Using Automated Fix Generation to Secure SQL Statements

Stephen Thomas and Laurie Williams
Department of Computer Science
{smthomas, lawilli3}@ncsu.edu
North Carolina State University, Raleigh, NC, USA

Abstract

Since 2002, over 10% of total cyber vulnerabilities were SQL injection vulnerabilities. Since most developers are not experienced software security practitioners, a solution for correctly fixing SQL injection vulnerabilities that does not require security expertise is desirable. In this paper, we propose an automated method for removing SQL injection vulnerabilities from Java code by converting plain text SQL statements into prepared statements. Prepared statements restrict the way that input can affect the execution of the statement. An automated solution allows developers to remove SQL injection vulnerabilities by replacing vulnerable code with generated secure code. In a formative case study, we tested our automated fix generation algorithm on five toy Java programs which contained seeded SQL injection vulnerabilities and a set of object traceability issues. The results of our case study show that our technique was able remove SQL injection vulnerabilities in five different statement configurations.

1. Introduction

Since 2002, over 50% of total cyber vulnerabilities were input validation vulnerabilities [10]. SQL injection vulnerabilities (SQLIVs) account for 20% of the input validation vulnerabilities and, therefore, 10% of total cyber vulnerabilities since 2002 [10]. A SQLIV allows input to a SQL statement to change the structure of the statement and allows malicious users to gain unauthorized information and abilities. As the trend of providing web-based services continues, the prevalence of SQLIVs is likely to increase [7].

Another concern facing the software development industry is that the number of developers inexperienced in software security outnumbers the number of experienced software security practitioners [1]. As a result, as significant portion of developers fixing SQLIVs will not be experienced with solving security issues. Therefore, a solution that fixes SQLIVs without requiring expertise is desirable. An automated fix generation solution allows both experienced and inexperienced developers to work proactively and reactively. As vulnerable SQL statements are discovered and written by the developer, an automated fix generation method will suggest replacement code that will remove the vulnerability while maintaining the same functionality. A typical developer will also use an automated fix generation method proactively, by replacing legacy code with more secure code to remove potentially vulnerable code.

The objective of this research is to develop an automated fix generation method for removing SQLIVs from plain text SQL statements. Our automated solution gathers information about known vulnerable SQL statement, generates a fix, and replaces the vulnerability with the generated code. We hypothesize that automation will allow developers to fix SQLIVs without requiring security expertise and secure legacy code in a standard way.

In this paper, we present the results of a formative case study of our solution. We developed five toy Java programs with a seeded vulnerable SQL statement in each program to test the ability of our algorithm to convert the vulnerable statements into secure prepared statements. Further, we tested that the generated statements were functionally equivalent to the vulnerable plain text SQL statement. The results from the tests showed that the generated statements were functionally equivalent to the vulnerable statements and were not exploited by the SQL injection attacks (SQLIAs).

The remainder of the paper is organized as follows: Section 2 contains background and related work; Section 3 describes our proposed approach;

Section 4 discusses our formative case study findings; and Section 5 presents our conclusions and future work.

2. Background and related work

In this section, we provide background on SQLIVs, SQLIAs, prepared statements, and automated fix generation and suggestion. We also review several other solutions to mitigate SQLIAs.

2.1. SQLIVs and SQLIAs

In this section, we introduce a typical SQLIV and an associated SQLIA. A SQLIV is the combination of dynamic SQL statement compilation and a lack of input validation, which allows input to change the structure of a SQL query [7]. An example of this combination commonly found in Java is shown in the following code. The code shows us that we have a plain text SQL statement that dynamically creates the SQL query based upon variable input (`userISBN`). In addition, the code creates the SQL query using string concatenation and does so without any input validation.

```
Statement stmt =
conn.createStatement();
ResultSet rs =
stmt.executeQuery("select  amount
from books where isbn = '" +
userISBN + "'");
```

