THE EXPERT'S VOICE® IN WEB DEVELOPMENT

Securing PHP Apps

Ben Edmunds

Apress®

Securing PHP Apps

Ben Edmunds

Apress®

Securing PHP Apps

Ben Edmunds
Brooklyn, New York, USA

ISBN-13 (pbk): 978-1-4842-2119-8 ISBN-13 (electronic): 978-1-4842-2120-4
DOI 10.1007/978-1-4842-2120-4

Library of Congress Control Number: 2016948186

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Technical Reviewer: Massimo Nardone
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Mark Powers
Copy Editor: Mary Bearden
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com/9781484221198. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

Contents at a Glance

nafisspour@bluewin.ch

Contents

About the Author

Ben Edmunds [1] leads development teams to create cutting-edge web and mobile applications. He is an active leader, developer, and speaker in various development communities. He has been developing software professionally for over 10 years and in that time has worked on everything from robotics to government projects.

Ben is the CTO at Mindfulware, PHP Town Hall podcast co-host, CodeIgniter Framework Security Counsel member, open source advocate, human.

Ben offers security auditing and consulting on a limited basis each year. If you're interested, please get in touch. He can be reached via e-mail at `consulting@benedmunds.com`.

[1] `http://benedmunds.com`

nafisspour@bluewin.ch

About the Technical Reviewer

Massimo Nardone holds a master's of science degree in computing science from the University of Salerno, Italy. He has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years. He currently works in the Chief Information Security Office (CISO) for Cargotec Oyj. He has more than 22 years of work experience in IT including security, SCADA, cloud computing, IT infrastructure, mobile, security, and WWW technology areas for both national and international projects. He worked as a visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years. He holds four international patents (PKI, SIP, SAML, and Proxy areas).

He is the co-author of *Pro Android Games* (Apress, 2015).

nafisspour@bluewin.ch

Constructor

Several years ago I was writing a web application for a client in the CodeIgniter PHP framework, shudder, but CodeIgniter didn't include any type of authentication system built in. I, of course, did what any good/lazy developer would do and went on the hunt for a well-made library to supply authentication capabilities. To my chagrin, I discovered that there weren't any clean, concise libraries that fit my needs for authentication in CodeIgniter. Thus began my journey of creating Ion Auth, a simple authentication library for CodeIgniter, and a career-long crusade for securing web applications as well as helping other developers do the same.

Here we are years later, a lot of us have moved on to other frameworks or languages, but I still repeatedly see basic security being overlooked. So let's fix that. I want to make sure that you'll never have to live the horror of leaking user passwords, have someone inject malicious SQL into your database, or experience the suite of other "hacks" that could have been easily avoided. Let's make sure we all get home on time and sleep well at night.

The intended audience for this book is someone who knows PHP and has developed for the web before. A large breadth of knowledge is not needed, however, and this will be applicable for a junior developer through to a senior developer. This will be a framework-agnostic guide to help you learn the basics of securing web applications built into PHP and learning about the common security pitfalls that a senior developer usually acquires over years of experience. This book will be a quick read with handbook-style references to specific items you can act on. It is meant to be something you can read in a couple hours and then reference later as needed. I'll also try to make sure we have some fun in the process.

Format

All code samples in the indented blocks can be assumed to be in PHP unless otherwise noted. Line numbers are shown on coding blocks for reference.

Lines starting with a dollar sign

```
$ ls -al
```

are examples of using the command line as a normal user. Lines starting with a pound sign

```
# ls -al
```

are examples of using the command line as the root user. Server command-line examples will assume some type of *nix (centos, redhat, ubuntu, osx, etc.) operating system.

I'm trying to keep the code examples from wrapping where possible so method arguments will be on their own lines. This may seem odd but it is much easier to read than wrapped code with this book format.

