# Why shouldn't I use mysql_* functions in PHP?

What are the technical reasons that I shouldn't use `mysql_*` functions? (like `mysql_query()`, `mysql_connect()` or `mysql_real_escape_string()`)?

Why should I move away from them if they work on my site?

php mysql database pdo

edited Apr 25 at 20:11
Mark Amery
9,121 6 54 88

asked Oct 12 '12 at 13:18
Madara Uchiha
66.6k 25 108 161

We're looking for long answers that provide some explanation and context. Don't just give a one-line answer; explain why your answer is right, ideally with citations. Answers that don't include explanations may be removed.

## 9 Answers

The MySQL extension is:

- Not under active development
- **Officially deprecated** (as of PHP 5.5. Will be removed in PHP 7.)
- Lacks an OO interface
- Doesn't support:
  - Non-blocking, asynchronous queries
  - **Prepared statements or parameterized queries**
  - Stored procedures
  - Multiple Statements
  - Transactions
  - All of the functionality in MySQL 5.1

Since it is deprecated, using it makes your code less future proof.

Lack of support for prepared statements is particularly important as they provide a clearer, less error prone method of escaping and quoting external data than manually escaping it with a separate function call.

See **the comparison of SQL extensions**.

edited Jan 27 at 7:54

community wiki
13 revs, 8 users 56%
Quentin

107 Deprecated alone is reason enough to avoid them. They will not be there one day, and you will not be happy if you rely on them. The rest is just a list of things that using the old extensions has kept people from learning. – Tim Post ♦ Oct 12 '12 at 13:26

36 Deprecation isn't the magic bullet everyone seems to think it is. PHP itself will not be there one day, yet we rely on the tools we have at our disposal today. When we have to change tools, we will. – Lightning Racis in Obrit Dec 24 '12 at 14:29

40 @LightnessRacesinOrbit — Deprecation isn't a magic bullet, it is a flag that says "We recognise this sucks so we aren't going to support it for much longer". While having better future proofing of code is a good reason to move away from the deprecated features, it isn't the only one (or even the main one). Change tools because there are better tools, not because you are forced to. (And changing tools before you are forced to means that you aren't learning the new ones just because your code has stopped working and needs fixing yesterday … which is the worst time to learn new tools). – Quentin Dec 24 '12 at 17:43

35 Lacking an OO interface is a non-argument in this discussion. – nl-x May 29 '13 at 8:40

4 @symcbean, It surely does **not** support prepared statements. That's in fact the main reason why it's

deprecated. Without (easy to use) prepared statements the mysql extension often falls victim to SQL injection attacks. – rustyx Apr 5 '14 at 20:30

PHP offers three different APIs to connect to MySQL. There are the `mysql` , `mysqli` , and `PDO` extensions.

The `mysql_*` functions are very popular, but their use is not encouraged anymore. The documentation team is discussing the database security situation, and educating users to move away from the commonly used ext/mysql extension is part of this (check *php.internals: deprecating ext/mysql*).

And the later PHP developer team has taken the decision to generate `E_DEPRECATED` errors when users connect to MySQL, whether through `mysql_connect()` , `mysql_pconnect()` or the implicit connection functionality built into `ext/mysql` .

**`ext/mysql`** is now **officially deprecated as of PHP 5.5**!

**See the Red Box?**

When you go on any mysql_* function manual page, you see a red box, explaining it should not be used anymore.

## Why

Moving away from `ext/mysql` is not only about security, but also about having access to all the features of the MySQL database.

`ext/mysql` was built for **MySQL 3.23** and only got very few additions since then while mostly keeping compatibility with this old version which makes the code a bit harder to maintain. Missing features that is not supported by `ext/mysql` include:(from PHP manual)

- Stored procedures (can't handle multiple result sets)
- Prepared statements
- Encryption (SSL)
- Compression
- Full Charset support

**Reason to not use `mysql_*` function**

- Not under active development
- In deprecation process (so the intention is to remove it from a future version of PHP)
- Lacks an OO interface
- Doesn't support non-blocking, asynchronous queries
- Doesn't support prepared statements or parametrized queries
- Doesn't support stored procedures
- Doesn't support multiple statements
- Doesn't support transactions
- Doesn't support all of the functionality in MySQL 5.1

Above point quoted from Quentin's answer

Lack of support for prepared statements is particularly important as they provide a clearer, less error prone method of escaping and quoting external data than manually escaping it with a separate function call.

See the comparison of SQL extensions

**Suppressing deprecation warnings**

While code is being converted to MySQLi/PDO, `E_DEPRECATED` errors can be suppressed by setting error_reporting in **php.ini** to exclude `E_DEPRECATED:`

```
error_reporting = E_ALL ^ E_DEPRECATED
```

Note that this will also hide **other deprecation warnings**, which, however, may be for things other than MySQL.(from PHP manual)

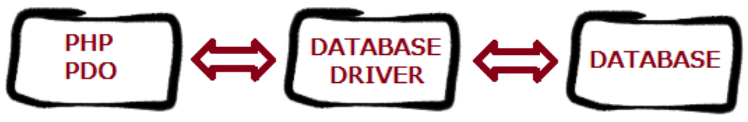
The article *PDO vs. MySQLi: Which Should You Use?* by **Dejan Marjanovic** will help you to choose.

And a better way is `PDO` , and I am now writing a simple PDO tutorial.

## A simple and short PDO tutorial

### Q.First question in my mind was: what is PDO?

A. "**PDO – PHP Data Objects** – is a database access layer providing a uniform method of access to multiple databases."



### Connecting to MySQL

With `mysql_*` function or we can say it the old way (deprecated in PHP 5.5 and above)

```
<?php
    $link = mysql_connect('localhost', 'user', 'pass');
    mysql_select_db('testdb', $link);
    mysql_set_charset('UTF-8', $link);
```

With `Pdo` : All you need to do is create a new PDO object. The constructor accepts parameters for specifying the database source PDO's constructor mostly takes four parameters which are DSN (data source name) and optionally username, password.

Here I think you are familiar with all except DSN; this is new in PDO. A DSN is basically a string of options that tell PDO which driver to use, and connection details. For further reference, check PDO MYSQL DSN.

```
<?php
    $db = new PDO('mysql:host=localhost;dbname=testdb;charset=utf8', 'username',
'password');
```

**Note:** you can also use `charset=UTF-8` , but sometimes it causes an error, so better to use `utf8` .

If there any connection error it will throw PDOException object it can be cached to handle Exception further.

**Good read** : Connections and Connection management ¶

You can also pass in several driver options as an array to the fourth parameter. I recommend passing the parameter which puts PDO into exception mode. Because some PDO drivers don't support native prepared statements, so PDO performs emulation of the prepare. It also lets you manually enable this emulation. To use the native server-side prepared statements, you should explicitly set it `false`

The other is to turn off prepare emulation which is enabled in the MySQL driver by default, but prepare emulation should be turned off to use PDO safely.

