Preventing SQL Injections

By Adam Kolawa

When SQL statements are dynamically created as software executes, there is an opportunity for a security breach: if the attacker is able to pass fixed inputs into the SQL statement, then these inputs can become part of the SQL statement. If the attacker knows his SQL, he can use this technique to gain access to privileged data, login to password-protected areas without a proper login, remove database tables, add new entries to the database, or even login to an application with admin privileges.

When SQL statements are dynamically created as software executes, there is an opportunity for a security breach...

SQL INJECTION EXAMPLES

The example in Figure 1 shows a system that was designed to restrict site access to only registered users. Unfortunately, an attacker can use SQL injection to do much more.

The database table that stores user accounts is shown in Figure 2.

FIGURE 1: SAMPLE SYSTEM

```
Http login form:

<FORM name=login action=login.jsp METHOD=post>
User name: <input name="USER">
<br>
Password: <input type="password" name="PASSWORD">
<br>
<input type="submit" value="Go">
</FORM>

login.jsp:

<%@ page import="java.sql.*" %>
<%@ page import="java.io.*" %>

<%
    //MySQL
    String DRIVER = "org.gjt.mm.mysql.Driver";
    String DBURL = "jdbc:mysql://localhost:3306/fruits";
    String LOGIN = "fruits";
    String PASSWORD = "fruits";

    Class.forName(DRIVER);
    Connection connection = DriverManager.getConnection(DBURL, LOGIN, PASSWORD);

    String sUsername = request.getParameter("USER");
    String sPassword = request.getParameter("PASSWORD");
    int iUserID = -1;
    String sLoggedUser = "";

    String s = "SELECT User_id, Username FROM USERS WHERE Username = '" +
                sUsername + "' AND Password = '" + sPassword + "'";
    Statement selectStatement = connection.createStatement ();
    ResultSet resultSet = selectStatement.executeQuery(s);
    if (resultSet.next()) {
        iUserID = resultSet.getInt(1);
        sLoggedUser = resultSet.getString(2);
    }

    PrintWriter writer = response.getWriter ();
    if (iUserID >= 0) {
        writer.println ("User logged in: " + sLoggedUser);
    } else {
        writer.println ("Access Denied!");
    }
%>
```

The intended usage is that when a user provides the inputs user=john and password=doe, the string select User_id, Username from users where Username='john' and Pass-word='doe' is formed and the user will be logged in as John.

However, there are many ways that an attacker can use SQL injection to perform actions that the developer did not anticipate.

LOGGING IN WITHOUT A VALID USERNAME AND PASSWORD

First, imagine that the attacker submits the following inputs:

```
user = ' or 1=1 #
password = (ANY)
```

The resulting SQL statement would be select User_id, Username from users where Username='' or 1=1 #' and Password='(ANY)'. As a result, the attacker can log in as John without providing a valid username or password.

Note that we are using '#' because this example connects to a MySQL database, which takes '#' as a comment. For other databases, other comment delimiters may apply (for example, '--').

LOGGING IN AS THE ADMINISTRATOR

The information exposed by error messages can be very useful to an attacker. Imagine that the attacker submits the following inputs:

```
user = ' error!
password = (ANY)
```

The resulting SQL statement would be select User_id, Username from users where User-name='' error! ' and Password='(ANY)'. As a result, the attacker receives a valuable error message shown in FIGURE 3.

Now, the attacker knows that the column names are based on their functions (for example, that a column with passwords is named "password"), so he can make educated guesses about the column names. For instance, he might try to submit the inputs in FIGURE 4.

As a result, he will receive another error message as shown in FIGURE 5.

This response tells the attacker that Username was a good guess. He then submits another set of inputs:

```
user = ' or Username like 'a%' #
password = (ANY)
```

The resulting SQL statement would be select User_id, Username from users where Username='' or Username like 'a%' # ' and Password='(ANY)'. As a result, the attacker would be logged in as admin, and gain all associated privileges.

DELETING TABLES

Even without logging in as admin, the attacker can remove database tables. Assume that the attacker provides the inputs shown in FIGURE 6.