SQLIAs are attempts to input known SQL characters and keywords into SQL statements with the intention to maliciously access or modify critical information in databases [7]. In the previous code, the SQL query can be modified by using SQL characters and keywords to malevolently change the structure of the query or execute queries that the program does not allow. An example of this malicious input could be inputting the values of `111’ OR ‘1’=’1` for `userISBN`. The malicious input contains a value for the SQL query, then exits out of that part of the query and appends an `OR ‘1’=’1` clause to the statement. The `OR ‘1’=’1` clause will always evaluate to true, turning the entire where clause into a true clause, and resulting in the query executing as if there was not a where clause. The resulting dynamically-created SQL statement is, `"select amount from books where isbn = 111' OR ‘1’=’1’"`, which is equivalent to the statement, `"select amount from books"`. These statements are equivalent since `‘1’=’1’` is always true, which opens up the statement to return all results without distinguishing between which results are supposed to be returned.

2.2. Prepared statements

Prepared statements are SQL statements that have been pre-compiled with the SQL query. A SQL statement is the object that will be run by the database while a SQL query is the plain text representation of the statement written by the developer. The SQL query for a prepared statement can have bind variables that allow for input to be put into the query at a later time. The bind variable is the question mark (?) in the query, `"select amount from books where isbn = ?"`, which is noted as a place for input into the query at a later time. In Java, the input for bind variables is set by input set methods that are specific to the input type, such as a ‘`setString(index, input)`’ call for a `String` type input variable. Set methods provide the added security of validating each input variable with respect to its declared type. Prepared statements are used primarily for security and efficiency reasons. Prepared statements were originally created to allow for the execution of the same statement multiple times while only compiling the statement once, a feature that plain text SQL statements do not have [2].

In Java, SQL statements are managed through the `Statement` and `PreparedStatement` interfaces, the former containing the majority of SQLIVs. `PreparedStatement`s were introduced in Java 1.1 to improve the security and efficiency of SQL statements. Prepared statements implement the same functionality as plain text SQL statements, but have a more structured way of using input. The structured handling of the SQL query prevents input from manipulating the structure of the pre-compiled query, thus preventing SQLIVs.

A prepared statement can only be created if the structure of the statement is known before the creation of the statement. Therefore, statements that are dynamically created or rely on input for a part of the structure are not able to be prepared statements. We define the structure of the SQL query to be the clauses, such as select, from, and where, the identifiers, such as what to select, table name, and attribute name, and the comparators, such as equals, like, and, and or. We define the literals that the attribute identifiers are compared to as not part of the structure of the statement and therefore can be missing and not affect whether the statement can be a prepared statement. In Java, once a prepared statement is created by the `Connection` object, the prepared statement is pre-compiled. Once all of the input is set into the statement and the statement is executed, the statement is sent to the database to be executed and the result set or return code is sent back from the database.

2.3. Automated fix generation and suggestion

Automated fix generation and suggestion is used in several development applications. The Eclipse IDE has included a “Quick Fix” feature that suggests replacement code for common development bugs [3]. FindBugs™ has also extended this “Quick Fix” Eclipse IDE feature with “Quick Fixes” for several of the FindBugs™ bug types.

The benefit of automated code generation has been shown to be positive in the related field of automated test generation [6]. Automated test generation is a method that has been used to significantly reduce the cost of software testing [6] which can account for 50% or more of total project costs [11]. Therefore, we surmise that automated fix generation has similar potential to reduce the cost of software fixing.

2.4. Related solutions

Current work on mitigating SQLIAs involve wrapping the vulnerable SQL statement with validation code or putting constraints on the runtime environment to prevent invalid SQL statements from running [2, 7, 12]. Proposed solutions to solving the vulnerability, based on a mixture of these two approaches, have been successful in stopping modern attacks. However, these approaches are designed to *fortify against* SQLIAs while our solution is designed to *remove* the SQLIVs. In addition, each proposed solution to this vulnerability adds developer and computational overhead [2, 7]. In comparison, our proposed method requires minimal developer overhead and will potentially make the code more efficient. The potential increase in efficiency is because `PreparedStatement`-based SQL statements are pre-compiled and can be executed multiple times without any extra statement compilation costs [4].

