|  |
| --- |
| **[How to Prevent SQL Injection in PHP](https://www.wikihow.com/Prevent-SQL-Injection-in-PHP)**  <https://www.wikihow.com/Prevent-SQL-Injection-in-PHP>   * **Using Prepared Statements in PHP.** * **Prepared Statements combine the variable with the compiled SQL statement, so that the SQL and the variables are sent separately.** * **The variables are then interpreted as mere strings and not part of the SQL statement.** * **Using the methods in the steps below, you will not need to use any other SQL injection filtering techniques** (such as the mysql\_real\_escape\_string() ) |
| **Understanding SQL Injection**  PART 1.1  **SQL Injection is a type of vulnerability in applications that use a SQL database.** The vulnerability arises when a user input is used in a SQL Statement:  $name = $\_GET['username'];  $query = "SELECT password FROM tbl\_user WHERE name = '$name' "; |
| **PART 1.2 :**  **The value a user enters into the URL variable username will be assigned to the variable $name.** It's then placed directly into the SQL statement, making it possible for the user to edit the SQL statement.  $name = "admin' OR 1=1 -- ";  $query = "SELECT password FROM tbl\_user WHERE name = '$name' "; |
| **PART 1.3 :**  **The SQL database will then receive the SQL statement as follows:**  SELECT password FROM tbl\_users WHERE name = 'admin' OR 1=1 -- '   * This is valid SQL, but instead of returning one password for the user, the statement will return all the passwords in the table *tbl\_user*. This is not something you want in your web applications. |
| **PART 2 Using mySQLi to Create Prepared Statements** Image titled 2542820 1 |
| **Create the mySQLi SELECT Query.**  Use the code below to SELECT data from a table using mySQLi Prepared Statements. |
| **PART 2.1**  $name = $\_GET['username'];    if ($stmt = $mysqli->prepare("SELECT password FROM tbl\_users WHERE name=?")) {    // Bind a variable to the parameter as a string.  $stmt->bind\_param("s", $name);    // Execute the statement.  $stmt->execute();    // Get the variables from the query.  $stmt->bind\_result($pass);    // Fetch the data.  $stmt->fetch();    // Display the data.  printf("Password for user %s is %s\n", $name, $pass);    // Close the prepared statement.  $stmt->close();    } |
| Image titled 2542820 2 |
| **Create the mySQLi INSERT Query.** Use the code below to INSERT data into a table using mySQLi Prepared Statements.  $name = $\_GET['username'];  $password = $\_GET['password'];    if ($stmt = $mysqli->prepare("INSERT INTO tbl\_users (name, password) VALUES (?, ?)")) {    // Bind the variables to the parameter as strings.  $stmt->bind\_param("ss", $name, $password);    // Execute the statement.  $stmt->execute();    // Close the prepared statement.  $stmt->close();    } |
| Image titled 2542820 3 |
| **Create the mySQLi UPDATE Query.** Use the code below to UPDATE data in a table using mySQLi Prepared Statements.  $name = $\_GET['username'];  $password = $\_GET['password'];    if ($stmt = $mysqli->prepare("UPDATE tbl\_users SET password = ? WHERE name = ?")) {    // Bind the variables to the parameter as strings.  $stmt->bind\_param("ss", $password, $name);    // Execute the statement.  $stmt->execute();    // Close the prepared statement.  $stmt->close();    } |
| Image titled 2542820 4 |
| **Create the mySQLi DELETE Query.** The below script is how to DELETE data from a table using mySQLi Prepared Statements.  $name = $\_GET['username'];  $password = $\_GET['password'];    if ($stmt = $mysqli->prepare("DELETE FROM tbl\_users WHERE name = ?")) {    // Bind the variable to the parameter as a string.  $stmt->bind\_param("s", $name);    // Execute the statement.  $stmt->execute();    // Close the prepared statement.  $stmt->close();    } |
|  |
| **SQL Injection**  If user input is inserted without modification into an SQL query, then the application becomes vulnerable to SQL injection. |
| example:  $unsafe\_variable = $\_POST['user\_input'];  mysql\_query("INSERT INTO `table` (`column`) VALUES ('$unsafe\_variable')");  That's because the user can input something like value'); DROP TABLE table;--, and the query becomes:  INSERT INTO `table` (`column`) VALUES('value'); DROP TABLE table;--')  What can be done to prevent this from happening? |
| Solution :  Use prepared statements and parameterized queries.  These are SQL statements that are sent to and parsed by the database server separately from any parameters. This way it is impossible for an attacker to inject malicious SQL. |
| You basically have two options to achieve this: |
| Option 1 :    Using PDO (for any supported database driver):  $stmt = $pdo->prepare('SELECT \* FROM employees WHERE name = :name');  $stmt->execute(array('name' => $name));  foreach ($stmt as $row)  {  // do something with $row  } |
