MySQL MySQLi PHP Security

PHP MySQLi Prepared Statements Tutorial to Prevent SQL Injection

Nov 8, 2017

Table of Contents

Introduction

Before I start, if you'd like to see an even easier way to use MySQLi prepared statements, check out my wrapper class (https://github.com/WebsiteBeaver/Simple-MySQLi). Also, here's a great resource to learn PDO prepared statements (https://websitebeaver.com/php-pdo-prepared-statements-to-prevent-sql-injection), which is the better choice for beginners and most people in general.

A hack attempt has recently been discovered, and it appears they are trying to take down the entire database. An impromptu staff meeting has been called at 2am, and everyone in the company is freaking out. Ironically, as the database manager, you remain the calmest. Why?

You know that these scrubs are are no match for those prepared statements you coded! In fact, you find this humorous, as these hackers will likely be annoyed that they wasted their time with futile attempts.

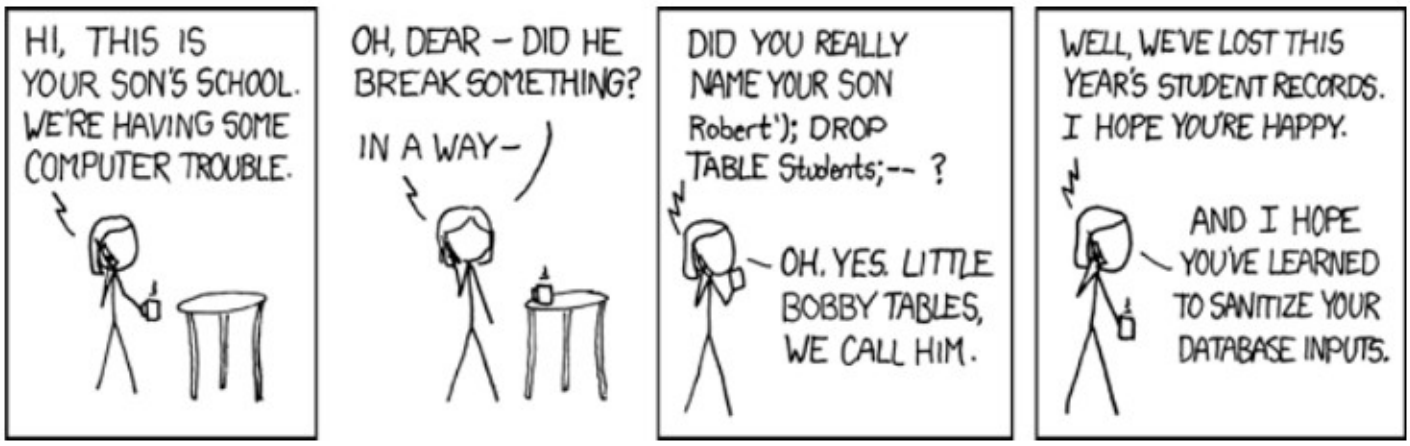
Hopefully this scenario will never happen to your website. However, it is undoubtedly a good idea to take proper precautions. If implemented correctly, prepared statements (aka parameterized queries) offer superior protection against SQL injection. You basically just create the query template with placeholder values, and then replace the dummy inputs with the real ones. Escaping is not necessary, since it will treat the values as literals; all attempts to inject sql queries will be interpreted as such.

Prepared statements may seem intimidating at first, but once you get hang of it, it'll seem like second nature to you. The goal of this tutorial is to transform someone with little to no knowledge of prepared statements, into an expert.

Disclaimer: Don't actually be as laid back as this database manager. When it comes to security, you should never be complacent, no matter how secure you think your system is.

How SQL Injection Works

The following iconic comic, known as *Bobby Tables*, is an excellent portrayal of how an SQL injection attack might work. All credit for the image goes to this site (https://xkcd.com/327/) for this classic piece of rhetoric.



Now that we're done with the theory, let's get to practice. Before I start, if you're wondering exactly how the "*Bobby Tables* Attack" works, check out this explanation (https://stackoverflow.com/a/332367).

