Information and Software Technology 51 (2009) 589–598

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

On automated prepared statement generation to remove SQL injection vulnerabilities

Stephen Thomas *, Laurie Williams, Tao Xie

Department of Computer Science, Box 8206, North Carolina State University, Raleigh, NC 27695, USA

ARTICLE INFO

Article history:
Received 18 December 2007
Received in revised form 5 August 2008
Accepted 8 August 2008
Available online 27 September 2008

Keywords:
SQL injection
Prepared statement
Fix automation

ABSTRACT

Since 2002, over 10% of total cyber vulnerabilities were SQL injection vulnerabilities (SQLIVs). This paper presents an algorithm of prepared statement replacement for removing SQLIVs by replacing SQL statements with prepared statements. Prepared statements have a static structure, which prevents SQL injection attacks from changing the logical structure of a prepared statement. We created a prepared statement replacement algorithm and a corresponding tool for automated fix generation. We conducted four case studies of open source projects to evaluate the capability of the algorithm and its automation. The empirical results show that prepared statement code correctly replaced 94% of the SQLIVs in these projects.

1. Introduction

SQL injection vulnerabilities (SQLIVs) accounted for 20% of input validation vulnerabilities and 10% of total cyber vulnerabilities between 2002 and 2007 [25]. SQLIVs can lead to data theft and data corruption, costing time and resources to correct. In 2006, CardSystems Solutions, a credit card processing company, had one SQLIV exploited, causing 40 million credit card numbers to be stolen and millions of dollars of fraudulent purchases [18]. Additionally, one SQLIV was exploited in a help desk system for the University of Missouri, causing theft of information for 22,000 students in 2007 [17].

Structured query language (SQL) is a standard interactive language used with relational databases [1]. An SQL statement is a unit of execution that returns a single result set from a database [1]. A vulnerability is the result of a software defect that gives an attacker unintended access to a computer system [3,5]. An SQLIV is a specific SQL vulnerability that exists when an attacker can insert SQL character and keywords into an SQL statement and change the logic of the SQL statement [1]. SQLIVs allow attacker input to modify SQL structure through an SQL injection attack (SQLIA), which changes the logic of the SQL statement. An attack is a sequence of actions that exploits a vulnerability, typically with devastating consequences [5]. An SQLIA is an attempt to inject SQL characters and/or keywords into an SQL statement's input and modify the statement's structure [9].

Several solutions that mitigate the risk posed by SQLIAs have already been proposed [6,9–13,16,26]. All of these solutions have been successful in mitigating SQLIAs. However, none of these solutions address the actual SQLIVs that exist in the source code. A common way to remove SQLIVs is to separate the SQL structure from the SQL input by using prepared statements. A prepared statement is an SQL statement structure with placeholders for variables [24]. Prepared statements declare the SQL structure explicitly and can remove SQL statements against SQLIVs as long as good coding practices are followed [13]. However, the process of manually converting vulnerable SQL statements to prepared statements is tedious, time consuming, complex, and therefore likely to be error-prone. As a result, automation of this task would be beneficial.

The objective of this research is to propose a prepared statement replacement algorithm and corresponding automation for removing SQL injection vulnerabilities from vulnerable SQL statements by replacing them with secure prepared statements. The source code generation approach to removing the SQLIVs is distinct from approaches that mitigate the risk posed by SQLIAs because the approach removes the SQLIV instead of fortifying against the SQLIA. Additionally, the benefit of automated fix generation is that the generators "deliver a predictable, consistent and repeatable process" and "relieve humans from manually performing tedious and error-prone actions" [2,4]. We created the prepared statement replacement algorithm (PSR-Algorithm), which gathers information from source code containing SQLIVs and generates secure prepared statement code that maintains functional integrity. Correspondingly, we created the Prepared Statement Replacement Generator[1] (PSR-Generator), which

* Corresponding author. Tel.: +1 919 513 4151.
E-mail addresses: stephen.smthomas@gmail.com (S. Thomas), williams@csc.ncsu.edu (L. Williams), xie@csc.ncsu.edu (T. Xie).

[1] The PSR-Generator source code can be found at: http://agile.csc.ncsu.edu/sqlivf/SQLIVF-Generator.zip.

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.08.002

automates the generation of the prepared statement-based code in Java, which results from the PSR-Algorithm. The PSR-Generator was written for Java due to the availability of open source Java projects containing SQLIVs. However, the logic in the algorithm is not limited to any specific language and could be extended to fit the syntax of any language.

We conducted four case studies on open source Java projects using the PSR-Algorithm and the PSR-Generator on each project. The open source Java projects used in the case studies are Nettrust,[2] iTrust,[3] WebGoat,[4] and Roller.[5] We use code inspection and static analysis to find the SQLIVs in each project and used the PSR-Generator to create replacement code for each project. Unit test suites are used to test the security of the replacement code and integrity of the original functionality for each project. The remainder of this paper is organized as follows: Section 2 contains background information on SQLIVs and the proposed solution. Section 3 examines related work. Section 4 describes the logic of the PSR-Algorithm. Section 5 discusses the case studies used to evaluate the PSR-Algorithm, and Section 6 presents the conclusions.

2. Background

This section provides background on SQL injection vulnerabilities, SQL injection attacks, prepared statements, and SQL structure.

2.1. SQL structure

An SQL statement's structure is the SQL code containing the logical purpose of the statement. For example, the SQL statement:

```
SELECT password FROM users WHERE userName = ‘user1’
```

has the structure:

```
SELECT password FROM users WHERE userName =
```

The structure is composed of the `SELECT`, `FROM`, and `WHERE` clauses, which define the purpose of the SQL statement. The SQL statement's structure can contain the following elements:

- clauses, such as `SELECT`, `FROM`, and `WHERE`;
- identifiers, such as table and attribute names; and
- comparators, such as equals (=), `AND`, `OR`, and `LIKE`.

An SQL statement's input is the part of a statement that is expected to change based on user input. In the example SQL statement, `‘user1’` is the SQL input, which is combined with the identifier `WHERE` and the comparator = to change the result of the conditional `WHERE` clause. An SQL statement's input is not considered part of the structure of the statement. SQL input is commonly used in the following places:

- VALUES clause: The new inserted values in an INSERT statement;
- SET clause: What an attribute is set to;
- WHERE clause: The values that the comparators are compared to; and
- ORDER BY clause: The attribute that the result set is ordered by.

