure of a SQL statement using SQL reserved words and special characters [1, 2]. The CWE categorizes SQL injection vulnerabilities as a subset of *input validation vulnerabilities*, which occur when a system does not assert that input falls within an acceptable range, allowing the system to be exploited to perform unintended functionality [3]. *Error message information leak vulnerabilities* are caused when an application does not correctly handle an exceptional condition and, as a result, sensitive information is revealed to the attacker [4, 5]. We contend that in web

[1] The CWE/SANS Top 25 can be found at http://cwe.mitre.org/top25/.

F. Massacci, D. Wallach, and N. Zannone (Eds.): ESSoS 2010, LNCS 5965, pp. 192–200, 2010.

applications, where security is paramount, input validation is comprised of *both* ensuring that input falls within an acceptable range (e.g. "integer") and that the application fails gracefully when input is *not* within said range.

To expose and mitigate SQL injection vulnerabilities at the white box level, a development team can execute unit tests that assert that malicious input is rejected by the components that communicate with the database [6]. In some development methodologies, components are constructed in horizontal slices that emanate from the ground up—the components that perform logic and interact with the database are composed and tested long before the user interface. However, in an *iterative development methodology*, teams build software on a feature-by-feature basis in vertical slices that extend from the database to the user interface [22]. Additionally, *test-driven development* implies the incremental creation of tests throughout the development process [7].

The goal of this research is to assess the relative effectiveness of system and unit level testing of web applications to reveal both SQL injection vulnerabilities and error message information leakage vulnerabilities when used with an iterative test automation practice by a feature development team. We conducted a case study on four Java-based open source web applications: iTrust[2], Hispacta[3], LogicServices[4], and TuduLists[5]. In our case study, we executed and compared JUnit[6] unit tests and HtmlUnit[7] system level tests. The purpose of this study is to determine whether system level testing[8] could be used in an iterative or test-driven development scenario to expose both parts of input validation earlier in the lifecycle—an important component of building security in from the beginning [8].

The rest of this paper is organized as follows. Section 2 presents the required background for understanding our study procedure. After that, Section 3 describes the case study, including the subject applications and experimental setup. Next, Section 4 presents the results of our case study. Section 5 presents limitations of the study. Finally, Section 6 describes the conclusions we reached from our study.

2 Background

In this section, we demonstrate an example of a SQL injection vulnerability and discuss error information leakage vulnerabilities.

SQL Injection Vulnerabilities. Consider a Java method used for the deletion of a patient's information in a medical record system. We present the relevant source

[2] http://sourceforge.net/projects/itrust
[3] http://sourceforge.net/projects/hispacta
[4] http://sourceforge.net/projects/logicservice
[5] http://sourceforge.net/projects/tudu
[6] http://www.junit.org
[7] http://htmlunit.sourceforge.net/
[8] The approach we propose in this paper tests the web application in the context of its server; a system level technique. However, our approach also targets specific areas ("hotspots") of the web application; a unit level technique. Thus, the way we use HtmlUnit in our case study is a hybrid of system level and unit level approaches, which is technically considered grey box testing [8, 9].

code for this operation in Figure 1 (assume that patients are deleted by their names). The vulnerability in this example was introduced in the line defining the SQL statement. The example we have presented in Figure 1 performs no input validation and, as a result, the example contains a SQL injection vulnerability relative to the use of the name parameter. An attacker could cause change to the interpretation of the SQL query by entering the SQL command fragment "' OR TRUE --" in the input field instead of any valid user name in the web form.

The single quotation mark (') indicates to the SQL parser that the character sequence for the username column is closed, the fragment OR TRUE is interpreted as always true, and the fragment of the query after the hyphens (--) is a comment. The altered WHERE clause of the SQL statement will be interpreted as always true and thus every patient is deleted from the table. Because no input validation was performed, the attacker can exploit the system by inserting the malicious input in the name field, and cause truncation of the Patients table. Thus, the bolded statement in Figure 1 is an example of a SQL injection hotspot (or just "hotspot" in this paper)—any source code location that may contain a SQL injection vulnerability [1, 2].

```
...
java.sql.Connection mySQLConnector = DriverManager.getConnection();
java.sql.Statement s = mySQLConnector.createStatement("DELETE FROM
Patients WHERE Name = '" + name + "';");
int result = s.executeUpdate();
       return 1 == result;
...
```

Fig. 1. Patient Deletion Code in Java; hotspot is bolded

Error message information leak vulnerabilities. These vulnerabilities occur when an application does not correctly handle exceptional conditions and subsequently leaks sensitive information to a user [4, 5]. This information can be obviously dangerous in the case of error messages that contain system or application passwords, or it may seem more benign, containing only version numbers or stack traces. Unfortunately, even these seemingly benign error information leaks can provide valuable information to an attacker and could expose additional attack vectors. Since a tester cannot tell what information an attacker needs to conduct future attacks, a good policy is to treat all error information leakage vulnerabilities as if they contain obviously dangerous information such as passwords.