Su and Wassermann [12] propose a solution to prevent SQLIAs by analyzing the parse tree of the statement, generating custom validation code, and wrapping the vulnerable statement in the validation code. They conducted a study using five real world web applications and applied their SQLCHECK wrapper to each application. They found that their wrapper stopped all of the SQLIAs in their attack set without generating any false positives. While their wrapper was effective in preventing SQLIAs with modern attack structures, we hope to shift the focus from the structure of the attacks and onto removing the SQLIVs.

Haliford and Orso [7, 8] secure vulnerable statements by combining static analysis with statement generation and runtime monitoring. Their AMNESIA solution analyzes the vulnerable SQL statement, generates a general acceptable SQL statement model, and allows or denies each statement based on how it compares to the model at runtime. They conducted a study using five real world web applications and two student-created web applications and applied their AMNESIA solution to each application. They found that their solution stopped all of the SQLIAs in their attack set without generating any false positives. Their solution throws an exception for each SQLIA, which the developer handles and builds in attack recovery logic. While their solution stopped current attacks, we recognize that determining whether a statement is valid or not is a difficult problem since new attacks are created frequently. We propose that the problem of determining statement validity can be avoided by removing the vulnerability that the attacks exploit. Further, their solution uses exceptions to indicate potential attacks, which is a developer overhead we would like to avoid by rendering the attacks harmless and allowing the system to run oblivious to attacks.

Buehrer et al. [2] secure vulnerable SQL statements by comparing the parse tree of a SQL statement before and after input and only allowing a SQL statements to execute if the parse trees match. They conducted a study using one real world web application and applied their SQLGUARD solution to each application. They found that their solution stopped all of the SQLIAs in their test set without generating any false positives. While it stopped all of the SQLIAs, their solution required the developer to rewrite all of their SQL code to use their custom libraries, which is an overhead we are trying to eliminate with code generation. We are also trying to avoid the computational overhead of dynamic statement validation by removing the vulnerability and allowing all input.

3. Proposed solution

Our proposed solution generates a `PreparedStatement`-based fix to a vulnerable SQL statement and suggests the fix to the developer. We assume all plain text SQL statements can be considered vulnerable since plain text SQL statements are dynamically created after input has been included, which allows input to potentially alter the statement structure. Nevertheless, if a better granularity of which SQL statements are most likely to be vulnerable is desired by the developer, code analyzers, such as FindBugs™ [9], a static code analyzer, can achieve this granularity. We have developed an algorithm for generating the `PreparedStatement`-based solution for Java programs and are currently working on implementing the algorithm in an Eclipse IDE plug-in. The following sections discuss the assumptions, three main parts, and limitations of our solution generating algorithm.

3.1. Assumptions

We make several assumptions about the runtime environment that our automated code will run in. We assume that the language, database connector, and database all support prepared statements. Further, we assume that the vulnerable code has the equivalent data types for input as the corresponding database column types. If the data types do not match, we assume that the runtime environment can handle type conversions. In addition, we assume that the database connector and database support prepared statement pre-compilation. Our solution will still work if pre-compilation is not supported, but the security of dynamically-compiled prepared statements is weaker since it is based on input formatting, which stops SQLIAs instead of removing SQLIVs.

3.1. Information gathering

Initially, our algorithm collects three types of information: (1) the plain text SQL statement, (2) the SQL query, and (3) the execution call. We use this information later in our algorithm, and we if we do not have all three parts we cannot generate the `PreparedStatement`-based solution and we will exit the algorithm. In addition, we determine if the statement is a batch job, since we cannot handle batch queries as discussed in Section 3.4.

