Using JML to protect Java code against SQL injection

Johan Janssen
0213888
jjanssen@sci.ru.nl

June 26, 2007

Abstract

There are a lot of potential solutions against SQL injection. The problem is that not all programmers use them, for various reasons. To check for bad code which can lead to SQL injection one could use JML specification. The specification can be added to the code, or more favorable to the Java API. After the programmer has written a piece of code, you can check with a tool like ESC/Java2 if the code is vulnerable to SQL injection. Or you could use one of the other solutions I proposed to make sure that the programmer only uses 'safe' functions.

Contents

1 Introduction

In today's world there are a lot of websites who depend on databases like MySQL. The problem with these databases is that you do not want users to access information that is not designated for them. One technique that is used to 'illegally' access information from MySQL databases is SQL injection. If you are unfamiliar with SQL injection, it is an attack technique used to exploit web sites by altering backend SQL statements through manipulating application input.[1]

SQL injection is widely used and a real threat to privacy sensitive information. Both for small sites from enthusiast, but also for big companies with lots of private information. When you look at The Common Vulnerabilities and Exposures site: `http://cve.mitre.org/`, and search in their database with the keywords "SQL injection" then you will find more than 2200 results and that is from 2000 until 2007.

Nearly all web applications are security critical, but only a small fraction of deployed web applications can afford a detailed security review. Even when such a review is possible, it is tedious and can overlook subtle security vulnerabilities.[1] This is the reason that you cannot depend on the programmer to produce secure code. Some website builders don't understand the security concerns and others might forget to implement them correctly.

That is why it is usefull to look at methods which can prevent SQL injection. Of course, if you use the right techniques and functions, then it should work fine. But that is the problem, in reality we see that not only small sites, but also the big ones have security flaws. Everyone makes mistakes and forgets things, therefore we would like to see that the security measures are applied in such a manner that the programmer should not bother about it.

So I wanted to see if it is possible to check if a programmer has protected his code against SQL injection. I wanted to see if it is possible to specify a piece of code with the help of JML to see if it is vulnerable to SQL injection.

2 Design by contract

There are different methods for designing computer software, Design by contract is one of them. Design by contract is also often mentioned as DBC and sometimes as Programming by contact. The principal idea behind DBC is that a class and its clients have a 'contract' with each other. The client must guarantee the pre-condition before calling a method defined by the class, and the class has to guarantee the post condition. [2]

Quality of software is quite important, reliability is needed, the code should work according to the specification and should deal with abnormal situations. Therefore a systematic method to specify and implement software would be of quite some support. One method to do so is Design by Contract.[3]

[1] Web Application Security Consortium `http://www.webappsec.org/`

[2] For more information: Design By Contract with JML[2]

[3] Eiffel Software, the pioneers of DBC: `http://archive.eiffel.com/doc/manuals/technology/contract/`

2.1 Contracts

Contracts according to DBC are precisely defined specifications of the mutual obligations of interacting components. It is an agreement between two parties a client and a supplier. If the client provides A, then the supplier guarantees B.

Besides preconditions and postconditions, there is a third assertion called invariant which holds of all instances of a class. An example type of an invariant is '0 <= *count*', where count should always be bigger then 0.

Contracts are a good way of documentation. You can provide client programmers with a good description of the interface properties of a class. It is even possible to make a so called 'short form' which includes only headers and assertions of exported features and invariants, the rest of the code isn't necessary to use the methods of the specified class.
An example of DBC is the following:

Listing 1: Design By Contract

```
1 //@ requires (* x is positive *);
2 /*@ ensures (* \result is an
3   @             approximation to
4   @             the square root of x *)
5   @       && \result >= 0;
6   @*/
7 public static double sqrt(double x) {
8   return Math.sqrt(x);
9 }
```

3 JML

The Java Modeling Language (JML) uses the ideas of Design By Contract. It is a formal specification language for Java which uses the preconditions, postconditions and invariants. JML can be used to specify the behavior of Java modules. It is also usefull to record decisions about the design and the implementation. The JML specifications can be added as annotation comments to Java modules. After that you can compile your Java program with any compiler. It is also possible to use one of the verification tools for Java, for example ESC/Java2, which I will using this paper, to check your code.[4]

Using preconditions and postconditions in JML gives us the opportunity to establish a contract between a class and its clients. Where the client must ensure the precondition and may assume the postcondition and the class must ensure the postcondition and may assume the precondition. This is also called Design by contract, which was explained earlier.

