The Hitchhiker's Guide to SQL Injection prevention

Disclaimer: My English is far from perfect and this text has been written when it was even worse. If you can't stand such poor grammar and can afford to do a bit of proofreading, here is the source on Github (https://github.com/colshrapnel/phpdelusions.net/blob/master/sql_injection.md), all pull requests to which will be accepted with gratitude.

In this article I will try to explain the nature of SQL injection; show how to make your queries 100% safe; and dispel numerous delusions, superstitions and bad practices related to the topic of SQL Injection prevention.

Don't panic.

Honestly, there is not a single reason to panic or to be even worried. All you need is to get rid of some old superstitions and learn a few simple rules to make all your queries safe and sound. Strictly speaking, you don't even need to protect from SQL injections at all! That is, no dedicated measures intended exclusively to protect from SQL injection, have to be taken. All you need is to **format your query** properly. It’s as simple as that. Don't believe me? Please read on.

What is an SQL injection?

SQL Injection is an exploit of an improperly formatted SQL query.

The root of SQL injection is the mixing of code and data.
In fact, an SQL query is *a program.* A fully legitimate program - just like our familiar PHP scripts. And so it happens that we are *creating this program dynamically,* adding data to this program on the fly. Naturally, this data may interfere with the program code and even alter it - and such an alteration would be the very SQL injection itself.

But such a thing can only happen if we don't format query parts properly. Let's take a look at a canonical example (http://xkcd.com/327/),

```
 $name  = "Bobby';DROP TABLE users; -- ";
$query = "SELECT * FROM users WHERE name='$name'";
```

which compiles into the malicious sequence

```
SELECT * FROM users WHERE name='Bobby';DROP TABLE users; -- '
```

Call it an injection? Wrong. It's **an improperly formatted string literal.**
Which, once properly formatted, won't harm anyone:

```
SELECT * FROM users WHERE name='Bobby\';DROP TABLE users; -- '
```

Let's take another canonical example,

```
 $id    = "1; DROP TABLE users;"
$id    = mysqli_real_escape_string($link, $id);
$query = "SELECT * FROM users where id = $id";
```

with no less harmful result:

```
SELECT * FROM users WHERE id =1;DROP TABLE users; -- '
```

Call it an injection again? Wrong yet again. It's **an improperly formatted numeric literal.** Be it properly formatted, an honest

```
SELECT * FROM users where id = 1
```

statement would be positively harmless.

But the point is, **we need to format out queries anyway** - no matter if there is any danger or not. Say there was no Bobby Tables around, but an honest girl named `Sarah O'Hara` - who would never get into a class if we don’t format our query, simply because the statement

```
INSERT INTO users SET name='Sarah O'Hara'
```

will cause an ordinary syntax error.

> So we have to format just for sake of it. Not for Bobby but for Sarah. That is the point.

While SQL injection is just a *consequence* of an improperly formatted query.

> Moreover, **all the danger is coming from the very statement in question:** zounds of PHP users still do believe that the notorious `mysqli_real_escape_string()` function's only purpose is "to protect SQL from injections" (by means of escaping some fictional "dangerous characters"). If only they knew the real purpose of this honest function, there would be no injections in the world! If only they were *formatting* their queries properly, instead of "protecting" them - they'd have real protection as a result.

So, to make our queries invulnerable, we need to format them properly and make such formatting **obligatory**. Not as just an occasional treatment at random, as it often happens, but as a strict, inviolable rule. And your queries will be perfectly safe just as a *side effect*.

What are the formatting rules?

The truth is, formatting rules are not that easy and cannot be expressed in a single imperative. This is the most interesting part, because, in the mind of the average PHP user, SQL query is something homogeneous, something as solid as a PHP string literal that represents it. They treat SQL as a solid medium (http://kunststube.net/encoding/), that require whatever all-embracing "SQL escaping" only. While in fact SQL query a *program*, just like a PHP script. A program with its distinct syntax for its distinct parts, and **each part require distinct formatting, inapplicable for the others!**

For Mysql it would be:

1. Strings
    - have to be added via a native prepared statement
      or
    - have to be enclosed in quotes
    - special characters (frankly - the very delimiting quotes) have to be escaped
    - a proper client encoding has to be set
      or
    - could be hex-encoded (https://dev.mysql.com/doc/refman/5.0/en/hexadecimal-literals.html)
2. Numbers
    - have to be added via a native prepared statement
      or
    - should be filtered out to make sure only numerical characters, a decimal delimiter, and a sign

3. Identifiers
    - have to be enclosed in backticks
    - special characters (frankly - the very delimiting backticks) have to be escaped
4. Operators and keywords.
    - there are no special formatting rules for the keywords and operators beside the fact that they have to be legitimate SQL operators and keywords. So, they have to be *whitelisted*.

As you can see, there are *four* whole *sets* of rules, not just one single statement. So, one cannot stick to a magic chant like "escape your inputs" or "use prepared statements".