2.2. SQL injection attacks (SQLIAs)

Consider the following SQL statement:

```
SELECT password FROM users WHERE userName = ‘’’ + inputUserName + ‘‘’
```

If the value dynamically assigned to the `inputUserName` variable via user input is `111’` OR true#, the SQL statement executed is:

```
SELECT password FROM users WHERE userName = ‘111’ OR true#’
```

When the database executes this SQL statement, the WHERE clause will always be true and the structure of the SQL statement will be interpreted by the relational database management system as:

```
SELECT password FROM users
```

The SQLIA is able to remove the `WHERE` clause from the previous SQL statement since the single quote SQL character ‘ escapes out of the variable `inputUserName`, the keyword OR makes the `WHERE` clause conditional, the keyword `true` makes the conditional always true, and the character # comments out of the rest of the SQL statement. The result of executing the modified SQL statement would contain the entire password column from the users table, regardless of `inputUserName`.

2.3. SQL injection vulnerabilities (SQLIVs)

An SQLIV exists when an SQL statement does not keep statement structure and input separate. An SQL statement is vulnerable to having the logic of the statement changed by input at runtime when the application sends the structure and input of the statement together in a combined request to the database. An SQLIV is caused by dynamic SQL statement construction combined with inadequately-verified input, which allows the input to change the structure and logic of a statement [9,15]. In the SQL statement discussed in Section 2.2, the vulnerable statement concatenates the input `inputUserName` with the statement structure before sending the statement to the database, which allows `inputUserName` to change the `WHERE` clause and the ending of the statement. Additionally, an SQL statement can contain a logical SQLIV if a developer creates a statement with the intent to have the structure of the statement to change based on input. A developer has to change the logic of the SQL statement and limit the range of acceptable SQL structures to remove this type of SQLIV.

2.4. Prepared statements

Prepared statements are SQL statements that separate statement structure from statement input. Prepared statements have a static structure when they are executed and take type-specific input parameters. When prepared statements are created and the statement structure is explicitly set before runtime, the statement structure cannot be changed by input variables and the statement is mitigated from the risk posed by SQLIVs. A prepared statement is "prepared" by declaring the structure of the statement and putting bind variables, placeholders for input, in the places where SQL input goes [15]. The SQL statement structure with the bind variables included is then sent to the database, which compiles and saves the statement structure for future execution with input variables. A prepared statement may look like this:

```
SELECT password FROM users WHERE userName = ?
```

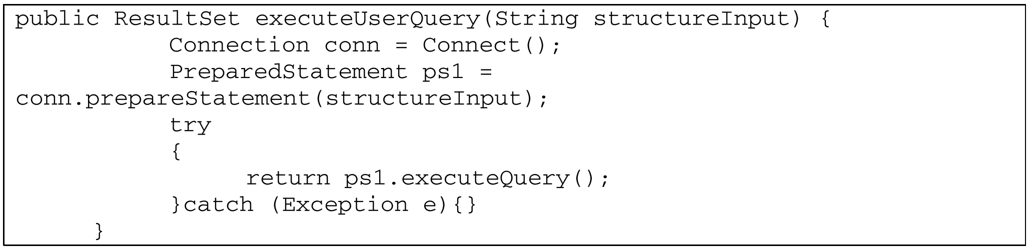
where the question mark (`?`) is the bind variable. A setter method sets a bind variable as well as performs strong type checking and will nullify the effect of invalid characters, such as single quotes in the middle of a string. The setter method, `setString(index, input)`, sets the bind variable in the SQL structure indicated by the `index` to `input`. For example, a call to `setString(1, ‘‘user1’’)` would set the bind variable in the above example to `‘‘user1’’`. Additionally, the setter method `setObject(index, input)` will call the appropriate setter method based on the object type of the input. After the SQL statement has been prepared, one setter method is used per bind variable

[2] Nettrust can be found at: http://code.google.com/p/nettrust/.

[3] iTrust can be found at: http://agile.csc.ncsu.edu/rose/.

[4] WebGoat can be found at: http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.

[5] Roller can be found at: http://rollerweblogger.org/.



```
public ResultSet executeUserQuery(String structureInput) {
            Connection conn = Connect();
            PreparedStatement ps1 =
conn.prepareStatement(structureInput);
            try
            {
                  return ps1.executeQuery();
            }catch (Exception e){}
      }
```

Fig. 1. Example logically vulnerable prepared statement.

to fill the bind variable with input. The static nature of a prepared statement's structure is the characteristic that prevents SQLIVs. Further, since `PreparedStatements`,[6] the Java implementation of prepared statements, can be compiled once and executed multiple times, `PreparedStatements` are used for efficiency as well as security [8].

Although `PreparedStatements`' static structure enables `PreparedStatements` to avoid SQLIVs, `PreparedStatements` can be created which have SQLIVs if they are not developed carefully [22]. If a developer uses input strings as part of the structure of a prepared statement, then the input changes the structure and nature of the statement before it is "prepared" and the `PreparedStatement` would reflect the changes. Fig. 1 shows a logically vulnerable `PreparedStatement` using an insecure prepared statement. The function of the method is to execute a user-specified SQL statement and allows the user to set the structure of the SQL statement. Therefore, even though the SQL structure is "prepared" before execution, the input string `structureInput` changes the structure of the statement at runtime.

3. Related work

This section presents other existing solutions to mitigate the risk posed by SQLIAs.

3.1. SQLIA risk mitigation solutions

Su and Wassermann [26] propose a solution to mitigate the risk posed by SQLIAs which involves performing static analysis of an SQL statement's parse tree, generating custom input validation code, and wrapping the statement in the validation code. Su and Wassermann create an SQL parse tree by parsing an SQL statement into its grammatical parts and arranging it into a tree to reveal the structure of the statement. They use the revealed structure to determine how to filter input and generate the input validation code. They conducted a study using five open-source Web application projects on GotoCode.com, the same five projects used by Halfond and Orso [9], and applied their wrapper, SQLCHECK, to each application. They found that their wrapper stopped all 18,424 SQLIAs in their attack suite without generating any false positives. While their wrapper was effective in preventing their SQLIA attack suite, they noted that their attack suite was created by independent researchers, and it may not contain all possible attacks. The parse tree approach is effective in identifying the structure of the SQL statement and using structure comparisons to detect potential SQLIAs. However, the parse tree approach focuses on the structure of the attacks instead of the removal of the SQLIVs.

Huang et al. [16] secure potential vulnerabilities by combining static analysis with runtime monitoring. Their solution, WebSSARI, statically analyzes source code, finds potential vulnerabilities, including SQLIVs, and inserts runtime guards into the source code, which sanitizes input. They conducted a study using 230 open-source Web application projects on SourceForge.net and applied their solution, WebSSARI, to each application. They found security vulnerabilities in every application. The WebSSARI approach is effective in preventing general input manipulation attacks through sanitizing input. However, the solution relies on white and black listing input, instead of removing the vulnerability.