The goal of JML is that it is easy to use for every Java programmer, so that it will be of practical use. To achieve this goal the following commands can be used inside .java files:

- Assertions are added to the Java code in the following annotations /*@ <JML specification> @*/, or //@ <JML specification>.

[4]The official JML site: `http://www.cs.iastate.edu/~leavens/JML/`

- Properties are specified as Java boolean expressions and can be extended with a few operators (\ old, \ forall, \ result, ...)
- New 'JML' keywords can be used, such as: requires, ensures, signals, assignable, pure, invariant, non_null,...)

For an example of JML you can see appendix A, where JML is used to protect against SQL injection.[5]

4 Extended Static Checking

The Extended Static Checker for Java also called ESC/Java2 is a tool for finding errors in Java programs. ESC/Java2 can detect programming errors at compile time that are normally not detected before runtime and sometimes they aren't even found during runtime. Examples of these errors include null dereference errors, array bounds errors, type cast errors etcetera. [6]

Extended Static Checking can be positioned between type checking and program verification:

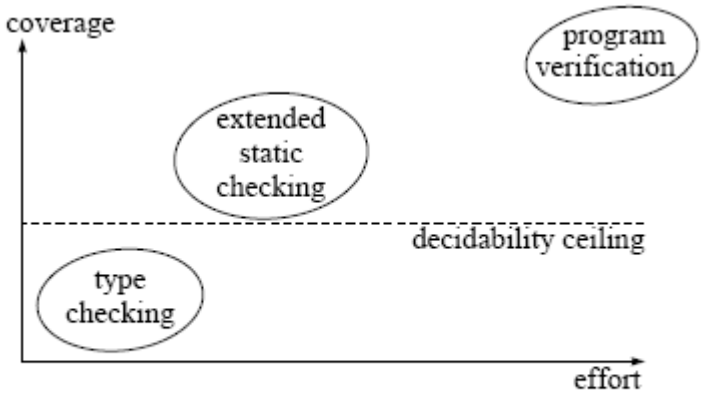

Figure 1: Static checkers plotted along the two dimensions coverage and effort (not to scale).

ESC/Java2 tries to prove the correctness of specifications at compile time and does that fully automatic. It can find lots of potential bugs quickly. It is good at proving the absence of runtime exceptions and verifying relatively simple properties. But it is unfortunately not sound (it can miss an error that is in the program) and it is not complete (it can give an error that is not possible).

ESC/Java2 operates in three steps, the first is the parsing phase where the syntax is checked. The second is the typechecking phase where type and usage is checked. The third is the static checking phase, which runs a prover called Simplify to find potential bugs. Parsing and typechecking can produce cautions or errors while static checking can produce warnings.

Extended checking has some advantages on runtime checking. For example if you specify assertions in your program but make a wrong assumption for instance:

[5]For more information you could also read the JML Reference Manual[3]

[6]The official ESC/Java2 site: `http://secure.ucd.ie/products/opensource/ESCJava2/`

Listing 2: Wrong Assumptions

```
1 if (length > 0 && width > 0){
2         //@ assert length > 0 && width < 0;
3 }
```

Then with runtime checking the wrong assertion may be detected with a good test suite, but ESC/Java2 will detect the error at compile-time.

ESC/Java2 checks specifications during compile-time, it proves the correctness of specifications. ESC/Java2 is independent of any test suite, which eliminates the problem of runtime checking where testing is only as good as the test suite. Therefore ESC/Java2 provides a higher degree of confidence. The disadvantage is that you have to specify everything you make and use, all (API) methods must be specified using pre- and postconditions and invariants.

5 SQL injection

SQL injection is a technique that exploits a security vulnerability occurring in the database layer of an application. The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed. It is in fact an instance of a more general class of vulnerabilities that can occur whenever one programming or scripting language is embedded inside another.[7]

SQL injection is possible when input data is untrusted and when the data is used to construct a SQL query dynamically.

