**SQL INJECTION IN PHP**

**SQL Injection Before Prevention**

The following code is a very simple PHP application that accepts an id and shows the name of the user. The application uses GET but it could use POST or any other HTTP method. This example is based on the MySQL database but the same principles apply for other databases. The sample database is called users and has the following structure and content.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **username** | **password** | **first\_name** | **last\_name** |
| 1 | johnsmith | $2a$10$SakFH.Eatq3QnknC1j1uo.rjM4KIYn.o8gPb6Y2YBnNNNY.61mR9K | John | Smith |
| 2 | maryjohnson | $2a$10$hA/hwCzhr6F23BsbRZBjdOA5eqTgV01cv30sy/O2EcL2/zG9k0aGy | Mary | Johnson |
| 3 | jameswilliams | $2a$10$OkV5tCMMsy91pkkMXHa94OgcunNtuhxsQcxaOW6tJimuaCO0FMDZm | James | Williams |
| 4 | lindabrown | $2a$10$2NgAjstT9NcN58zMcF/Rq.pYt5bg3iQ6OmdRgR3YWfT.ZVgmJR4FK | Linda | Brown |

The following is the PHP code of the application that contains an SQL Injection vulnerability.

<?php/\*

\* Check if the 'id' GET variable is set

\* Example - http://localhost/?id=1

\*/

if (isset($\_GET['id'])){

$id = $\_GET['id'];

/\* Setup the connection to the database \*/

$mysqli = new mysqli('localhost', 'dbuser', 'dbpasswd', 'sql\_injection\_example');

/\* Check connection before executing the SQL query \*/

if ($mysqli->connect\_errno) {

printf("Connect failed: %s\n", $mysqli->connect\_error);

exit();

}

/\* SQL query vulnerable to SQL injection \*/

$sql = "SELECT username

FROM users

WHERE id = $id";

/\* Select queries return a result \*/

if ($result = $mysqli->query($sql)) {

while($obj = $result->fetch\_object()){

print($obj->username);

}

}

/\* If the database returns an error, print it to screen \*/

elseif($mysqli->error){

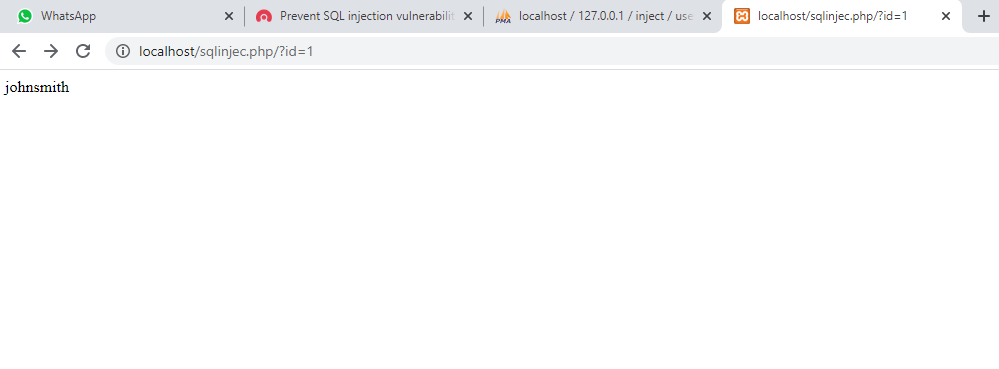
print($mysqli->error);

}

}

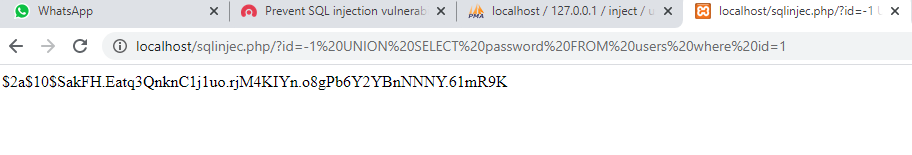
Legitimate HTTP request: <http://localhost/?id=1>

Output:



Malicious HTTP request: http://localhost/?id=-1 UNION SELECT password FROM users where id=1

Output:



## SQL Injection Prevention in PHP

### **Parameterized queries**

To prevent and/or fix SQL Injection vulnerabilities, start by reading advice in our [Defence in Depth](https://www.acunetix.com/websitesecurity/defence-in-depth-and-how-it-applies-to-web-applications/) series: [Parameterize SQL queries](https://www.acunetix.com/blog/articles/defence-in-depth-part-4-validate-everything-parameterize-sql-queries/). Parameterized queries are simple to write and understand. They force you to define the SQL query and use placeholders for user-provided variables in the query. After the SQL statement is defined, you can pass each parameter to the query. This allows the database to distinguish between the SQL command and data supplied by a user. If an attacker inputs SQL commands, the parameterized query treats them as untrusted input and the database does not execute injected SQL commands. If you properly parametrize SQL queries, all user input that is passed to the database is treated as data and can never be confused as being part of a command.

### **PHP Data Objects (PDO)**

Many PHP developers access databases using mysql or mysqli extensions. It is possible to use parameterized queries with the mysqli extension but PHP 5.1 introduced a better way to work with databases: PHP Data Objects (PDO). PDO provides methods that make parameterized queries easy to use. It also makes the code easier to read and more portable – it works with several databases, not just MySQL.The following example represents the same application as the one presented at the beginning of the article. This improved code uses PDO with parameterized queries to prevent the SQL Injection vulnerability.