3 Case Study

In this section, we present information about our case study. Each part of the case study was conducted using Eclipse v3.3 Europa executed using Java v1.6 running on an IBM Lenovo T61p running Windows Vista Ultimate with a 2.40Ghz Intel Core Duo processor and 2GB of RAM. We used MySQL[9] v5.0.45-community-nt for our research database management system.

[9] http://www.mysql.com

Table 1. Information about the Test Subjects (n=4)

Project	iTrust	Hispacta	LogicServices	TuduLists
Version	4.0	0.0.3	1.8	2.2
Lines of Code[đ]	7707	1991	5011	6178
Production Classes[*]	143	42	155	132
Database Classes	20	4	1	5
Hibernate[10]	No	Yes	Yes	Yes

[đ] Source Lines of Code calculated by NLOC: http://sourceforge.net/projects/nloc/

[*] A production class is any class that is required to be on the class path for the web application to function correctly (excluding test classes and utility classes)

To obtain our case study applications, we collected information about 12 enterprise Java web applications, which we found by searching SourceForge[11] with the query "Java web application," and sampling the first 12 projects that contained the Eclipse webtools[12] project file structure. We then rejected eight subjects from our study because they did not meet one or many of the following criteria:

- Could be compiled, built, and deployed.
- Contained automated unit tests, written in JUnit, which were distributed with the source code.
- Relied upon a relational DBMS to store its data.

We were left with the four subjects presented in Table 1. In an attempt to reveal both SQL injection and error message information leak vulnerabilities in our test subjects, as stated in Section 1, we created the following systematic, system level, security testing procedure and executed it on our subjects. By design, this procedure produces an automated system level test suite that executes all reachable hotspots with normal and malicious input. We note here that by *intrinsic*, we mean that we did not augment or modify the existing test set in any way; values for these measures were achieved by the unit tests that were distributed with each system.

1. **Identify and Instrument Hotspots.** We manually inspected the source code to discover any point where the system interacts with the database. We note here that hotspots can take many forms; we explain this issue more below. We have written the Java program `SQLMarker`, introduced in our earlier work [9]. `SQLMarker` keeps a record of the execution state at runtime for each uniquely identified hotspot[13]. `SQLMarker` has a method, `SQLMarker.mark()`, which passes the line number and file name to a research database that stores whether the hotspot has been executed.

[10] http://www.hibernate.org/
[11] http://sourceforge.net/
[12] http://www.eclipse.org/webtools/
[13] For larger applications, one could use a static analyzer to determine hotspots' locations.

2. **Record Hotspots.** A second class we wrote, called `Instrumenter`, provides each manually marked hotspot with a unique identifier comprised of the filename and line number, and outputs the number of hotspots found. Once we manually marked each, we executed `Instrumenter` to store a record of each of these hotspots.
3. **Execute Original Unit Tests.** After instrumenting each subject to mark its executed SQL hotspots, we executed the intrinsic unit tests and recorded the resultant number of executed statements.
4. **Create Test Cases.** We used the stored file name and line number of the hotspot from Step 1 to construct an automated system level test with HtmlUnit[14] that executed the SQL statement located at the stored file name and line number. We constructed an initial automated test for each hotspot by using a call hierarchy and manual testing to make web requests until the hotspot was marked as being executed and then modeled our automated test after the use case we discovered[15].
5. **Apply Malicious Input.** We modified the test defined in Step 4 to emulate a malicious user by using 132 forms of malicious input in an attack list from NeuroFuzz [10] in place of normal input. This part of the procedure is similar to "fuzzing". The difference here is that fuzzing is a semi-random, black box activity; our approach is targeted to specifically attack the areas where user input might reach a hotspot.
6. **Record Result.** We then marked each test that caused incorrect SQL operations or an application error in Step 5 as a successful attack and its corresponding SQL statement as a vulnerability.

Identifying hotspots may seem trivial, but in fact can be difficult because hotspots may not always take the same form. One way of discovering vulnerabilities is automated static analysis tools, which can be designed to check for a particular hotspot or vulnerability type [11]. We executed static analysis tools on our subjects and the tools reported no input validation vulnerabilities in any project we examined.