Errata

If you find any errors don't hesitate to get in touch with me via e-mail.[2]

Sample Code

All of the examples are in PHP unless otherwise noted. I will use native PHP code where possible, even if it creates more boilerplate. If something requires too much work to succinctly explain in native PHP I will use the Laravel framework because it has an elegant syntax and should be easy to understand.

Some of the code examples are broken up for explanation. To view complete code examples you can reference the GitHub repository[3] or download the source code from the book's apress.com product page, located at www.apress.com/9781484221198.

Let's do this.

[2] feedback@buildsecurephpapps.com
[3] https://github.com/benedmunds/Building-Secure-PHP-Apps-Examples

CHAPTER 1

Never Trust Your Users. Sanitize ALL Input!

Let's start with a story. Mike is the system administrator 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 in computer lingo 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.

SQL Injection

SQL injection is an attack preformed against an application by having it change the SQL statements it is running. This can cause data exposure, modification, and lose of user data.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-2120-4_1) contains supplementary material, which is available to authorized users.

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

B. Edmunds, *Securing PHP Apps*, DOI 10.1007/978-1-4842-2120-4_1

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]

```
1   mysql_query('UPDATE users
2     SET first_name="' . $_POST['first_name'] . '"
3     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.

[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.

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 you can use a better method known as **parameter binding** using prepared statements. Parameter binding is definitely the way I recommend, as it offers more security. Using PHP's PDO class,[4] your code now becomes:

```
$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 and 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. I'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 messing up your day. Bound parameters are pretty awesome right? You know what else is awesome, Funyuns are awesome.

Best Practices and Other Solutions

Prepared statements and parameterized queries (as described above) are by far the best practice method of protecting against SQL injection.

Stored procedures are another way to protect against SQL injection. A stored procedure is a function built into your database. Using a stored procedure means you're less likely to be susceptible to SQL injection, since your data aren'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 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.

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

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 them in one go.

Your form might look like this:

```
1   <form  action="...">
2     <input name="first_name" />
3     <input name="last_name" />
4     <input name="email" />
5   </form>
```

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

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

Quick and easy right? But what if a malicious user modifies the form, giving themselves administrator permissions, as would happen with this code?

```
1   <form  action="...">
2     <input type="text" name="first_name" />
3     <input type="text" name="last_name" />
4     <input type="text" name="email" />
5     <input type="hidden" name="permissions" value="{\
6   'admin':'true'}" />
7   </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) is 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 support from your application code/framework completely so that parameters sent through an HTTP request are not automatically processed
- 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:

```
1   class User extends Eloquent {
2
3     protected $table = 'users';
4
5     protected $fillable = ['first_name', 'last_name'\
6   , '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:

```
1   class User extends Eloquent {
2
3     protected $table = 'users';
4
5     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 yourself 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., and it is determined by the PHP engine based on context.

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

By typecasting the value, you can verify that the data matches what you expect. In the previous example, if the ID was coming from a variable, it would make sense to typecast it if you 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 you are defining the ID yourself, so you know it's an integer. But if the ID came from a posted form or another source, this would give you additional peace of mind.

PHP supports a number of types that you can cast to. They include:

```
$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 also throughout your application. Just because PHP is dynamically typed doesn't mean that you can't enforce typing in certain places. Yeah science!

In addition to typecasting variables, you can also check their type via the `gettype()` function. PHP provides a host of convenient functions for checking types and handling variables. You can view a full reference in the PHP manual at http://php.net/manual/en/ref.var.php. An example of checking the type for the `$id` variable in the example above would look like this:

```
if (gettype($id) !== "integer") {
   //$id doesn't equal expected type
   //handle error and stop processing
   return false;
}
```

Sanitizing Output

Just as you should take care to sanitize the data coming into your application, you need to sanitize the output of your system. This protects both your user experience and the security of your users from themselves and others.