5.1 Overview

SQL injection affects any programming language that is used as an interface with a database. High-level languages are most vulnerable to the attack, but sometimes even low-level languages can be compromised. Even the SQL language itself can be sinful.

Mostly an application receives some malformed data, which the application uses to build an SQL statement with the help of string concatenation. With this, the attackers can change the semantics of the SQL query. "Some people don't know there are better methods than string concatenation and others don't use them, because string concatenation is so easy." [4]

SQL injection can have serious consequences on 4 security characteristics:

- Confidentiality: sensitive data from SQL databases can be compromised.
- Integrity: information can be changed or deleted.
- Authentication: it is possible to login as another user without knowing the password.
- Authorization: authorization information can be changed.

[7] Wikipdia `http://en.wikipedia.org/wiki/SQL_injection`

5.2 Example

A simple example of Java servlet code which is vulnerable to SQL injection:

Listing 3: Vulnerable Java code

```
String query = "select * from user where username='" +
    username +"' and password='" + password + "'";
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(query);
if (resultSet.next()) {
        out.println("You're logged in");
} else {
        out.println("Wrong username or password");
}
```

This piece of code accepts user input without performing input validation or escaping meta-characters. With this code it is possible for a user/attacker to provide a username which contains SQL meta-characters that change the intended function of the SQL statement. One example to to this is to provide the following username:

```
admin' OR '1'-'1
```

This together with a blank password generates the following SQL statement:

```
select * from user where username='admin' OR '1'='1'
        and password=' '
```

Which allows the attacker to login to the site without a password, because the 'OR' expressions is always true. With this technique attackers can inject SQL commands that are powerful enough to extract, modify of delete data within the database.

6 Countermeasures against SQL injection

There are a lot of ways to prevent SQL injection. The most common are the mentioned in the next subsections.

6.1 Validating input

The 'deny by default' principle should be applied in this case. All data should be rejected unless it matches the criteria for 'good' data. With this method you can define a strict range for valid data 'white list' and reject all other data. The valid data should be constrained by:

- Type: String, integer, unsigned integer etc.
- Length.
- Character set: for instance only alphabetic characters [a-zA-Z]*.
- Format: you could constrain the data by specifying some format for example 'dd-mm-jjjj' where you can enter the day with 2 digits, the month with two digits and the year with four digits.

- Reasonableness: compare values to expected ranges, to filter 'strange' values.

For protection against SQL injection it is best to only allow letters and numbers. Other items should be converted by HTML encoding, so ¿ becomes "¿" and so on. For some user input, like email-addresses you have to allow items such as @,.,-,_ these should be allowed, but only if they are converted using HTML encoding.

6.2 Prepared statements

If you use prepared statements, then the variables that are passed as arguments to the statements will automatically be escaped for meta-characters by the JDBC driver. They should be used when stored procedures cannot be used and the SQL commands are dynamic. Prepared statements are not only usefull for security but also for performance. Because the contents of the variable or not interpreted, the database will just use the value of the bind variable. That makes prepared statements immune to SQL injection attacks.

An example of the use of prepared statements:

Listing 4: Prepared statements

```
1 String selectStatement = "SELECT * FROM Catalog WHERE
    userID = ? ";
2 Connection connection = DriverManager.getConnection(url,
    username, password);
3 PreparedStatement preparedStatement = connection.
    prepareStatement(selectStatement);
4 preparedStatement.setString(1,userID);
5 Resultset resultSet = preparedStatement.executeQuery();
```

But be carefull not to pass input variables directly to the prepared statement, because then SQL injection attacks are still possible. The following example shows such a mistake:

Listing 5: Wrong use of prepared statements

```
1 Connection connection = DriverManager.getConnection(url,
    username, password);
2 String userName = request.getParameter("UserName");
3 PreparedStatement preparedStatement = connection.
    prepareStatement("SELECT * FROM catalog WHERE userID =
     '+userName+'");
```

6.3 Stored procedures

Developers should not allow user input to change the syntax of SQL statements. The best solution is to separate the web application and SQL completely. The solution is to use stored procedures on the database server for every SQL statement. Then the application has to execute the stored procedures by using a safe interface such as callable statements of JDBC or CommandObject of ADO.