Below is a sample SQL statement found in one of the five toy vulnerable Java programs we used for forming our algorithm

```
Statement stmt =
conn.createStatement();
ResultSet rs =
stmt.executeQuery("select amount
from books where isbn = '" +
userISBN +
"'");
```

The example statement contains all three parts:

- `stmt` is the plain text SQL statement
- `"select amount from books where isbn = '" + userISBN + "'"` is the SQL query
- `executeQuery` is the execution call

We can proceed to the "fix generation" step with the information gathered. Further, since the execution call is not an `executeBatch` call, we know that the `Statement` is not a batch job and can be converted. We have designed our algorithm to use the Java AST provided by the Eclipse IDE for our initial data collection. We will analyze the Java AST to find statement, query, and execution call and we will use the AST to determine if there are any scope or method signature conflicts that we will have replacing the vulnerable code with our generated code.

3.2. Fix generation

In this stage, we initially analyze the plain text SQL query to determine the structure of the query in the form of a parse tree. We find any input variables and determine the structure of the query without input. We then create a prepared statement and a functionally-equivalent query with the same structure and use bind variables in the place of input. However, if we cannot find any input variables, then we cannot determine the structure of the query without input and thus we cannot create the `PreparedStatement`-based solution.

We have designed our method to use a SQL parse tree analysis technique to determine the structure of the SQL query as well as the input variables [2]. We will use a SQL parser, such as the freely-available Java SQL parser Zql [5], to build the SQL parse tree. Buehrer et al. use parse trees in a dynamic way to make runtime comparisons to determine whether two queries are functionally equivalent. We plan on building upon their research and using parse trees in a static way to determine the structure and input variables. Parse trees make this determination easy since all input variables are clearly marked as leaves off of 'literal' nodes.

Once we have a functionally-equivalent SQL query with bind variables for input, we create the set calls for each bind variable and put input into the proper bind variable locations. These set calls are based on the data type of each input variable. For instance, we would match up an input variable of type `String` with a '`setString(index, input)`' call. Finally, we generate the `PreparedStatement`-based execute call that is functionally equivalent to the vulnerable statement execute call. The `PreparedStatement`-based execute call will both execute in the same manner as the vulnerable statement execute call and return the same data type.

When we ran our algorithm on one of the same toy Java programs we used for feasibility testing in Section 3.1, we generated the following functionally equivalent code:

```
String PSsql = "select amount from
books where isbn = ?";
PreparedStatement preparedStmt =
conn.prepareStatement(PSsql);
preparedStmt.setString(1,
userISBN);
ResultSet rs =
preparedStmt.executeQuery();
```

The example code illustrates the creation of the PreparedStatement-based SQL statement, modification of the SQL query to include a bind variable in the form of a question mark (*?*), and creation of the set call. Finally, this code shows that the result of the secured SQL statement is the same data type as the result of the vulnerable statement.

3.3. Vulnerability replacement

In this final stage of our algorithm, we either replace the entire vulnerable statement or append the secured statement to the vulnerable statement. We append the secured statement to the vulnerable statement when the database `Connection` object is not in the scope of the execute call or when the vulnerable statement is in a method signature as will be explained in Section 3.4. In addition, if the vulnerable statement is in any detectable method signature then we will not attempt to replace the statement since that would change the API. The vulnerable statement can be secured without modifying the statement creation code, but a full replacement of the plain text SQL statement is preferred to remove redundant objects.

In either of these two cases, we will replace the execute call for the `Statement` with a '`PreparedStatement preparedStmt =Statement.getConnection().prepareStatement(psSQL);`' prepared statement creation call. `Statement` is the actual `Statement` object in the code and `PSsql` is the generated SQL query with bind variables, followed by the rest of the `PreparedStatement`-based solution. The creation call code bypasses the `Statement` and uses it to refer to the proper `Connection` variable and generate the secure `PreparedStatement`-based SQL statement. Nevertheless, if the `Connection` is in scope of the plain text execute call and the `Statement` is not in any detectible method signature, then the `Statement`-based code is replaced with the `PreparedStatement`-based code. Finally, after both cases, we finish and can use the algorithm on other vulnerable SQL statements.

