PHP MySQLi Prepared Statements to Prevent SQL Injection

Nov 8, 2017

https://websitebeaver.com/prepared-statements-in-php-mysqli-to-prevent-sql-injection

Introduction

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.

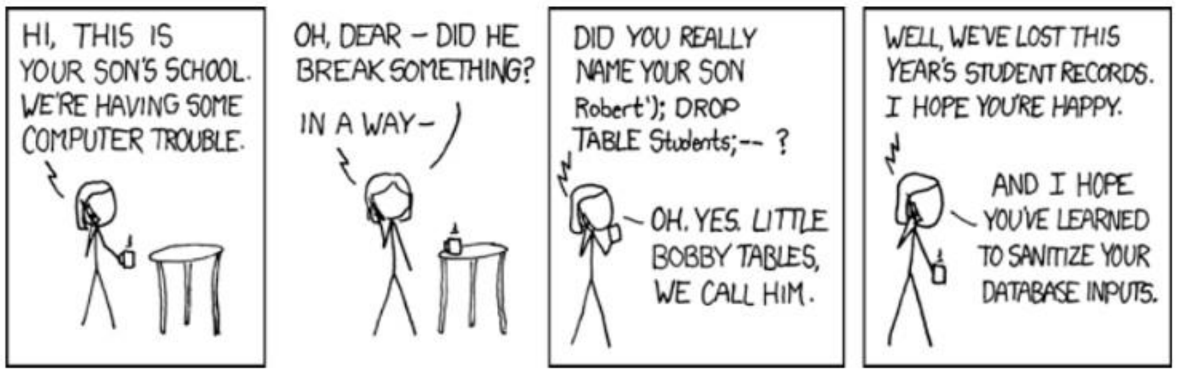
Hopefully this scenario will never happen to your website. However, it is undoubtedly a good idea to take proper precautions. If implemented correctly, prepared statements (aka parameterized queries) offer superior protection against SQL injection. You basically just create the query template with placeholder 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 for this classic piece of rhetoric.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

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.

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

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

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'
```
Copy

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:

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

```
Copy
```

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.

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)$var` for integers. Just don't forget to set the default character set. This can be set in either the php.ini (should be the default value) like `default_charset = "utf-8"` or by using `$mysqli->set_charset('utf8')` 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 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 `$mysqli->real_escape_string()` (Not properly setting the charset is one of the causes.).

If you'd like to learn PDO prepared statements, check out this post.

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();
```
Copy

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.

```
$mysqli = new mysqli("localhost", "username", "password", "databaseName");
if($mysqli->connect_error) {
  exit('Error connecting to database'); //Should be a message a typical user could understand in production
}
$mysqli->set_charset("utf8");
```
Copy

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, since if you are in development, it will just print out the error anyway; in production, the error message will be appended to your error log. 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.

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`.

```
mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
try {
  $mysqli = new mysqli("localhost", "username", "password", "databaseName");
  $mysqli->set_charset("utf8");
} catch(Exception $e) {
  echo $e->getMessage(); //use in development
  error_log($e->getMessage()); //use in production
  exit('Error connecting to database'); //Should be a message a typical user could understand in production
}
Copy
```

If you prefer using set_exception_handler() 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() to revert back to the built in PHP one.

```
mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
set_exception_handler(function($e) {
  echo $e->getMessage(); //use in development
  error_log($e->getMessage()); //In production
  exit('Error connecting to database'); //Should be a message a typical user could
understand in production
});
$mysqli = new mysqli("localhost", "username", "password", "databaseName");
$mysqli->set_charset("utf8");
Copy
```

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 `$mysqli->connect_error` (password not shown) and just place `mysqli_report()` *after* `new mysqli()`.
2. If you call `mysqli_report()` *before* creating a connection, then you need to ensure that it's in a try/catch block and you specifically print out `$e->getMessage()`, not `$e`, which still contains your sensitive information.
3. Use `set_exception_handler()` in the same manner as 2 and use `$e->getMessage()`.

I strongly recommend doing one of these methods (preferably 2 or 3). 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();
```

Copy

Update

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

Copy

Delete

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

Copy

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) echo 'No rows updated';
$stmt->close();
```

Copy

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 been executed

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

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.

```
$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()) {
  $id[] = $row['id'];
  $name[] = $row['name'];
  $age[] = $row['age'];
}
$stmt->close();
Copy
```

bind_result()

You might be wondering, why even use `bind_result()`? This is strictly due to preference, as the syntax is considered to be more readable by some. Also, before `get_result()` existed and mysqlnd became built into PHP, this was your only option.

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()) {
  $id[] = $idRow;
  $name[] = $nameRow;
  $age[] = $ageRow;
}
$stmt->close();
```
Copy

Fetch all Selected Rows in Associative Array

I find this to be the most common use case typically.

```
$stmt = $mysqli->prepare("SELECT id, name, age 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()) {
  $arr[] = $row;
}
$stmt->close();
```
Copy

You can even shorten this up a little more. I am providing both methods so you can decide which one you like more.

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

```
if($result->num_rows === 0) exit('No rows');
$arr = $result->fetch_all(MYSQLI_ASSOC);
$stmt->close();
```

Copy

You can actually do this using `bind_result()` as well, although it was clearly not designed for it. Here's [a clever solution](), though I personally feel like it's something that's cool to know is possible, but realistically shouldn't be used.

Selecting 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.