In a normal MySQL call, you would do something like:

```
1 $name = $_POST['name'];
2 $mysqli->query("SELECT * FROM myTable WHERE name='$name'");
```

The problem with this, is that if it is based off of user input, like in the example, then a malicious user could do `' OR '1'='1`. Now this statement will always evaluate to true, since `1=1`. In this case, the malicious user now has access to your entire table. Just imagine what could happen if it were a `DELETE` query instead. Take a look at what is actually happening to the statement.

```
SELECT * FROM myTable WHERE name='' OR '1'='1'
```

A hacker could do a lot of damage to your site if your queries are set up like this. An easy fix to this would be to do:

```
1 $name = $mysqli->real_escape_string($_POST['name']);
2 $mysqli->query("SELECT * FROM myTable WHERE name='$name'");
```

Notice how similar to the first example, I still added quotes to the column value. **Without quotes, strings are still equally susceptible to SQL injection**. If you'll be using a LIKE clause, then you should also do `addcslashes($escaped, '%_')`, since `mysqli::real_escape_string` won't do this as stated here (http://php.net/manual/en/mysqli.real-escape-string.php#96429).

This covers strings, as the function name implies, but what about numbers? You could do `(int)$mysqli->real_escape_string($_POST['name'])`, which would certainly work, but that's redundant. If you're casting the variable to an int, you don't need to escape anything. You are already telling it to essentially make sure that the value will be an integer. Doing `(int)$_POST['name']` would suffice. Since it is an integer you also obviously do not need to add quotes to the sql column `name`.

In reality, if you follow these instructions perfectly, it should be enough to use `mysqli::real_escape_string` for strings and `(int)$var` for integers. Just don't forget to set the default character set. This can be set in either the php.ini (should be the default value) like `default_charset = "utf-8"` and by using `$mysqli->set_charset('utf8mb4')` on each page that

uses `$mysqli->real_escape_string()`. But only for things that are legal in prepared statements, which are values in a WHERE statement or column *values*; **don't use this for table/column *names* or SQL keywords**.

Regardless, I still *strongly* suggest using prepared statements, as they are clearly more suited to protect against SQL injection and less prone to mistakes, since you don't have to worry about manually formatting — instead you just have to replace dummy placeholders with your values. Of course you'll still want to filter and sanitize your inputs to prevent XSS however. With prepared statements statements, there are fewer aspects to consider, along with some edge cases to break (http://stackoverflow.com/questions/5741187/sql-injection-that-gets-around-mysql-real-escape-string/12118602#12118602) `$mysqli->real_escape_string()` (Not properly setting the charset is one of the causes.). In summation, **there's absolutely no good reason to be using `real_escape_string()` over prepared statements.** I merely showed how to manually format your queries with it, to show that it's possible. In reality, it would be foolish to not use prepared statements to prevent SQL injection.

How MySQLi Prepared Statements Work

In plain English, this is how MySQLi prepared statements work in PHP:

1. Prepare an SQL query with empty values as placeholders (with a question mark for each value).
2. Bind variables to the placeholders by stating each variable, along with its type.
3. Execute query.

The four variable types allowed:

- i - Integer
- d - Double
- s - String
- b - Blob

A prepared statement, as its name implies, is a way of preparing the MySQL call, without storing the variables. You tell it that variables will go there eventually — just not yet. The best way to demonstrate it is by example.

```
$stmt = $mysqli->prepare("SELECT * FROM myTable WHERE name = ? AND age = ?");
$stmt->bind_param("si", $_POST['name'], $_POST['age']);
$stmt->execute();
//fetching result would go here, but will be covered later
$stmt->close();
```

If you've never seen prepared statements before, this may look a little weird. Basically what's happening is that you are creating a template for what the SQL statement will be. In this case, we are selecting everything from `myTable`, where `name` and `age` equal `?`. The question mark is just a placeholder for where the values will go.

The `bind_param()` method is where you attach variables to the dummy values in the prepared template. Notice how there are two letters in quotes before the variables. This tells the database the variable types. The `s` specifies that name will be a string value, while the `i` forces age to be an integer. This is precisely why I didn't add quotation marks around the question mark for name, like I normally would for a string in an SQL call. You probably thought I just forgot to, but the reality is that there is simply no need to (In fact, it actually won't work if you do put quotes around the `?`, since it will be treated as a string literal, rather than a dummy placeholder.). You are already telling it that it will be a string literal when you call `bind_param()`, so even if a malicious user tries to insert SQL into your user inputs, it will still be treated as a string. `$stmt->execute()` then actually runs the code; the last line simply closes the prepared statement. We will cover fetching results in the Select section.