Outputting to the Browser

Not only should you take precautions when processing the data you take in from users—whether that is through web forms, API endpoints, or any other user provided data—you should also sanitize 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 it 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 or 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 `escapeshellcmd()`[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. The `escapeshellarg()` function is used to wrap arguments to ensure they are escaped correctly, and don't open your application up to manipulation of the structure of the commands.

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

CHAPTER 2

HTTPS/SSL/BCA/JWH/SHA and Other Random Letters; Some of Them Actually Matter

Once again, it's time for a little story. In October 2010, Eric Butler released a Firefox extension named Firesheep to highlight a huge problem on the Web that most people hadn't been paying enough attention to. Firesheep allowed any regular ol' user to watch the nonencrypted traffic on their local network and then hijack other users' sessions. Firesheep exploits a type of man-in-the-middle attack called sidejacking. Sound scary? It should, because it is. Maybe you're thinking, well this is conjecture. Alright fine, facts in. Let's walk through an illustration to make the point.

It's December 2010, Jane is out of town on a work trip for Achme Inc. and is staying at the Garden Inn. It just so happens to be the same hotel that John is staying at. John is in the running for a position that Jane is also trying to get. Jane recently heard about Firesheep on the news and is in a mischievous mood. She logs on to the hotel Wi-Fi and runs Firesheep. Luckily for Jane, John is using the Wi-Fi and she sees that he has an unsecured connection to their company's web e-mail portal. With one click she is now logged in to John's e-mail account. Just take a second and think of the trouble she could cause him, the private things she has access to, the general control/chaos e-mail can exert in someone's life.

This type of exploit, session hijacking via unencrypted network traffic (aka sidejacking), has always been possible by those who knew what they were doing. Now, with the release of Firesheep, this is possible by anyone who knows how to download an extension and click a button.

While you go download Firesheep (yeah, that's right, I know what you're doing!), you might be thinking that this is a horrible thing to happen. Quite the opposite actually; this has spurred web companies to finally get off of their respective laurels and take HTTPS seriously. Gmail, Facebook, and Twitter now all default to using HTTPS throughout their entire site. Previously, the standard had been to only encrypt login pages, which secured the user's login credentials but left their current session open to hijacking, as in the example above.

B. Edmunds, *Securing PHP Apps*, DOI 10.1007/978-1-4842-2120-4_2

What Is HTTPS?

I'm going to cover some of the more basic concepts of SSL/HTTPS here for those who aren't familiar with it. If you're well versed in this, feel free to skip ahead.

Normal interweb traffic is transferred over HTTP. When you type `http://www.google.com` into your browser, you're using HTTP, notice the http:// at the beginning there. Normal HTTP traffic uses port 80; HTTPS, on the other hand, uses port 443. HTTP is not secure in the least, everything you do is sent free and clear for anyone listening to see what you're doing. HTTPS is "HTTP Secure" or "HTTP on SSL"; acronym semantics can be argued, but they both mean the same thing—HTTP using SSL to secure it.

I'm only going to cover how HTTPS works at a very high level because the details won't matter to most people. If you're interested in learning more, please do; searching `google.com` is a good place to start.

A real life example to explain how SSL works is a diplomatic bag.[1] The contents are secured and can only be opened on either end of the transfer by the person with the proper credentials. The bag is secured by international law, as well as physical means, just as the SSL-encrypted message body is protected by a strong algorithm and keys.

A certificate authority will sign your web site's certificate to prove that it is valid. The user's web browser already knows the major certificate authorities and will verify the site's certificate against the root certificate that the certificate authority provides. The traffic will then be encrypted with this key on both ends, so the only traffic going across the network is encrypted traffic. If you've ever used SSH with public keys for authentication, you are already familiar with the process. You have a public and private key that is used to verify your identity with a remote server.

This will protect you from man-in-the-middle attacks, including the session hijacking I mentioned above, if all of your site is encrypted with HTTPS.

Limitations

There are a few limitations when using HTTPS that may make it infeasible in certain circumstances.