"All right" - you would say - "I am following all these rules already. What's all the fuss is about"? All the fuss is about the *manual* formatting. In fact, you should never apply these rules manually, in the application code, but must have a mechanism that will do this formatting for you instead. Why?

Why manual formatting is bad?

Because it is manual. And manual means error prone. It depends on the programmer's skill, temper, mood, number of beers last night and so on. As a matter of fact, manual formatting is the very and the only reason for the most injection cases in the world. Why?

1. **Manual formatting can be incomplete.** Let's take the Bobby Tables' case. It's a perfect example of incomplete formatting: a string we have added to the query was only quoted, but not escaped! While, as we just learned from the above, quoting and escaping should always go together (along with setting the proper encoding for the escaping function). But in a usual PHP application which does SQL string formatting separately (partly in the query and partly somewhere else), it is very likely that some part of formatting may be simply overlooked.

2. **Manual formatting can be applied to the wrong literal.** Not a big deal as long as we are using *complete formatting* (as it will cause an immediate error which can be fixed in the development phase), but when combined with incomplete formatting it's a real disaster. There are hundreds of answers on the great site of Stack Overflow, suggesting **to escape identifiers** the same way as strings. Which would be completely useless and would cause an SQL injection.

3. **Manual formatting is essentially a non-obligatory measure**. First of all, there is an obvious lack of attention case, where proper formatting can be simply forgotten. But there is a really weird case - many PHP users often *intentionally* refuse to apply any formatting, because up to this day they still separate the data to "clean" and "unclean", "user input" and "non-user input", etc. Thinking that a "safe" data don't need any formatting. Which is a plain nonsense - remember Sarah O'Hara. From the formatting point of view, it's the *destination* that matters. A developer has to mind the *type of SQL literal*, not the data source. Is it a string going to the query? It has to be formatted as a string then. No matter if it's from user input or just mysteriously appeared out of nowhere amidst the code execution.

4. **Manual formatting can be separated from the actual query execution by a considerable distance.**

The most underestimated and overlooked issue. Yet most essential of them all, as it can spoil all the other rules alone, if not followed.

Almost every PHP user is tempted to do all the "sanitization" in a single place, far away from the actual query execution, and such a wrong approach is a source of innumerable faults alone:

- first of all, having no query at hand, one cannot tell what kind of SQL literal this a certain piece represents - and thus we have both formatting rules (1) and (2) violated at once.
- having more than one place for sanitizing (it could be either a centralized facility or in-place formatting), we are calling for a disaster, as one developer would think it was done by another or was already made somewhere else, etc.
- having more than one place for sanitizing, we're introducing another danger, of double-sanitizing data (say, one developer formatted it at the entry point and another - before query execution), which is not dangerous but make one's site look extremely unprofessional
- premature formatting will spoil the source variable, making it unusable for anything else.

5. After all, manual formatting will always take extra space in the code, making it entangled and bloated. All right, now you trust me that manual formatting is bad. What do we have to use instead?

Prepared statements.

Here I need to stop for a while to emphasize the important difference between the implementation of **native** prepared statements supported by the major DBMS and **the general idea of using a placeholder** to represent the actual data in the query. And to emphasize **real benefits** of a prepared statement.

The idea of a native prepared statement is smart and simple: the query and the data are sent to the server separated from each other, and thus there is no chance for them to interfere. Which makes SQL injection outright impossible. But at the same time, native implementation has its limitations, as it supports only two kinds of literals (strings and numbers, namely) which **renders them insufficient and insecure for the real-life usage.**

There are also some wrong statements about native prepared statements:

- they are "faster". **Not in PHP**, which simply won't let you reuse a prepared statement between separate calls. While repeated queries within the same instance occurred too seldom to talk about.
- they are "safer". This is partially true, but not because they are *native*, but because they are *prepared statements* - as opposed to manual formatting.

And here we came to the main point of the whole article: the general idea of creating an SQL query out of constant part and placeholders, which will be substituted with actual data, which will be **automatically formatted** is indeed a Holy Grail we were looking for.

The main and most essential benefit of prepared statements is the elimination of all the dangers of manual formatting:

- a prepared statement does the complete formatting - all without programmer's intervention! Just fire and forget.
- a prepared statement does the adequate formatting (as long as we're binding our data using the proper type)
- a prepared statement makes the formatting inviolable!
- a prepared statement does the formatting in the only proper place - right before query execution.

This is why manual formatting is so much despised nowadays and prepared statements are so honored.

There are two additional but non-essential benefits of using prepared statements:

- a prepared statement doesn't spoil the source data which can be used safely somewhere else: shown back in the browser, stored in a cookie, etc.
- a programmer who is lazy enough can make their code dramatically shorter by means of using prepared statements (however, the opposite is true too - a diligent user can write a code for the simple insert of the size of an average novel).

So, "nativeness" of a prepared statement is not that essential, as it proven by PDO, which can just *emulate* a prepared statement, sending the regular query to the server at once, substituting placeholders with the actual data, if `PDO::ATTR_EMULATE_PREPARES` configuration variable (http://php.net/manual/en/pdo.setattribute.php) is set to `TRUE` . But the data gets **properly formatted** in this case - and therefore this approach is equally safe!