Buehrer et al. [6] secure vulnerable SQL statements by comparing the parse tree of a statement at runtime with the parse tree of the original statement and allow a statement to execute only if the parse trees match. They conducted a study using one student-created web application and applied their solution, SQLGuard, to the application. They found that their solution stopped the four SQLIA types described in their paper without generating any false positives. The parse tree approach, the same approach Su and Wassermann use, is effective in identifying the SQL statement structure and detecting when the structure has been changed by SQLIAs. However, SQLGuard does add a computational overhead of dynamic SQL statement validation and also uses white and black listing, similar to Huang et al.

Halfond and Orso [9–13] secure vulnerable SQL statements by combining static analysis with statement generation and runtime monitoring. Their solution, AMNESIA, analyzes a vulnerable SQL statement, generates a generalized statement structure model for the statement, and allows or denies each statement based on how it compares to the model at runtime. Their solution throws an exception for each SQLIA, which the developer handles and builds in attack recovery logic. They conducted a study using five open-source Web application projects on GotoCode.com and two student-created web applications, and applied their solution, AMNESIA, to each application. They found that their solution stopped all of the SQLIAs in their attack set, a set ranging from 140 to 280 elements for each application, without generating any false positives. Their model generation and runtime comparison approach is effective at detecting SQLIAs and does a comparison similar to Buehrer et al.'s SQLGuard. AMNESIA, like SQLGuard, also adds a computational overhead by including an additional process that has to be integrated into the runtime environment. Additionally, AMNESIA adds the capability for developers to add logic to how SQLIAs are handled, by throwing an exception in the vulnerable code.

The solutions discussed in this section have mitigated the risk posed by SQLIAs by cleansing input before it is put into vulnerable SQL statements or by runtime monitoring of potentially compromised SQL statements [6,9–13,16,26]. These solutions have had positive results in stopping the attack methods known at the time of the case studies. These solutions also are approaches that we used to build our solution. We separate the SQL structure from the rest of the SQL statement, the same way that the parse tree approach creates a tree of the SQL structure and compares structures to find SQLIAs. Additionally, we created our solution to work within the existing code, similar to SQLCheck and AMNESIA. Like SQLCheck and AMNESIA, we allow our solution to be extended by developers,

[6] `PreparedStatements` can be found at: http://java.sun.com/j2se/1.4.2/docs/api/java/sql/PreparedStatement.html.

by inserting our solution into the vulnerable code. However, these approaches are designed to *fortify against* SQLIAs while our proposed solution is designed to preventively *remove* the SQLIVs.

4. Prepared statement replacement algorithm

In this section, we provide the PSR-Algorithm details, the PSR-Generator details, and the limitations of the PSR-Algorithm.

4.1. Algorithm-generated code structure

The PSR-Algorithm is targeted to the environment where existing source code contains SQLIVs that need to be removed. The PSR-Algorithm analyzes source code containing SQLIVs and generates a specific recommended code structure containing prepared statements. The PSR-Algorithm separates the SQL statement's input from the SQL structure in the generated code structure. The PSR-Algorithm creates an additional string object for each string object used to create the SQL statement. The new string object contains the raw string data of the original string object and any identifiers found in the original string object the PSR-Algorithm identifies as SQL structure. The PSR-Algorithm creates an assistant vector for each new string object. The assistant vector is created to contain any SQL input found in the original string object. The PSR-Algorithm-generated string objects can contain other PSR-Algorithm-generated string objects based on how the original string objects are used. Therefore, assistant vectors can contain other assistant vectors, creating a tree. The significance of the assistant vector tree is that it can branch based on conditionals, which makes the tree contain the proper variables for each decision path of the conditional.

When the PSR-Algorithm generates a new string object and assistant vector for a string object used to create the SQL statement, the PSR-Algorithm declares and assigns the new string object and assistant vector in each code location the original string object is declared and assigned. The PSR-Algorithm declares and assigns the new string object and assistant vector in these locations to make the new string object and assistant vector have the same decision paths for conditional code as the original string object. Therefore, the assistant vector tree changes dynamically at runtime and ensures that the SQL input elements are in the proper order. Figs. 2 and 3 show an example of PSR-Algorithm-generated code and tree structure, depicting the ability of the code to change based on conditionals and match the structure of the original statement. The PSR-Algorithm-generated code in Fig. 2 keeps the original code intact, which is in bold in the example, and adds the new code structure around the existing code. Fig. 3 shows an example of how the tree structure of the generated code can change based on conditionals, where Tree1 is created when the conditional is true and Tree2 is created when the conditional is false.

Once the PSR-Algorithm creates and assigns all of the string and vector objects, the algorithm creates a prepared statement using

```
Vectors vector1, vector2, and vector3 are created as new Vectors
String originalString2 is set to SQLInput2_1
String string2 is set to "?" #comment: "?" is a bind variable
SQLInput2_1 is added to Vector vector2
String originalString3 is set to SQLInput3_1 concatenated with " "
 concatenated with SQLInput3_2
String string3 is set to "? ?"
SQLInput3_1 and then SQLInput3_2 are added to Vector vector3
String string1 is declared
String originalString1 is declared
if conditional is true
    originalString1 is set to "SELECT * FROM Table1 where value = "
     concatenated with SQLInput1_1 concatenated with
     originalString2 concatenated with SQLInput1_2
    string1 is set to "SELECT * FROM Table1 where value = "
     concatenated with "?" concatenated with string2 concatenated
     with "?"
    SQLInput1_1, vector2, and then SQLInput1_2 are added to vector1
else
    originalString1 is set to "SELECT * FROM Table1 where value = "
     concatenated with SQLInput1_3 concatenated with originalString3
    string1 is set to "SELECT * FROM Table1 where value = "
     concatenated with "?" concatenated with string3
    SQLInput1_3 and then vector3 are added to vector1
endif
Statement stmt is created from a Connection to the database
 executing originalString1 #comment: this execution is removed in
 the final code
Prepared statement psStmt is created from stmt's Connection using
 string1 as the structure of the SQL statement
Vector returnVector is declared and set to the return vector of a
 call to the traverseInputTree on vector1
for each element in returnVector
   set the next psStmt bind variable to the element
endfor
execute psStatement, replace stmt result set with psStmt result set
```

Fig. 2. Example PSR-Algorithm-generated code.