6.4 Least privilege

Another way to protect (partly) against SQL injection attacks is mentioned in 'How to Break Web Software'[5]. They suggest that you should give users the rights they need (and nothing more). This security principal is known as 'least privilege'. You should restrict the amount of data that a user can access, create different categories of users and give them their own databases and rights. If SQL injection occurs then it is at least restricted to the data that the user has privileges for.

7 Countermeasures using JML

One way is to use an extra 'ghost' field that tells you if the input is trusted. That ghost field should be added to the String class of the Java API in the following way:

```
1 //@ ghost public boolean trusted;
```

Then you have to create a function which checks if the user input is trusted. The best way is to use white listing and allow only a minimum of different characters as input. The function I used only accepts digits and characters from the alphabet which may be uppercase or lowercase. For practical use, you may want to add more characters to the white list. If the userinput is trusted then the boolean will be set to true, else an exception is thrown.

```
1  //@ ensures userinput.trusted;
2  //@ signals (Exception) !userinput.trusted;
3  //**** Check if the user input is trusted, else throw an
       exception ****//
4  public static void checkUserInput(/*@ non_null @*/ String
       userinput) throws Exception
5  {
6    if(userinput.matches("[a-zA-Z0-9]*"))
7    {
8      //@ set userinput.trusted = true;
9    }
10   else
11   {
12     //@ set userinput.trusted = false;
13     throw new Exception("Userinput not trusted");
14   }
15 }
```

So when calling this function with userinput containing " ", "__", "-" or other characters, the checkUserinput function will produce an exception and the program will be terminated. The problem with this solution is that for instance email-addresses containing special characters are rejected. To build a whitelisting function that is absolutely secure against SQL injection and is usable seems to be an impossible task. That is because of the fact that you would like to accept user input containing special characters, although they pose a threat of SQL injection.

The last step contains the part of the program that shows how checkUserInput in combination with JML is used to make sure that user input is checked before executing the query with the given user input.

```
1  //**** Input which is normally given by the user ****//
2  String usernameInput = "$usernameInput";
3  String passwordInput = "$passwordInput";
4  checkUserInput(usernameInput);
5  checkUserInput(passwordInput);
6
7  //**** The query including user input ****//
8  String queryWithUserInput = "select * from loginData
       where user='" + usernameInput + "' AND password='" +
       passwordInput + "'";
9
10 Connection connection = DriverManager.getConnection(url,
       username, password);
11 //@ assume connection != null;
12 Statement st = connection.createStatement();
13 //@ assume queryWithUserInput != null && st != null;
14 //@ assume usernameInput.trusted && passwordInput.trusted
       ==> queryWithUserInput.trusted;
15 // **** If both the usernameInput and the passwordInput
       are trusted then the resulting queryWithUserInput
       should be trusted ****/
16 executeQuery(queryWithUserInput,st);
```

Checking if there are any problems can be done dynamically which is called input tainting and can be checked using runtime assertion checking, or statically using ESC/Java2.

8 General JML rules for the Java API

The same JML that I used in the previous example can be used in the Java API. It is a matter of adding the JML to the executeQuery functions in the API. That can be done in the same way as used in the example in Appendix A.

```
1  //@ requires query.trusted;
2  //**** If the query is trusted then execute the query
       ****//
3  public static void executeQuery(/*@ non_null @*/ String
       query, /*@ non_null @*/ Statement st)
4  {
5    try{
6      ResultSet rs;
7      rs = st.executeQuery(query);
8
9      //@ assume rs != null;
10     while(rs.next())
11     {
```

```
12        System.out.println("User " + rs.getString(1));
13        System.out.println("Password " + rs.getString(2));
14      }
15    }
16    catch(Exception e)
17    {
18      System.out.println(e.getMessage());
19    }
20  }
```

Then you need to change the string concatenation function(s) in the API, as shown below. I assume that if 2 trusted strings are added that the resulting string is also trusted. So I added some JML to the concat function to ensure that when two trusted strings are concatenated then the resulting string is also trusted.

```
1  //@ ensures s1.trusted && s2.trusted ==> \result.trusted;
2  public static String concat(String s1, String s2)
3  {
4      String s = s1+s2;
5      //@ assume s1.trusted && s2.trusted ==> s.trusted;
6      return s;
7  }
```