I will later explain why prepare emulation should be turned off To find reason please check this post .

It is only usable if you are using an old version of MySQL which I do not recommended.

Below I am showing how you can do it.

```
<?php
    $db = new PDO('mysql:host=localhost;dbname=testdb;charset=UTF-8',
                  'username',
                  'password',
                  array(PDO::ATTR_EMULATE_PREPARES => false,
                  PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
```

**Can we set attributes after PDO construction?**

**Yes**, we can also set some attributes after PDO construction with the `setAttribute` method:

```
<?php
    $db = new PDO('mysql:host=localhost;dbname=testdb;charset=UTF-8',
                  'username',
                  'password');
                  $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
                  $db->setAttribute(PDO::ATTR_EMULATE_PREPARES, false);
```

### Error Handling

Error handling is much easier in PDO than `mysql_*` .

A common practice when using `mysql_*` is:

```
<?php
    //Connected to MySQL
    $result = mysql_query("SELECT * FROM table", $link) or die(mysql_error($link));
```

`OR die()` is not a good way to handle the error since we can not handle the thing in `die` . It will just end the script abruptly and then echo the error to the screen which you usually do NOT want to show to your end users, and let bloody hackers discover your schema. Alternately, the return values of `mysql_*` functions can often be used in conjunction with mysql_error() to handle errors.

PDO offers a better solution: exceptions. Anything we do with PDO should be wrapped in a `try` - `catch` block. We can force PDO into one of three error modes by setting the error mode attribute. Three error handling modes are below.

- `PDO::ERRMODE_SILENT` . It's just setting error codes and acts pretty much the same as `mysql_*` where you must check each result and then look at `$db->errorInfo();` to get the error details.
- `PDO::ERRMODE_WARNING` Raise `E_WARNING` . (Run-time warnings (non-fatal errors). Execution of the script is not halted.)
- `PDO::ERRMODE_EXCEPTION` : Throw exceptions. It represents an error raised by PDO. You should not throw a `PDOException` from your own code. See *Exceptions* for more information about exceptions in PHP. It acts very much like `or die(mysql_error());` , when it isn't caught. But unlike `or die()` , the `PDOException` can be caught and handled gracefully if you choose to do so.

**Good read**

- Errors and error handling ¶
- The PDOException class ¶
- Exceptions ¶

Like:

```
$stmt->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_SILENT );
$stmt->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING );
$stmt->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
```

And you can wrap it in `try` - `catch` , like below:

```
<?php
try {
    //Connect as appropriate as above
    $db->query('hi'); //Invalid query!
}
catch (PDOException $ex) {
    echo "An Error occured!"; //User friendly message/message you want to show to user
    some_logging_function($ex->getMessage());
}
```

You do not have to handle with `try` - `catch` right now. You can catch it at any time appropriate, but I strongly recommend you to use `try` - `catch` . Also it may make more sense to catch it at outside the function that calls the PDO stuff:

```
<?php
    function data_fun($db) {
       $stmt = $db->query("SELECT * FROM table");
       return $stmt->fetchAll(PDO::FETCH_ASSOC);
    }

    //Then later
    try {
       data_fun($db);
    }
    catch(PDOException $ex) {
       //Here you can handle error and show message/perform action you want.
    }
```

Also, you can handle by `or die()` or we can say like `mysql_*` , but it will be really varied. You can hide the dangerous error messages in production by turning `display_errors off' and just reading your error log.

Now, after reading all the things above, you are probably thinking: what the heck is that when I just want to start leaning simple `SELECT` , `INSERT` , `UPDATE` , or `DELETE` statements? Don't worry, here we go:

## Selecting Data

OBJECTS
PHP PDO
ARRAYS

So what you are doing in `mysql_*` is:

```
<?php
    $result = mysql_query('SELECT * from table') or die(mysql_error());

    $num_rows = mysql_num_rows($result);

    while($row = mysql_fetch_assoc($result)) {
        echo $row['field1'];
    }
```

Now in PDO, you can do this like:

```
<?php
    $stmt = $db->query('SELECT * FROM table');

    while($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
        echo $row['field1'];
    }
```

Or

```
<?php
    $stmt = $db->query('SELECT * FROM table');
    $results = $stmt->fetchAll(PDO::FETCH_ASSOC);

    //Use $results
```

**Note:**: if you are using the method like below ( `query()` ), this method returns a `PDOStatement` object. So if you want to fetch the result, use it like above.

```
<?php
    foreach($db->query('SELECT * FROM table') as $row) {
        echo $row['field1'];
    }
```

In PDO Data, it is obtained via the `->fetch()` , a method of your statement handle. Before calling fetch, the best approach would be telling PDO how you'd like the data to be fetched. In the below section I am explaining this.

## Fetch Modes

Note the use of `PDO::FETCH_ASSOC` in the `fetch()` and `fetchAll()` code above. This tells `PDO` to return the rows as an associative array with the field names as keys. There are many other fetch modes too which I will explain one by one.

First of all, I explain how to select fetch mode:

```
$stmt->fetch(PDO::FETCH_ASSOC)
```

In the above, I have been using `fetch()` . You can also use:

- `PDOStatement::fetchAll()` - Returns an array containing all of the result set rows
- `PDOStatement::fetchColumn()` - Returns a single column from the next row of a result set
- `PDOStatement::fetchObject()` - Fetches the next row and returns it as an object.
- `PDOStatement::setFetchMode()` - Set the default fetch mode for this statement

Now I come to fetch mode:

- `PDO::FETCH_ASSOC` : returns an array indexed by column name as returned in your result set
- `PDO::FETCH_BOTH` (default): returns an array indexed by both column name and 0-indexed column number as returned in your result set

There are even more choices! Read about them all in `PDOStatement` Fetch documentation..

### Getting the Row Count

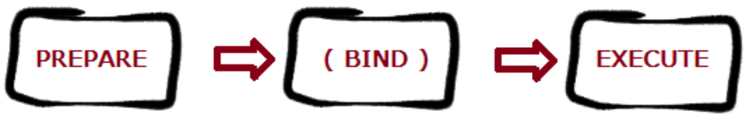
Instead of using mysql_num_rows to get the number of returned rows, you can get a `PDOStatement` and do `rowCount();` , like:

```
<?php
    $stmt = $db->query('SELECT * FROM table');
    $row_count = $stmt->rowCount();
    echo $row_count.' rows selected';
```

### Getting the Last Inserted ID

```
<?php
    $result = $db->exec("INSERT INTO table(firstname, lastname) VAULES('John', 'Doe')");
    $insertId = $db->lastInsertId();
```

## Insert and Update or Delete statements



What we are doing in `mysql_*` function is:

```
<?php
    $results = mysql_query("UPDATE table SET field='value'") or die(mysql_error());
    echo mysql_affected_rows($result);
```

And in pdo, this same thing can be done by:

```
<?php
    $affected_rows = $db->exec("UPDATE table SET field='value'");
    echo $affected_rows;
```

In the above query `PDO::exec` (execute an SQL statement and returns the number of affected rows)

(Insert and delete will be covered later.)