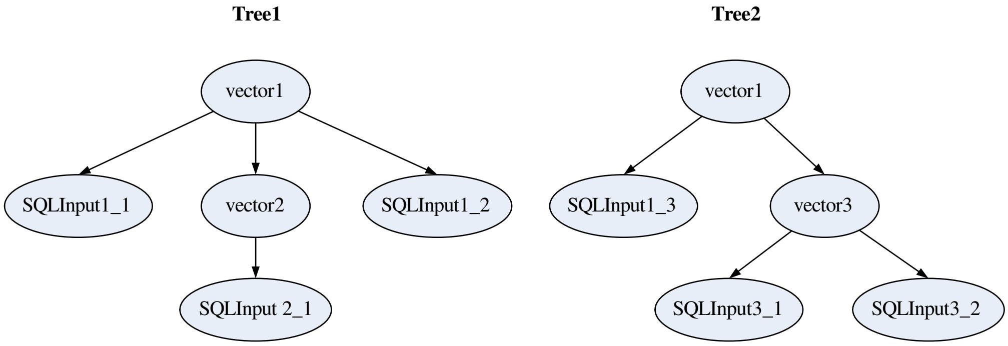


Fig. 3. Conditional tree structure.

the new string objects that contain only SQL structure. Additionally, as shown in Fig. 2, the PSR-Algorithm-generated code does an in-order transversal of the tree and places all of the elements into a single vector in the order that they are traversed. Since the tree exists at runtime in the executing code, the PSR-Algorithm creates a recursive method that the PSR-Algorithm places into the source code under analysis. The method traverses the tree and returns a vector with the elements in the traversed order.

Once the PSR-Algorithm-generated code has all of the elements in a singular vector, the PSR-Algorithm inserts code that loops through the vector and sets each SQL input element to the proper bind variable. The PSR-Algorithm-generated code then executes the prepared statement and replaces the SQLIV execution. Fig. 2 shows the final execution replacement.

4.2. Algorithm details

The PSR-Algorithm uses the objects involved in the SQLIV to create the prepared statement. The required objects include the execution method, the string objects containing the SQL statement, and the `Connection` or `Statement` object. The PSR-Algorithm cannot work without these objects.

The PSR-Algorithm starts separating the SQL statement structure from input by iterating through each string object used in the SQL structure and creating a new string object and a new vector object for each existing string object. The PSR-Algorithm parses each existing string object into raw string data, and identifiers. The PSR-Algorithm leaves raw string data and any guaranteed secure identifiers in the new string object as structure. A guaranteed secure identifier is an identifier the developer determines is secure through manual static analysis. The PSR-Algorithm allows all guaranteed secure identifiers to be part of the SQL structure. Of the remaining identifiers, the PSR-Algorithm identifies all non-string identifiers, assumes they are SQL input, puts them into the assistant vector, and replaces each identifier with a bind variable. The PSR-Algorithm determines if any of the remaining string identifiers have already had string and vector pairs made for them. If so, the PSR-Algorithm replaces the existing identifier with the PSR-Algorithm-created identifier and puts the assistant vector into the current assistant vector.

For all of the string identifiers that the PSR-Algorithm has not converted yet, the PSR-Algorithm recursively repeats the new string and vector creation process until all string objects have associated PSR-Algorithm-generated string and vector objects. Additionally, the PSR-Algorithm recursively repeats the assignment of the new string and vector pairs for each assign of the existing string object. Fig. 2 shows an example of the type of string and vector objects the PSR-Algorithm creates.

With the SQL input separated from the SQL structure via the new string and vector objects, the PSR-Algorithm creates a `PreparedStatement` from the new SQL structure string objects. Further, the PSR-Algorithm creates a call to the `traverseInputTree` method, which does an in-order traversal of the assistant vector tree and returns a single vector with the elements in the proper order. The PSR-Algorithm then creates a loop that goes through the single assistant vector and assigns each element to each bind variable in order. The loop assigns each element to a bind variable using the setter method `setObject(index, input)`, which calls the appropriate setter method based on the object type of the input. The PSR-Algorithm creates a call to the `execute` method of the `PreparedStatement` and replaces the call to the Statement execute method with a call to the `PreparedStatement` `execute` method. Fig. 4 shows an example of the PSR-Algorithm preparing the `PreparedStatement`, setting the bind variables to the appropriate values, and replacing the SQLIV with the `PreparedStatement` execute. Fig. 4 shows the `PreparedStatement` `ps0` created by `stmt`'s `Connection`, set to the SQL structure string `PSqueryUniqueID0`, the bind variables set to the proper inputs, and `ps0`'s execution inserted into the `if` statement that the SQLIV was in. Fig. 4 shows the original line of code in the bottom cell of the figure.

The PSR-Algorithm inserts the `traverseInputTree` method into the source code since the tree grows dynamically at runtime and needs to be traversed at runtime. The PSR-Algorithm finishes after the SQLIV execution is replaced.

4.3. The PSR-Algorithm implementation

We have implemented the PSR-Algorithm as an automated fix generator in Java called the PSR-Generator. We implemented the PSR-Algorithm in Java due to the availability of open source Java projects containing SQLIVs. The PSR-Generator takes in a Java file and line numbers of SQLIV method calls and outputs a Java source file with the PSR-Algorithm-generated code structure included and the SQLIV method calls removed. The first step of the process of securing vulnerable source code with the PSR-Generator starts with formatting the source code in preparation for the generator. A developer imports the source code into an Eclipse[7] Integrated Development Environment (IDE) project and uses the Eclipse IDE Formatter, a feature of the Eclipse IDE, to make sure that the source code was organized in a standardized way that makes analyzing the code a regular and repeatable process.[8]

After the Eclipse IDE Formatter formats the code, static analysis and code inspection is conducted to find the SQLIVs. A developer can use the FindBugs™ static analyzer [14] Eclipse IDE plugin and/or code inspection to analyze the source code and find the line

[7] Eclipse can be found at: http://www.eclipse.org.