The resulting SQL statement would be select User_id, Username from users where Username='' or 1=1; drop table users; # ' and Password='(ANY)'. As a result, the attacker would be logged in as John, and the users table would be deleted if the database engine and driver allow multiple SQL statements to be passed as one. Fortunately, most JDBC drivers do not.

FIGURE 2: DATABASE TABLE

```
+---------+--------------+----------+
| User_id | Username     | Password |
+---------+--------------+----------+
|       1 | john         | doe      |
|       2 | jan's friend | friend   |
|       3 | admin        | qaz456   |
+---------+--------------+----------+
```

FIGURE 3: ERROR MESSAGE

```
java.sql.SQLException: Syntax error or access violation:
You have an error in your SQL syntax near 'error! ' AND
Password = ''' at line 1
```

FIGURE 4: SAMPLE INPUTS

```
user = ' or Username like '%' or User_name like '%' #
password = (ANY)
```

FIGURE 5: ERROR MESSAGE

```
java.sql.SQLException: Column not found: Unknown column
'User_name' in 'where clause'
```

FIGURE 6: SAMPLE INPUTS

```
user = ' or 1=1; drop table users; #
password = (ANY)
```

SQL INJECTION PREVENTION

The traditional attempt to avoid this problem is to validate all user inputs. This is generally an effective way of dealing with malicious user input. However, it's possible to prevent these attacks altogether by building the statements in such a way that it is impossible for attackers to hijack them—even with the most well-designed and malicious inputs.

A simple way to ward off SQL injection attacks is to ensure that all SQL statements recognize user inputs as variables, and that statements are precompiled before the actual inputs are substituted for the variables. Typically, this is implemented as a two-stage process. In the first stage, the SQL statement should be built and parsed with variables in place of the expected user inputs. Then, in the second stage—before the statement is passed to the database—the variables should be replaced with the user inputs. When you implement this strategy, you ensure that user inputs are never parsed as the actual SQL statement, so even malicious user inputs are rendered ineffective.

For instance, in Java, a secure way to build SQL statements is to construct all queries with PreparedStatement instead of Statement and/or to use parameterized stored procedures. Parameterized stored procedures are compiled before user input is added, making it impossible for an attacker to modify the actual SQL statement. When PreparedStatement is used, most JDBC drivers will prepare a statement with the server, and then supply the parameters separately. In either case, after the initial parsing, there is a clear distinction between the SQL statement and the variable. The variables are encapsulated and special characters within them are automatically escaped in a manner suited to the target database. Consequently, it is impossible for an attacker to pass malicious input and have it treated as if it were the actual SQL statement—treatment that is required if the attacker is going to succeed with SQL injection attacks.

Even if you use PreparedStatement, you still need to pay attention to the way in which you build arguments. All parameters should be inserted through appropriate JDBC calls. If you concatenate the SQL sentence and omit the JDBC calls, then an attempted SQL injection could be parsed as SQL, and the attacker could succeed.

For example, the code in FIGURE 7 illustrates both the correct and incorrect ways of using PreparedStatement.

Even if you use PreparedStatement, you still need to pay attention to the way in which you build arguments.

How do you ensure that code follows these best practices? Most SQL statements are created dynamically; consequently, you need to execute the application paths that create, collect and examine the SQL statements, and verify whether they are being constructed in a secure manner (i.e., that the statements are precompiled with variables before user input is added). In addition, you need to inspect the database-related code to verify that secure coding practices are being followed (for instance, the Java best practices of using PreparedStatement instead of Statement, and for using PreparedStatement correctly).

Both of these types of verification can be automated. For instance, in Java, available technologies can statically analyze the code that is responsible for forming the SQL statements. This static analysis can be used to verify that if you build SQL for JDBC, you always use PreparedStatement. It can also verify whether all available PreparedStatements it includes are built properly (with all parameters inserted through appropriate JDBC calls, rather than string concatenation).