The above method is only useful when you are not using variable in query. But when you need to use a variable in a query, do not ever ever try like the above and there for **prepared statement or parameterized statement** is.

## Prepared Statements

**Q.** What is a prepared statement and why do I need them?
**A.** A prepared statement is a precompiled SQL statement that can be executed multiple times by sending only the data to the server.

The typical workflow of using a prepared statement is as follows (quoted from Wikipedia three 3 point):

1. **Prepare**: The statement template is created by the application and sent to the database management system (DBMS). Certain values are left unspecified, called parameters, placeholders or bind variables (labelled "?" below):

   ```
   INSERT INTO PRODUCT (name, price) VALUES (?, ?)
   ```

2. The DBMS parses, compiles, and performs query optimization on the statement template, and stores the result without executing it.

3. **Execute**: At a later time, the application supplies (or binds) values for the parameters, and the DBMS executes the statement (possibly returning a result). The application may execute the statement as many times as it wants with different values. In this example, it might supply 'Bread' for the first parameter and '1.00' for the second parameter.

You can use a prepared statement by including placeholders in your SQL. There are basically three ones without placeholders (don't try this with variable its above one), one with unnamed placeholders, and one with named placeholders.

**Q.** So now, what are named placeholders and how do I use them?
**A.** Named placeholders. Use descriptive names preceded by a colon, instead of question marks. We don't care about position/order of value in name place holder:

```
  $stmt->bindParam(':bla', $bla);
```

`bindParam(parameter,variable,data_type,length,driver_options)`

You can also bind using an execute array as well:

```
<?php
    $stmt = $db->prepare("SELECT * FROM table WHERE id=:id AND name=:name");
    $stmt->execute(array(':name' => $name, ':id' => $id));
    $rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

Another nice feature for OOP friends is that named placeholders have the ability to insert objects directly into your database, assuming the properties match the named fields. For example:

```
class person {
    public $name;
    public $add;
    function __construct($a,$b) {
        $this->name = $a;
        $this->add = $b;
```

```
    }

}
$demo = new person('john','29 bla district');
$stmt = $db->prepare("INSERT INTO table (name, add) value (:name, :add)");
$stmt->execute((array)$demo);
```

**Q.** So now, what are unnamed placeholders and how do I use them?
**A.** Let's have an example:

```
<?php
    $stmt = $db->prepare("INSERT INTO folks (name, add) values (?, ?)");
    $stmt->bindValue(1, $name, PDO::PARAM_STR);
    $stmt->bindValue(2, $add, PDO::PARAM_STR);
    $stmt->execute();
```

and

```
 $stmt = $db->prepare("INSERT INTO folks (name, add) values (?, ?)");
 $stmt->execute(array('john', '29 bla district'));
```

In the above, you can see those `?` instead of a name like in a name place holder. Now in the first example, we assign variables to the various placeholders ( `$stmt->bindValue(1, $name, PDO::PARAM_STR);` ). Then, we assign values to those placeholders and execute the statement. In the second example, the first array element goes to the first `?` and the second to the second `?` .

**NOTE**: In **unnamed placeholders** we must take care of the proper order of the elements in the array that we are passing to the `PDOStatement::execute()` method.

## SELECT,INSERT,UPDATE,DELETE prepared queries

1 **SELECT**

```
<?php
    $stmt = $db->prepare("SELECT * FROM table WHERE id=:id AND name=:name");
    $stmt->execute(array(':name' => $name, ':id' => $id));
    $rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

2 **INSERT**

```
$stmt = $db->prepare("INSERT INTO table(field1,field2) VALUES(:field1,:field2)");
$stmt->execute(array(':field1' => $field1, ':field2' => $field2));
$affected_rows = $stmt->rowCount();
```

3 **DELETE**

```
<?php
    $stmt = $db->prepare("DELETE FROM table WHERE id=:id");
    $stmt->bindValue(':id', $id, PDO::PARAM_STR);
    $stmt->execute();
    $affected_rows = $stmt->rowCount();
```

4 **UPDATE**

```
<?php
    $stmt = $db->prepare("UPDATE table SET name=? WHERE id=?");
    $stmt->execute(array($name, $id));
    $affected_rows = $stmt->rowCount();
```

## NOTE:

However PDO/MySQLi are not completely safe. Check the answer *Are PDO prepared statements sufficient to prevent SQL injection?* by ircmaxell. Also, I am quoting some part from his answer:

```
$pdo->setAttribute(PDO::ATTR_EMULATE_PREPARES, false);
$pdo->query('SET NAMES GBK');
$stmt = $pdo->prepare("SELECT * FROM test WHERE name = ? LIMIT 1");
$stmt->execute(array(chr(0xbf) . chr(0x27) . " OR 1=1 /*"));
```

edited Apr 25 at 20:36
Mark Amery
9,121 6 54 88

answered Jan 1 '13 at 11:52
NullPoiиteя
27k 11 58 94

2 What the good read above should propably mention: prepared statement take away any meaningful use of the `IN (...) construct`. – Eugen Rieck Dec 14 '13 at 19:35

46 This is the longest and most complete answer I've ever seen on Stackoverflow ! Good job dude ;) – Amine Mar 18 '14 at 14:11

1 such a great explain !! – Up_One Mar 30 '14 at 21:24

3 @Amine, No, it is not! :] While *NullPoiteя* really did a great job of writing it, this is most certainly isn't a good reading, because it is way to long. I'm pretty sure, that 8 of 10 visitors will simply skip it. And you also have explanation, why this answer isn't top voted. A `tl;dr` part in the beginning would be a good

idea, I think. – trejder Jul 4 '14 at 11:00

1 The question was "Why shouldnt I use mysql_* functions in PHP". This answer, while impressive and full of helpful information, goes WAY out of scope and like @trejder says - 8 out of 10 people are going to miss out on that information simply because they don't have 4 hours to spend trying to work through it. This would be far more valuable broken up and used as answers to several, more precise, questions. – Alex McMillan Nov 19 '14 at 22:33

First, let's begin with the standard comment we give everyone:

> **Please, don't use `mysql_*` functions in new code**. They are no longer maintained and are officially deprecated. See the **red box**? Learn about *prepared statements* instead, and use PDO or MySQLi - this article will help you decide which. If you choose PDO, here is a good tutorial.

Let's go through this, sentence by sentence, and explain:

- **They are no longer maintained, and are officially deprecated**

  This means that the PHP community is gradually dropping support for these very old functions. They are likely to not exist in a future (recent) version of PHP! Continued use of these functions may break your code in the (not so) far future.