[8] The details of the Eclipse IDE Formatter settings can be found at: http://www4.ncsu.edu/~smthomas/EclipseFormatterSettings.xml.

```
String PSqueryUniqueID0;
 java.util.Vector PSInput00 = new java.util.Vector();
{
PSInput00 = new java.util.Vector();
PSInput00.add(HCPID);
PSInput00.add(loggedInUser.getMID());
PSqueryUniqueID0 =  "SELECT * FROM DeclaredHCP WHERE HCPID =  " +"?"
+ "  AND PatientID =  " +"?" + " " ;
}
java.util.Vector returnVector0 = new java.util.Vector();
traverseInputTree(PSInput00, returnVector0);
java.sql.PreparedStatement ps0 =
stmt.getConnection().prepareStatement(PSqueryUniqueID0);
for(int i = 0; i < returnVector0.size(); i++){
ps0.setObject((i + 1), returnVector0.get(i));
}
if ( ps0.executeQuery().isBeforeFirst() )
```

```
   if ( stmt.executeQuery( "SELECT * FROM DeclaredHCP WHERE HCPID =
'" + HCPID + "' AND PatientID = '" + loggedInUser.getMID() + "'"
).isBeforeFirst() )
```

Fig. 4. The PSR-Algorithm-generated `PreparedStatement` preparation and execution code inserted into existing code.

numbers of all SQLIVs in the source code. Once all of the SQLIV line numbers have been collected, a developer runs the PSR-Generator on each Java file that has SQLIVs. For each vulnerable Java file, the developer passes the generator the path to the Java file, all of the SQLIV line numbers in ascending order, and any secure reference IDs. The PSR-Generator creates a converted Java file with the SQLIV execution calls removed. Once the PSR-Generator has converted all of the vulnerable Java files, the vulnerable files are replaced with the converted files and the before and after projects can be analyzed.

4.4. Assumptions and limitations

The PSR-Algorithm and PSR-Generator assumptions and limitations are discussed in this section.

4.5. Code analysis assumptions and limitations

The PSR-Algorithm assumes the language, database connector, and database all support prepared statements. The PSR-Generator only analyzes and creates Java code. Additionally, the PSR-Generator is limited to the knowledge of a single file and the PSR-Algorithm only considers variables, methods, and method calls that can be found in a single file. The PSR-Generator's analysis of the source code relies solely on pattern matching and does not take into consideration call graphs, Abstract Syntax Trees, or other advanced code analysis. Since the PSR-Generator uses pattern matching, the PSR-Generator assumes that the developer removes all non-compiled parts of the code such as comments or documentation before the PSR-Generator converts the file.

4.6. Language specific assumptions and limitations

Additionally, since the PSR-Generator is limited to Java, the proposed solution is further limited since Java currently does not have the `PreparedStatement` equivalent of the `Statement` batch job functionality. While `PreparedStatements` currently allow multiple batches of input for a single prepared statement structure, `PreparedStatements` do not have the ability to specify multiple statement structures, like `Statements` do. Creating this functionality is additionally difficult since `Statement` batch jobs are dependent on the database type. `Statement` batch jobs determine whether to exit after one of the SQL statements causes an error in the database or to continue despite errors based on the database type, which is a decision that the solution cannot mimic. Therefore, `PreparedStatements` cannot be used to create functionally equivalent code for `Statement` batch job code. However, we did not encounter batch jobs in any of the four case studies we conducted.

The PSR-Algorithm assumes that all SQL structure is contained within string objects as raw string data. Further, the PSR-Algorithm assumes the code sets all of the SQL structure `String`s explicitly in the file. Additionally, since the PSR-Algorithm-generated code creates and inserts duplicate prepared statement code for the SQLIV code without removing the SQLIV code, the PSR-Algorithm-generated code is unable to handle iterator data structures, or any data structures that shift pointers when getting data. However, we only encountered SQLIVs with iterator data structures in three of the 56 SQLIVs discovered. Table 1 summarizes the PSR-Algorithm and the PSR-Generator code challenges.

5. Empirical studies

We evaluated the PSR-Algorithm with four empirical studies of open source projects. The PSR-Algorithm and the PSR-Generator removed 94% of the SQLIVs of the case studies. The 6% of the SQLIVs not removed from the case studies are SQLIVs that are

Table 1
Code challenges for the PSR-Algorithm

SQLIV code challenge	Success
Static SQLIV structure	✓
Complex dynamic SQLIV structure	✓
Conditional SQL input	✓
Java file scope knowledge	✓
Manually determines Java objects	✓
Manually separates SQL structure/input	✓
Standardized treatment of all SQL structure types	✓
Standardized treatment of conditional and non-conditional code	✓
Instance variables containing SQL structure	✓
SQL Structure containing Strings declared and assigned in any order	✓
Batch queries	☒
Non-explicit setting of SQL structure	☒
Non-string containing SQL structure	☒
Project level knowledge SQL structure	☒
Iterator data structure code	☒

Table 2
SQLIA type code examples

SQLIA type	Code	Example
Conditional where	Vulnerability	`‘‘SELECT HCPID FROM DeclaredHCP WHERE PatientID =’’’ + loggedInUser.getMID() + ‘‘’’’`
	Attack	3’ or true#
Additional column insert	Vulnerability	`‘‘INSERT INTO TransactionFailureAttempts (medicalID, failureCount) VALUES (’’ + id + ‘‘, 0);’’`
	Attack	`4, 800)#`
Additional row insert	Vulnerability	`‘‘INSERT INTO OfficeVisits (notes, HCPID, PatientID, visitDate) VALUES (’’’ + notes.replace(‘‘’’’, ‘‘’’’’) + ‘‘’,’’’ + hcpid + ‘‘’,’’’ + mid + ‘‘’,’’’ + visitDate + ‘‘’);’’`
	Attack	`0000-00-00’), (‘patient died’, ‘2’, ‘2’, ‘0000-00-00’)#`
Conditional order keyword	Vulnerability	`getBroadCasterName.executeQuery(‘‘select BroadCasterName, broadcasters.BroadCasterID from broadcasters,broadcasternymmap where broadcasternymmap.BroadCasterID = broadcasters.BroadCasterID and NymID=’’’ + strNymID + ‘‘’ order by’’ + strCriteria +’’ desc’’);`
	Attack	`broadcastername asc#`

known limitations noted in Table 1 for the PSR-Algorithm and PSR-Generator. This section describes each case study and the tools used in each study. To analyze the security of the PSR-Algorithm, unit-level test suites were created for each case study. Specific methods within the study projects were tested for security, the secure execution of SQLIAs, functional integrity, and the expected execution results of normal data.

5.1. SQLIA types

Throughout each case study, a variety of SQLIAs were used to exploit SQLIVs to analyze the effectiveness of the `PreparedStatement`-based code. Each SQLIV in all four case studies was exploited through one of four types of SQLIAs, selected based on static analysis of the SQLIV, as will be explained: conditional where, additional column insert, additional row insert, and additional order keyword. The SQLIA types are based on Chris Anley's SQLIA types [1]. For testing, we considered each of these types as an equivalence class and ensured we had coverage of each of these types. Table 2 shows an example of each SQLIA type and an example SQLIV that is exploitable by that type.