Creating a New MySQLi Connection

Creating a new MySQLi is pretty simple. I suggest naming a file called `mysqli_connect.php` and place this file outside of your root directly (html, public_html) so your credentials are secure. We'll also be using exception handling, by utilizing `mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT)`. This might look weird to you, especially if you've never used a bitwise operator (http://php.net/manual/en/language.operators.bitwise.php) before. But all it's doing is reporting all errors, while converting them to exceptions, using the mysqli_sql_exception class (http://php.net/manual/en/class.mysqli-sql-exception.php).

```
$mysqli = new mysqli("localhost", "username", "password", "databaseName");
if($mysqli->connect_error) {
  exit('Error connecting to database'); //Should be a message a typical user could understan
}
mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
$mysqli->set_charset("utf8mb4");
```

A lot of tutorials, including the PHP manual, show how to use `$mysqli->connect_error()` by printing it in `exit()` or `die()`. But this isn't really necessary (not to mention incredibly stupid, as you will be printing out this info to the world), since the error message will be appended to your error log anway. The message in `exit()` should be something a normal user could understand, like `exit('Something weird happened')`.

You would think that setting the charset to `utf-8` in your php.ini would suffice, along with `utf8mb4` for your entire database, but sometimes weird errors happen if you don't set it in your php file too, as noted here (http://php.net/manual/en/mysqli.construct.php#115143).

You can alternatively instantiate it in a try/catch block if you enable internal reporting, which I mention in the error handling section. **Please don't ever report errors directly on your site in production.** You'll be kicking yourself for such a silly mistake, since it will print out your sensitive database information (username, password and database name). Here's what your php.ini file should look like in production: do both **`display_errors = Off`** and **`log_errors = On`**. Also, do not echo the error in production.

```
mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
try {
  $mysqli = new mysqli("localhost", "username", "password", "databaseName");
  $mysqli->set_charset("utf8mb4");
} catch(Exception $e) {
  error_log($e->getMessage());
  exit('Error connecting to database'); //Should be a message a typical user could understan
}
```

If you prefer using set_exception_handler() (http://php.net/manual/en/function.set-exception-handler.php) instead of `try/catch`, you can do the following to avoid nesting. If you are using this method, you need to understand that it will affect every page its in included in. Therefore,

you must either reuse the function again with a custom message for each page or use restore_exception_handler() (http://php.net/manual/en/function.restore-exception-handler.php) to revert back to the built in PHP one. If you made multiple ones, it will go to the previous one your made.

```
mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
set_exception_handler(function($e) {
  error_log($e->getMessage());
  exit('Error connecting to database'); //Should be a message a typical user could understan
});
$mysqli = new mysqli("localhost", "username", "password", "databaseName");
$mysqli->set_charset("utf8mb4");
```

There's a very of serious repercussion of using `mysqli_report()`, which is that it will report your sensitive database information. You have three options to still use it but *not* report your password.

1. You can also use `mysqli_report()` strictly on everything *except* for creating the connection if you do it the first method I showed with `$mysqli->connect_error` (password not shown) and just place `mysqli_report()` *after* `new mysqli()`.
2. If you call `mysqli_report()` *before* creating a connection, then **you need to ensure that it's in a try/catch block and you specifically print in your error log `$e->getMessage()`, not `$e`, which still contains your sensitive information**. This obviously strictly applies to the constructor.
3. Use `set_exception_handler()` in the same manner as 2 and use `$e->getMessage()`.

I strongly recommend doing one of these methods. Even if you are diligent and ensure all your errors only goes in your error log, I personally don't see why anyone would need to log their password. You'll already know what the issue is anyway.

Insert, Update and Delete

Inserting, updating and deleting have an identical syntax, so they will be combined.

Insert

```
$stmt = $mysqli->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
$stmt->bind_param("si", $_POST['name'], $_POST['age']);
$stmt->execute();
$stmt->close();
```

Update

```
$stmt = $mysqli->prepare("UPDATE myTable SET name = ? WHERE id = ?");
$stmt->bind_param("si", $_POST['name'], $_SESSION['id']);
$stmt->execute();
$stmt->close();
```

Delete

```
$stmt = $mysqli->prepare("DELETE FROM myTable WHERE id = ?");
$stmt->bind_param("i", $_SESSION['id']);
$stmt->execute();
$stmt->close();
```

Get Number of Affected Rows

You may also want to check the status of a row you inserted, updated or deleted. Here's how you would it if you're updating a row.

```
$stmt = $mysqli->prepare("UPDATE myTable SET name = ?");
$stmt->bind_param("si", $_POST['name'], $_POST['age']);
$stmt->execute();
if($stmt->affected_rows === 0) exit('No rows updated');
$stmt->close();
```

In this case, we checked to see if any rows got updated. For reference, here's the usage for `mysqli::$affected_rows` return values.