```
$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 !== 1) exit('More or less than 1 row');
$stmt->bind_result($id, $name, $age);
$stmt->fetch();
$stmt->close();
```

Copy

Now you can use just simply use the variables in `bind_result()`, since you know they will only contain one value, not an array. It is comparable to `$id = $id[0]`.

Here's the `get_result()` version:

```
$stmt = $mysqli->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
$stmt->bind_param("s", $_POST['name']);
$stmt->execute();
$result = $stmt->get_result();
if($result->num_rows !== 1) exit('More or less than 1 row');
$row = $result->fetch_assoc();
$stmt->close();
```

Copy

You would then use the variable as `$row['id']` for example. Similar to `bind_result()`, this variable is equivalent to `$row['id'] = $row['id'][0]`.

Conclusion

bind_result() - best used for fetching single row without too many columns or `*`; extremely inelegant for associative arrays; avoids `$row['val']` syntax.

get_result() - best used for multiple rows and fetching associative arrays.

Like

You would probably think that you could do something like:

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

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();
$result = $stmt->get_result();
if($result->num_rows === 0) exit('No rows');
$arr = $result->fetch_all(MYSQLI_ASSOC);
$stmt->close();
Copy
```

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 for argument unpacking, which requires PHP 5.6+. If you are using a version lower than that (I really hope not in this day and age), then you can substitute it with `call_user_func_array()`.

```
$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();
$result = $stmt->get_result();
if($result->num_rows === 0) exit('No rows');
$resArr = $result->fetch_all(MYSQLI_ASSOC);
$stmt->close();
```
Copy

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?

```
$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();
$result = $stmt->get_result();
if($result->num_rows === 0) exit('No rows');
$resArr = $result->fetch_all(MYSQLI_ASSOC);
$stmt->close();
```
Copy

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.

```
mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
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)
  echo $e; //use in development
  error_log($e); //use in production
}
```
Copy

Reuse Same Template, Different Values

```
mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
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)
  echo $e; //use in development
  error_log($e); //use in production
}
```
Copy

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?

Error Handling is fairly easy, as all of the functions you need to check return false. To get a meaningful error message, you can simply use the `error` and `errno` functions on the MySQLi functions. Furthermore, you can even get error messages from the prepared statement itself after `execute()` is called.

Regular Error Handling

```
if ( !$stmt = $mysqli->prepare("SELECT * FROM myTable WHERE name = ?") )
 echo "Prepare Error: ($mysqli->errno) $mysqli->error";
if ( !$stmt->bind_param("s", $_POST['name']) )
  echo "Binding Parameter Error: ($mysqli->errno) $mysqli->error";
if ( !$stmt->execute() )
 echo "Execute Error: ($stmt->errno)  $stmt->error";
if ( !$stmt->get_result() ) //Only for select with get_result()
 echo "Getting Result Error: ($mysqli->errno) $mysqli->error";
if ( !$stmt->store_result() ) //Only for select with bind_result()
 echo "Storing Result Error: ($mysqli->errno) $mysqli->error";
$stmt->close();
```
Copy

Exception Handling

There's also 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`.

```
mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
try {
  $stmt = $mysqli->prepare("DELETE FROM myTable WHERE id = ?");
  $stmt->bind_param("i", $_SESSION['id']);
  $stmt->execute();
  $stmt->close();
} catch (Exception $e) {
  echo $e; //use in development
  error_log($e); //use in production
  exit('Error deleting');
}
```
Copy

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.

```
//include mysqli_connect.php
mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
set_exception_handler(function($e) {
  echo $e; //use in development
  error_log($e); //use in production
  exit('Error deleting');
});
$stmt = $mysqli->prepare("DELETE FROM myTable WHERE id = ?");
$stmt->bind_param("i", $_SESSION['id']);
$stmt->execute();
$stmt->close();
```

```
Copy
```

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 and for the the parameterized query. 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.

Should I Always be Using Prepared Statements?

I can see the confusion for this — the answer is absolutely not. I mean, you can if you want, but there doesn't really seem to be a need to write extra code and be less efficient (though I'm sure negligible for most people, unless you're named Mark Zuckerberg). As per the PHP docs:

Using a prepared statement is not always the most efficient way of executing a statement. A prepared statement executed only once causes more client-server round-trips than a non-prepared statement.

My favorite example of showing this is when you only need to select a scalar (one value) from your table. Check out this sick one liner to select a the oldest person from your table.

```
$oldest = $mysqli->query("SELECT MAX(age) AS oldest FROM myTable")->fetch_object()->oldest;
Copy
```

Pretty neat, right? So you if you don't have any user-inputted values, why not just utilize this awesomeness, if a prepared statement isn't needed.

The only argument I can see for using a prepared statement when it's not needed, is if you are anticipating that you will eventually be adding user-inputted values. That could be considered excellent preemptive thinking in my opinion.

Should I use mysqli::$insert_id or mysqli_stmt::$insert_id?

It almost seems like they are aliases, which is true in almost every use case. They both will get the primary key of the latest row inserted. The only difference is that it will be incorrect if you do `$stmt->insert_id` on a prepared statement statement that is used multiple times. I had no idea about this behavior, until I read the following comment on the PHP docs.

It should be noted that using `mysqli_stmt->insert_id` will not result in a unique ID being returned for each execution of a prepared insert statement. In practice, it appears that the first insertion ID is returned. If you are performing multiple inserts with the same prepared statement (one invocation of `mysqli_stmt::prepare` and multiple invocations of `mysqli_stmt::execute()` for a given statement), and need to keep the unique ID for each insert, use `mysqli_connection->insert_id`.

Thus, you should probably just stick with `$mysqli->insert_id`, since it performs the same task, without that gotcha.

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. To help prevent persistent XSS, you can use functions like filter_var() to validate *before* inserting it into the database and htmlspecialchars() to sanitize *after* retrieving it.

It should be noted that these are just my opinions on these matters. I am not a believer in dogmatic philosophies when it comes to programming, so I encourage all readers to understand the facts, and make his or her own conclusions.