The **conditional where** attack inserts an attack string into an SQL statement `WHERE` clause, which turns the conditional of the `WHERE` clause into an always true. The vulnerable SQL statement in Table 2 puts the return of loggedInUser.getMID() directly into a `WHERE` clause for the MID attribute. The SQLIA in Table 2 turns the `WHERE` clause of the SQL statement to always be true.

The **additional column insert** attack inserts an attack string into an SQL statement `VALUES` clause that sets the value for the attribute as well as the following attribute(s). The vulnerable statement in Table 2 puts the `id` method variable directly into a `VALUES` clause for the `id` attribute. The SQLIA in Table 2 sets the `medicalID` to 4 and the `failureCount` to 800, as opposed to the intended 0, which injects invalid data.

The **additional row insert** attack inserts an attack string into an SQL statement `VALUES` clause that finishes the set of values, and adds an additional set of values to be inserted as a second row in the table. The vulnerable SQL statement in Table 2 puts the value `visitDate` directly into a `VALUES` clause that sets the value for the attribute `visitDate`. The SQLIA in Table 2 sets the `visitDate` to `‘0000-00-00’`, exits out of the `visitDate`, and adds the additional value set `(‘patient died’, ‘2’, ‘2’, ‘0000-00-00’)` and then exits out of the rest of the SQL statement. The SQL statement inserts both value sets into the table.

The **conditional order keyword** attack inserts an attack string into an SQL statement `ORDER BY` clause that sets the value for the attribute and adds an order keyword to the SQL statement. The vulnerable SQL statement in Table 2 puts the value `strCriteria` directly into the `ORDER BY` clause that sets the attribute with which to order by in descending order. The vulnerable SQL statement in Table 2 sets the attribute to order by to `broadcastername` and adds the order keyword `asc`, which changes the order to ascending and then exits out of the rest of the SQL statement. The SQL statement returns the values in ascending order instead of the descending order of the original statement.

Table 3
SQLIVs vulnerable to SQLIA types by project

SQLIA type	Number SQLIVs vulnerable to each SQLIA type:			
	Nettrust	iTrust	WebGoat	Roller
Conditional where SQLIAs	14	19	6	1
Additional column	3	1	0	0
Additional row	2	4	1	0
Conditional order	1	0	0	0

Table 4
Case study projects details

Project	Version	Lines of code	Classes	SQLIVs	SQLIAs	Converted classes
Nettrust[a]	Fortify Review	1,603	11	31 (20 converted)	20	10
iTrust[b]	Fall 2006	2,213	18	24	24	4
WebGoat[c]	0.9	19,440	77	7	7	5
Roller[d]	0.9.9	52,089	276	6 (2 converted 1 exploitable)	1	1

[a] http://www4.ncsu.edu/~smthomas/Nettrust_versions.zip.
[b] http://www4.ncsu.edu/~smthomas/iTrust_versions.zip.
[c] http://arches.csc.ncsu.edu/smthomas/roller_versions.zip.
[d] http://arches.csc.ncsu.edu/smthomas/roller_versions.zip.

Table 3 shows the number of SQLIVs in each case study project that was vulnerable to each type of the SQLIA types.

5.2. Evaluative empirical studies

To evaluate the security and integrity aspects of the PSR-Algorithm-generated code, we assessed the results of the PSR-Algorithm on four open source Java applications. Statistics and relevant details for each project can be seen in Table 4. Study 1 was an evaluative empirical study on the Java server code of a trust management project Nettrust. A purpose of Nettrust is to use social trust between collaborators to create online trust. Therefore, security is an important requirement of the system. Study 2 was an evaluative empirical study using a role-based, web-based, open-source healthcare application iTrust. iTrust catalogues sensitive medical information, which makes security a high priority. Study 3 was an evaluative empirical study using the open source security teaching Java web application WebGoat, a project from the Stanford SecuriBench[9] [19–23]. Study 4 was an evaluative empirical

[9] SecuriBench can be found at: http://suif.stanford.edu/~livshits/securibench/.

study using the open source Java blog server Roller, also a project from the Stanford SecuriBench [19–23]. The original and converted versions of each project can be found online, as noted in Table 4. WebGoat and Roller are part of the SecuriBench [19–23] set of test projects, along with Blojsom, BlueBlog, jboard, Pebble, PersonalBlog, and SnipSnap. We used code inspection and static analysis to find the SQLIVs in the entire SecuriBench [19–23] set. However, we determined that only WebGoat and Roller had the vulnerabilities that we needed to test the PSR-Algorithm. Each case study is reviewed in more detail in the subsequent sections. Additionally, each test project was tested with SQLIAs from each of the different SQLIA types.

5.2.1. Study setups

As mentioned in Table 4, the version of Nettrust used for Study 1 is the version of Nettrust Fortify Software used in the Java Open Review Project.[10] Fortify Software analyzed Nettrust and published Nettrust as having 30 defects, with SQLIVs being a majority of defects in Nettrust. A similar analysis of Nettrust was conducted in Study 1 to find the SQLIVs found by Fortify Software and see if the SQLIVs could be removed.

The version of iTrust used for Study 2 is iTrust 2.0. The input filter code was removed from iTrust to make security validation of iTrust more obvious. The input filter code would make sure that the input was in the proper format (e.g., between 8 and 20 characters) before it would enter the application. The input filter code would have restricted the SQLIAs to more complex and harder-to-understand attacks to bypass the restrictions to expose the SQLIVs. Since the goal of the testing was to expose the SQLIVs, regardless of whether they are latent or not, and compare the security of the pre- and post-algorithm versions of iTrust, the case study did not benefit from the input filter code existing in iTrust.

The version of WebGoat used for Study 3 is version 0.9, found in the Stanford SecuriBench [19–23] version 0.91a. The version of Roller used for the case study is version 0.9.9, found in the Stanford SecuriBench [19–23] version 0.91a.

The PSR-Generator was used to convert all projects. Therefore, each project was imported into the Eclipse IDE and the Eclipse IDE Formatter was run. The FindBugs™ [14] Eclipse IDE plugin was then run on each project to identify SQLIVs. The plugin found 31 SQLIVs in Nettrust, 24 SQLIVs in iTrust, 7 SQLIVs in WebGoat, and 6 SQLIVs in Roller. Each SQLIV was confirmed through code inspection.