  **NEW! - ext/mysql is now *officially deprecated as of PHP 5.5!***

  ### Newer! ext/mysql *will be removed in PHP 7*

- **Instead, you should learn of prepared statements** -

  `mysql_*` extension does not support **prepared statements**, which is (among other things) a very effective countermeasure against **SQL Injection**. It fixed a very serious vulnerability in MySQL dependent applications which allows attackers to gain access to your script and perform **any possible query** on your database.

  For more information, see **How can I prevent SQL-injection in PHP?**

- **See the Red Box?**

  When you go on any `mysql` function manual page, you see a red box, explaining it should not be used anymore.

- **Use either PDO or MySQLi**

  There are better, more robust and well built alternatives, **PDO - PHP Database Object**, which offers a complete OOP approach to database interaction, and **MySQLi**, which is a MySQL specific improvement.

edited Jan 14 at 10:19

answered Oct 12 '12 at 13:28

Madara Uchiha
**66.6k** 25 108 161

6 "Deprecation process" is scaremongering (and it's not like php-dev has actual processes anyway). The use of mysql_* functions has been ***discouraged***. That's not news however. That's a PHP4.4 era development. An actual selling point is *ease of use*, and the obvious security benefit as by-product. Which I think is more likely to convince newbies than the hypothetical function drop, or the link-ladden 08/15 comment (more likely to overcharge our typical php.net/manual eschewers). – mario Nov 5 '12 at 22:19

2 There is one more thing: i think that function still exists in PHP for only one reason - compatibility with old, outdated but still running CMS, e-commerce, bulletin board systems etc. Finally it will be removed and you will have to rewrite your application... – Kamil Nov 13 '12 at 11:49

@Kamil: That's true, but it's not really a reason why you shouldn't use it. The reason not to use it is because it's ancient, insecure, etc. :) – Madara Uchiha Nov 13 '12 at 11:50

@Madara Uchiha i completely agree with that. I just added some background information. – Kamil Nov 13 '12 at 11:51

1 @Mario -- the PHP devs do have a process, and they've just voted in favour of formally deprecating ext/mysql as of 5.5. It's no longer a hypothetical issue. – SDC Dec 10 '12 at 15:00

## Ease of use

The analytic and synthetic reasons were already mentioned. For newcomers there's a more significant incentive to stop using the dated mysql_ functions.

**Contemporary database APIs are just *easier* to use.**

It's mostly the *bound parameters* which can simplify code. And with excellent tutorials (as seen above) the transition to PDO isn't overly arduous.

Rewriting a larger code base at once however takes time. Raison d'être for this intermediate alternative:

### Equivalent pdo_* functions in place of ~~mysql_*~~

Using <pdo_mysql.php> you can switch from the old mysql_ functions with *minimal effort*. It adds `pdo_` function wrappers which replace their `mysql_` counterparts.

1. Simply `include_once( "pdo_mysql.php" );` in each invocation script that has to interact with the database.

2. Remove the ~~`mysql_`~~ function prefix *everywhere* and replace it with `pdo_`.

   - ~~`mysql_`~~ `connect()` becomes **`pdo_`** `connect()`
   - ~~`mysql_`~~ `query()` becomes **`pdo_`** `query()`
   - ~~`mysql_`~~ `num_rows()` becomes **`pdo_`** `num_rows()`
   - ~~`mysql_`~~ `insert_id()` becomes **`pdo_`** `insert_id()`
   - ~~`mysql_`~~ `fetch_array()` becomes **`pdo_`** `fetch_array()`
   - ~~`mysql_`~~ `fetch_assoc()` becomes **`pdo_`** `fetch_assoc()`
   - ~~`mysql_`~~ `real_escape_string()` becomes **`pdo_`** `real_escape_string()`
   - *and so on...*

3. Your code will work alike and still mostly look the same:

```
include_once("pdo_mysql.php");

pdo_connect("localhost", "usrABC", "pw1234567");
pdo_select_db("test");

$result = pdo_query("SELECT title, html FROM pages");

while ($row = pdo_fetch_assoc($result)) {
    print "$row[title] - $row[html]";
}
```

Et voilà.
Your code is *using* PDO.
Now it's time to actually *utilize* it.

### Bound parameters can be easy to use

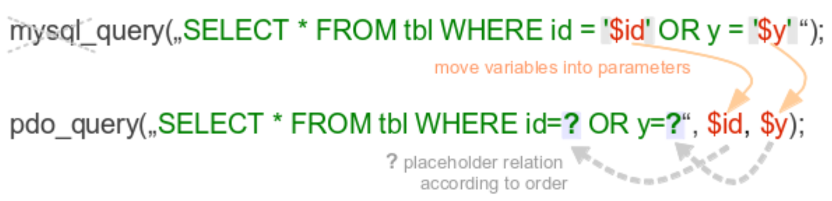
->prepare() ->bindValue() ->bindValue()
->bindValue() ->bindValue() ->bindValue
->bindValue()
->bindValue()
->bindValue()
->bindValue()
->bindValue()
->execute()
Ain't nobody got time for that

~~`mysql_query`~~ requires to mix values into SQL queries. Due to omnipresent horridly outdated tutorials newcomers seldomly learn about escaping variables. And even then it's often forgotten. It's a cumbersome approach anyway.

Prepared statements, or more specifically *bound parameters*, make that redundant. But once you check out `PDO` or particularly `MYSQLI` howtos you'd get the impression that'll be even more overhead.

You just need a less unwieldy API.

`pdo_query()` adds very facile support for bound parameters. Converting old code is straightforward:



```
mysql_query(„SELECT * FROM tbl WHERE id = '$id' OR y = '$y' “);

pdo_query(„SELECT * FROM tbl WHERE id=? OR y=?“, $id, $y);
```

Move your variables out of the SQL string.

- Add them as comma delimited function parameters to `pdo_query()`.
- Place question marks `?` as placeholders where the variables were before.
- Get rid of `'` single quotes that previously enclosed string values/variables.

The advantage becomes more obvious for lengthier code.

Often string variables aren't just interpolated into SQL, but concatenated with escaping calls in between.

```
pdo_query("SELECT id, links, html, title, user, date FROM articles
   WHERE title='" . pdo_real_escape_string($title) . "' OR id='".
```

```
    pdo_real_escape_string($title) . "' AND user <> '" .
    pdo_real_escape_string($root) . "' ORDER BY date")
```

With `?` placeholders applied you don't have to bother with that:

```
pdo_query("SELECT id, links, html, title, user, date FROM articles
   WHERE title=? OR id=? AND user<>? ORDER BY date", $title, $id, $root)
```

Remember that pdo_* still allows *either or*.
Just don't escape a variable *and* bind it in the same query.

- The placeholder feature is provided by the real PDO behind it.
- Thus also allowed `:named` placeholder lists later.

