Web Security 2016

from php[architect] magazine

Edited By Oscar Merida

a php[architect] anthology

Are you keeping up with modern security practices? This anthology collects articles first published in php[architect] magazine. Each one touches on a security topic to help you harden and secure your PHP and web applications. Your users' information is important, make sure you're treating it with care.

This anthology includes:

- An overview of the **attacks you should be familiar with** and how to protect against exploits.
- Using a PHP-based **Intrusion Detection System** to monitor and reject requests that attempt to breach your site.
- How to protect against **SQL Injection** from user-supplied data by using prepared statements.
- A case study in how the **Drupal security** team keeps core and contributed modules safe.
- How to **securely store passwords** and understanding the techniques used to crack credentials.
- Using **OAuth 2.0** to connect to web services and fetch information for your users without asking for a password.
- How **web service security** differs from traditional web application security and advice for effectively protecting one from malicious users.
- Identifying the right kind of **cryptography** to implement in your application and doing it correctly.

Each month in php[architect] magazine, experts from the PHP community and wider web development community share their knowledge and experience with our readers. Leverage their expertise in building and protecting websites for all types of organizations.

a php[architect] anthology

www.phparch.com

Web Security 2016

From php[architect] Magazine

Edited By
Oscar Merida

a php[architect] anthology

Table of Contents

Introduction

"The mantra of any good security engineer is: 'Security is a not a product, but a process.' It's more than designing strong cryptography into a system; it's designing the entire system such that all security measures, including cryptography, work together."

— Bruce Schneier

Whenever I discuss computer security, I like to remind people of the sentiment in the quote above. Security is not just a box you can check off and be done with prior to a release. On the web, attackers probe and discover new vulnerabilities on a daily basis not just in PHP itself but in the applications built on it.

Sensitive information, from financial to medical records, is migrating to the 24/7 connected, online world. Keeping your application—and more importantly your users' data—secure is an iterative process requiring regularly reviewing every component in your stack for new vulnerabilities, keeping them patched and updated, and vetting new parts to ensure they don't compromise the overall system.

In the following pages, we've collected security articles from the pages of php[architect] magazine. Read on to see how to make sure you use modern techniques to monitor your systems and keep them secure.

To start, Nicola Pietroluongo asks *Is Your Website Secure from Hackers?* He provides an overview of the attacks you should know and also how to protect your applications against them.

In *Basic Intrusion Detection with Expose*, Greg Wilson makes the case for having an Intrusion Detection System in place and shows you how to setup Expose, a PHP-based IDS.

David Stockton explores SQL injection and how to prevent it in *DeLoreans, Data, and Hacking Sites*. If you still have legacy code that concatenates strings to build SQL queries, don't miss this article.

Cathy Theys looks at the Drupal ecosystem in *Drupal Security: How Open Source Strengths Manage Software Vulnerabilities*. She explains how the Drupal project's security team leverages Open Source to keep Drupal core and contributed modules secure.

Leszek Krupiński writes about how passwords are stored and the techniques used to crack them in *Keep Your Passwords Hashed and Salted*. Learn how passwords can be reversed, given enough computing password, and how you can mitigate this risk.

In *Mastering OAuth 2.0*, Ben Ramsey shows how to use the league/oauth2-client library to connect to Instagram. OAuth is now a de facto standard for connecting your application to web services, and this article is a step-by-step example explaining how it all works.

Edward Barnard starts a three-part series with *Learn from the Enemy: Securing Your Web Services, Part One*. In this part, he'll show you why your website and web service should be treated differently when talking about security.

If your decoupled application is talking to or providing one or more APIs, don't miss *Security Architecture: Securing your Web Services, Part Two* by Edward Barnard. In this part, he has advice for an effective web services security approach.

In his third feature on security, Edward Barnard gives advice on *Implementing Cryptography*. Cryptography certainly seems like some magical math stuff that helps keep our data secure, but doing it correctly can be really tricky.