Only two of the six SQLIVs in Roller were convertible, while all of the SQLIVs in Nettrust, iTrust, and WebGoat were convertible. Three of the four remaining SQLIVs used an iterator as part of the code and the PSR-Algorithm is unable to create logically equivalent prepared statement code for code containing an iterator. Further, one of the remaining SQLIVs contained a logical SQLIV and the PSR-Algorithm is unable to create equivalent code for SQLIV code that is logically vulnerable. Additionally, of the two SQLIVs that were determined to be convertible, one of the two SQLIVs contained a logical vulnerability as well as an implementation vulnerability. A logical vulnerability exists when the logic of the code intentionally allowed input to determine part of the SQL structure. Therefore, only the implementation vulnerability was removed from the SQLIV and the logical vulnerability was ignored, since the logic of the code would have to be changed before the logical vulnerability could be removed.

The PSR-Generator was run on each SQLIV in each of the files containing SQLIVs in each of the projects.

5.2.2. Functional integrity and security unit testing

A unit test suite was created for the purpose of this research for each project to test the integrity of the functionality of each converted project. A unit test was created for each SQLIV that the PSR-Algorithm converted to use prepared statements. The unit test suite each study had 100% coverage of the SQLIVs, which is the only code modified by the PSR-Algorithm. The JUnit[11] [7] testing framework was used for all unit testing to make the unit testing repeatable. JUnit [7] is a framework that allows developers to create and execute a repeatable unit test suite.

Additionally, a JUnit [7] SQLIA test suite was created for each project to test the security of each converted project compared to each original project. The security unit test suites contained one SQLIA per SQLIV, to verify that each vulnerable method call was exploitable. The SQLIA test suites were created by analyzing each vulnerable SQL statement and determining which SQLIA type could be use to exploit the statement.

As Table 4 shows, only 20 of the 31 SQLIVs were determined to be directly exploitable through attack string input. The other 11 SQLIVs, while vulnerable, were not directly exploitable through the basic SQLIA techniques known. The 11 SQLIVs were nested inside of code where no input variables could reach the SQLIVs and test the SQLIVs for security. However, if Nettrust was refactored, then the SQLIVs could be tested for security. Additionally, only one of the two convertible SQLIVs in Roller was determined to be exploitable. The other SQLIV, while vulnerable, was not exploitable through direct input SQLIAs.

When the functional integrity unit test suites were run on both the original and converted projects for each study, both the original and converted projects returned the same results for all tests. All functional integrity unit test suites had 100% coverage of the SQLIV code, which is the only code modified by the PSR-Algorithm. Therefore, the replacement code in the converted project has functional integrity.

When the SQLIA test suites were run on each original project, all SQLIVs in each project were exploited. When the SQLIA test suites were run on each converted project, none of the SQLIAs were successful. The results of the SQLIA test suites show that each of the SQLIVs was vulnerable. The result that none of the SQLIAs was successful when being run on the converted projects demonstrates that the PreparedStatements prevented SQL characters and keywords from modifying the prepared statement structures. The extent that the SQLIA test suites can validate the security of the converted projects is that the PSR-Algorithm removed only the tested vulnerabilities that were in the system. Table 5 shows the overall test results of the studies, including the functional integrity and security test results for both the original and converted projects.

5.2.3. SQL statement logical equivalency

To test the equivalency of the PSR-Algorithm-generated code, the queries executed by the MySQL[12] database were logged for the functional integrity test suite of both the original and converted Nettrust, and the queries were compared. The queries executed by both projects were equivalent with the exception of two distinctions. The developers of Nettrust created the SQL statement:

```
‘‘delete from nyminfo where nyminfo.NymID=’’’ + strNymID + ‘‘’ and BuddyID=’’’ + strBuddyID + ‘‘’’’’
```

which takes in the string strBuddyID for setting the BuddyID in the statement. However, when strBuddyID is null, the SQL statement makes the null string a ‘null’ while Java’s PreparedStatement, when taking in a null string, sets the input to the SQL value NULL. Java’s PreparedStatement can set a

[10] Fortify Software’s Java Open Review Project can be found at: http://opensource.fortifysoftware.com/.

[11] JUnit can be found at: http://www.junit.org.

[12] MySQL can be found at: http://www.mysql.com/.

Table 5
Summary of the results of the evaluative empirical studies

Project	Test suite	Original project	Converted project
Nettrust	Equivalency	Passed	Passed
	Security	20 attacks successful	No attacks successful
iTrust	Equivalency	Passed	Passed
	Security	24 attacks successful	No attacks successful
WebGoat	Equivalency	Passed	Passed
	Security	7 attacks successful	No attacks successful
Roller	Equivalency	Passed	Passed
	Security	1 attack successful	Attack not successful

'`null`' as the `PreparedStatement` variable if `strBuddyID` is passed in as a '`null`'. Additionally, when Java's `PreparedStatement` takes in a float that has the decimal value of zero, Java's `PreparedStatement` will round the float to the integer while Java's `Statement` leaves the .`0` on the float. These two distinctions were found during the analysis of the logical equivalency of the Nettrust queries.

5.2.4. Stanford SecuriBench analysis

To evaluate how well the solution presented in this paper works on a well-known set of security test projects, we analyzed the Stanford SecuriBench [19–23] to determine how the solution could secure the set of test projects. The Stanford SecuriBench [19–23] analysis is distinct from Study 1 through Study 4 since we were able to analyze the results of the PSR-Algorithm in Study 1 through Study 4 and the Stanford SecuriBench [19–23] analysis was only intended to analyze the usability of the PSR-Algorithm. The SecuriBench [19–23] project set consists of eight projects: Blojsom, BlueBlog, jboard, Pebble, PersonalBlog, Roller, SnipSnap, and WebGoat. Each test project was imported into the Eclipse IDE and the FindBugs™ [14] Eclipse IDE plugin was run on each test project to identify SQLIVs. Five of the test projects: BlueBlog, Blojsom, Pebble, PersonalBlog, and SnipSnap, contained no SQLIVs. The project jboard contained three SQLIVs; however, the SQLIVs were not convertible since the SQLIVs used non-`String` Objects to combine both the SQL structure and SQL input, which breaks the algorithm's assumption that all SQL structure will be contained in raw String data in `String`s. Therefore, including the SQLIVs from Study 3 and Study 4, 9 out of the 16 SQLIVs in the Stanford SecuriBench [19–23] set of projects were removed using the PSR-Algorithm. Four of the SQLIVs were unconvertible because they were logic vulnerabilities, while three of the SQLIVs were unconvertible because they were outside the assumptions and limitations of the PSR-Algorithm.