More importantly you can pass $_REQUEST[] variables safely behind any query. When submitted `<form>` fields match the database structure exactly it's even shorter:

```
pdo_query("INSERT INTO pages VALUES (?,?,?,?,?)", $_POST);
```

So much simplicity. But let's get back to some more rewriting advises and technical reasons on why you may want to get rid of ~~`mysql_`~~ and escaping.

## Fix or remove any oldschool `sanitize()` function

Once you have converted all ~~`mysql_`~~ calls to `pdo_query` with bound params, remove all redundant `pdo_real_escape_string` calls.

In particular you should fix any `sanitize` or `clean` or `filterThis` or `clean_data` functions as advertised by dated tutorials in one form or the other:

```
function sanitize($str) {
   return trim(strip_tags(htmlentities(pdo_real_escape_string($str))));
}
```

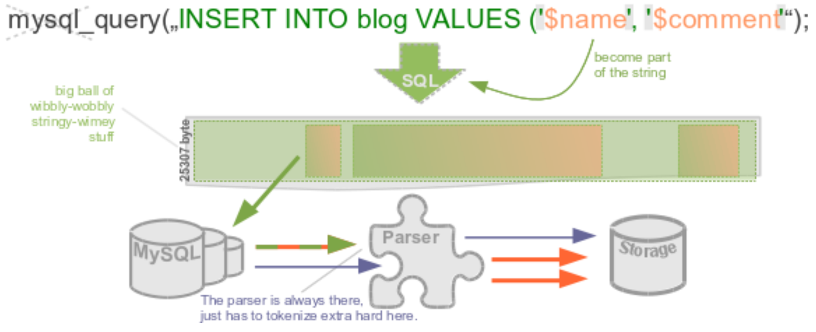
Most glaring bug here is the lack of documentation. More significantly the order of filtering was in exactly the wrong order.

- Correct order would have been: deprecatedly `stripslashes` as the innermost call, then `trim` , afterwards `strip_tags` , `htmlentities` for output context, and only lastly the `_escape_string` as its application should directly preceed the SQL intersparsing.
- But as first step just **get rid of the `_real_escape_string`** call.
- You may have to keep the rest of your `sanitize()` function for now if your database and application flow expect HTML-context-safe strings. Add a comment that it applies only HTML escaping henceforth.
- String/value handling is delegated to PDO and its parameterized statements.
- If there was any mention of `stripslashes()` in your sanitize function, it may indicate a higher level oversight.
    - That was commonly there to undo damage (double escaping) from the deprecated `magic_quotes` . Which however is best fixed centrally, not string by string.
    - Use one of the userland reversal approaches. Then remove the `stripslashes()` in the `sanitize` function.

> **Historic note on magic_quotes.** That feature is rightly deprecated. It's often incorrectly portrayed as failed *security* feature however. But magic_quotes are as much a failed security feature as tennis balls have failed as nutrition source. That simply wasn't their purpose.
>
> The original implementation in PHP2/FI introduced it explicitly with just "*quotes will be automatically escaped making it easier to pass form data directly to msql queries*". Notably it was accidentially safe to use with mSQL, as that supported ASCII only. Then PHP3/Zend reintroduced magic_quotes for MySQL and misdocumented it. But originally it was just a convenience feature, not intend for security.

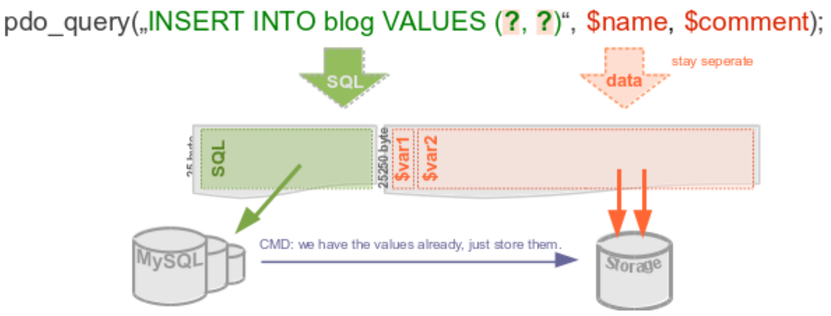
## How prepared statements differ

When you scramble string variables into the SQL queries, it doesn't just get more intricate for you to follow. It's also extraneous effort for MySQL to segregate code and data again.


mysql_query(„INSERT INTO blog VALUES ('$name', '$comment'");
SQL
become part of the string
big ball of wibbly-wobbly stringy-wimey stuff
25307 byte
MySQL
Parser
Storage
The parser is always there, just has to tokenize extra hard here.

SQL injections simply are when *data bleeds into code* context. A database server can't later spot where PHP originally glued variables inbetween query clauses.

With bound parameters you separate SQL code and SQL-context values in your PHP code. But it doesn't get shuffled up again behind the scenes (except with PDO::EMULATE_PREPARES). Your database receives the unvaried SQL commands and 1:1 variable values.


pdo_query(„INSERT INTO blog VALUES (?, ?)", $name, $comment);
SQL
data
stay seperate
25 byte
SQL
25250 byte
$var1
$var2
MySQL
CMD: we have the values already, just store them.
Storage

While this answer stresses that you should care about the readability advantages of dropping `~~mysql_~~` . There's occasionally also a performance advantage (repeated INSERTs with just differing values) due to this visible and technical data/code separation.

Beware that parameter binding still isn't a magic one-stop solution against *all* SQL injections. It handles the most common use for data/values. But can't whitelist column name / table identifiers, help with dynamic clause construction, or just plain array value lists.

### Hybrid PDO use

These `pdo_*` wrapper functions make a coding-friendly stop-gap API. (It's pretty much what `MYSQLI` could have been if it wasn't for the idiosyncratic function signature shift). They also expose the real PDO at most times.
Rewriting doesn't have to stop at using the new pdo_ function names. You could one by one transition each pdo_query() into a plain $pdo->prepare()->execute() call.

It's best to start at simplifying again however. For example the common result fetching:

```
$result = pdo_query("SELECT * FROM tbl");
while ($row = pdo_fetch_assoc($result)) {
```

Can be replaced with just an foreach iteration:

```
foreach ($result as $row) {
```

Or better yet a direct and complete array retrieval:

```
$result->fetchAll();
```

You'll get more helpful warnings in most cases than PDO or mysql_ usually provide after failed queries.

### Other options

So this hopefully visualized some *practical* reasons and a worthwile pathway to drop `~~mysql_~~` .

Just switching to `pdo` doesn't quite cut it. `pdo_query()` is also just a frontend onto it.

Unless you also introduce parameter binding or can utilize something else from the nicer API, it's a pointless switch. I hope it's portrayed simple enough to not further the discouragement to newcomers. (Education usually works better than prohibition.)