Virtual Hosts

Under normal configurations, virtual hosts cannot be used with SSL. This is a problem if you're using shared hosting or simply running multiple sites on the same server. The reason for this is because the server can't determine the host header until the connection has been completed, which requires the SSL authentication. Because certificates can only have one host, this means it will simply not work. The easiest way around this is to set up multiple IP addresses and use IP-based hosts instead of the name-based host resolution you're probably used to. I usually recommend setting up a separate server for secure sites though; if you need HTTPS, you are probably at the point of needing a dedicated server as well.

[1] `http://en.wikipedia.org/wiki/Diplomatic_bag`.

There are, however, some hosting providers with shared certificates that can be used across the sites hosted with them. This can enable you to quickly and inexpensively support HTTPS. The main issue with this is that the domain would need to reflect the hosting provider's domain name. For example instead of

`https://yourApp.com/login`

the URL would be something like

`https://yourHost.com/yourApp/login`

This may or may not be a concern depending on your application and branding needs.

Speed

HTTPS connections require SSL handshakes to establish the connection, thus making the overall transfer slower. Once that initial handshake is performed, additional connections only require the encryption and decryption of the content, meaning that once the initial connection is complete, subsequent connections aren't much slower. The performance impact is *incredibly* low though, but this is not a valid reason to discredit the use of HTTPS.

Caching

Cheddar. Fat stacks. Dead Presidents. Cash money. Nah, actually we're talking about cache, the secret sauce behind your super quick load times. But you have to say it with a British accent. Modern browsers will cache HTTPS content the same as HTTP content, so there is no disconnect there. To cause your older browser to support caching, set the Cache-Control header, for example:

```
header('Cache-Control: max-age=31536000');
```

This would tell the browser to cache for one year.

The real issue comes with proxy caching. Proxy caching might come from an ISP or a service meant to speed up connections. This is mostly used in rural parts of the world that have slow Internet connection speeds. Using HTTPS, this type of caching is impossible because all the traffic the proxy sees is encrypted. This is not a major issue for most sites, but if you have a large global user base or an application that targets users in remote locations, this should be considered carefully.

There is one other thing you should think about. There is a good chance that there are parts of your site that should NOT be cached. This means that you shouldn't just let the browser cache everything. You need to sit down and plan out which parts of your application should be cached and for how long. For example, CSS and JavaScript should probably be cached for a significant amount of time; whereas the user's timeline view should update very often.

Certificate Types

There are two types of SSL certificates.

Domain Validated Certificates do not verify as much information as their counterparts, but they are substantially less expensive. Usually starting around $50 per year, they will likely be the best option for small sites. The main down side from a user perspective is that there is usually some distinction in the browser between the two, for example, a Domain Validated Certificate might only show a lock symbol in the address bar, while an Extended Validation Certificate will show the full green address bar.

Extended Validation Certificates are the gold standard of SSL certificates. They not only validate that you are the owner of the domain, but they also verify the identity and legitimacy of the domain owner. Because this usually requires a personal effort on the part of the certificate authority, these certificates are significantly more expensive. Usually Extended Validation Certificates start at around $500 per year. This will be the certificate of choice for most large and reputable companies. Browsers will display the full green address bar when an Extended Validation Certificate is in use, giving users more peace of mind.

When to Use HTTPS

The traditional view has been to use HTTPS anywhere credentials or other sensitive data is passed to the server. For many years this has meant that login pages and shopping carts were all that was encrypted. These are still valid and necessary places to use encryption, but this will leave the rest of the user's session open to man-in-the-middle attacks. Recently, there has been a movement to use HTTPS everywhere, which is just a marketed way of stating that every page of your site would be encrypted on HTTPS. This is a good rule in many cases, but the limitations of HTTPS should be considered, and don't just blindly implement HTTPS everywhere without evaluating the trade-offs. If you determine that the limitations discussed above are offset by the enhanced security throughout for your specific application, then using HTTPS on each page is strongly recommended.