6. Conclusion

In this paper, we presented the problem of SQLIVs and the unique solution of using a prepared statement replacement algorithm to remove the vulnerabilities. We also presented the algorithm details and the logic of the generated code. We conducted four case studies to evaluate the algorithm and the conclusions drawn from each study are also presented. Additionally, we presented the automated fix generator, the PSR-Generator, used to implement the PSR-Algorithm, and the process of using the generator to convert vulnerable source code files, successfully converting 94% of the SQLIVs found in 20 files.

The main benefit of the PSR-Algorithm outlined in this paper is that the SQLIVs are removed with minimal manual intervention. The PSR-Algorithm only has to be used once to remove the SQLIV and does not have to be integrated into the runtime environment, unlike several of the existing solutions [4,6,7]. The PSR-Algorithm has the benefit of consuming a small footprint: the generated prepared statement code inserted into the existing source code. Additionally, the solution is a systematic way of removing SQLIVs. Finally, the prepared statement generated code creates equivalent queries for standard data as the original SQLIVs, except for the two distinctions noted in Section 5.2.3. Further, the solution was developed to remove the method call producing the SQLIV, not modify or remove any other lines of code, and inject the generated code into the existing code. The PSR-Algorithm could be expanded beyond the current solution while implementing the solution in multiple languages. Additionally, any positive results gained through the solution in Java, we expect that other languages could gain similar results. The PSR-Algorithm also benefits from automated fix generation as a way of implementing the prepared statement replacement algorithm.

Acknowledgements

We would like to thank Rada Chirkova and Michael Gegick for their contributions to this research. We would also like to thank the NCSU Software Engineering Realsearch group for their careful review and helpful suggestions for the paper. Funding for this research was provided by the North Carolina State Center for Advanced Computers and Communications (CACC) and the National Science Foundation.

References

[1] C. Anley, Advanced SQL Injection in SQL Server Applications, 2002, <http://www.ngssoftware.com/papers/advanced_sql_injection.pdf>, accessed January 21, 2007.
[2] N. Audsley, I. Bate, S. Crook-Dawkins, Automatic code generation for airborne systems, in: IEEE Aerospace Conference, New York, NY, 2003, pp. 6_2863–6_2873.
[3] S. Barnum, G. McGraw, Knowledge for software security, Security and Privacy Magazine, IEEE 3 (2) (2005) 74–78.
[4] M. Bordin, T. Vardanega, Real-time Java from an automated code generation perspective, in: International Workshop on Java Technologies for Real-Time and Embedded Systems, Vienna, Austria, 2007, pp. 63–72.
[5] R.E. Bryant, S. Jha, T.W. Reps, S.A. Seshia, V. Ganapathy, Automatic discovery of API-level exploits, in: 27th International Conference on Software Engineering (ICSE'05), St. Louis, MO, 2005, pp. 312–321.
[6] G. Buehrer, B.W. Weide, P.A.G. Sivilotti, Using parse tree validation to prevent SQL injection attacks, in: 5th International Workshop on Software Engineering and Middleware, Lisbon, Portugal, 2005, pp. 106–113.
[7] Y. Cheon, G.T. Leavens, A simple and practical approach to unit testing: the JML and JUnit way, in: 16th European Conference on Object-Oriented Programming, Spain, 2002, p. 29.
[8] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, J. Carey, A. Sundararajan, The BEA streaming XQuery processor, The VLDB Journal 13 (3) (2004) 294–315.
[9] W.G.J. Halfond, A. Orso, AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks, in: 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, CA, USA, 2005, pp. 174–183.
[10] W.G.J. Halfond, A. Orso, Combining static analysis and runtime monitoring to counter SQL-injection attacks, in: Third International Workshop on Dynamic Analysis, St. Louis, MO, 2005, pp. 1–7.
[11] W.G.J. Halfond, A. Orso, P. Manolios, Using positive tainting and syntax-aware evaluation to counter SQL-injection attacks, in: 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Portland, Oregon, 2006, pp. 175–185.
[12] W.G.J. Halfond, A. Orso, P. Manolios, WASP: protecting web applications using positive tainting and syntax-aware evaluation, IEEE Transactions on Software Engineering 34 (1) (2008) 65–81.
[13] W.G.J. Halfond, J. Viegas, A. Orso, A classification of SQL-injection attacks and countermeasures, in: International Symposium on Secure Software Engineering Raleigh, NC, USA, 2006.
[14] D. Hovemeyer, W. Pugh, Finding bugs is easy, in: 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Vancouver, BC, Canada, 2004, pp. 92–106.
[15] M. Howard, D. LeBlanc, Writing Secure Code, second ed., Microsoft Corporation, Redmond, 2003.
[16] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, S.-Y. Kuo, Securing web application code by static analysis and runtime protection, in: 13th International Conference on World Wide Web, New York, NY, 2004, pp. 40–52.
[17] G. Keizer, One-at-a-time Hacker Grabs 22,000 IDs from University of Missouri, first ed., Retrieved Issue 1, vol. 1, 2007, <http://computerworld.com/action/

article.do?command=viewArticleBasic&taxonomyName=cybercrime_and_hacking &articleId=9018982&taxonomyId=82&intsrc=kc_top>, accessed June 30, 2008.

[18] J. Kirk, Databases Assaulted by SQL Injection Attacks, first ed., Retrieved Issue 1, Volume 1, 2006, <http://www.cio.com/article/23133/Databases_Assaulted_by_SQL_Injection_Attacks>, accessed June 30, 2008.

[19] M.S. Lam, J. Whaley, V.B. Livshits, M. Martin, D. Avots, M. Carbin, C. Unkel, Context-sensitive program analysis as database queries, in: Principles of Database Systems (PODS), Baltimore, Maryland, 2005, p. 12.

[20] B. Livshits, Defining a set of common benchmarks for web application security, in: Workshop on Defining the State of the Art in Software Security Tools, Baltimore, 2005, p. 1.

[21] V.B. Livshits, Findings security errors in Java applications using lightweight static analysis, in: Computer Security Applications Conference, Tucson, AZ, 2004, p. 2.

[22] V.B. Livshits, M.S. Lam, Finding security vulnerabilities in Java applications with static analysis, in: 14th Usenix Security Symposium, Baltimore, MD, 2005, pp. 271–286.

[23] M. Martin, V.B. Livshits, M.S. Lam, Finding application errors and security flaws using PQL: a program query language, in: 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, San Diego, CA, 2005, p. 19.

[24] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, D. Evans, Automatically hardening web applications using precise tainting, in: 20th IFIP International Information Security Conference, Chiba, Japan, 2005, p. 12.

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

[26] Z. Su, G. Wassermann, The essence of command injection attacks in web applications, in: 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, SC, USA, 2006, pp. 372–382.