While it qualifies for the simplest-thing-that-could-possibly-work category, it's also still very experimental code. I just wrote it over the weekend. There's a plethora of alternatives however. Just google for PHP database abstraction and browse a little. There always have been and will be lots of excellent libraries for such tasks.

If you want to simplify your database interaction further, mappers like Paris/Idiorm are worth a try. Just like nobody uses the bland DOM in JavaScript anymore, you don't have to babysit a raw database interface nowadays.

answered Dec 24 '13 at 23:30

mario
**85.8k** 7 92 173

1 Very good answer! When on Getty back to my pc I'll have it accepted for a while. – Madara Uchiha Dec 25 '13 at 5:25

Thanks. I'm mainly fond of the abstract pictograms visualizing parameterized queries. But I'll probably expand more on a later second conversion from pdo_query() to real PDO:: usage. – mario Dec 25 '13 at 21:32

1 Be careful with the `pdo_query("INSERT INTO pages VALUES (?,?,?,?,?)", $_POST);` function - ie: `pdo_query("INSERT INTO users VALUES (?, ?, ?), $_POST); $_POST = array( 'username' => 'lawl', 'password' => '123', 'is_admin' => 'true');` – rickyduck Jan 22 '14 at 16:35

---

The `mysql_` functions are:

1. out of date - they're not maintained any more
2. don't allow you to move easily to another database backend
3. don't support prepared statements, hence
4. encourage programmers to use concatenation to build queries, leading to SQL injection vulnerabilities

answered Oct 12 '12 at 13:22

Alnitak
**168k** 34 221 302

11 #2 is equally true of `mysqli_` – eggyal Oct 12 '12 at 13:35

7 to be fair, given the variations in SQL dialect, even PDO doesn't give you #2 with any degree of certainty. You'd need a proper ORM wrapper for that. – SDC Oct 23 '12 at 14:55

1 @SDC indeed - the problem with standards is that there's *so* many of them... – Alnitak Oct 23 '12 at 15:06

3 xkcd.com/927 – Lightning Racis in Obrit Dec 24 '12 at 14:30

the `mysql_*` function are a shell into mysqlnd functions for newer PHP versions. So even if the old client library is not maintained any more, mysqlnd is maintained :) – hakre May 31 '13 at 14:25

---

Speaking of *technical* reasons, there are only few, extremely specific and rarely used. Most likely you will never ever use them in your life.
May be I am too ignorant, but I never had an opportunity to use them things like

- non-blocking, asynchronous queries
- stored procedures returning multiple resultsets
- Encryption (SSL)
- Compression

If you need them - these are no doubt technical reasons to move away from mysql extension toward something more stylish and modern-looking.

**Nevertheless, there are also some non-technical issues, which can make your experience a bit harder**

- further use of these functions with modern PHP versions will raise deprecated-level notices. They simply can be turned off.
- in a distant future they can be possibly removed from the default PHP build. Not a big deal too, as mydsql ext will be moved into PECL and every hoster will be happy to complie PHP with it, as they don't want to lose clients whose sites were working for decades.
- strong resistance from Stackoverflow community. Every time you mention these honest functions, you being told that they are under strict taboo.
- being an average php user, most likely your idea of using these functions is error-prone and wrong. Just because of all these numerous tutorials and manuals which teach you wrong way. Not the functions themselves - I have to emphasize it - but the way they are used.

This latter issue is a problem.
But, to my opinion, proposed solution is no better either.
It seems to me **too idealistic** a dream that all those PHP users will learn how to handle SQL queries properly at once. Most likely they would just change mysql_* to mysqli_* mechanically, **leaving the approach the same**. Especially because mysqli makes prepared statements usage incredible painful and troublesome.
Not to mention that *native* prepared statements **aren't enough to protect** from SQL injections,

and neither mysqli nor PDO offers a solution.

So, instead of fighting this honest extension, I'd prefer to fight wrong practices and educate people in the right ways.

**Also, there are some false or non-significant reasons, like**

- Doesn't support Stored Procedures (we were using `mysql_query("CALL my_proc");` for ages)
- Doesn't support Transactions (same as above)
- Doesn't support Multiple Statements (who need them?)
- Not under active development (so what? does it affect *you* in any practical way?)
- Lacks an OO interface (to create one is a matter of several hours)
- Doesn't support Prepared Statements or Parametrized Queries

A latter one is an interesting point. Although mysql ext do not support *native* prepared statements, they aren't required for the safety. We can easily fake prepared statements using manually handled placeholders (just like PDO does):

```
function paraQuery()
{
    $args  = func_get_args();
    $query = array_shift($args);
    $query = str_replace("%s","'%s'",$query);

    foreach ($args as $key => $val)
    {
        $args[$key] = mysql_real_escape_string($val);
    }

    $query  = vsprintf($query, $args);
    $result = mysql_query($query);
    if (!$result)
    {
        throw new Exception(mysql_error()." [$query]");
    }
    return $result;
}

$query  = "SELECT * FROM table where a=%s AND b LIKE %s LIMIT %d";
$result = paraQuery($query, $a, "%$b%", $limit);
```

*voila*, everything is parameterized and safe.

But okay, if you don't like the red box in the manual, a problem of choice arises: mysqli or PDO?

Well, the answer would be as follows:

- If you understand the necessity of using a *database abstraction layer*, and looking for an API to create one, **mysqli** is a very good choice, as it indeed supports many mysql-specific features.
- If, like wast majority of PHP folks, you are using raw API calls right in the application code (which is essentially wrong practice) - **PDO is the only choice**, as this extension pretends to be not just API but rather a semi-DAL, still incomplete but offers many important features, with two of them makes PDO critically distinguished from mysqli:
  - unlike mysqli, PDO can bind placeholders *by value*, which makes dynamically built queries feasible without several screens of quite messy code.
  - unlike mysqli, PDO can always return query result in a simple usual array, while mysqli can do it only on mysqlnd installations.

So, if you are an average PHP user and want to save yourself a ton of headaches when using native prepared statements, PDO - again - is the only choice.
However, PDO is not a silver bullet too, and has it's hardships.
So, I wrote solutions for all the common pitfalls and complex cases in the **PDO tag wiki**

Nevertheless, everyone talking of extensions always missing the **2 important facts** about Mysqli and PDO:

1. Prepared statement **isn't a silver bullet**. There are dynamical identifiers which cannot be bound using prepared statements. There are dynamical queries with unknown number of parameters which makes query building a difficult task.

2. **Neither mysqli_* nor PDO functions should be appeared in the application code.**
   There ought to be an **abstraction layer** between them and application code, which will do all the dirty job of binding, looping, error handling, etc. inside, making application code DRY and clean. Especially for the complex cases like dynamical query building.