When we ran the algorithm on a sample vulnerable program that had the database `Connection` in scope of the execution call and had the vulnerable statement not within any method signature, we replaced the vulnerable code with the secured code. In the toy vulnerable Java program we used as an example in Sections 3.1 and 3.2, we replaced the code in Section 3.1 with the code generated in Section 3.2. Nevertheless, when we ran the algorithm on a small vulnerable Java program that did not have the `Connection` object in scope of the execute call and had the vulnerable statement in a method signature, we replaced the execution call with the following code:

```
String psSQL = "select amount from
books where isbn = ?";
PreparedStatement preparedStmt =
globalStmt.getConnection().prepare
Statement(psSQL);
preparedStmt.setString(1,
userISBN);
ResultSet rs =
preparedStmt.executeQuery();
```

The example code illustrates the same functionally-equivalent code generated in stage 2 as well as the use of the vulnerable statement to reference the `Connection` variable to create the secured statement.

3.4. Limitations

While we can convert a significant amount of vulnerable SQL statements with our conversion algorithm, we cannot convert batch SQL statements. Batch jobs cannot be handled since the JDBC `PreparedStatement` interface does not currently allow for multiple independent queries in the same batch `PreparedStatement`. In addition, when we cannot trace the input from the execution call to the variables, we cannot change those variables into bind variables. One instance of this traceability issue is when the developer uses a custom method to put the input into the SQL query and executes the new combined string.

Additionally, if the `Connection` is not in scope of the execute method, then the vulnerable statement is generated by some code outside the scope of the algorithm, which we will not attempt to modify. We will not attempt to find or modify the out of scope code since that code may not always be accessible.

4. Formative case study

To develop our approach, we performed a case study in which we evolved our solution generation algorithm. Our case study setup consisted of five vulnerable toy Java programs constructed by the first author with seeded SQLIVs and the required information for the algorithm. Each toy Java program was less than 50 lines of code and was seeded with exactly one vulnerable SQL statement. These toy Java programs are available online[1].

We determined that our algorithm can be tested with six equivalence classes. Three of the classes tested the ability of the algorithm to automate the secure code and three of the classes tested the ability of the algorithm to secure different types of statements. The three automation equivalence classes were:

- ❖ a `Connection` out of scope;
- ❖ `Statement` in method signature; and
- ❖ all objects localized with no method signatures issue.

The three security equivalence classes were:

- ❖ a `select` statement with a vulnerable where clause;
- ❖ a `delete` statement with a vulnerable where clause; and
- ❖ an `insert` statement with a vulnerable values clause

Each automation equivalence class was tested with an individual toy program with one seeded select statement with a vulnerable where clause. Each security equivalence class was tested with an individual toy program with all objects localized with no method signature issues with one seeded statement. We determined that since update and delete statements were both manipulating data inside the database and were vulnerable to the same attacks, we could group those statements into one equivalence class. We used a delete statement for our test of that class. However, we grouped select statements in a separate class since they did not manipulate the database and insert statements in a separate class since they were vulnerable to attacks that input data into the database. We should note that no equivalence classes were created for testing the algorithm when the required objects, SQL query structure, and input variable traceability is not available, since the algorithm exits in those situations.

We tested each program before and after the conversion with a valid input test as well as a SQLIA and collected the results. The valid input test contained data without any SQL characters or keywords and made the program correctly perform the desired tasks. The SQLIA test contained a custom-made SQLIA for each type of statement. The SQLIA for `select` and `delete` SQL queries contained SQL characters and keywords to make the where clause of the query always return true, such as, `asfd' or '1'='1`. The SQLIA for `insert` queries contained SQL characters and keywords to make the values clause contain un-validated data, such as `test6','f','f',5)#`.

We found that the programs had the same results for the valid input test before and after the conversion. However, we found that the programs returned sensitive information or inserted un-validated data for the SQLIA test before the conversion while all programs returned non-sensitive results or inserted validated data after the conversion. These results cannot prove that the generated solutions are secure and the prepared statements do not contain SQLIVs, but they do show five instances of the algorithm stopping a sample attack.