Are you thinking that at this point it would be easier to just forget about this whole HTTPS thing? Okay. Okay. Let's just slow down. Slow down. Regardless of your constraints, you have an obligation to your users to implement the best security you possibly can. If you run a shopping cart or collect credit card information, for instance, HTTP is not even an option. More and more even for what isn't considered sensitive data, like a social media account, it is becoming standard to encrypt. Don't be left behind; use HTTPS whenever you can.

Implementing HTTPS

What Kind of SSL Certificate Do I Need?

The main question to ask yourself is do you need to secure subdomains or not? If you need to secure multiple subdomains, such as:

```
api.yourApp.com
docs.yourApp.com
```

```
yourMom.yourApp.com
cart.yourApp.com
```

then you'll need a Wildcard SSL Certificate. If you don't need that capability and only need to secure something like

```
yourApp.com
```

then a standard certificate will work just fine. The only deterrent to getting the Wildcard just in case you need it later is the cost.

Generating Your Server Certificate

In order for the certificate authority to generate your certificate, you'll need to generate keys on your server and then upload those to the certificate authority.

This will require OpenSSL. If you don't have it on your server, you'll need to install it. Installing applications across various server operating systems and distributions is out of the scope of this book, hopefully if your at the point of needing to set up HTTPS, you know your way around your server well. If you don't know your server operating system or distribution well, it might be a good idea to hire someone to help you set up SSL certificates.

First, create a directory to store your keys. People have differing opinions on the best place to store these, but for this example, let's stick with

```
/usr/bin/ssl/
```

Now let's generate the private RSA key with the code:

```
$ openssl genrsa -out yourApp.key 1024
```

Then you will generate the CSR using the RSA key:

```
$ openssl req -new -key yourApp.key -out yourApp.c\ sr
```

You'll now be asked several questions about smart defaults. The main one to pay attention to is "Common Name," which should match your domain name, for example, "yourApp.com".

Now you should have two new files:

```
/usr/bin/ssl/yourApp.key
/usr/bin/ssl/yourApp.csr
```

Before you do anything else, make a backup copy of the `.key` file somewhere. Seriously, make two backup copies. If you lose the private key, you'll need to buy a new certificate, and servers crash all the time.

Obtaining an SSL Certificate

The first step to getting up and running on HTTPS is to obtain a certificate. There are inexpensive/free certificates available from some certificate authorities, but many of them won't come preinstalled on the popular web browsers. So that makes them useless for external-facing sites. If you're running an internal application, then less expensive alternatives and self-signed certificates are valid options. For everyone else, you may need to purchase a certificate. On the bright side, Let's Encrypt recently launched, which is a free certificate authority.

First, I recommend checking with your DNS provider to see if they offer any type of discounted or easy to set up certificates. For example, DNSimple is the DNS provider I use, and they offer subscription payments for certificates at a large discount.

If your DNS provider does not provide certificates, Symantec/VeriSign is a well-respected certificate authority.

Now go get one.

You'll then need to walk through whatever process your chosen certificate authority has in place for setting up your certificate. Usually you'll just upload your server certificate (yourApp.csr) and they will e-mail you the signed certificate. It's worth noting here that Let's Encrypt offers an automated process for set up as well.

Your certificate authority will provide you with the signed certificate, here we will call it `yourAppSigned.crt`. Copy this to your server; for this example, I'll use the following path:

```
/usr/bin/ssl/yourAppSigned.crt
```

Verifying a Certificate

There are many web sites and tools for verifying SSL certificates. DigiCert (`https://www.digicert.com/`) provides a quick and easy interface where you enter your web site address and it returns information about your certificate. Most browsers support similar functionality as well by visiting the site and then clicking the green lock symbol to drill down into the certificate information.