Then it is necessary that literals ("text") are trusted by default, that should be added to the Java language itself. That is necessary because you want that the pieces of the query that you enter in the source code are trusted by default. Because that is not so easy to modify, I just set the literals to trusted manually. Normally you would want to implement this in the checker. Before executeQuery is executed you should check whether all user input is trusted, with the help of some function like checkUserInput and the ghost field which I added to the JAVA API already.

```
1  String driverName = "com.mysql.jdbc.Driver";
2  Class.forName(driverName).newInstance();
3
4  String serverName = "$serverName";
5  String mydatabase = "$mydatabase";
6  String url = "jdbc:mysql://" + serverName +  "/" +
       mydatabase;
7  String username = "$username";
8  String password = "$password";
9
10 //**** Input which is normally given by the user ****//
11 String usernameInput = "$usernameInput";
12 String passwordInput = "$usernameInput";
13 checkUserInput(usernameInput);
14 checkUserInput(passwordInput);
15
16 //**** Literals should be trusted ****//
```

```
String queryPart1 = "select * from loginData where user='
    ";
String queryPart2 = "' AND password='";
String queryPart3 = "'";
//@ set queryPart1.trusted = true;
//@ set queryPart2.trusted = true;
//@ set queryPart3.trusted = true;

//**** The query including user input ****//
String queryWithUserInput = concat(concat(concat(concat(
    queryPart1, usernameInput),queryPart2),passwordInput),
    queryPart3);

Connection connection = DriverManager.getConnection(url,
    username, password);
//@ assume connection != null;
Statement st = connection.createStatement();
//@ assume queryWithUserInput != null && st != null;
executeQuery(queryWithUserInput,st);
```

The above is working well, but it has one disadvantage: the whitelist function. That function is hard to build, because of the problem that to make it secure you have to give up to much on usability. Therefore I looked at another example to stop SQL injection with the help of JML. This one uses the prepared statements which are safe against SQL injection. So we could combine prepared statements and JML to force programmers to use prepared statements. To force the use of prepared statements, you have to disallow the use of executeQuery for statements and other unsafe objects. Every executeQuery method in the Java API should be modified by adding some JML code to it. The executeQuery function of preparedStatement is save and may be used by the programmer. The following JML code should be added above the executeQuery function in the prepared statement part of the JAVA API. This piece of JML code always allows the use of executeQuery for prepared statements.

```
//@ requires true;
```

Above all other executeQuery functions in the JAVA API, such as the one in Statement the following JML code should be added. This JML code ensures a JML warning when the specific executeQuery is called.

```
//@ requires false;
```

The only thing you have to be carefull of with prepared statements in Java is not to pass input variables directly to the Prepared Statement. The possible solution for this might be to add an extra check that disallows the use of "" in the strings which are given as an argument for the preparedStatement. This can be done with a function like checkUserInput which was mentioned before. Unfortunately I could not test this easily, because the ESC/Java2 specifications do not cover the PreparedStatement, but the ideas are the same as the ideas behind the code that I wrote, so I assume that it will work. Then with the help of ESC/Java2 you can check very fast if a programmer has used the executeQuery

function with statements or other unsafe objects, and correct the sourcecode with the use of prepared statements.

9 Is JML an effective solution?

Partially, it can be of great help, but it introduces some new problems. The programmer for instance can easily break security and add something like the following. Then the JML code to detect SQL injection is quite useless.

```
1 //@ assume queryWithUserInput.trusted;
```

This can be detected by ESC/Java2, but takes some extra effort. ESC/Java2 should check the programmers code for the `assume` keyword. If that is detected then ESC/Java2 should give a warning. What follows are the three alternatives that I looked at to counter possible SQL injection.

1. Using a whitelisting function proves difficult. It is hard to build a function using whitelisting that is safe and usable. To make it safe you would want to disallow symbols like @,.,-,_ because allowing them would allow an attacker to use SQL injection. If you only allow characters and numbers then SQL injection is not possible any more. But for some input you want to allow special symbols; for instance email-addresses use them. To disallow all special symbols makes the whitelist very restrictive. So it is safe versus usable, a secure whitelisting function is hard to use in practice. You would propably need to restrict to many symbols to make it usable any more. Another problem is that you want the programmer to call a function to check the user input. He might not want that and just add some JML that indicates that the given user input is trusted.