So, just switching to PDO or mysqli is not enough. One have to use an ORM, or a query builder, or whatever database abstraction class instead of calling raw API functions in their code.
And contrary - if you have an abstraction layer between your application code and mysql API - **it doesn't actually matter which engine is used.** You can use mysql ext until it goes deprecated and then easily rewrite your abstraction class to another engine, **having all the application code intact.**

Here are some examples based on my safemysql class to show how such an abstraction class ought to be:

```
$city_ids = array(1,2,3);
$cities   = $db->getCol("SELECT name FROM cities WHERE is IN(?a)", $city_ids);
```

Compare this one single line with amount of code you will need with PDO.
Then compare with crazy amount of code you will need with raw Mysqli prepared statements.
Note that error handling, profiling, query logging already built in and running.

```
$insert = array('name' => 'John', 'surname' => "O'Hara");
$db->query("INSERT INTO users SET ?u", $insert);
```

Compare it with usual PDO inserts, when every single field name being repeated six to ten times - in all these numerous named placeholders, bindings and query definitions.

Another example:

```
$data = $db->getAll("SELECT * FROM goods ORDER BY ?n", $_GET['order']);
```

You can hardly find an example for PDO to handle such practical case.
And it will be too wordy and most likely unsafe.

So, once more - it is not just raw driver should be your concern but abstraction class, useful not only for silly examples from beginner's manual but to solve whatever real life problems.

edited Sep 17 '13 at 9:03 answered Jan 1 '13 at 17:42
Your Common Sense
1

11 `mysql_*` makes vulnerabilities very easy to come by. Since PHP is used by a whole lot of novice users, `mysql_*` is actively harmful in practice, even if in theory it can be used without a hitch. – Madara Uchiha Jan 1 '13 at 17:48

`everything is parameterized and safe` - it may be parameterized, but your function doesn't use ***real*** prepared statements. – uınbeƆs Jan 3 '13 at 6:07

2 How is `Not under active development` only for that made-up '0.01%'? If you build something with this stand-still function, update your mysql-version in a year and wind up with a non-working system, I'm sure there are an awful lot of people suddenly in that '0.01%'. I'd say that `deprecated` and `not under active development` are closely related. You can say that there is "no [worthy] reason" for it, but the fact is that when offered a choice between the options, `no active development` is almost just as bad as `deprecated` I'd say? – Nanne Feb 1 '13 at 10:21

@MadaraUchiha: Can you explain how vulnerabilities are very easy to come by? Especially in the cases where those same vulnerabilities don't affect PDO or MySQLi... Because I'm not aware of a single one that you speak of. – ircmaxell Feb 4 '13 at 12:42

2 @ShaquinTrifonoff: sure, it doesn't use prepared statements. But neither does PDO, which most people recommend over MySQLi. So I'm not sure that has a significant impact here. The above code (with a little more parsing) is what PDO does when you prepare a statement by default... – ircmaxell Feb 4 '13 at 12:44

There are many reasons, but perhaps the most important one is that those functions encourage insecure programming practices because they do not support prepared statements. Prepared statements help prevent SQL injection attacks.

When using `mysql_*` functions, you have to remember to run user-supplied parameters through `mysql_real_escape_string()`. If you forget in just one place or if you happen to escape only part of the input, your database may be subject to attack.

Using prepared statements in `PDO` or `mysqli` will make it so that these sorts of programming errors are more difficult to make.

edited Oct 12 '12 at 13:41 answered Oct 12 '12 at 13:23
Trott
19.2k 12 49 99

1 Unfortunately the poor support in MySQLi_* for passing a variable number of parameters (such as when you want to pass a list of values to check against in an IN clause) encourages non use of parameters, encouraging the use of exactly the same concatenated queries that leave MySQL_* calls vulnerable. – Kickstart Jun 27 '13 at 9:31

2 But, once again, insecurity is not an inherent problem of mysql_* functions, but a problem of incorrect usage. – Agamemnus Feb 2 '14 at 5:29

@Agamemnus The problem is that mysql_* makes it easy to implement that "incorrect usage", especially for inexperienced programmers. Libraries that implement prepared statements make it harder to make that type of error. – Trott Feb 2 '14 at 16:33

Because (amongst other reasons) it's much harder to ensure the input data is sanitized. If you use parametrized queries, as one does with PDO or mysqli you can entirely avoid the risk.

As an example, some-one could use "enhzflep); drop table users" as a user name. The old functions will allow executing of multiple statements per query, so something like that nasty bugger can delete a whole table.

If one were to use PDO of mysqli, the user-name would end-up being "enhzflep); drop table users"

See here: http://bobby-tables.com/

answered Oct 12 '12 at 13:24
enhzflep
6,138 1 9 27

8 `The old functions will allow executing of multiple statements per query` - no, they won't. That kind of injection is not possible with ext/mysql - the only way this kind of injection is possible with PHP and MySQL is when using MySQLi and the `mysqli_multi_query()` function. The kind injection that is possible with ext/mysql and unescaped strings is things like `' OR '1' = '1` to extract data from the database that was not meant to be accessible. In certain situations it is possible to inject sub queries, however it is still not possible to modify the database in this way. – DaveRandom Dec 30 '12 at 20:58

---

This answer is written to show just how trivial it is to bypass poorly written PHP user-validation code, how (and using what) these attacks work and how to replace the old MySQL functions with a secure prepared statement - and basically, why StackOverflow users (probably with a lot of rep) are barking at new users asking questions to improve their code.

First off, please feel free to create this test mysql database (I have called mine prep):

```
mysql> create table users(
    -> id int(2) primary key auto_increment,
    -> userid tinytext,
    -> pass tinytext);
Query OK, 0 rows affected (0.05 sec)

mysql> insert into users values(null, 'Fluffeh', 'mypass');
Query OK, 1 row affected (0.04 sec)

mysql> create user 'prepared'@'localhost' identified by 'example';
Query OK, 0 rows affected (0.01 sec)

mysql> grant all privileges on prep.* to 'prepared'@'localhost' with grant option;
Query OK, 0 rows affected (0.00 sec)
```

With that done, we can move to our PHP code.

Lets assume the following script is the verification process for an admin on a website (simplified but working if you copy and use it for testing):

```
<?php

    if(!empty($_POST['user']))
    {
        $user=$_POST['user'];
    }
    else
    {
        $user='bob';
    }
    if(!empty($_POST['pass']))
    {
        $pass=$_POST['pass'];
    }
    else
    {
        $pass='bob';
    }

    $database='prep';
    $link=mysql_connect('localhost', 'prepared', 'example');
    mysql_select_db($database) or die( "Unable to select database");

    $sql="select id, userid, pass from users where userid='$user' and pass='$pass'";
    //echo $sql."<br><br>";
    $result=mysql_query($sql);
    $isAdmin=false;
    while ($row = mysql_fetch_assoc($result)) {
        echo "My id is ".$row['id']." and my username is ".$row['userid']." and lastly, my
password is ".$row['pass']."<br>";
        $isAdmin=true;
        // We have correctly matched the Username and Password
        // Lets give this person full access
    }
    if($isAdmin)
    {
        echo "The check passed. We have a verified admin!<br>";
    }
    else
    {
        echo "You could not be verified. Please try again...<br>";
    }
    mysql_close($link);
```

```
?>

<form name="exploited" method='post'>
    User: <input type='text' name='user'><br>
    Pass: <input type='text' name='pass'><br>
    <input type='submit'>
</form>
```

Seems legit enough at first glance.