Apache Set Up

If you're using Apache, follow the steps below. If you're using a different web server, skip this section and keep reading. Open your `httpd.conf` file in your favorite text editor. Note, some Linux distributions may use separate config files for HTTPS. For example, my laptop running OSX uses a `httpd-ssl.conf` file.

Add a VirtualHost similar to the following. It will likely closely match your existing VirtualHost for your HTTP site:

```
1   <VirtualHost *:443>
2     DocumentRoot "/path/to/your/app/htdocs"
3     ServerName yourApp.com
4     SSLEngine on
5     SSLCertificateFile /usr/bin/ssl/yourAppSigned.crt
```

```
6     SSLCertificateKeyFile /usr/bin/ssl/yourApp.key
7   </VirtualHost>
```

Now restart Apache:

```
$ apachectrl restart
```

or use:

```
$ service apache restart
```

which will usually do the trick.

Try your site out with https://yourApp.com. You should be good to go!

NGINX Set Up

If you're using NGINX, use the following steps. If you're using a different web server, you'll need to research how to set this up with your server, sorry!

Open your NGINX virtual hosts file in your favorite text editor. Add a virtual host similar to the following, it should closely match your existing site set up:

```
1   server {
2
3     listen   443;
4
5     server_name yourApp.com;
6     location / {
7       root   /path/to/your/app/htdocs;
8       index  index.php;
9     }
10
11    ssl                                  on;
12    ssl_certificate           /usr/bin/ssl/yourAppSigned.crt
13    ssl_certificate_key    /usr/bin/ssl/yourApp.key
14
15  }
```

Now restart NGINX:

```
$ sudo /etc/init.d/nginx restart
```

or use:

```
$ service nginx restart
```

which will usually handle it.

Try your site out with https://yourApp.com. It should be ready!

Additional Resources

For Apache, the best source is the docs:

http://httpd.apache.org/docs/current/ssl/ssl_howto\.html

For NGINX the WIKI is a great starting place:

http://wiki.nginx.org/HttpSslModule

For anything else just replace yourWebServerName in the text below with the name of the software you are using to serve web pages, then paste the full URL into your web browser:

http://lmgtfy.com/?q=yourWebServerName+SSL+certifi\cate+setup

Paths

Base Path

You should ensure that users are on the HTTPS version of your site whenever it is needed. This can be done in your web server config using redirects. Another simpler option is to set the base path of your application to use your HTTPS URL, for example, `https://yourApp.com`, and force a redirect using the base path if a user comes in on HTTP.

Quite often you will want to allow HTTP on certain pages and require HTTPS on others. This is where your web server configs and proper routing in your code come in.

Relative Paths

There is one more thing to mention that isn't necessarily security related but will make your life a lot easier when using both HTTP and HTTPS on one site. URLs for assets, for example, CSS or JS, can begin with double forward slashes instead of http:// or https:// to reference the current protocol. For example, on your home page you might have:

```
<link type="text/css" rel="stylesheet" href="//ass\ ets/main.css" />
```

Navigating to `https://yourApp.com` would cause this to load:

https://yourApp.com/assets/main.css

whereas navigating to `http://yourApp.com` would load:

http://yourApp.com/assets/main.css

That's just a little trick to make your life a little easier, because I care.

Done

You are done, good job! Pat yourself on the back, mix your favorite drink, and take a well deserved nap.

CHAPTER 3

Password Encryption and Storage for Everyone

You should know how this works by now—first, a story. Chris is a junior developer working for Marvel Comics[1] web team. It's an abnormally hot summer in Burbank. He has just been tasked with building the login functionality for the new web/tablet comic portal his team is building. His "team" really means Chris and the other developer. Chris might have forgotten to wear deodorant today, why is it so hot?

Chris plans out how the login system will work. It'll have the normal things you would expect, login/logout/forgot password/etc. Regarding passwords, he'll need to store the user's password, compare it on log in, and then e-mail it back to the user if they forget it. Minutes pass. As he thinks through each part of the login process, he starts to worry about the security implications of having users' passwords available to read by anyone who has, or gains, access to the database. He knows he should encrypt the passwords, but what about decrypting for log in? Or when a user forgets their password?