-1 - query returned an error; redundant if there is already error handling for `execute()`

0 - no records updated on `UPDATE`, no rows matched the `WHERE` clause or no query has been executed

Greater than 0 - returns number of rows affected; comparable to `mysqli_result::$num_rows` for `SELECT`

Get Rows Matched

A common problem with `$mysqli->affectedRows` is that it makes it impossible to know why it returned zero on an UPDATE. This is due to the fact that it prints the amount of rows changed, so it makes ambiguous if you update your value(s) with the same data.

An awesome feature that is unique to MySQLi, and doesn't exist in PDO, is the ability to get more info about a query. You can technically achieve it in PDO, but it can only be done in the connection, therefore you can't choose.

```
$stmt = $mysqli->prepare("UPDATE myTable SET name = ?");
$stmt->bind_param("si", $_POST['name'], $_POST['age']);
$stmt->execute();
$stmt->close();
echo $mysqli->info;
```

This will print:

```
Rows matched: 1 Changed: 0 Warnings: 0
```

I find this to be a rather imprudent implementation, as it's extremely inelegant to use it as is. Luckily we can change that, by converting it to an associative array. All credit goes do this helpful commenter (http://php.net/manual/en/mysqli.affected-rows.php#116152) on the PHP docs. While using `mysqli->info` for UPDATE is by far its most common use case, it can be used for some other query types (http://php.net/manual/en/mysqli.info.php#refsect1-mysqli.info-description) as well.

```
preg_match_all('/(\S[^:]+): (\d+)/', $mysqli->info, $matches);
$infoArr = array_combine ($matches[1], $matches[2]);
var_export($infoArr);
```

Now this will output an array.

```
['Rows matched' => '1', 'Changed' => '0', 'Warnings' => '0']
```

Get Latest Primary Key Inserted

```
$stmt = $mysqli->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
$stmt->bind_param("si", $_POST['name'], $_POST['age']);
$stmt->execute();
echo $mysqli->insert_id;
$stmt->close();
```

Check if Duplicate Entry

This is useful if you were to create a unique constraint on a table, so duplicates aren't allowed. You can even do this for multiple columns, so it will have to be that exact permutation. If exception handling were turned off, you'd check the error code with `$mysqli->errno`. With exception handling turned on, you could choose between that or the generic exception method `$e->getCode()`. Note, this differs from PDOException, which will print the SQLSTATE, rather than the error code.

Here's a list of error messages (https://dev.mysql.com/doc/refman/5.5/en/error-messages-server.html#error_er_dup_entry). The error code for a duplicate row entry from either an update or insert is **1062** and SQLSTATE is **23000**. To specifically check for SQLSTATE, you must use `$mysqli->sqlstate`.

```
try {
  $stmt = $mysqli->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
  $stmt->bind_param("si", $_POST['name'], $_POST['age']);
  $stmt->execute();
  $stmt->close();
} catch(Exception $e) {
  if($mysqli->errno === 1062) echo 'Duplicate entry';
}
```

This is how you would set a unique constraint:

```
ALTER TABLE myTable ADD CONSTRAINT unique_person UNIQUE (name, age)
```

Select

All select statements in parameterized queries will start off about the same. However, there is a key difference to actually storing and fetching the results. The two methods that exist are `get_result()` and `bind_result()`.

get_result()

This is the more versatile of the two, as it can be used for any scenario. It should be noted that this requires mysqlnd, which has been included in PHP since 5.3 and has been the default native driver since 5.4, as stated here (https://dev.mysql.com/downloads/connector/php-mysqlnd/). I doubt many people are using older versions than that, so you should generally stick with `get_result()`.

This essentially exposes the regular, non-prepared mysqli_result api. Meaning, that once you do `$result = get_result()`, you can use it exactly the same way you'd use `$result = $mysqli->query()`.

Now you can use the following methods for fetching one row at a time or all at once. Here's just some of the most common ones, but you can take a look at the entire mysqli_result class (http://php.net/manual/en/class.mysqli-result.php) for all of its methods.