Moreover, **even with the old MySQL extension** we can use prepared statements all right! Here is a small function that can offer rock-solid security with this old good extension:

```
 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);
```

Look - everything is parameterized and safe, at least to the degree PDO can offer.

So the all-embracing rule for the protection:

EVERY DYNAMICAL ELEMENT SHOULD GO INTO QUERY VIA **PLACEHOLDER**

> Here I need to stop again to make a very important statement: we need to distinguish a **constant** query element from a **dynamical** one. Obviously, our primary concern has to be dynamical parts, just because of their very dynamic nature. While constant value cannot be spoiled by design, and whatever formatting issue can be fixed at development phase, dynamical query element is a distinct matter. Due to its variable nature, **we never can tell if it contains a valid value, or not.** This is why it is so important to use placeholders for all the dynamical query parts.

"All right" - says you - "I am using prepared statements all the way already. And so what?" Be honest - you aren't.

One Step Beyond.

In fact, our queries sometimes are not as primitive as primary key lookups. Sometimes we have to make them even more dynamical, adding identifiers or whatever complex structures, like arrays. What regular drivers offer to solve this problem?

Nothing.

For the identifier, it would be nothing else... **but old good manual formatting:**

```
 $field = "`".str_replace("`","``",$field)."`";
$sql   = "SELECT * FROM t ORDER BY $field";
$data  = $db->query($sql)->fetchAll();
```

For the arrays, it will be a whole *program* written to create a query on the fly:

```
 $ids = array(1,2,3);
$in  = str_repeat('?,', count($arr) - 1) . '?';
$sql = "SELECT * FROM table WHERE column IN ($in) AND category=?";
$stm = $db->prepare($sql);
$ids[] = $category; //adding another member to array
$stm->execute($ids);
$data = $stm->fetchAll();
```

And still, we have a variable interpolated in the query string, which makes me shivers in the back, although I am sure this code is safe. And it should make you shiver too!

So, in all these cases **we are forced to fall back into the stone age of manual formatting!**

But as long as we have learned that manual formatting is bad and prepared statements are good, there is only one possible solution: **we have to implement placeholders for these types as well!**

Imagine we have a placeholder for the above cases. These two snippets become as simple and *safe* as any other code with prepared statements:

```
 $sql   = "SELECT * FROM t ORDER BY ?";
$data  = $db->query($sql, $field)->fetchAll();

$stm = $db->prepare("SELECT * FROM table WHERE column IN (?) AND category=?");
$stm->execute([$ids, $category]);
$data = $stm->fetchAll();
```

So, here can we make be but one conclusion: regular drivers **have** to be extended to support a wider range of data types to be bound. Having devised an idea of new types, we can think of what these types can be:

- identifier placeholder (single identifier)
- identifier list (comma separated identifiers)
- integer list (comma separated integers)
- strings list (comma separated strings)
- the special SET type consists of a comma-separated `idientifier=string value` pairs
- you name it

Quite a lot, in addition to existing types.

One little trick.

So far so good. But here we face another problem. Even with regular types sometimes we need to set data type explicitly, to let driver understand how to format this particular value. A classical example for PDO:

```
 $stm = $db->prepare('SELECT * FROM table LIMIT ?, ?');
$stm->execute([1,2]);
```

won't work in emulation mode, as PDO will format data as strings, whose aren't allowed in this query part. It isn't much a trouble though, as most of time default string formatting is all right:

```
 $stm = $db->prepare('SELECT * FROM t WHERE id=? AND email=?');
$stm->execute([$id,$email]);
```

But with our new complex types, there is no way to use this approach anymore. Because, for example, an identifier can never be bound as a string. So, we have to always set placeholder type explicitly. And it makes a problem: although both PDO and Mysqli offer their own solutions, I wouldn't call any of them viable.

- with PDO's tons of rows with `bindValue` our code is no better than old `mysql_real_escape_string` approach in terms of code size and a number of repetitions.
- mysqli's bind_param() is a nightmare when you have to bind a variable number of values.
- the worst part: all this manual binding makes *application* code bloated, as we cannot encapsulate these bindings into some internals.

So, we need a better solution again. And here it is:

TO MARK PLACEHOLDER WITH ITS TYPE!

Not quite a fresh idea, I have to admit. Well-known `printf()` function has been using this very principle since Unix Epoch: `%s` will be formatted as a string, `%d` as a digit and so on. So, we only have to borrow that brilliant idea for our needs.

To solve all the problems, we need to extend a regular driver with simple *parser* which will parse a query with type-hinted placeholders, extract type information and use it to format a value.

I am not the first to use this approach with DBAL either. There is Dibi (http://dibiphp.com/), DBSimple (http://en.dklab.ru/lib/DbSimple/) or NikiC's PDO wrapper (https://github.com/nikic/DB) and some other examples. But it seems to me that their authors underestimated the significance of type-tinted placeholder, taking it as some sort of syntax sugar, not as essential and cornerstone feature, which it has to be.

