A brief look at SQL Injection

A SQL injection attack consists of insertion or "injection" of a SQL query via data sent from the client to the application. A successful attack can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shut down the DBMS), recover the content of a given file present on the DBMS file system, and in some cases issue commands to the operating system.

As an example of a simple SQL injection attack, consider the following server-side code that uses a last name submitted from a textbox on a previous page to pull an employee's data from a MySQL database:

```php
<![CDATA[
$name = $_POST['emp_name'];
$sql = "SELECT * FROM `employee` WHERE emp_lname='" . $name . "';
$result = mysql_query($sql);

while($row = mysql_fetch_array($result)) {
    print "<u>Full employee name</u>: " . $row['emp_fname'] . " " . $row['emp_lname'] . "<br />";
}
]]>
```

The developer is expecting the user to submit just a last name (Hahn, for example), in which case the code will dynamically insert the value into the SQL statement between the single quotes and return the correct results from the database.

Suppose, however, if instead of submitting just a last name, the user submits something like:

```
Hahn' OR '1'='1
```

This will result in the following query being created instead:

```
"SELECT * FROM `employee` WHERE emp_lname='Hahn' OR '1'='1"
```

Because the logical statement in the WHERE clause will always be evaluated as true, this will return all the records for all of the employees, rather than just those for employee Hahn (and the while loop will print all the information to the screen).

This same idea potentially also can be used to make undesired changes to the database. For example, suppose that the remote user entered the following string in the text box instead:

```
Hahn'; DROP TABLE employee;--
```

resulting in:

```
"SELECT * FROM `employee` WHERE emp_lname='Hahn'; DROP TABLE employee;--"
```

The semicolon is used to separate two different SQL statements, and the double dash (--) serves as a SQL comment – this causes the database to ignore all characters that follow the
comment (and thus to ignore the trailing single quote in the dynamic SQL statement). The employee table may thus be permanently deleted from the database.

*** Neither MySQL nor MS Access support allowing multiple SQL statements separated by semicolons to be executed at once, thus they are not affected by the problem exhibited in this second example. Other DBMSs, such as SQL Server and Oracle, do support this however, so it is important to understand that it may be a potential issue in those environments.

**Protecting against SQL injection attacks**

There are several defenses against SQL injection attacks – the two that are most often discussed are the following:

1. **Escaping user-supplied input values and/or data validation**
2. **Using parameterized prepared statements (prepared queries)**

We briefly discuss each of these below.

1. **Escaping user-supplied input values and/or data validation**

(a) Escaping user-supplied input

MySQL, which is typically accessed through a web page running a scripting language such as PHP, has a built-in function, `mysql_real_escape_string()` which can be called after a database connection has been established. This function will add in backslashes to change characters such as a single quote to their escape character equivalents: ' → \'

Thus, by using:

```php
$name = mysql_real_escape_string($_POST["emp_name"]);
```

in the example on the first page, `Hahn' OR '1'='1` will be changed to: `Hahn\' OR \'1\'='1` before it is inserted into the SQL statement.

This also protects against errors from names that actually do include single quotes (such as O'Neill), by preventing the database from thinking that the SQL statement has ended prematurely.

PHP includes a similar function: `addslashes()`, which will accomplish the same thing. It is recommended to use `mysql_real.escape_string()` to pre-process database input for a MySQL database because it is specifically designed for that purpose, but some database management systems (i.e., Oracle and SQL Server) do not include an escape function – in this case, `addslashes()` can and should be used.

(b) Data validation

Although escaping user-supplied input works well much of the time, it does not always help. Consider the following example:
A brief look at SQL injection

```php
<?
$id = mysql_real_escape_string($_POST["emp_id"]);  
$sql = "SELECT * FROM `employee` WHERE emp_id=" . $id;  
$result = mysql_query($sql);

while($row = mysql_fetch_array($result)) {
    print "<u>Full employee name</u>: " . $row["emp_fname"] . " " . $row["emp_lname"] . "<br/>";
}
?>
```

Because the SQL statement is restricting by an integer value, single quotes are not needed around that value. Thus, instead of submitting just an integer employee ID, the user could submit something like:

```
3 OR 1=1
```

resulting in:

```
"SELECT * FROM `employee` WHERE emp_id=3 OR 1=1"
```

The `mysql_real_escape_string( )` function will not change the variable's value in this case because there are no "special" characters to escape, and thus the SQL statement will return all employees, rather than just employee 3.

