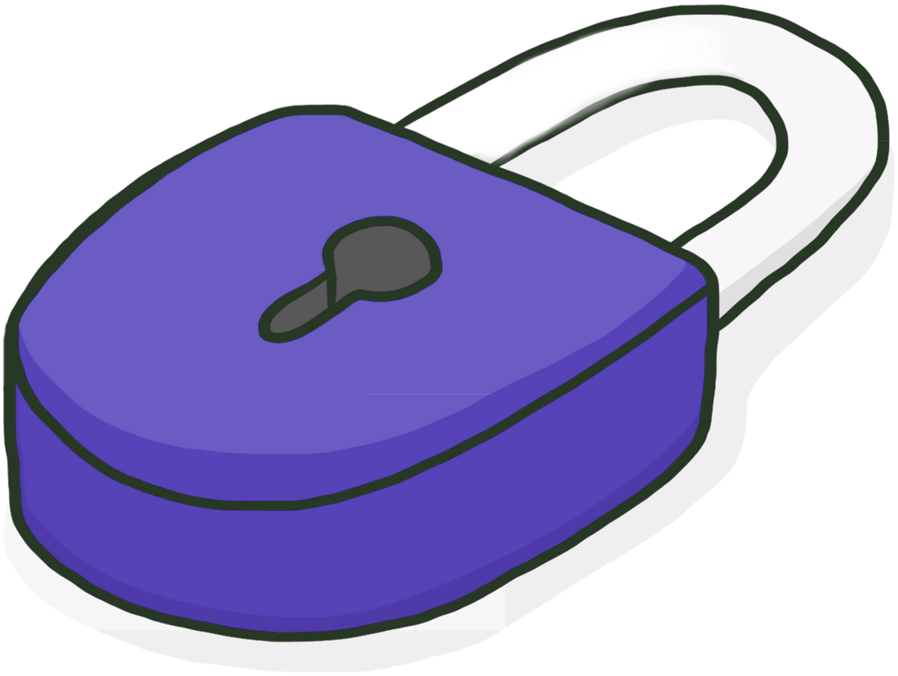
BUILDING SECURE PHP APPS



a practical guide

Ben Edmunds

Building Secure PHP Apps

is your PHP app truly secure? Let's make sure you get home on time and sleep well at night.

Ben Edmunds

This book is for sale at
http://leanpub.com/buildingsecurephpapps

This version was published on 2016-12-10

Tweet This Book!

Please help Ben Edmunds by spreading the word about this book on Twitter!

The suggested hashtag for this book is #buildingsecurephpapps.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#buildingsecurephpapps

Contents

Chapter 1 - Never Trust Your Users. Sanitize ALL Input!

Let's start with a story. Mike is the system admin for a small private school in Oklahoma. His main responsibility is keeping the network and computers working. Recently he started automating various tasks around the school by building a web application for internal use. He doesn't have any formal training and just started programming about a year ago, but he feels pretty good about his work. He knows the basics of PHP and has built a pretty stable customer relationship manager for the school. There are still a ton of features to add, but the basics are covered. Mike even received kudos from the superintendent for streamlining operations and saving the school money.

Everything was going well for Mike until a particular new student started. The student's name is Little Bobby Tables[1]. One day, Jon from the admin office called Mike to ask why the system was down. After inspecting, Mike found that the table containing all the students' information was missing entirely. You see, Little Bobby's full name is actually "Robert'); DROP TABLE students;–". There aren't any backups of the database; it has been on Mike's "to do" list for a while, but he hadn't gotten around to it yet. Mike is in big trouble.

[1]http://xkcd.com/327/

SQL Injection

Real World

While it's unlikely a real child's name will contain damaging SQL code, this kind of **SQL injection vulnerability** happens in the real world all the time:[2]

- In 2012, LinkedIn leaked over 6 million users' data due to an undisclosed SQL injection vulnerability
- In 2012, Yahoo! exposed 450,000 user passwords
- In 2012, 400,000 passwords were compromised from Nvidia
- In 2012, 150,000 passwords were compromised from Adobe
- In 2013, eHarmony had roughly 1.5 million user passwords exposed

How SQL Injection Works

If you use input directly from your users without modification, a malicious user can pass unexpected data, and fundamentally change your SQL queries.

If your code looks something like this:[3]

[2]For most of these precise details were undisclosed, so we can't be certain these were due to SQL injection attacks. Chances are the majority were though.