For an example of our test setup and results, our security equivalence class toy program contained the vulnerable insert statement:

```
 insert         into         books
(isbn,name,publisher,amount)
VALUES    ('"    +    userISBN    +
"','asdf','asdf',5)
```

This insert statement was intended to insert a the string for `userISBN` and then insert the default values `'asdf','asdf',5`. When the SQLIA, `test6','f','f',5)#`, was run on the program, the resulting insert statement was:

```
insert         into         books
(isbn,name,publisher,amount)
VALUES
('test6','f','f',5)#','asdf','asd
f',5)"
```

This new statement inserted the values `'f','f',5` instead of the default values and made the default values a comment with the comment character #. We ran this SQLIA on the same program after the conversion, with the insert statement `"insert into books (isbn,name,publisher,amount) VALUES (?,'asdf','asdf',5)"`. The resulting statement used `test6','f','f',5)#` as the isbn and the default values were inserted for the other attributes. This result is positive since the attack was unsuccessful in preventing our program from inserting the default values.

[1] http://www4.ncsu.edu/~smthomas/SESS07_CaseStudyCode.zip

5. Conclusion and future work

In this paper, we presented an automated fix generation solution for replacing vulnerable `Statement`-based SQL statements with secured `PreparedStatement`-based SQL statements. We use a conversion algorithm where we secure a statement without knowledge of the context while maintaining the logic of the statement. The intuition behind this solution is that automated fix generation can be a means for securing vulnerable SQL statements without requiring expertise in software security. In addition, since our solution replaces the vulnerable statement with a secured prepared statement, then our solution removes the SQLIVs and is not prone new types of SQLIAs. Further, our solution is not disadvantaged by the same computational overhead that dynamic statement analysis solutions generate since our solution pre-compiles the SQL statement.

In our future work, we will develop our solution into an Eclipse IDE plug-in that we will use to conduct further tests of our solution. In addition, we are going to refine our conversion algorithm to reduce the cases of vulnerable SQL statements that it cannot handle to a minimal set. We are looking for corporate partners that we could work with to test our solution.

We would also like to note that while this work is initially for Java, the concept and general algorithm could be applied to other languages that meet the environment assumptions as well. We surmise that for any positive results that are gained through this solution in Java, similar results could be gained in other languages.

6. References

[1] S. Barnum, McGraw, G., "Knowledge for Software Security," *Security & Privacy Magazine, IEEE*, vol. 3, no. 2, 2005, pp. 74 - 78.

[2] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks," 5th International Workshop on Software Engineering and Middleware, Lisbon, Portugal, 2005, pp. 106-113.

[3] Eclipsepedia, What is a Quick Fix?, 2006, http://wiki.eclipse.org/index.php/FAQ_What_is_a_Quick_Fix%3F, accessed January 16, 2007.

[4] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, J. Carey, and A. Sundararajan, "The BEA Streaming XQuery Processor," *The VLDB Journal*, vol. 13, no. 3, 2004, pp. 294-315.

[5] P.-Y. Gibello, Zql: A java sql parser, 2006, http://www.experlog.com/gibello/zql/, accessed January 16, 2007.

[6] N. Gupta, A. P. Mathur, and M. L. Soffa, "Automated Test Data Generation Using an Iterative Relaxation Method," 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lake Buena Vista, Florida, United States, 1998, pp. 231-244.

[7] W. G. J. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks," 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, CA, USA, 2005, pp. 174-183.

[8] W. G. J. Halfond and A. Orso, "Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks," 3rd International Workshop on Dynamic Analysis, St. Louis, Missouri, 2005, pp. 1-7.

[9] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Vancouver, BC, CANADA, 2004, pp. 92-106.

[10] NIST, National Vulnerability Database, 2007, http://nvd.nist.gov/, accessed January 16, 2007.

[11] R. Ramler and K. Wolfmaier, "Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost," 2006 International Workshop on Automation of Software Test, Shanghai, China, 2006, pp. 85-91.

[12] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, 2006, pp. 372-382.