So, I endeavored my own implementation, to emphasize the idea, and called it SafeMysql (https://github.com/colshrapnel/safemysql), because type-hinted placeholders make it indeed safer than the regular approach. It's also aimed for better usability, to make code cleaner and shorter.

Although there are some improvements still pending for implementation (such as native placeholders support; named placeholders (like in PDO); un-typed default placeholder treated as a string; a couple another data types to support), it's already a complete solution that can be used in any production code.

Exception that proves the rule.

Unfortunately, a prepared statement is not a silver bullet too, and cannot offer 100% protection alone. There are two cases where it's either not enough or not applicable at all:

1. Remember the last kind of query literals from the section #3 list of formatting rules: SQL keywords. There is no way to format them.
2. A somewhat tricky case, when we are creating our *list of identifiers* dynamically, based on user input, and there could be fields a user isn't allowed to. The very common case is creating an insert query dynamically based on the keys and values of the `$_POST` array. Although we can format both properly, there is still a possibility for some field like 'admin' or 'permissions' that can be set by site admin only. And these fields should be never allowed from user input.

And to solve these both cases, we have to implement another approach which is called *whitelisting*. In this case, every dynamic parameter has to be pre-written in your script already, and all the possible values have to be chosen from that set.
For example, to do dynamic ordering:

```
 $orders  = array("name","price","qty"); //field names
$key     = array_search($_GET['sort'],$orders)); // see if we have such a name
$orderby = $orders[$key]; //if not, first one will be set automatically. smart enough :)
$query   = "SELECT * FROM `table` ORDER BY $orderby"; //value is safe
```

For keywords the rules are same, but of course there is no formatting available - thus, only whitelisting is possible and ought to be used:

```
 $dir = $_GET['dir'] == 'DESC' ? 'DESC' : 'ASC';
$sql = "SELECT * FROM t ORDER BY field $dir"; //value is safe
```

Note that aforementioned SafeMysql lib supports two functions for whitelisting, one for key=>value arrays and one for single keywords, but now I am thinking of making these functions less advisory but more strict in usage.

Dynamically built queries.

As this text pretends to be a complete guide for injection protection, I cannot avoid complex queries creation. A usual case of multiple-criteria search, for example. Or any other case where we have to have *arbitrary query parts* added or taken out from the query. There is no way to use placeholders in this case - so, we need another mechanism.

First, a **query builder** is a king here. The exact case query builders were invented for

```
 $query = $users = DB::table('users')->select('*');
if ($fname = input::get('first_name'))
{
    $query->where('first_name = ?', $fname);
}
if ($lname = input::get('last_name'))
{
    $query->where('last_name = ?', $lname);
}
// and so on
$results = $query->get();
```

But sometimes we have a query that is so complex that makes it to painful to write it using query builders. But either way we have to remember the only rule - all dynamical query parts are going into query via placeholders. And for a raw query we can use a very smart trick (http://stackoverflow.com/a/11231629/285587):

```
 SELECT * FROM people
WHERE (first_name = :first_name or :first_name is null)
AND (last_name = :last_name or :last_name is null)
AND (age = :age or :age is null)
AND (sex = :sex or :sex is null)
```

Written this way, we only have to bind our variables, either contains a value or a NULL, to these placeholders. For the NULL values, the smart engine would just throw away their conditions, leaving only ones with values!

Either way, one has to bear in mind that the resulting query should be always built from only two sources - either constant part or a placeholder.

Conclusion.

In short, we can formulate two simple rules:

Even dynamically created, an SQL query has to consist of 2 possible kinds of data only:

- constant parts hardcoded in the script
- placeholders for every dynamical value

When followed, these rules will guarantee 100% protection.

Appendix 1. Vocabulary.

Almost everyone, who have fancy to talk on the topic, uses the wide range of words, hardly bothering to comprehend their meaning, or - worse of that - having their own idea on the meaning at all.

Surely, the most notorious term is "escaping". Everyone is taking it as "making data clean", "making data safe", "escaping dangerous characters". While all these terms are essentially vague and ambiguous. At the same time, in the mind of average PHP user, "escaping" is strongly connected to the old `mysql_real_escape_thing` , so they have all these matters intermixed, resulting in the oldest and most callous of PHP superstitions - " `*_escape_string` prevents from injection attack" while we already learned it is far from it.

Instead, the only proper term should be "formatting". While "escaping" should designate only particular part of string formatting.

The same goes for every other word from the list of "sanitizing", "filtering", "prepared statements" and the like. Think of the meaning (or, rather, if it has any certain meaning at all) before using a word.

Even "SQL injection" term itself is extremely ambiguous. It seems most users take injection literally as adding a second query as shown in the Bobby Tables example. And thus trying to protect by means of forbidding multiple query execution (which would be useless rubbish, of course). Somewhat similar delusion is shared by another lot of people, whose belief is that only DML queries can be considered as injection prone (and thus they don't care for injections in SELECT queries at all).