4 Results

This section presents the results of our case study. We first observed, as shown in Table 2, that there were no intrinsic JUnit test cases that used malicious input. We conducted all of our 272 system level tests by using HtmlUnit to inject our attack list into a request parameter, or in the case of TuduLists, to conduct an AJAX request where the malicious input was injected into an asynchronous JavaScript call. Using our technique, we found no instances of SQL injection vulnerabilities at the system

[14] HtmlUnit is a "GUI-Less browser for Java programs". It models HTML documents and provides an API that allows you to invoke pages, fill out forms, click links, etc, just like you do in your "normal" browser. http://htmlunit.sourceforge.net.

[15] However, some hotspots were not used by the JSPs in the application, perhaps because these hotspots were used for database administration only, or the development team had not finished implementing the use case that required the query. If we could not reach the SQL statement through the web interface, we augmented the white box test plan to include a malicious test that directly calls the database class.

level. No application allowed us to issue commands to the database management system because prepared statements and Hibernate both perform strong type checking on the variables used in their hotspots. Hibernate allows developers to create persistent classes in the object-oriented paradigm that represent individual database records [12]. However, we found 17 error message information leak vulnerabilities among the four applications in our case study, summarized in Table 2.

Table 2. Results for the Test Subjects

Project	**iTrust**	**Hispacta**	**LogicServices**	**TuduLists**
Hotspots	92	23	48	13
Covered by Intrinsic Tests	89	20	47	3
Statement Coverage (EclEmma)	84%	49%	53%	40%
Test Cases with Malicious Input	0	0	0	0
New System Level Test Cases (Normal and Malicious)	149	29	80	14
Confirmed Vulnerabilities	2	2	9	4

We found that unit testing could have identified none of the 17 confirmed vulnerabilities; rather, these confirmed vulnerabilities are system level vulnerabilities that had to involve the application server. A missing exception handler for pages or Servlets within Apache Tomcat caused each of the vulnerabilities we discovered. The example presented in Table 3 and Table 4 helps illustrate why these vulnerabilities cannot be exposed at the unit level.

Table 3 presents the relevant code from one of the confirmed vulnerabilities that we found in iTrust, in the file `editHCPs.jsp`. In other pages within iTrust, there is a JSP directive declared at the top of the page's code (along with various navigational toolbars and headers) that declares an exception handler:

```
<%@page errorPage="/auth/exceptionHandler.jsp"%>
```

This directive does not appear in editHCPs.jsp (see Table 3). At the moment an exception is thrown, Apache Tomcat forwards the user to the page declared in this directive, if this directive is declared. Otherwise, Apache Tomcat outputs a revealing stack trace to the user's browser window, also known as an error message information leakage.

Since the omission of an exception handler is something that happens in the JSP code and not the Java code, some form of interaction is required with the application server (Apache Tomcat) in order to expose the vulnerabilities. One may view each JSP as a unit, but still the exception handler is a JSP page directive that involves a separate page; the unit therefore cannot be tested in isolation. The confirmed

vulnerabilities, then, are caused by a system level error: the absence of an exception handler in the JSP or Servlet code of the application. Consider a JUnit test case that is written to execute `undeclareHCP` (see Table 4). This JUnit test case would pass, but would not expose the vulnerability even if it uses the some malicious input, such as `' UNION SELECT`. However, an HtmlUnit test case that targets `editHCPs.jsp` (see Table 3), produced by our system level testing technique, would expose the vulnerability using the same attack. That is, the vulnerability is not that an exception is thrown, but rather that the exception is not correctly handled by the JSP.

Table 3. JSP for Example Vulnerability (editHCPs.jsp)

```
DeclareHCPAction action = // Action class for declaring the HCP.
String confirm = ""; // used to store the result from the DAO.
String removeHCP = request.getParameter("removeID");
if(removeHCP!=null && !removeHCP.equals("")){
        confirm = action.undeclareHCP(removeHCP);
}
List<PersonnelBean> hcps = action.getDeclaredHCPS();
```

Table 4. Java Method undeclareHCP

```
//given: patientDAO, a DAO pertaining to the patients table
//given: iTrustException, a custom-build Exception class for
//       handling alternate flow errors
public String  undeclareHCP(String input) throws iTrustException {
try {
long hcpID = Long.valueOf(input);
boolean confirm = patientDAO.undeclareHCP(loggedInMID, hcpID);
if (confirm) {
       return "HCP successfully undeclared";
} else
       return "HCP not undeclared";
} catch (NumberFormatException e) {
       throw new iTrustException("HCP's MID not a number");
} catch (DBException e) {
       throw new iTrustException(e.getMessage());
}
}
```

5 Limitations

Future case studies should examine much larger web applications than the ones in this study. In addition, the selective criteria as described in Section 3 could have biased the data. For example, perhaps the fact that all of our test subjects were Tomcat Servlet applications caused or prevented some security vulnerabilities that would not have been observable in another architectural setup. In addition, if stored procedures had been used in any of our test subjects, our results may have been different. The development teams for each project may have been using other testing techniques to improve the security posture of our subjects, or security may not have been high on their list of requirements.