One Row

- **`$result->fetch_assoc()`** - Fetch an associative array
- **`$result->fetch_row()`** - Fetch a numeric array
- **`$result->fetch_object()`** - Fetch an object array

All

- **`$result->fetch_all(MYSQLI_ASSOC)`** - Fetch an associative array
- **`$result->fetch_all(MYSQLI_NUM)`** - Fetch a numeric array

```
$stmt = $mysqli->prepare("SELECT * FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$result = $stmt->get_result();
if($result->num_rows === 0) exit('No rows');
while($row = $result->fetch_assoc()) {
  $ids[] = $row['id'];
  $names[] = $row['name'];
  $ages[] = $row['age'];
}
var_export($ages);
$stmt->close();
```

Output:

```
[22, 18, 19, 27, 36, 7]
```

bind_result()

You might be wondering, why even use `bind_result()`? I personally find it to be far inferior to `get_result()` in every scenario, except for when fetching a single row into separate variables. Also, before `get_result()` existed and mysqlnd became built into PHP, this was your only

option, which is why a lot of legacy code might be using it.

The most annoying part about using `bind_result()` is that you must bind *every* single column you select and then traverse the values in a loop. This is obviously not ideal for a plethora of values or to use with `*`. The star selector is especially annoying to use with `bind_result()`, since you don't even know what those values are without looking in the database. Additionally, this makes your code exceedingly unmaintainable with changes to the table. This usually won't matter, as you shouldn't be using the wildcard selector in production mode anyway (but you know you are).

```
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$stmt->store_result();
if($stmt->num_rows === 0) exit('No rows');
$stmt->bind_result($idRow, $nameRow, $ageRow);
while($stmt->fetch()) {
  $ids[] = $idRow;
  $names[] = $nameRow;
  $ages[] = $ageRow;
}
var_export($ids);
$stmt->close();
```

Output:

```
[106, 221, 3, 55, 583, 72]
```

Fetch Associative Array

I find this to be the most common use case typically. I will also be utilizing chaining in the following, though that's obviously not necessary.

```
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$arr = $stmt->get_result()->fetch_all(MYSQLI_ASSOC);
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

If you need to modify the result set, then you should probably use a while loop with `fetch_assoc()` and fetch each row one at a time.

```
$arr = [];
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$result = $stmt->get_result();
while($row = $result->fetch_assoc()) {
  $arr[] = $row;
}
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

Output:

```
[
  ['id' => 27, 'name' => 'Jessica', 'age' => 27],
  ['id' => 432, 'name' => 'Jimmy', 'age' => 19]
]
```

You can actually do this using `bind_result()` as well, although it was clearly not designed for it. Here's a clever solution (http://php.net/manual/en/mysqli-stmt.bind-result.php#85470), though I personally feel like it's something that's cool to know is possible, but realistically shouldn't be used.

Fetch Numeric Array

This follows the same format as an associative array. To get the entire array in one command, without a loop, you'd use `mysqli_result->fetch_all(MYSQLI_NUM)` . If you need to fetch the results in a loop, you must to use `mysqli_result->fetch_row()` .

```
$stmt = $mysqli->prepare("SELECT location, favorite_color, age FROM myTable WHERE name = ?")
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$arr = $stmt->get_result()->fetch_all(MYSQLI_NUM);
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

And of course, the while loop adaptation.

```php
$arr = [];
$stmt = $mysqli->prepare("SELECT location, favorite_color, age FROM myTable WHERE name = ?")
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$result = $stmt->get_result();
while($row = $result->fetch_row()) {
  $arr[] = $row;
}
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

Output:

```
[
  ['Boston', 'green', 28],
  ['Seattle', 'blue', 49],
  ['Atlanta', 'pink', 24]
]
```

Fetch Single Row

I personally find it simpler to use `bind_result()` when I know for fact that I will only be fetching one row, as I can access the variables in a cleaner manner.

```php
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$stmt->store_result();
if($stmt->num_rows === 0) exit('No rows');
$stmt->bind_result($id, $name, $age);
$stmt->fetch();
echo $name; //Output: 'Ryan'
$stmt->close();
```

Now you can use just simply use the variables in `bind_result()`, like `$name` since you know they will only contain one value, not an array.