So, reading recommendations from various sources, always keep in mind that you have to understand the terminology and - more than that - understand what the author meant (and if there is *any* particular meaning at all).

Appendix 2. How to protect from of injection of [xxx] type.

You may be heard of different kinds of injections - "blind", "time-delay","second order" and thousands of others. One has to understand that all these aren't different ways to *perform* an injection, but just different ways to *exploit* it. While there is only one way to perform an injection - to break query integrity. So, if you can keep query integrity, you'll be safe against all thousands of different "kinds" of injections all at once. And to keep the query integrity you have just to format query literals properly.

Appendix 3. False measures and bad practices.

1. **Escaping user Input.**
   (https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Defense_Option_3:_Escaping_All_User_Supplied_Input)
   This is the King. A grave delusion, still shared by many PHP users (and even OWASP, as you can see). Consists of two parts: "escaping" and "user input":
   - escaping: as we noted above, it does only *part* of the job, for only *one* type of SQL literal. And when used alone or not in the proper place is a sure call for disaster.
   - user input: there should be no such words in the context of injection protection. Every variable is potentially dangerous - no matter of the source! Or, in other words - every variable have to be properly formatted to be put into query - no matter the source again. It's the destination that matters. The moment a developer starts to separate the sheep from the goat, he does his first step to disaster.
   - moreover, even the wording suggests bulk escaping at the entry point, resembling the very `magic quotes` feature - already despised, deprecated and removed from the language.
2. **magic quotes -** the very material incarnation of the above principle. Thank goodness, it's already removed from the language.
3. **data validation**. One have to understand, that input (in the meaning of user input) data validation has absolutely nothing to do with SQL. Really. No validation rule can help against SQL injection if a free-form text is allowed. Yet we have to format our SQL despite any validations anyway - remember Sarah O'Hara who bears a name which is perfectly valid from the user input point of view. Also, remember that **validation rules may change**.
4. **htmlspecialchars** (and also `filter_var()` , `strip_tags()` and the like).
   Folks. It's **HTML** special characters encoding if you didn't notice yet. It has absolutely nothing to do with **SQL.** It helps nothing in the matter, and should never be used in the context of SQL injection protection. It's absolutely inapplicable for SQL, and cannot protect your query even if used as a string escaping function. Leave it for other parts of your application.
   Also, please understand that SQL formatting should never ever alter the data. Then you put your jewelry in a safe, you want it back **intact**, not some parts amended or replaced "for your own safety!". Same here. A database is intended to store your data, not to "protect" it. And it is essential to store the exact data you want back (means your silly `base64` attempt is wrong as well, by the way).
5. **Universal** "clean'em all" **sanitization function.**
   Such a function just should have never existed. First, there are too much different contexts/mediums our data could be used in (SQL query, HTML code, JS code, JSON string, etc. etc. etc.) each require different formatting. So you just cannot tell beforehand what kind of formatting/sanitization will be required. Moreover, there are different literals even in a single medium, which, in turn, require it's own formatting too. Which just makes it impossible to create a single all-embracing function to format 'em all at once, without the risk of spoiling the source data yet leaving huge security holes at the same time.
6. **Filtering out malicious characters and sentences.**
   This one is simple - it's an absolutely imaginary measure. It fails the reality check a big one. Nobody ever used it on a more-or-less viable site. Just because it will spoil the user experience.
7. **Stored procedures.**
   NOT in the web-development. This suggestion is from another realm, where one would have a fancy of limiting database users to a set of stored procedures. It is more like a general security measure, pointed against local users, rather than against an intruder. Anyway, the approach is just inviable in the web development. It can add nothing to what prepared statements already offer, yet in a way inconvenient and toilsome way.

8. **Separate DB accounts** for running SELECT and DML queries. Again not a protection measure, but a just a [worthless] attempt to soften the consequences of the already performed attack. Worthless because of SELECT-based injection as a disaster alone. And of course useless because we are protected already by formatting our queries properly.

Appendix 4. ORMs and Query Builders.

Although this article is on the raw SQL queries only, in the modern applications you may find not a trace of them. Because all SQL is hidden behind the scenes by an ORM or a Query Builder. If your application is of such kind - you're lucky. However, it doesn't make this article less significant due to the following reasons:

- First of all, even ORMs and Query Builders have to be written by someone. And prepared statements will make such writing simpler and safer.
- Speaking of ORM, as long as you can stick to its methods, you're all right. Unfortunately, in the real life you can't use it all the way - sometimes you have to run raw queries as well. So, always have a good library that supports type-hinted placeholders along with your ORM.
- Query Builders are using placeholders already - so, they would only benefit from adding some new placeholder types.

TL;DR

1. There is no such thing like injection but improperly formatted query only
2. To make formatting inviolable we have to use prepared statements
3. Prepared statements have to support much more data types than simple strings and numbers
4. To make formatting usable we need to use type-hinted placeholders
5. In case when formatting is inapplicable, white-listing have to be used.

GOT A QUESTION?