<?php

/\*\*

\* Check if the 'id' GET variable is set

\* Example - http://localhost/?id=1

\*/

if (isset($\_GET['id'])){

$id = $\_GET['id'];

/\*\*

\* Validate data before it enters the database. In this case, we need to check that

\* the value of the 'id' GET parameter is numeric

\*/

if ( is\_numeric($id) == true){

try{ // Check connection before executing the SQL query

/\*\*

\* Setup the connection to the database This is usually called a database handle (dbh)

\*/

$dbh = new PDO('mysql:host=localhost;dbname=sql\_injection\_example', 'dbuser', 'dbpasswd');

/\*\*

\* Use PDO::ERRMODE\_EXCEPTION, to capture errors and write them to

\* a log file for later inspection instead of printing them to the screen.

\*/

$dbh->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

/\*\*

\* Before executing, prepare statements by binding parameters.

\* Bind validated user input (in this case, the value of $id) to the

\* SQL statement before sending it to the database server.

\*

\* This fixes the SQL injection vulnerability.

\*/

$q = "SELECT username

FROM users

WHERE id = :id";

// Prepare the SQL query string.

$sth = $dbh->prepare($q);

// Bind parameters to statement variables.

$sth->bindParam(':id', $id);

// Execute statement.

$sth->execute();

// Set fetch mode to FETCH\_ASSOC to return an array indexed by column name.

$sth->setFetchMode(PDO::FETCH\_ASSOC);

// Fetch result.

$result = $sth->fetchColumn();

/\*\*

\* HTML encode our result using htmlentities() to prevent stored XSS and print the

\* result to the page

\*/

print( htmlentities($result) );

//Close the connection to the database.

$dbh = null;

}

catch(PDOException $e){

/\*\*

\* You can log PDO exceptions to PHP's system logger, using the

\* log engine of the operating system

\*

\* For more logging options visit http://php.net/manual/en/function.error-log.php

\*/

error\_log('PDOException - ' . $e->getMessage(), 0);

/\*\*

\* Stop executing, return an Internal Server Error HTTP status code (500),

\* and display an error

\*/

http\_response\_code(500);

die('Error establishing connection with database');

}

} else{

/\*\*

\* If the value of the 'id' GET parameter is not numeric, stop executing, return

\* a 'Bad request' HTTP status code (400), and display an error

\*/

http\_response\_code(400);

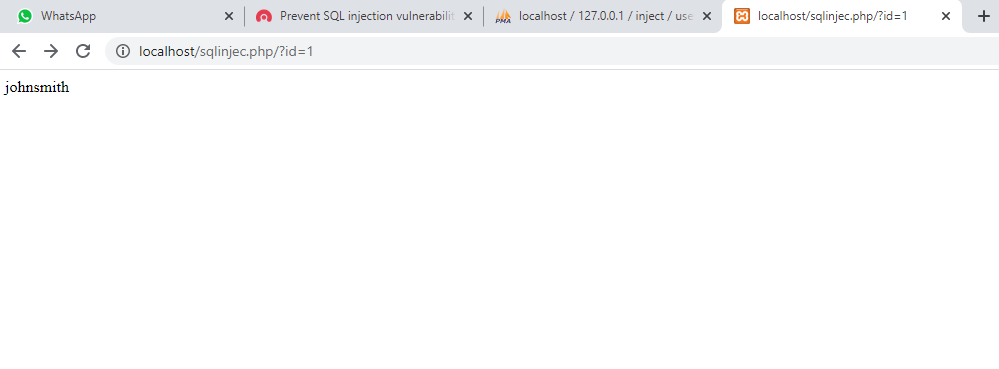
die('Error processing bad or malformed request');

}

}

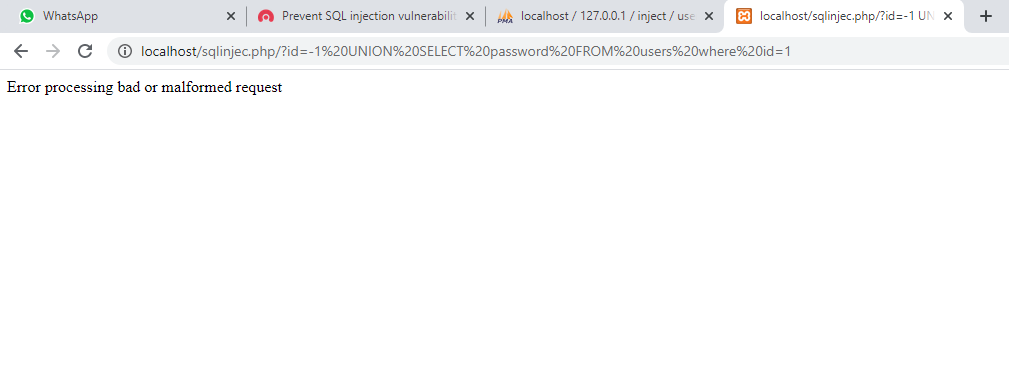
Legitimate HTTP request: <http://localhost/?id=1>

Output:



Malicious HTTP request: http://localhost/?id=-1 UNION SELECT password FROM users where id=1

Output:



## Conclusions

Parameterized queries solve SQL Injection vulnerabilities. This example uses PDO to fix the vulnerability but you can still use mysqli functions to prevent SQL Injection. However, PDO is easier to use, more portable, and supports the use of named parameters (in this example, we used :id as a named parameter).

**Reference**

<https://www.acunetix.com/blog/articles/prevent-sql-injection-vulnerabilities-in-php-applications/>