Here's the `get_result()` version:

```php
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$arr = $stmt->get_result()->fetch_assoc();
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

You would then use the variable as `$arr['id']` for example.

Output:

```
['id' => 36, 'name' => 'Kevin', 'age' => 39]
```

Fetch Array of Objects

This very similar to fetching an associative array. The only main difference is that you'll be accessing it like `$arr[0]->age` . Also, in case you didn't know, objects are pass by value, while arrays are by reference.

```php
$arr = []
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE id = ?");
$stmt->bind_param("s", $_SESSION['id']);
$stmt->execute();
$result = $stmt->get_result();
while($row = $result->fetch_object()) {
  $arr[] = $row;
}
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

Output:

```
[
  stdClass Object ['id' => 27, 'name' => 'Jessica', 'age' => 27],
  stdClass Object ['id' => 432, 'name' => 'Jimmy', 'age' => 19]
]
```

You can even add property values to an existing class as well. However, it should be noted that there is a potential gotcha, according to this comment (http://php.net/manual/en/mysqli-result.fetch-object.php#109186) in the PHP docs. The problem is that if you have a default value in your constructor with a duplicate variable name, it will fetch the object first and *then*

set the constructor value, therefore overwriting the fetched result. Weirdly enough, there was a "bug" (https://bugs.php.net/bug.php?id=72151) from PHP 5.6.21 to 7.0.6 where this wouldn't happen. Even though this violates principles of OOP, some people would like this feature, even though it was bug in certain versions. Something like `PDO::FETCH_PROPS_LATE` in PDO should be implemented in MySQLi to give you the option to choose.

```
class myClass {}
$arr = [];
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE id = ?");
$stmt->bind_param("s", $_SESSION['id']);
$stmt->execute();
$result = $stmt->get_result();
while($row = $result->fetch_object('myClass')) {
  $arr[] = $row;
}
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

As the comment states, this is how you would do it correctly. All you need is a simple if condition to check if the variable equals the constructor value — if it doesn't, just don't set it in the constructor. This is essentially the same as using `PDO::FETCH_PROPS_LATE` in PDO.

```
class myClass {
  private $id;
  public function __construct($id = 0) {
    if($this->id === 0) $this->id = $id;
  }
}
$arr = [];
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE id = ?");
$stmt->bind_param("s", $_SESSION['id']);
$stmt->execute();
$result = $stmt->get_result();
while($row = $result->fetch_object('myClass')) {
  $arr[] = $row;
}
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

Another unexpected, yet potentially useful behavior of using `fetch_object('myClass')` is that you can modify private variables. I'm really not sure how I feel about this, as this seems to violate principles of encapsulation.

Conclusion

bind_result() - best used for fetching single row without too many columns or `*`; extremely inelegant for associative arrays.

get_result() - is the preferred one for almost every use-case.

Like

You would probably think that you could do something like:

```
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE Name LIKE %?%");
```

But this is not allowed. The `?` placeholder must be the entire string or integer literal value. This is how you would do it correctly.

```
$search = "%{$_POST['search']}%";
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name LIKE ?");
$stmt->bind_param("s", $search);
$stmt->execute();
$arr = $stmt->get_result()->fetch_all(MYSQLI_ASSOC);
if(!$arr) exit('No rows');
var_export($arr);
$stmt->close();
```

Where In Array

This is definitely something I'd like to see improved in MySQLi. For now, using MySQLi prepared statements with `WHERE IN` is possible, but feels a little long-winded.

Side note: The following two examples use the splat operator (http://php.net/manual/en/migration56.new-features.php#migration56.new-features.splat) for argument unpacking, which requires PHP 5.6+. If you are using a version lower than that, then you can substitute it with `call_user_func_array()`.

```php
$inArr = [12, 23, 44];
$clause = implode(',', array_fill(0, count($inArr), '?')); //create 3 question marks
$types = str_repeat('i', count($inArr)); //create 3 ints for bind_param
$stmt = $mysqli->prepare("SELECT id, name FROM myTable WHERE id IN ($clause)");
$stmt->bind_param($types, ...$inArr);
$stmt->execute();
$resArr = $stmt->get_result()->fetch_all(MYSQLI_ASSOC);
if(!$resArr) exit('No rows');
var_export($resArr);
$stmt->close();
```