I am the only person to hold a gold badge in ● pdo (http://stackoverflow.com/help/badges/4220/pdo), ● mysqli (http://stackoverflow.com/help/badges/6342/mysqli) and ● sql-injection (http://stackoverflow.com/help/badges/5981/sql-injection) on Stack Overflow and I am eager to show the right way for PHP developers.
Besides, your questions let me make my articles even better, so you are more than welcome to ask any question you got.

Click here to ask!

LATEST ARTICLE:
PHP Error Reporting (/articles/error_reporting)

LATEST COMMENTS:

18.04.20 00:46
Lonestar Jack for How to connect to MySQL using PDO: (/pdo_examples/connect_to_mysql#comment-846)
What is he best coding to use two queries on one page? One connection and then close it and do...
read more (/pdo_examples/connect_to_mysql#comment-846)

17.04.20 19:02
Gregory Stathes for How to use mysqli properly: (/mysqli#comment-845)
My issue is this, been trying to solve for days. And I am more than happy to pay for a solution,...
read more (/mysqli#comment-845)

16.04.20 20:47
Phil for How to use mysqli properly: (/mysqli#comment-844)
Which is better for performance mysqli or PDO?
read more (/mysqli#comment-844)

12.04.20 04:28
eljaydee for Mysqli examples: (/mysqli_examples#comment-843)

I cannot make this work: I have been told a million times to "review you query" but cannot find...
read more (/mysqli_examples#comment-843)

12.04.20 04:27
Elizabeth for PDO Examples: (/pdo_examples#comment-842)
Hi, my php programs which worked fine on my local server are now no longer passing through...
read more (/pdo_examples#comment-842)

Your Common Sense
@ShrapnelCol

Replying to @ShrapnelCol

Sometimes abstract class and an interface can be almost indistinguishable, and sometimes they can have nothing in common, like in case of such interfaces as Iterable and such, where interface defines kind of trait. Oh, wait...

21h

Your Common Sense
@ShrapnelCol

Replying to @ShrapnelCol

And the answer is: abstract class is a thing itself, "strong is-a". It defines what a class is. While an interface is "can do", some specific behavior supported by class.

Add a comment

Please refrain from sending spam or advertising of any sort.
Messages with hyperlinks will be pending for moderator's review.

Markdown is now supported:

- `>` before and an empty line after for a quote
- four spaces to mark a block of code

Your name

Are you a robot?

Message

Address

If you want to get a reply from admin, you may enter your E—mail address above

Send

Comments:

John, 08.10.18 10:06

Hello, when using function 'run' from your PDO wrapper like this:

$this->db->run('INSERT INTO `posts` (`title`, `body`, `author`) VALUES (?, ?, ?)', [$this->title, $this->body, $this->author]);

How it should be protected? Thanks

REPLY:

Hello John!

Speaking of SQL injection, it as already protected. This code is 100% safe because you put ? marks instead of the actual variables in the query, and put these variables in the second argument. So the proper binding process is performed internally. That's the very point of this function - to do all the necessary stuff under the hood.

Hope it's clear now! Feel free to ask any other questions you have!

Daniel, 21.05.18 17:25

I've made my own PDO wrapper, inspired by your SafeMySql wrapper, which "prepares" queries. Thank you for taking the time to teaching people the proper way of doing things.

I would have used your library, but:

1. I wanted static methods
2. I wanted to use "LIKE %?s%"
3. I did not understand how to use it properly...

Daniel, 20.05.18 13:59

Thank you for this great article !

P.S. Please fix the grammar issues... Or if you want, i'll do it for you.

Himanshu Kubavat, 07.03.18 12:29

This is simply awesome.. thanks for elaborating it.

hitanshi mehta, 25.02.18 21:29

thank you so much!!!!!!!! it was very helpful.

Jordan, 08.01.18 23:02

Hello there, I'm curious as to whether or not something like MSSQL would be superior over MySQL, as it has a Transaction Log. It would appear MySQL doesn't (some binary log from what I hear, which... I don't have binary memorized, neither do I have the time to do so...) (Perhaps that's a noob statement... I have no clue.)

Say someone DID successfully somehow drop a table, would the transaction not be able to recover the database to a certain point? (Obviously code Sanitization as a prevention is a better option... but say somehow they get past it anyways?)

That being said, does sanitization of code completely prevent injections?

Thanks!

JKB

REPLY:

Hello Jordan!

Thank you for the good question. There are some fallacy in your reasoning and I will try to make it clear one by one:

1. Mysql's binary log is pretty much the same as a Transaction Log, so both can used to handle transactions
2. That said, transactions are useless in recovering dropped tables as every transaction explicitly gets committed when a drop table command is encountered. in other words, dropped tables are never recorded in transactions.
3. Apart from that, SQL injection is not limited to dropping tables. most of time it's the information leak, so you can tell that even if transactions of any help for the dropped tables, they won't prevent every possible exploit.
4. I don't really get is the sanitization of code you are talking about. To prevent sql injection you should use prepared statements and it will make your code 100% safe from sql injections

Hope it is clear now bu feel free to ask if not!