[3]The `mysql_*` extension and it's methods are officially deprecated. Please don't use them.

```
mysql_query('UPDATE users
  SET first_name="' . $_POST['first_name'] . '"
  WHERE id=1001');
```

You would expect the generated SQL to be:

```
UPDATE users set first_name="Liz" WHERE id=1001;
```

But if your malicious user types their first name as:

```
Liz", last_name="Lemon"; --
```

The generated SQL then becomes:

```
UPDATE users
SET first_name="Liz", last_name="Lemon"; --"
WHERE id=1001;
```

Now all of your users are named Liz Lemon, and that's just not cool.

How To Guard Against It

The single requirement for guarding against SQL injection is to **sanitize input** (also known as **escaping**). You can escape each input individually, or use a better method known as **parameter binding**. Parameter binding is definitely the way I recommend, as it offers more security. Using PHP's `PDO` class[4], your code now becomes:

[4]http://us1.php.net/manual/en/intro.pdo.php

```
$db = new PDO(...);
$query = $db->prepare('UPDATE users
  SET first_name = :first_name
  WHERE id = :id');

$query->execute([
  ':id'         => 1001,
  ':first_name' => $_POST['first_name']
]);
```

Using bound parameters means that each value will be escaped, quoted properly, and only one value is expected. Keep in mind, bound parameters protect your query, but they don't protect the input data after it enters your database. Remember, *any* data can be malicious. You will still need to strip out and/or escape data that will be displayed back to the user. You can do this when you save the data to the database, or when you output it, but don't skip this very important step. We'll cover this more in the "Sanitizing Output" section coming up.

Your code is now a little longer, but it's safe. You won't have to worry about another Little Bobby Tables screwing up your day. Bound parameters are pretty awesome right? You know what else is awesome, Funyuns are awesome.

Best Practices and Other Solutions

Stored procedures are another way to protect against SQL injection. A stored procedure is a function built in your database. Using a stored procedure means you're less likely to be susceptible to SQL injection, since your data isn't passed directly as SQL. In general, stored procedures are frowned upon. The main reasons for which include:

1. Stored procedures are difficult to test
2. They move the logic to another system outside of the application
3. They are difficult to track in your version control system, since they live in the database and not in your code
4. Using them can limit the number of people on your team capable of modifying the logic if needed

Client-side JavaScript is NOT a solution for validating data, ever. It can be easily modified or avoided by a malicious user with even a mediocre amount of knowledge. Repeat after me: I will NEVER rely on JavaScript validation; I will NEVER EVER rely on JavaScript validation. You can certainly use JavaScript validation to provide instant feedback and present a better user experience, but for the love of your favorite deity, check the input on the back end to make sure everything is legit.

Mass Assignment

Mass assignment can be an incredibly useful tool that can speed up development time, or cause severe damage if used improperly.

Let's say you have a `User` model that you need to update with several changes. You could update each field individually, or you could pass all of the changes from a form and update it in one go.

Your form might look like this:

```html
<form action="...">
  <input name="first_name" />
  <input name="last_name" />
  <input name="email" />
</form>
```

Then you have back end PHP code to process and save the form submission. Using the Laravel framework, that might look like this:

```php
$user = User::find(1);
$user->update(Input::all());
```

Quick and easy right? But what if a malicious user modifies the form, giving themselves administrator permissions?

```
<form action="...">
  <input type="text" name="first_name" />
  <input type="text" name="last_name" />
  <input type="text" name="email" />
  <input type="hidden" name="permissions" value="\
{'admin':'true'}" />
</form>
```

That same code would now change this user's permissions erroneously.

This may sound like a dumb problem to solve, but it is one that a lot of developers and sites have fallen victim to. The most recent, well-known exploit of this vulnerability was when a user exposed that Ruby on Rails was susceptible to this. When Egor Homakov originally reported to the Rails team that new Rails installs were insecure, his bug report was rejected. The core team thought it was a minor concern that would be easier for new developers to leave enabled by default. To get attention to this issue, Homakov hilariously "hacked" Rails' GitHub account (GitHub is built on Rails) to give himself administrative rights to their repositories. Needless to say, this proved his point, and now Rails (and GitHub) are protected from this attack by default.

How do you protect your application against this? The exact implementation details depend on which framework or code base you're using, but you have a few options:

- Turn off mass assignment completely
- Whitelist the fields that are safe to be mass assigned
- Blacklist the fields that are not safe to be mass assigned

Depending on your implementation, some of these may be used simultaneously.