With Other Placeholders

The first example showed how to use the `WHERE IN` clause with dummy placeholder solely inside of it. What if you wanted to use other placeholders in different places?

```php
$inArr = [12, 23, 44];
$clause = implode(',', array_fill(0, count($inArr), '?')); //create 3 question marks
$types = str_repeat('i', count($inArr)); //create 3 ints for bind_param
$types .= 'i'; //add 1 more int type
$fullArr = array_merge($inArr, [26]); //merge WHERE IN array with other value(s)
$stmt = $mysqli->prepare("SELECT id, name FROM myTable WHERE id IN ($clause) AND age > ?");
$stmt->bind_param($types, ...$fullArr); //4 placeholders to bind
$stmt->execute();
$resArr = $stmt->get_result()->fetch_all(MYSQLI_ASSOC);
if(!$resArr) exit('No rows');
var_export($resArr);
$stmt->close();
```

Multiple Prepared Statements in Transactions

This might seem odd why it would even warrant its own section, as you can literally just use prepared statements one after another. While this will certainly work, this does not ensure that your queries are atomic. This means that if you were to run ten queries, and one failed, the other nine would still succeed. If you want your SQL queries to execute only if they all succeeded, then you must use transactions.

```php
try {
  $mysqli->autocommit(FALSE); //turn on transactions
  $stmt1 = $mysqli->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
  $stmt2 = $mysqli->prepare("UPDATE myTable SET name = ? WHERE id = ?");
  $stmt1->bind_param("si", $_POST['name'], $_POST['age']);
  $stmt2->bind_param("si", $_POST['name'], $_SESSION['id']);
  $stmt1->execute();
  $stmt2->execute();
  $stmt1->close();
  $stmt2->close();
  $mysqli->autocommit(TRUE); //turn off transactions + commit queued queries
} catch(Exception $e) {
  $mysqli->rollback(); //remove all queries from queue if error (undo)
  throw $e;
}
```

Reuse Same Template, Different Values

```php
try {
  $mysqli->autocommit(FALSE); //turn on transactions
  $stmt = $mysqli->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
  $stmt->bind_param("si", $name, $age);
  $name = 'John';
  $age = 21;
  $stmt->execute();
  $name = 'Rick';
  $age = 24;
  $stmt->execute();
  $stmt->close();
  $mysqli->autocommit(TRUE); //turn off transactions + commit queued queries
} catch(Exception $e) {
  $mysqli->rollback(); //remove all queries from queue if error (undo)
  throw $e;
}
```

Error Handling

Fatal error: Uncaught Error: Call to a member function bind_param() on boolean

Anyone who's used MySQLi prepared statements has seen this message at some point, but what does it mean? Pretty much nothing at all. So how do you fix this, you might ask? To start, don't forget to turn on exception handling, instead of error handling `mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT)` when you create a new connection.

Exception Handling

All of the mysqli functions return false on failure, so you could easily just check for truthiness on each function and report errors with `$mysqli->error`. However, this is very tedious, and there's a more elegant way of doing this if you enable internal reporting. I recommend doing it this way, as it's much more portable from development to production.

This can be used in production too, as long as you have an error log set up for all errors; this needs to be set in the php.ini. **Please don't ever report errors directly on your site in production.** You'll be kicking yourself for such a silly mistake. The placement of `mysqli_report()` matters also. if you place it *before* creating a new connection then it will output your password too; otherwise, it will just report everything after, like your queries.

Here's what your php.ini file should look like in production: do both `display_errors = Off` and `log_errors = On`. Also, keep in mind that each page should really only be using a single, global, `try/catch` block, rather than wrapping each query individually. The only *exception* to this is with transactions, which would be nested, but throw its own exception, so the global `try/catch` can "catch" it.

```
try {
  $stmt = $mysqli->prepare("DELETE FROM myTable WHERE id = ?");
  $stmt->bind_param("i", $_SESSION['id']);
  $stmt->execute();
  $stmt->close();

  $stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
  $stmt->bind_param("s", $_POST['name']);
  $stmt->execute();
  $arr = $stmt->get_result()->fetch_all(MYSQLI_ASSOC);
  $stmt->close();

  try {
    $mysqli->autocommit(FALSE); //turn on transactions
    $stmt = $mysqli->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
    $stmt->bind_param("si", $name, $age);
    $name = 'John';
    $age = 21;
    $stmt->execute();
    $name = 'Rick';
    $age = 24;
    $stmt->execute();
    $stmt->close();
    $mysqli->autocommit(TRUE); //turn off transactions + commit queued queries
  } catch(Exception $e) {
    $mysqli->rollback(); //remove all queries from queue if error (undo)
    throw $e;
  }
} catch (Exception $e) {
  error_log($e);
  exit('Error message for user to understand');
}
```