| Option 2 :    Using MySQLi (for MySQL):  $stmt = $dbConnection->prepare('SELECT \* FROM employees WHERE name = ?');    $stmt->bind\_param('s', $name); // 's' specifies the variable type => 'string'  $stmt->execute();  $result = $stmt->get\_result();  while ($row = $result->fetch\_assoc())  {  // do something with $row  } |
|  |
| **Warning:** This answer's sample code (like the question's sample code) uses PHP's mysql extension, which was deprecated in PHP 5.5.0 and removed entirely in PHP 7.0.0.  $unsafe\_variable = $\_POST["user-input"];  $safe\_variable = mysql\_real\_escape\_string($unsafe\_variable);  mysql\_query("INSERT INTO table (column) VALUES ('" . $safe\_variable . "')"); |
|  |
| To use the parameterized query :  <?php  $mysqli = new mysqli("server", "username", "password", "database\_name");  // TODO - Check that connection was successful.  $unsafe\_variable = $\_POST["user-input"];  $stmt = $mysqli->prepare("INSERT INTO table (column) VALUES (?)");  // TODO check that $stmt creation succeeded  // "s" means the database expects a string  $stmt->bind\_param("s", $unsafe\_variable);  $stmt->execute();  $stmt->close();  $mysqli->close();  ?> |
|  |
| Use PDO and prepared queries.  ($conn is a PDO object)  $stmt = $conn->prepare("INSERT INTO tbl VALUES(:id, :name)");  $stmt->bindValue(':id', $id);  $stmt->bindValue(':name', $name);  $stmt->execute(); |
|  |
| **mysql\_real\_escape\_string**  (PHP 4 >= 4.3.0, PHP 5)  mysql\_real\_escape\_string — Escapes special characters in a string for use in an SQL statement  **Warning**  This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the [MySQLi](http://php.net/manual/en/book.mysqli.php) or [PDO\_MySQL](http://php.net/manual/en/ref.pdo-mysql.php) extension should be used. See also [MySQL: choosing an API](http://php.net/manual/en/mysqlinfo.api.choosing.php) guide and [related FAQ](http://php.net/manual/en/faq.databases.php#faq.databases.mysql.deprecated) for more information. Alternatives to this function include:   * [mysqli\_real\_escape\_string()](http://php.net/manual/en/mysqli.real-escape-string.php) * [PDO::quote()](http://php.net/manual/en/pdo.quote.php) |
|  |
| SQL Injection Example Below is a sample string that has been gathered from a normal user and a bad user trying to use SQL Injection. We asked the users for their login, which will be used to run a SELECT statement to get their information. MySQL & PHP Code: // a good user's name  $name = "timmy";  $query = "SELECT \* FROM customers WHERE username = '$name'";  echo "Normal: " . $query . "<br />";  // user input that uses SQL Injection  $name\_bad = "' OR 1'";  // our MySQL query builder, however, not a very safe one  $query\_bad = "SELECT \* FROM customers WHERE username = '$name\_bad'";  // display what the new query will look like, with injection  echo "Injection: " . $query\_bad; Display: Normal: SELECT \* FROM customers WHERE username = 'timmy' Injection: SELECT \* FROM customers WHERE username = '' OR 1''  The normal query is no problem, as our MySQL statement will just select everything from customers that has a username equal to *timmy*.  **However**, the injection attack has actually made our query behave differently than we intended. By using a single quote (') they have ended the string part of our MySQL query   * username = ' '   and then added on to our WHERE statement with an OR clause of 1 (always true).   * username = ' ' **OR 1**   **This OR clause of 1 will always be *true* and so every single entry in the "customers" table would be selected by this statement!** |
|  |
| <https://websitebeaver.com/prepared-statements-in-php-mysqli-to-prevent-sql-injection> |
|  |
| <https://www.acunetix.com/blog/articles/prevent-sql-injection-vulnerabilities-in-php-applications/> |
|  |
| **#1. Using PHP-MySQLi Driver** You can use PHP-MySQLi driver [prepared statements](https://php.net/manual/en/mysqli.quickstart.prepared-statements.php) to avoid these type of SQL injections. Use below PHP code which will prevent SQL injection.   |  |  | | --- | --- | |  | $accountId = $\_GET['account\_id'];  if ($stmt = $mysqli->prepare('SELECT accountNumber, balance FROM accounts WHERE accountId = ?')) {    $stmt->bind\_param("s", $accountId);  $stmt->execute();  $result = $stmt->get\_result();  while ($row = $result->fetch\_assoc()) {  // do something here  }      $stmt->close();  } | |
| #2. Using PHP-PDO Driver You can use PHP-PDO driver [prepare statements](https://php.net/manual/en/pdo.prepare.php) to avoid these type of SQL injections. Use below PHP code which will resolve above SQL injections.   |  |  | | --- | --- | |  | $accountId = $\_GET['account\_id'];  if ($stmt = $pdo->prepare('SELECT accountNumber, balance FROM accounts WHERE accountId = :accountId'))  {    $stmt->execute(array('name' => $name));  foreach ($stmt as $row)  {  // do something here  }    $stmt->close();  } | |