Chapter 3

DeLoreans, Data, and Hacking Sites

David Stockton

In the mid to late 1980s, Robert Zemeckis, Michael J. Fox, and Christopher Lloyd (and others) created a series of movies that explored time travel, paradoxes, and how if you mess up the past and your parents don't fall in love, you may cease to exist. The second movie in the series (spoiler alert!) involves a bit where Biff, the series antagonist, retrieves a sports almanac from the future, brings it to the past, and is able to place a large series of winning bets on the outcome of various events, placing him at the top of his own empire and greatly affecting the future in an arguably negative way.

Introduction

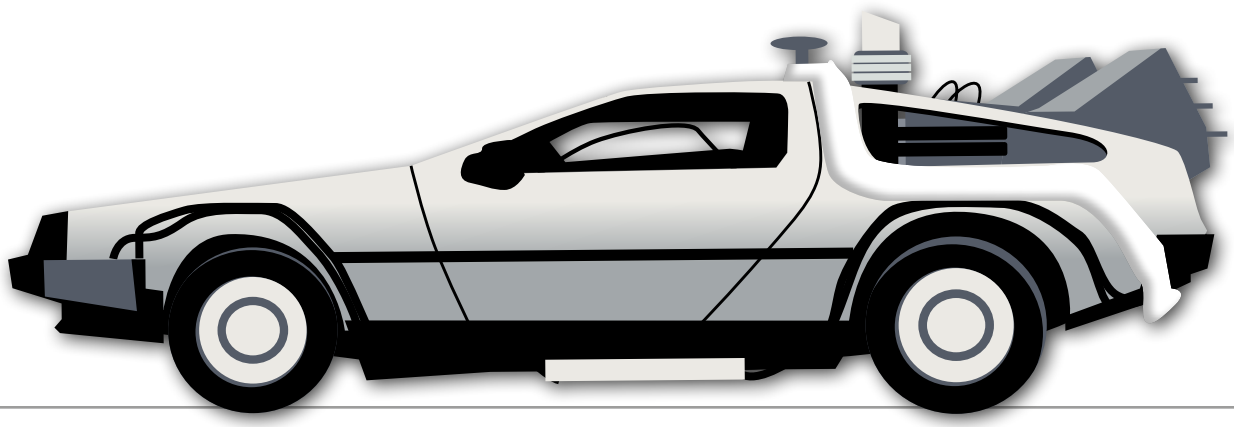
While the movies are excellent and beloved by many, they're not real. There's no actual stainless steel car that can send the occupants through time when it reaches 88 miles per hour. In the movie, Marty jumps forward to just a few weeks from now—October 21, 2015. In a few weeks, we'll know completely how accurate or ridiculous the predictions made in this movie may be. Some prognostications have come true, while others, not so much. For instance, fax machines, while they still exist, unfortunately do not play nearly as large of a role today as the movies predicted. We also still don't have a real hoverboard.

As much as I would love to revisit these movies and talk about them for hours, I do have a point—and a way to tie all this together. In Back to the Future, Part II, Biff uses his knowledge of the future to make himself a lot of money. It is not hard to imagine that if one could actually predict the future, it would not be difficult to use that knowledge for profit. However, we cannot predict the future. Mostly.

> *Prediction is difficult, especially when dealing with the future.*
>
> *- Danish Proverb*

On August 11, 2015, just a couple of weeks ago, federal authorities unsealed charges against 32 hackers and international traders who used their knowledge of the future to gain profits of over $100 million by trading stocks. Now, it's certainly not illegal to trade stocks, but it is illegal to trade stocks when you have insider information about the deals a company may make or how it will announce its performance during its annual or quarterly reviews. However, these people were not (to my knowledge) insiders in the traditional sense. They used knowledge that was released to the public via news articles to make trades that made them a lot of money. So what's the problem?

The information they were using was in the form of not-yet-released news articles, press releases, earnings statements, and more acquired by hacking the networks belonging to Marketwired and the PR Newswire Association. This gave them access to these articles before they were published, allowing them to trade based on easy-to-make predictions of what would happen to various companies' stock prices before the public knew about the news. If you have a



way of knowing with a fair amount of certainty that a stock price is going to get a good bump (or take a dive), it's not hard to make trades that will take advantage of that knowledge.