The container for the applications (in this case, Apache Tomcat) could also be emulated using a Mock Object pattern [13], and each individual servlet or JSP could be tested in isolation from one another. However, the quality of the testing results is entirely dependent on the quality of the mock object's ability to emulate the server [13]; additionally, mock objects may not be any less expensive than system testing. Prepared statements, which separate the user's input from the structure of the query at the application level [14], protected the applications in this study. However, prepared statements are only useful if developers are aware of them and choose to use them. Our own system level procedure may not have exposed all vulnerabilities latent in the four subjects. Our procedure was targeted towards SQL injection vulnerabilities, which did not exist in these sampled applications at the locations of the hotspots we identified, but other vulnerabilities of varying types may exist in our subjects.

6 Conclusion

In our investigation of the relative effectiveness of unit and system level testing techniques, we have discovered that developers sometimes miss the fact that input validation is comprised of *both* ensuring that input falls within an acceptable range (e.g. "integer") and that the application fails gracefully when input is *not* within said range. We found that all four of our study subjects use Hibernate and/or properly constructed prepared statements, which were completely effective for asserting that input falls within a safe (non-attack) range. Using a systematic system level security testing procedure to generate an HtmlUnit test suite, we found 17 error message information leakage vulnerabilities in the four web applications of our study. We found it impossible to replicate these same 17 vulnerabilities by augmenting the intrinsic unit test suites with additional malicious tests because vulnerabilities cannot be exposed at the system level though unit testing.

Our results show that ensuring that error messages resulting from SQL injection attacks do not reveal sensitive information is an inherently system level activity because the web server will dictate how and when error messages are displayed. *Thus, an iterative, a feature-based development team conducting a test-driven automation practice can use a system level test procedure like the one described in this paper to expose both SQL injection vulnerabilities and error message information leak vulnerabilities.* From a security perspective, unit testing would not be effective toward this aim, because it cannot take into account the production environment in which the system exists.

Acknowledgments. We would like to thank the North Carolina State University Realsearch group for their helpful comments on the paper. In addition, we would like to thank Yonghee Shin for the foundational work she performed by providing formal definitions for our SQL hotspot metrics and for her input on the content of this paper. This work is supported 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.

References

1. Halfond, W.G.J., Orso, A.: AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In: 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, CA, USA, pp. 174–183 (2005)
2. Kosuga, Y., Kono, K., Hanaoka, M., Hishiyama, M., Takahama, Y.: Sania: syntactic and semantic analysis for automated testing against SQL injection. In: 23rd Annual Computer Security Applications Conference, Miami Beach, FL, pp. 107–117 (2007)
3. Pietraszek, T., Berghe, C.V.: Defending against injection attacks through context-sensitive string evaluation. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 124–145. Springer, Heidelberg (2006)
4. Aslam, T., Krsul, I., Spafford, E.: Use of a taxonomy of security faults. In: 19th National Information Systems Security Conference, Baltimore, MD, pp. 551–560 (1996)
5. Tsipenyuk, K., Chess, B., McGraw, G.: Seven pernicious kingdoms: a taxonomy of software security errors. IEEE Security & Privacy 3, 81–84 (2005)
6. IEEE: IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology (1990)
7. Beck, K.: Test-driven development: By example. Addison-Wesley, Boston (2003)
8. McGraw, G.: Software security: Building security in. Addison-Wesley, Upper Saddle River (2006)
9. Smith, B., Shin, Y., Williams, L.: Proposing SQL statement coverage metrics. In: The 4th International Workshop on Software Engineering for Secure Systems at the 30th International Conference on Software Engineering, Leipzig, Germany, pp. 49–56 (2008)
10. Jiang, Y., Cukic, B., Menzies, T.: Fault Prediction using Early Lifecycle Data. In: The 18th IEEE International Symposium on Software Reliability, 2007. ISSRE 2007, pp. 237–246 (2007)
11. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in Java applications with static analysis. In: USENIX Security Symposium, Baltimore, MD, pp. 18–18 (2005)
12. Bauer, C., King, G.: Hibernate in Action. Manning Publications (2004)
13. Brown, M., Tapolcsanyi, E.: Mock object patterns. In: The 10th Conference on Pattern Languages of Programs, Monticello, USA (2003)
14. Thomas, S., Williams, L.: Using automated fix generation to secure SQL statements. In: Proceedings of the Third International Workshop on Software Engineering for Secure Systems, Minneapolis, MN (2007)