2. Building a function with the help of whitelisting to prevent SQL injection is hard. But it is possible to use one of the older techniques which were mentioned before. In this paper I tried to combine prepared statements and JML. This is intended to force the programmer to use prepared statements and thus create code which is safe against SQL injection. This seems to be the better solution, as it is still very usable, all symbols can be used as user input. The JML that is needed to accomplish this is even easier then the JML that is needed to build functions that use whitelisting. The disadvantage is that you force a programmer to use some kind of function, in this case prepared statements, although his code is possibly completely safe against SQL injection. However, the small amount of extra work for using prepared statements and the greater usability above whitelisting make this the better solution. Another positive point is that all SQL queries can be constructed using prepared statements.

3. Another solution would be to remove all the executeQuery's from the Java API, except from PreparedStatement. This can be done by Sun, who maintain Java. Or the company or another group of people who want to remove them can adjust the API that they use themself. After removing programmers are immediately forced to use prepared statements and you do not have to check it with a tool like ESC/Java2. The disadvantage is that some code will be unusable, since parts of the JAVA API are

deprecated. So all the code containing deprecated functions will need adjustments in order to work properly. But that is also the case when we use prepared statements in combination with JML. Deprecating functions is a possible solution very similar to the one using prepared statements. Before one of the solutions is used in practice it is important that further research is done, to find out which solution is the best one.

10 Conclusion

With the help of JML one can quite easily detect if a given piece of code has possible SQL injection. Using whitelisting one could check whether user input contains possible SQL injection. The disadvantage is that checking user input is a difficult task, you want to deny all input that contains possible SQL injection. This results in the problem that various input, like email-addresses, which contain special characters are rejected. This makes it hard to use the solution in practice. Another disadvantage is that you would need to add quite some JML to the JAVA API in order to work with this solution. And the help of the programmer is needed to call the function which checks the user input.

To combine JML with prepared statements, who are safe against SQL injection was the next step. This solves the whitelisting problem, so you can use every user input you want. Using this solutions requires less JML code for the JAVA API then the whitelisting solution. Further the programmer does not need to call a function which checks the user input. The disadvantage is that you cannot execute queries which are not build using prepared statements. So for instance Java `statements` cannot be executed as a query. But since it is a secure solution against SQL injection and you can use every user input it provides a better solution then whitelisting. The programmer only faces the task of making sure that every query uses a prepared statement. Since that is not so hard and should be done anyway, if you want to build secure code, I think it is a good solution.

The last possible solution I looked at was to deprecate functions from the JAVA API. This can be done by Sun, or just someone else who wants to change them. All functions that execute queries and do not use prepared statements should be deprecated. In that case it is impossible to execute queries that are not build with the help of prepared statements. This solution and the solution which uses prepared statements with JML seem very similar. But removing the functions seems to be better since you don't need the extra JML code. Extra JML code could introduce possible bugs and you do not want that. In the future it would be a good idea to find out what the advantages and disadvantages exactly are.

So the research question can be answered positively: it is possible to specify a piece of code with the help of JML to see if it is vulnerable to SQL injection. The only disadvantage is that sometimes user input is detected as SQL injection when that is not the case. To solve this and build a white list that is both completely secure and offers you 'complete usability' is difficult, if not impossible. Maybe with more research one could build a 'good' white list, or find another solution to detect input that contains SQL injection. But in the mean time using prepared statements with JML or deprecating specific functions in the JAVA API seems to be the better solutions.

The possible solutions are based on some Java code which I have written using whitelisting and which can detect SQL injection. This was done after reading and processing literature about SQL injection and JML. After that I looked at the use of prepared statements. The code I provided for the prepared statements solution will work, since it is quite straightforward. So I assume that this solution would also work when implemented completely. Unfortunately that was not so easy to check, since ESC/Java2 does not have the specifications for prepared statements. Maybe in the future these specification are build and then it is easy to check whether my solution really works correct. Building these specifications is not so hard. It would require some JML for the functions in the prepared statement class of the Java API.