By using a series of SQL injection attacks against the servers, over three years, the hackers gained access to about 150,000 draft news articles, which they used to make informed trades. They didn't do this all at once, and they didn't act on every article that was stolen, which made it harder for authorities to figure out that something shady was going on.

What Is SQLi?

Last month, I talked briefly about SQL injection, or SQLi attacks. This month, it's all about that. Like PHP, SQL (which stands for Structured Query Language) has certain keywords that mean something to the language on their own. These include words like `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `WHERE`, and `INTO`, among others. In between these keywords, you'll find words that were supplied by a user: names of tables, fields, functions, and more. In many queries, there are also values or patterns, which are used to control or limit the records that are affected by a particular query.

The database engine that runs the queries and gives back results or changes data is able to interpret a provided instruction string into keywords, identifiers, and data in order to do what we've asked it to do. The problem comes about when the SQL engine doesn't know what the intention of a query is, and it doesn't know the difference between the instruction parts of SQL that the developer wanted to run as instruction and data provided by a user, which may be misinterpreted as instruction, rather than data.

Let's take a look at how a SQL statement might be built and how it could be susceptible to an injection attack:

```
$query = "SELECT * FROM users WHERE username = '{$_REQUEST['user']}';"
// Run the $query
```

Now suppose we've set up a page with a field called `user`, and we have well-behaved users. As long as they're not messing around with us, a field containing a standard string representing a username will come into that query, and we'll have something that looks like this:

```
$query = "SELECT * FROM users WHERE username = 'dave';"
```

That's a perfectly legitimate query, and it will give back the results the developer was expecting: If there's some user named `'dave'`, then we'll get a row back. If not, we will get back something indicating that no such row exists. It's probably worth noting here early on that most of the code examples in this article will be bad. Don't use them in your code unless you're practicing making attacks, and certainly don't let this code get into any code you're running on public servers.

So let's jump out a bit to explain `$_REQUEST` just in case anyone reading is not familiar with it. `$_REQUEST` in PHP is what's known as a *Superglobal*. It's automatically set up and populated by PHP, and it's available everywhere. The `$_REQUEST` array will be filled with values from `$_GET` (query parameters) and `$_POST` (standard form data from HTTP POST requests where the Content-Type is `application/x-www-form-urlencoded` or `multipart/form-data`. If the request comes in with some other Content-Type, then `$_POST` won't be populated, and

`$_REQUEST` will not have any of the POSTed fields. It's also possible that `$_REQUEST` could be populated by cookie values, depending on your `php.ini` setting for `request_order`. By default, it's `GP`, which stands for GET and POST. So right there, it means that it's not possible to know from `$_REQUEST` if the variable is a query parameter, a POST field, or even a cookie. Not knowing where your variables come from is not a great idea on its own, but that's a topic for another day.

Now let's revisit the SQL above. Suppose we have a user who has an apostrophe in his or her name, like O'Reilly. This user fills out the form and submits it, and our query becomes the following:

```
$query = "SELECT * FROM users WHERE username = 'O'Reilly';";
```

The string itself is okay, but when executed as a SQL statement, the SQL engine will think the query is using `Reilly` as some sort of command and it will fail to run, since the `O` is contained by the single quotes. Even though there was (likely) no malicious intent from our O'Reilly user, he or she is not likely to have a good time on the site since it will not behave well with this username. Back in the early days of PHP, a function named `addslashes` was added, and uncountable tutorials on the language recommended its use. The function replaces single quotes with `\'`, which means our code above changes slightly:

```
$username = addslashes($_REQUEST['user']);
$query = "SELECT * FROM users WHERE username = '$username';";
```

Now this is a *tiny* bit better because our O'Reilly friend will be able to use the site. The resulting query string becomes the following:

```
$query = "SELECT * FROM users WHERE username = 'O\'Reilly';";
```