Custom Exception Handler

As stated earlier, you can alternatively use `set_exception_handler()` on each page (or a global redirect). This gets rid of the layer of curly brace nesting. If you are using transactions, you should still use a try catch with that, but then throw your own exception.

```
set_exception_handler(function($e) {
  error_log($e);
  exit('Error deleting');
});
$stmt = $mysqli->prepare("DELETE FROM myTable WHERE id = ?");
$stmt->bind_param("i", $_SESSION['id']);
$stmt->execute();
$stmt->close();
```

Gotcha with Exception Handling

You'd expect for all MySQLi errors to be converted to exceptions with `mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT)`. Oddly enough, I noticed that it still gave me a warning error when `bind_param()` had too many or too little bound variables or types. The message outputted is as follows:

> Warning: mysqli_stmt::bind_param(): Number of variables doesn't match number of parameters in prepared statement

A solution to this is to use a global error handler to trigger an exception. An example of this could be:

```
set_error_handler(function($errno, $errstr, $errfile, $errline) {
  throw new Exception("$errstr on line $errline in file $errfile");
});
```

This only happened on runtime warnings, but I converted all errors to exceptions. I see no problem doing this, but there are some people who are strongly against it.

Some Extras

Do I Need $stmt->close()?

Great question. Both `$mysqli->close()` and `$stmt->close()` essentially have the same effect. The former closes the MySQLi connection, while the latter closes the prepared statement. **TLDR;** both are actually generally not even necessary in most cases, since both will close once the script's execution is complete anyway. There's also a function to simply free the memory associated with the MySQLi result and prepared statement, respectively: `$result->free()` and `$stmt->free()`. I myself, will likely never use it, but if you're interested, here's the one for the result (http://www.php.net/manual/en/mysqli-result.free.php) and for the the parameterized query (http://www.php.net/manual/en/mysqli-stmt.free-result.php). The following should also be noted: both `$stmt->close()` and the end of the execution of the script will the free up the memory anyway.

Final verdict: I usually just do `$mysqli->close()` and `$stmt->close()`, even though it can be argued that it's a little superfluous. If you are planning on using the same variable `$stmt` again for another prepared statements, then you must either close it, or use a different variable name, like `$stmt2`. Lastly, I have never found a need to simply free them, without closing them.

Classes: mysqli vs. mysqli_stmt vs. mysqli_result

One thing you may have realized along the way is that there are certain methods that exist in two of the classes, like an alias almost. I personally believe it would be better to only have one version, like in PDO, to avoid confusion.

- **`mysqli::$affected_rows`** (http://php.net/manual/en/mysqli.affected-rows.php) or **`mysqli_stmt::$affected_rows`** (http://php.net/manual/en/mysqli-stmt.affected-rows.php) - Belongs to `mysqli_stmt`. Works the same with either, but will be an error if called *after* the statement is closed with either method
- **`mysqli_result::$num_rows`** (http://php.net/manual/en/mysqli-result.num-rows.php) or **`mysqli_stmt::$num_rows`** (http://php.net/manual/en/mysqli-stmt.num-rows.php) - `$result->num_rows` can only be used with `get_result()`, while `$stmt->num_rows` can only be used with `bind_result()`.
- **`mysqli::$insert_id`** (http://php.net/manual/en/mysqli.insert-id.phpp) or **`mysqli_stmt::$insert_id`** (http://php.net/manual/en/mysqli-stmt.insert-id.php) - Belongs to `mysqli`. Better to use `$mysqli->insert_id`, since it will still work even after `$stmt->close()` is used. There's also a note (http://php.net/manual/en/mysqli-stmt.insert-id.php#102299) on the PHP docs from 2011 stating that `$stmt->insert_id` will only get the first executed query. I tried this on my current version of 7.1 and this doesn't seem to be the case. The recommended one to use is the mysqli class version anyway.

So Using Prepared Statements Means I'm Safe From Attackers?

While you are safe from SQL injection, you still need validate and sanitize your user-inputted data. You can use a function like filter_var() (http://php.net/manual/en/function.filter-var.php) to validate *before* inserting it into the database and htmlspecialchars() (http://php.net/manual/en/function.htmlspecialchars.php) to sanitize *after* retrieving it.

Series