The user has to enter a login and password, right?

Brilliant, not enter in the following:

```
user: bob
pass: somePass
```

and submit it.

The output is as follows:

```
You could not be verified. Please try again...
```

Super! Working as expected, now lets try the actual username and password:

```
user: Fluffeh
pass: mypass
```

Amazing! Hi-fives all round, the code correctly verified an admin. It's perfect!

Well, not really. Lets say the user is a clever little person. Lets say the person is me.

Enter in the following:

```
user: bob
pass: n' or 1=1 or 'm=m
```

And the output is:

```
The check passed. We have a verified admin!
```

Congrats, you just allowed me to enter your super-protected admins only section with me entering a false username and a false password. Seriously, if you don't believe me, create the database with the code I provided, and run this PHP code - which at glance REALLY does seem to verify the username and password rather nicely.

So, in answer, THAT IS WHY YOU ARE BEING YELLED AT.

So, lets have a look at what went wrong, and why I just got into your super-admin-only-bat-cave. I took a guess and assumed that you weren't being careful with your inputs and simply passed them to the database directly. I constructed the input in a way tht would CHANGE the query that you were actually running. So, what was it supposed to be, and what did it end up being?

```
select id, userid, pass from users where userid='$user' and pass='$pass'
```

That's the query, but when we replace the variables with the actual inputs that we used, we get the following:

```
select id, userid, pass from users where userid='bob' and pass='n' or 1=1 or 'm=m'
```

See how I constructed my "password" so that it would first close the single quote around the password, then introduce a completely new comparison? Then just for safety, I added another "string" so that the single quote would get closed as expected in the code we originally had.

However, this isn't about folks yelling at you now, this is about showing you how to make your code more secure.

Okay, so what went wrong, and how can we fix it?

This is a classic SQL injection attack. One of the simplest for that matter. On the scale of attack vectors, this is a toddler attacking a tank - and winning.

So, how do we protect your sacred admin section and make it nice and secure? The first thing to do will be to stop using those really old and deprecated `mysql_*` functions. I know, you followed a tutorial you found online and it works, but it's old, it's outdated and in the space of a few minutes, I have just broken past it without so much as breaking a sweat.

Now, you have the better options of using mysqli_ or PDO. I am personally a big fan of PDO, so I will be using PDO in the rest of this answer. There are pro's and con's, but personally I find that the pro's far outweigh the con's. It's portable across multiple database engines - whether you are using MySQL or Oracle or just about bloody anything - just by changing the connection string, it has all the fancy features we want to use and it is nice and clean. I like clean.

Now, lets have a look at that code again, this time written using a PDO object:

```
<?php

    if(!empty($_POST['user']))
    {
        $user=$_POST['user'];
    }
    else
    {
        $user='bob';
    }
    if(!empty($_POST['pass']))
    {
        $pass=$_POST['pass'];
    }
    else
    {
        $pass='bob';
    }
    $isAdmin=false;

    $database='prep';
    $pdo=new PDO ('mysql:host=localhost;dbname=prep', 'prepared', 'example');
    $sql="select id, userid, pass from users where userid=:user and pass=:password";
    $myPDO = $pdo->prepare($sql, array(PDO::ATTR_CURSOR => PDO::CURSOR_FWDONLY));
    if($myPDO->execute(array(':user' => $user, ':password' => $pass)))
    {
        while($row=$myPDO->fetch(PDO::FETCH_ASSOC))
        {
            echo "My id is ".$row['id']." and my username is ".$row['userid']." and
lastly, my password is ".$row['pass']."<br>";
            $isAdmin=true;
            // We have correctly matched the Username and Password
            // Lets give this person full access
        }
    }

    if($isAdmin)
    {
        echo "The check passed. We have a verified admin!<br>";
    }
    else
    {
        echo "You could not be verified. Please try again...<br>";
    }

?>

<form name="exploited" method='post'>
    User: <input type='text' name='user'><br>
    Pass: <input type='text' name='pass'><br>
    <input type='submit'>
</form>
```

The major differences are that there are no more `mysql_*` functions. It's all done via a PDO object, secondly, it is using a prepared statement. Now, what's a prepred statement you ask? It's a way to tell the database ahead of running a query, what the query is that we are going to run. In this case, we tell the database: "Hi, I am going to run a select statement wanting id, userid and pass from the table users where the userid is a variable and the pass is also a variable.".

Then, in the execute statement, we pass the database an array with all the variables that it now expects.

The results are fantastic. Lets try those username and password combinations from before again:

```
user: bob
pass: somePass
```

User wasn't verified. Awesome.

How about:

```
user: Fluffeh
pass: mypass
```

Oh, I just got a little excited, it worked: The check passed. We have a verified admin!

Now, lets try the data that a clever chap would enter to try to get past our little verification system:

```
user: bob
pass: n' or 1=1 or 'm=m
```

This time, we get the following:

```
You could not be verified. Please try again...
```

This is why you are being yelled at when posting questions - it's because people can see that your code can be bypassed wihout even trying. Please, do use this question and answer to improve your code, to make it more secure and to use functions that are current.

Lastly, this isn't to say that this is PERFECT code. There are many more things that you could do to improve it, use hashed passwords for example, ensure that when you store sensetive information in the database, you don't store it in plain text, have multiple levels of verification - but really, if you just change your old injection prone code to this, you will be WELL along the way to writing good code - and the fact that you have gotten this far and are still reading gives me a sense of hope that you will not only implement this type of code when writing your websites and applications, but that you might go out and research those other things I just mentioned - and more. Write the best code you can, not the most basic code that barely functions.

edited Sep 18 '13 at 12:50

answered Sep 18 '13 at 12:28
Fluffeh
**23.2k** 13 36 61

1 Thank you for your answer! Have my +1! It's worth noting that `mysql_*` in on itself isn't insecure, but it does promote insecure code via bad tutorials and the lack of a proper statement prepare API. – Madara Uchiha Sep 18 '13 at 12:31

XKCD comic will tell you the same, but way more vivid and *shorter*. – Your Common Sense Sep 18 '13 at 14:57

1 unhashed passwords, oh the horror! =oP Otherwise +1 for detailed explanation. – cryptic ツ Sep 19 '13 at 5:42

**protected** by Madara Uchiha Oct 14 '12 at 21:32

Thank you for your interest in this question. Because it has attracted low-quality answers, posting an answer now requires 10 reputation on this site.

Would you like to answer one of these unanswered questions instead?