This is a legitimate, runnable SQL string (in MySQL). But `addslashes` is bad, so my first suggestion is to make sure that you're not using it anywhere in your code. If you find calls to it, work toward removing them and making your code safe. The prevalence of `addslashes` and the false assumption that just escaping (that's what the backslash is doing) single quotes was good enough resulted in a misguided concept called magic quotes. Magic quotes meant that PHP would automatically escape quotes in strings if found. Because the majority of code at the time was going against MySQL and this worked well enough, it stuck around in PHP for some time. It was deprecated in PHP 5.3.0 and removed in PHP 5.4.0. While it was around—and because it's a feature that could be turned on or off in php.ini—it led to a lot of problems, which, to many PHP developers, were indicators of other developers or admins who didn't really understand what they were doing. Indicators usually were strings, which, when viewed on the site, would have single quotes prefaced by one or many backslashes. This was typically caused by a developer working on a machine that didn't use magic quotes, manually calling `addslashes`, and then uploading to a server that was configured with magic quotes. This ultimately caused the backslashes to be escaped, as well.

As PHP evolved and gained more and more support for other flavors of databases, it became clear that a one-size-fits-all solution to escaping database input would not work and was not appropriate. Instead, it's important to filter input strings and escape output strings (output into

the database, I mean) through a database-specific method, which can ensure that SQL injection is avoided. More on that in a bit.

For now, though, enough of the history lesson. Let's get back to SQLi and recommendations on how to identify and fix issues in the code. If your code contains calls that start with `mysql_`, I would highly recommend fixing it. The mysql extension has been deprecated as of PHP 5.5.0 and will be removed in PHP 7. The `mysqli` extension is recommended over the `mysql` extension. If you're using MySQL on your sites or applications, the `mysqli` extension will work. Furthermore, it will support everything you can do with MySQL 5.1+. PDO doesn't support every bit of every database functionality, just most, but I would still recommend PDO over `mysqli` (or other database-specific functions) unless your application requires some of the functionality that PDO does not support. Chances are, though, PDO will work for anything you're doing.

The advantage that PDO provides is that you'll be able to work with a number of different database engines using the exact same set of method calls. In my previous position, we had a single application that needed to fetch data from MySQL, PostgreSQL, Oracle, and Microsoft SQL Server. To use each of these with their native drivers would mean learning `mysqli_*`, `pg_*`, `oci*`, and `mssql_*` functions. By using PDO, I was able to connect and send queries into all of these databases with the same set of methods. While the presence of the SQL dialects means that the queries needed to be built slightly differently, the PHP calls were all the same.

Identifying Potential SQL Injection

The easiest way to have a SQL injection vulnerability in your code is to build your queries using string concatenation with user-provided data. By "user-provided data", I am intending to cast a wider net than you might be thinking. Of course, all the standard `$_GET`, `$_REQUEST`, `$_POST`, and `$_COOKIE` values are suspect. Additionally, I also mean any value that we've stored in the database. You might be wondering why. It's because at some point, data in the database may have been inserted through some way that would not pass any restrictions in our own code. It could be DBAs directly inserting data, loading data from files or sometime in the past when your application was not quite as secure as it may be today. So with that in mind, I mean we need to look for queries in our code that are built using PHP variables directly in the query.

In order to find potential SQL injection candidates, you'll want to search your code for any queries that you're running. This means looking for calls to functions like `mysql_query` or `mysqli_query` or even PDO methods like `query` and `execute`. Additionally, searches for SQL keywords like `SELECT`, `UPDATE`, `INSERT`, `DELETE`, and `CALL` will help find other places where queries may have been built in a different place from the code that runs them. When you've found the queries in your code, look at how they are built. Some queries may have no variable portion of the query, which means that the query never changes based on any variable. Queries like the following, for instance, are not vulnerable to SQL injection:

```
SELECT site_title FROM configuration;
```

If you find PHP variables in the SQL string, such as:

```
$query = "SELECT $field FROM $table WHERE $whereField = '$whereValue'"
```

Index

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X