Kirinnee97, 30.11.17 00:53

Thanks for the information! Good article!

Don, 21.10.17 00:33

Excellent post. The information is right on. I also agree with everything said by dreftymac. If you send me an email I'll be happy to edit the text for you to clean up some of the English.

Anon, 28.09.17 04:46

Thanks for this article, it's been a great help!

eV, 30.06.17 09:49

Awesome article, thank you. I was about to make the same offer as dreftymac did on June 11. If you need another volunteer, I'll be happy to help.

REPLY:

Hello!

Thank you for reaching out. Unfortunately, dreftymac didn't answer to my mail, I suppose it's some communication problem.

Yet I am still thinking that the main idea of this article (that SQL injection is just a consequence of improper syntax, and all you need is to make sure that syntax is always correct) deserves more attention. So if you can fix some mistakes here I'll be very grateful!

Ihyongo suurshater, 24.06.17 03:27

I created a class that holds all database query functions. It works fine fetching results using "fetchALL" using foreach loop statement but fails when i try using while. please i need a helping hand here. Thanks!

REPLY:

Hello Ihyongo!

to be able to use while, you should use fetch() command instead of fetchAll().

So, instead of something like this

```
 $data = $stmt->fetchAll();
foreach ($data as $row)
```

it should be

```
 while ($row = $stmt->fetch())
```

dreftymac, 11.06.17 20:29

This is excellent material, but the non-native English is very distracting and it makes people think you are not as smart. Would you be willing to have someone re-write your content in native English?

I would be willing to do so at no cost, because I believe your content deserves better presentation.

REPLY:

Hello!

Thank you for your generous suggestion! Sent you an email.

Someone, 12.05.17 07:12

I wrote the first ever SQL injection cheat-sheet almost 15 years ago, and it seems PHP/SQL is still a mess today.

Dan Costinel, 22.12.16 16:21

Recently I'm having a discussion with a buddy, about this subject. He says, even though one pass $_GET variables to prepare() method, you can still be vulnerable to SQL Injection, even though you are correctly limiting the charset to utf8.

My thought is that he uses the "third order SQL Injection" here, with "1st order" being to pass directly the $_GET variables without any proper action, "2nd order" being not setting the charset correctly, and "3rd order" being that the one who codes (the developer), intentionally inject malicious code in order to achieve SQL Injection.

I really want to ask if would be to remove those backticks from before and after $table variable, in $sql = "SELECT * FROM `" . $table . "` LIMIT :offset, :perpage"; (from vulnerable.php script), would be the right ch4oice. I mean, if those backticks are present, then yes, there is SQL Injection, but without those backticks, you still can call that SQL Injection? I know, official docs says to wrap names in backticks, but I usually don't follow that recommendation.

REPLY:

Hello Dan!

Thank you for sharing an interesting case.

In fact you both are right.
As it's covered in this article, prepared statements work for the *data* literals only and utterly useless for anything else.

So, you are right, because if you prepare your query for real, *substituting every data literal with a placeholder*, while actual variables are sent into execute(), **then you can use $_GET variables directly without any fear.**

But if you are adding your variable directly to the query, just calling it via prepare() won't do any good by itself: prepare works only through placeholders, but never helps for the anything added to the query directly. And for the table name there are no placeholder in PDO.

What you see in the example is not a 3rd order injection but just plain and simple 1st order, a regular SQL injection.

Regarding backticks - as you can read in this very article, by no means you should remove them, if your table name is coming from a variable. But backticks alone are not enough: you have to escape them.

So, prepare your table name like this

```
$table = str_replace("`","``",$table);
```

strictly keep it in backticks the guy won't be able to inject anything anymore.

Feel free to ask any other questions, just keep in mind that messages that contain hyperlinks are not shown immediately. but I get informed of them anyway and will write an answer as soon as possible.

Andrew, 18.12.16 18:29

Can't repy to your comment so I'll write here then :)

Yes, my wrapper converts PHP null into mysql 'NULL' both ways, for input and for data output.

Regarding limit, what if I just check incoming value with is_numeric or even is_int and if it's not, then throw Exception? And only after that, construct query without quotes?

Thanks

REPLY:

You're welcome to comment here.

If you're working on the query builder that deliberately knows what a statement in question is LIMIT, then your idea surely works. But for a general purpose query where you just unable to tell which placeholder goes in where, I would rather suggest to introduce another placeholder type, just like it's done in my own wrapper. Remember - there are table names or arrays that should be safely processed as well. So you need more than one placeholder type. One more for integers looks a good idea.

Andrew, 17.12.16 18:23

If I use DB wrapper, which:

1. Sets UTF8
2. mysqli_real_escape every incoming value
3. escaped value is surrounded with single quotes and then added into query, by the wrapper.

The result is:

```
SELECT * FROM users WHERE id ='1;DROP TABLE users;'
```

Then I am 100% secure?

Thanks

REPLY:

Hello Andrew!