FIGURE 7: PREPARED STATEMENT

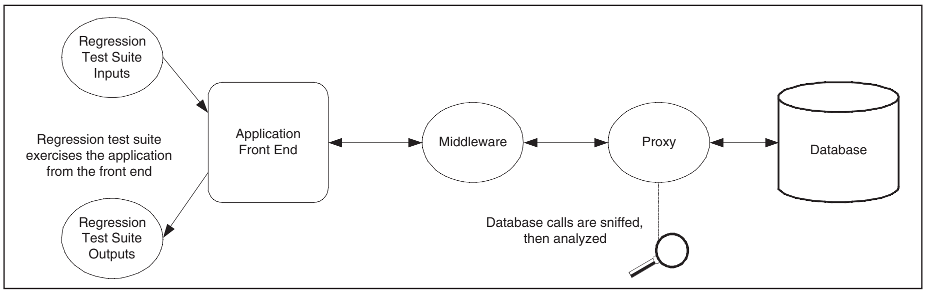
```
package fruits;

import java.sql.*;

public class CustomerDatabaseJDBC extends CustomerDatabase
{

public int getUserId(String user, String password)
     {
         ConnectionManager manager = ConnectionManager.getInstance ();
         Connection connection = manager.getConnection ();

PreparedStatement selectStatement = null;
         ResultSet resultSet = null;

int id = -1;
         try {
             /* Correct use of prepared statement. Both parameters are inserted
                through appropriate JDBC calls
             selectStatement = connection.prepareStatement(
                 "SELECT User_id FROM USERS WHERE Username = ? AND Password = ?");

selectStatement.setString (1, user);
             selectStatement.setString (2, password);
             */

/* Incorrect use of prepared statement. Developer is concatenating
               SQL sentence and omits JDBC calls */
             selectStatement = connection.prepareStatement(
                 "SELECT User_id FROM USERS WHERE Username = '" +
                 user +
                 "' AND Password = '" +
                 password + "'");

/* Still incorrect use of prepared statement.
                All parameters should be passed through JDBC calls
             selectStatement = connection.prepareStatement(
                 "SELECT User_id FROM USERS WHERE Username = '" +
                user + "' AND Password = ?");

selectStatement.setString (1, password);
             */

resultSet = selectStatement.executeQuery();

if (resultSet.next()) {
                 id = resultSet.getInt(1);
             }
         } catch (SQLException exception) {
             System.err.println ("Error looking for user: " +
                                         exception.getMessage ());
         } finally {
             if (resultSet != null) {
                 try {resultSet.close();} catch (SQLException ex) {}
             }
             if (selectStatement != null) {
                 try {selectStatement.close();} catch (SQLException ex) {}
             }
             manager.reclaimConnection (connection);
         }

         return id;
     }
}
```

Moreover, to determine whether SQL statements are being built in the recommended two-step process, available monitoring technologies can use dynamic analysis to watch database calls as the application is being tested during the integration phase. As the application is stimulated through a functional test suite that is run through a test client, the monitoring technologies watch the test suite coverage to see which application paths are covered, and trap all SQL statements that pass through the proxy. The database monitoring technologies can then analyze how the SQL statements were dynamically constructed, and identify any statements that were built in an unsafe way. In this way, you can identify security vulnerabilities and correct them before attackers have the opportunity to exploit your application. This analysis process is illustrated in FIGURE 8.

While the previous examples refer to Java and JDBC, the same principles can be applied to code built in C++, ASP, Visual Basic, etc. For instance, with C++, you would perform runtime instrumentation to enable the integration testing monitoring, and you would again supplement this dynamic analysis with a static analysis of database-related code. In all cases, the key to security is to ensure that the user input is not recognized and parsed as part of the SQL statement.

NaSPA member Dr. Adam Kolawa is the co-founder and CEO of Parasoft, a leading provider of Automated Error Prevention (AEP) software solutions.

FIGURE 8: ANALYSIS PROCESS


Regression Test Suite Inputs
Regression test suite exercises the application from the front end
Regression Test Suite Outputs
Application Front End
Middleware
Proxy
Database
Database calls are sniffed, then analyzed