In Laravel you add a `$fillable` property to your models to set the whitelist of fields that are mass assignable:

```
class User extends Eloquent {

  protected $table = 'users';

  protected $fillable = ['first_name', 'last_name\
', 'email'];
```

This would stop the "permissions" column from being mass assigned. Another way to handle this in Laravel is to set a blacklist with the `$guarded` property:

```
class User extends Eloquent {

  protected $table = 'users';

  protected $guarded = ['permissions'];
```

The choice is up to you, depending on which is easier in your application.

If you don't use Laravel, your framework probably has a similar method of whitelisting/blacklisting mass assignable fields. If you use a custom framework, get on implementing whitelists and blacklists!

Typecasting

One additional step I like to take, not just for security but also for data integrity, is to typecast known formats. Since PHP is a dynamically typed language[5], a value can be any type: string, integer, float, etc. By typecasting the value, we can verify that the data matches what we expect. In the previous example, if the ID was coming from a variable it would make sense to typecast it if we knew it should always be an integer, like this:

```
$id = (int) 1001;

$db    = new PDO(...);
$query = $db->prepare('UPDATE users
  SET first_name = :first_name
  WHERE id = :id');

$query->execute([
  ':id'         => $id, //we know its an int
  ':first_name' => $_POST['first_name']
]);
```

In this case it wouldn't matter much since we are defining the ID ourselves, so we know its an integer. But if the ID came from a posted form or another source, this would give us additional peace of mind.

PHP supports a number of types that you can cast to, they are

[5]http://stackoverflow.com/questions/7394711/what-is-dynamic-typing

```
$var = (array) $var;
$var = (binary) $var;
$var = (bool) $var;
$var = (boolean) $var;
$var = (double) $var;
$var = (float) $var;
$var = (int) $var;
$var = (integer) $var;
$var = (object) $var;
$var = (real) $var;
$var = (string) $var;
```

This is helpful not only when dealing with your database, but throughout your application. Just because PHP is dynamically typed doesn't mean that you can't enforce typing in certain places. Yeah science!

Sanitizing Output

Outputting to the Browser

Not only should you take precautions when saving the data you take in, you should sanitize / escape any user-generated data that is output back to the browser.

You can modify and escape your data prior to saving to the database, or in between retrieving it and outputting to the browser. It usually depends on how your data is edited and used. For example, if the user is editing the data later, it usually makes more sense to save it as-is, and sanitize upon output.

What security benefits come from escaping user-generated data that you output? Suppose a user submits the following JavaScript snippet to your application, which saves it for outputting later:

```
<script>alert('I am not sanitized!');</script>
```

If you don't sanitize this code before you echo it out to the browser, the malicious JavaScript will run normally, as if you wrote it yourself. In this case it's a harmless `alert()`, but a hacker won't be nearly as kind.

Another popular place for this type of exploit is in an image's XIFF data. If a user uploads an image and your application displays the XIFF data, it will need to be sanitized as well. Anywhere you are displaying data that came into your app from the outside, you need to sanitize it.

If you're using a templating library or a framework that handles templating, escaping may happen automatically, or there is a built-in method for doing so. Make sure to check the documentation for your library / framework of choice to determine how this works.

For those of you handling this yourself, PHP provides a couple of functions that will be your best friends when displaying data in the browser: `htmlentities()`[6] and `htmlspecialchars()`[7]. Both will escape and manipulate data to make it safer before rendering.

`htmlspecialchars()` should be your go-to function in 90% of cases. It will look for characters with special meaning (e.g., `<`, `>`, `&`) and encode these characters to HTML entities.

`htmlentities()` is like `htmlspecialchars()` on steroids. It will encode any character into its HTML entity equivalent if one exists. This may or may not be what you need in many cases. Make sure to understand what each one of these functions does exactly, then evaluate which is best for the type of data you are sending to the browser.

[6]http://us1.php.net/htmlentities

[7]http://us1.php.net/htmlspecialchars

Echoing to the Command Line

Don't forget to sanitize the output of any command line script you are running. The functions for this are `escapeshell-cmd()`[8] and `escapeshellarg()`[9].

They are both pretty self-explanatory. Use `escapeshellcmd()` to escape any commands that you are calling. This will prevent arbitrary commands from being executed. `escapeshel-larg()` is used to wrap arguments to ensure they are escaped correctly, and don't open your application up to manipulating the structure of the commands.

[8]http://us1.php.net/escapeshellcmd

[9]http://us1.php.net/escapeshellarg