In the end the solution is not completely what I wanted. Detecting SQL injection without rejecting false positives proves difficult. The solution is not to detect SQL injection in the input, but by making sure that secure Java functions are used. So I did not find a good solution to detect possible SQL injection in the input. However, I did find another way to secure code against SQL injection. Securing code against SQL injection can be done by using prepared statements. So forcing a programmer in some way to only use prepared statements for his queries will make his code secure against SQL injection.

A Sourcecode

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import java.sql.Statement;
6
7  public class SQLInjection {
8    //@ ensures userinput.trusted;
9    //@ signals (Exception) !userinput.trusted;
10   //**** Check if the user input is trusted, else throw
       an exception ****//
11   public static void checkUserInput(/*@ non_null @*/
       String userinput) throws Exception
12   {
13     if(userinput.matches("[a-zA-Z0-9]*"))
14     {
15         //@ set userinput.trusted = true;
16     }
17     else
18     {
19         //@ set userinput.trusted = false;
20         throw new Exception("Userinput not trusted");
21     }
22   }
23
24   //@ requires query.trusted;
25   //**** If the query is trusted then execute the query
       ****//
26   public static void executeQuery(/*@ non_null @*/ String
        query, /*@ non_null @*/ Statement st)
27   {
28     try{
29       ResultSet rs;
30       rs = st.executeQuery(query);
31
32       //@ assume rs != null;
33       while(rs.next())
34       {
35         System.out.println("User " + rs.getString(1));
36         System.out.println("Password " + rs.getString(2))
             ;
37       }
38     }
39     catch(Exception e)
40     {
41       System.out.println(e.getMessage());
42     }
```

```
43    }
44
45    //@ ensures s1.trusted && s2.trusted ==> \result.
          trusted;
46    public static String concat(String s1, String s2)
47    {
48        String s = s1+s2;
49        //@ assume s1.trusted && s2.trusted ==> s.trusted;
50        return s;
51    }
52
53
54    public static void main(String args[])
55    {
56      try{
57        String driverName = "com.mysql.jdbc.Driver";
58        Class.forName(driverName).newInstance();
59
60        String serverName = "$serverName";
61        String mydatabase = "$mydatabase";
62        String url = "jdbc:mysql://" + serverName +  "/" +
              mydatabase;
63        String username = "$username";
64        String password = "$password";
65
66        //**** Input which is normally given by the user
              ****//
67        String usernameInput = "$usernameInput";
68        String passwordInput = "$passwordInput";
69        checkUserInput(usernameInput);
70        checkUserInput(passwordInput);
71
72        //**** Literals should be trusted ****//
73        String queryPart1 = "select * from loginData where
              user='";
74        String queryPart2 = "' AND password='";
75        String queryPart3 = "'";
76        //@ set queryPart1.trusted = true;
77        //@ set queryPart2.trusted = true;
78        //@ set queryPart3.trusted = true;
79
80        //**** The query including user input ****//
81        String queryWithUserInput = concat(concat(concat(
              concat(queryPart1, usernameInput),queryPart2),
              passwordInput),queryPart3);
82
83        Connection connection = DriverManager.getConnection
              (url, username, password);
84        //@ assume connection != null;
85        Statement st = connection.createStatement();
```

```
86        //@ assume queryWithUserInput != null && st != null
              ;
87        executeQuery(queryWithUserInput,st);
88      }
89      catch (ClassNotFoundException e)
90      {
91        System.out.println(e.getMessage());
92      }
93      catch (SQLException e)
94      {
95        System.out.println(e.getMessage());
96      }
97      catch (InstantiationException e)
98      {
99        System.out.println(e.getMessage());
100     }
101     catch (IllegalAccessException e)
102     {
103       System.out.println(e.getMessage());
104     }
105     catch(Exception e)
106     {
107       System.out.println(e.getMessage());
108     }
109   }
110 }
```

References

[1] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. 2006.

[2] Gary T. Leavens and Yoonsik Cheon. Design By Contract with JML. 2006.

[3] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Mller, Joseph Kiniry, and Patrice Chalin. JML Reference Manual draft.

[4] Michael Howard, David LeBlanc, and John Viega. *19 Deadly Sins of Software Security.* McGraw-Hill, 2005.

[5] Mike Andrews and James A. Whittaker. *How to Break Web Software.* Addison-Wesley, 2006.