There are a couple of ways to address this issue, however. For example:

- PHP has a built-in `is_numeric( )` function that will return a value of FALSE if a variable contains any non-numeric characters (including spaces). A submitted variable can thus be checked before it is inserted into a dynamic SQL statement.

- PHP allows you to cast variables as a particular data type. If a variable is cast as either an integer or a float (decimal) and it has a leading numeric value, then the variable's value will be changed to that leading numeric value. If the variable's value doesn't begin with a number or if it contains only alphabetic or punctuation characters, then the variable's value will be changed to zero. The PHP functions `intval( )` (or `floatval( )`) and `strval( )` together thus allow us to check if a variable is purely numeric before using it:

```php
if($x == strval(intval($x))){
    // $x is an integer
} else{
    // $x is not an integer
}
```

2. **Using parameterized queries**

(adapted from [http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html](http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html))

Parameterized queries, or prepared statements, offer the ability to set up a SQL statement once, and then execute it many times with different parameters. They are designed to replace building dynamic query statements by concatenating strings together, and they accomplish the same behavior as concatenation but in a more secure and efficient manner. A typical prepared statement looks something like:

```
SELECT * FROM Country WHERE code = ?
```
The `?` is called a placeholder. When you execute the above query, you must supply the value for the database to match – this value then replaces the `?` in the query above. Because MySQL is expecting only the values for the missing parameters, it will not generally interpret inputs otherwise and this approach thus protects against many SQL injection attacks.

Prepared statements can also improve performance because the query only needs to be parsed once – new parameter values can then be bound (assigned) to the statement and run against the database, without incurring the processing overhead again. This pre-parsing can lead to a speed increase if you need to run the same query many times, such as when applying many INSERT statements.

Prepared statements are only available for PHP and MySQL by using the mysqli extension (an extension to the core PHP code that exposes a more powerful API for database access) - they are not available using just the standard functions from the mysql extension.

The following is an example of using a prepared statement in PHP with mysqli that uses a database to retrieve employee information based on a submitted last name:

**Example:** (using prepared statements with PHP and mysqli)

```php
<?

// with mysqli, the process of connecting to an actual database requires only one code statement
$db = mysqli_connect("localhost", "root", "mysql", "furnish", 3306);

if(isset($_POST["lname"])){
    // add slashes to the input parameter value as necessary (escapes quotes, etc.)
    $lastname = mysqli_real_escape_string($db, $_POST["lname"]);
    print "Name submitted: " . $lastname . "<br /><br />";

    // create a prepared statement
    if ($stmt = mysqli_prepare($db, "SELECT emp_fname, emp_lname
          FROM employee
          WHERE emp_lastname=?")) {
        // bind parameters for markers (assign dynamic parameter value)
        mysqli_stmt_bind_param($stmt, "s", $lastname);
        // execute query
        mysqli_stmt_execute($stmt);
        // bind result variables (assign columns to variables)
        mysqli_stmt_bind_result($stmt, $fname, $lname);
        if(mysqli_stmt_fetch($stmt)){
            // print current column values
            print "Full employee name: " . $fname . " " . $lname . "<br />";
        }else{
            // print "No results found";
        }
    }else{
        // print "Error occurred\n";
    }
}
```
A brief look at SQL injection

```php
32. print "No records returned<br />";
33. }
34. // close statement
35. mysqli_stmt_close($stmt);
36. }
37. // close connection
38. mysqli_close($db);
39. }
40. <br />
41. Please enter the last name of the agent you would like to retrieve:<br />
42. <form method="post" action="mysqli.php">
43. <input type="text" name="lname" />
44. <input type="submit" value="submit" />
45. </form>
```

Additional information


A stored procedure is simply a procedure that is stored within a database server – such procedures can typically be run from any application environment, and by using any programming language, without recreating the logic. The use of stored procedures can reduce network traffic and improve performance, particularly if an application involves many complex, repetitive tasks involving a database.

Stored procedures do not automatically provide protection against SQL injection attacks. If a stored procedure is set up to dynamically create SQL statements by concatenating user inputs into an existing text string, it is subject to the same SQL injection issues as dynamic SQL statement that is concatenated together within a PHP file. In order to protect against attacks, therefore, stored procedures must be carefully designed to either validate the data that has been submitted (by checking data types or formats), or to only allow parameterized SQL statements to be used.

Other suggestions for improved database security:

- Never connect to the database as a superuser or as the database owner. Always use customized users with very limited privileges.

Useful additional resources:

http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
http://dev.mysql.com/doc/refman/5.0/en/stored-routines.html