Thank you for the good question. Yes, if these conditions are met, then you are safe, as this is exactly how the escaping works.

You just have to keep in mind that sometimes you cannot use such scenario. Parameters for LIMIT clause is an example (because `LIMIT '10'` will cause a syntax error). But in most cases your wrapper will go all right.

You may also want to add a case for the `null` value, which is better to be translated into SQL `null`, as both mysqli and PDO prepared statements do and as it is done in my own wrapper (https://github.com/colshrapnel/safemysql/blob/master/safemysql.class.php#L556).

Feel free to ask any other questions!

Darren Edwards, 09.12.16 20:46

Excellent work, I have been coding PHP for many years now, and rarely have I seen such lengthy and justified arguments in favour of security. I have learnt the hard way to check every user input,variable and url entry point, and especially to `format` variables before they get anywhere near the queries that have been written.

Goran, 29.11.16 04:05

hi..im a newbie or noob in php...whatever you prefer I think I scratched the surface of PHP OOP and understood some general things of the matter. I'm little stuck with my DB class because it appears singleton is no god anymore.......tried to understand reading articles... reading over and over still $understand=FALSE. My question is, is this some general thing everybody should start to follow or its debate? If you can do the caveman answer understandable and to the point. Thanks Xigo

Deluded, 04.08.16 11:32

Thank You for fast anwer. I think your function paraQuery() is not sanitized (at least not compared with PDO/MySQLi). With your function You can play safe only withing ASCII charspace. Introducing multibyte strings would require proper escaping with implicit charsets.

REPLY:

Thank you for highlighting the particular issue.

mysql_real_escape_string() is designed especially to do the proper escaping with implicit charsets, and therefore there shouldn't be an issue as long as client encoding is properly set. Note that PDO by default does exactly the same - so the bottom line says.

However, I have to admit that such a function, although used only for demonstration, is rather misplaced. Or at least I should add a note stating that by no means I encourage its use.

Deluded, 04.08.16 10:59

I red only few paragraphs about prepared statements, and I found this content outdated, and filled with common misconceptions.

This days one may use PDO or mysqli and play way safer by following official documentation.

Beside this, you have bugs on code on very this page which allow SQL injection attack same as You have in you code examples here.

REPLY:

Nice try, but you need to pay more attention to what you read (and write).

This article says noting against mysqli or PDO, encouraging their usage. Besides, it goes a step further, revealing some edge cases where no way offered by the official documentation could be of any help.

There are some intentionally bugged codes, intended for demonstration purposes, as this is an educational article.

But yet again, thank you for the try. You are welcome to read a bit more and leave any relevant feedback :)

Cory, 27.07.16 22:50

I've been coding php for 13+ years, but all self taught and relatively new to the concept of prepared statements. Question, should you still used prepared statements for static queries to your database (queries that have no user input)?

REPLY:

Hi Cory! Thank you for your question.

Like it is said in the query() section of the article, if your query is completely static, and doesn't contain any variable in it, there is no need to use a prepared statement. In this case a query() method have to be preferred, allowing you to use the method chaining.

Hope it is clear now. But if you give a bit more context, what led you to this question, I'll be able to give more detail as well.

hardy1993, 26.07.16 11:21

Good read, and funny too!

andrew, 29.02.16 00:01

hey this is andrew from stackoverflow. sorry I don't have the time to go through all the articles. I think in chapter four, `Manual == error prone.` would better to be `Manual is identical to error prone.` , just imho

REPLY:

Thanks, mate! Fixed it.

Ash, 24.01.16 02:01

I enjoyed the article. Very informative. It might help, if it's your intention, to point the reader to SafeMysql (aside from a simple link shown once). In my particular case, I'm simply wondering how to implement the class.

Col. Shrapnel, 14.10.15 21:29

??lvaro G. Vicario thank you for the interesting question.

First of all, there is a potential problem with integers. There is a chance that some numerical string for some reason can be casted to the numeric type and then bound using such a deduction as int, and then, when compared to a string field to database, will return unwanted results. SELECT * FROM users WHERE password=0 query will match any user whose password starts from a letter. That's why I prefer to either bind a variable as a string by default, or set the type explicitly. I had a disastrous experience with such a deduction once, and so once bitten I am twice shy.

Identifiers just adds to the problem, as well as arrays. Although there could be implemented some magic, magic in general makes it hard for debugging. With explicit types, one can be sure about the data they are dealing with.

Either way, I come to the same conclusion myself, but from the other end - from the usability point of view: yes, it's just an overkill to set the type explicitly every time. So, I am going to alter the parser for my class, to make a single ? mark allowed as an alias for ?s placeholder. It will solve the problem, yet there is always a possibility to fall back into typed mode, for such a rare occasion when we need to set the type.

Hope I convinced you.

??lvaro G. Vicario, 14.10.15 15:09

Awesome article, I'll certainly link it. Yet I still don't understand why data types need to be determined manually (when you don't need to support identifiers) rather than just deduced from PHP data types.

wow, 08.10.15 12:24

really?