After researching for an excruciatingly boring 45 minutes, Chris decides that he needs to use PHP's built in `mcrypt_- encrypt()` and `mcrypt_decrypt()` methods. Chris is pretty stoked; secure encrypted passwords and all the dirty work on encrypting and decrypting is done by PHP. He's going to be done with this project in half the time quoted.

That, my friends (we're friends right?), is why when you reset your password on `Marvel.com` you get your plain-text password sent to you in an e-mail. AN E-MAIL. WITH YOUR PASSWORD IN IT. This is why you should go change your `Marvel.com` password to something you have never used anywhere else right now. Go ahead, I'll wait. I'll just be sobbing in the corner thinking about all of the people who will be exploited by this soon enough.

The moral of the story, don't store passwords. Store one-way hashes of users' passwords. Now let's walk through how to do this right.

The Small Print

I am not a cryptographic expert. This is my personal advice based on experience. These are opinionated web development best practices and are not meant to be used as directions for securely storing nuclear launch codes. Your most important tweets will be kept safe though.

[1]This is fiction built from truth. Please don't sue me Marvel.

B. Edmunds, *Securing PHP Apps*, DOI 10.1007/978-1-4842-2120-4_3

What Is a Hash?

First, I need to cover the basics. Hashing is not encryption; passwords should be one-way hashed. This means that it is impossible to decrypt, hence the one-way part of that. There is never a need to display a password back to a user/admin/mother/anyone ever. Once a password is entered, it becomes something totally different, a hash that can be re-created only by the original password being given as input.

Popular Attacks

Before discussing this any further, let's delve into the popular attacks against hashing algorithms.

Lookup Tables

A lookup table is simply a table of hashes where the password is known. It can be as simple as this:

```
password | hash
-----------------------
pass1    | bidfb2enkjnf
pass2    | psdfnojn3nod
etc...
```

This is then compared against the password hashes in your database to determine the password that was used. This attack is useless if you are using random salts, but it's easier than robbing a train if the hashes aren't salted.

Rainbow Tables

A rainbow table is technically sophisticated compared to a lookup table yet very similar. It is basically a less memory-intensive way of achieving a lookup table through mathematic means. You can think of them interchangeably as I mention them here. Rainbow tables are also thwarted through the use of random salts, so they are less relevant with modern hashing algorithms.

A rainbow table is a very complex exploit that is really out of the scope of this book to explain. If you want to learn more, you can read the original paper published by Martin Hellman[2] that introduces the concept. The Wikipedia article on Rainbow Tables[3] is also a good source for an easier-to-digest explanation.

Collision Attacks

A collision attack is the main security flaw found in most hashing algorithms. There are two types of collision attacks.

[2] http://www-ee.stanford.edu/~hellman/publications/36.pdf.
[3] http://en.wikipedia.org/wiki/Rainbow_table.

A classical collision attack is when two different values generate the same hash. A simple example is:

```
string1 = 'abc123'
string2 = 'bcd234'

hash(string1) === hash(string2)
```

A chosen-prefix collision attack is when different prefixes are used to cause two separate values to generate the same hash.

Here is an example of this:

```
prefix1 = 'zxy'
prefix2 = 'abc'

string1 = 'abc123'
string2 = 'bcd234'

hash(prefix1 + string1) === hash(prefix2 + string2)
```

A Pinch of Salt

That's not enough though. The next problems to look out for are lookup table and rainbow table exploits. Both of these exploits basically keep a list of popular passwords and their resulting hashes. Rainbow tables are a little more complex, but that's basically what's going on. How do we combat this? Salts. RANDOM SALTS.

A salt is something that is appended to the password hash to make it unique. Salt is also something that is added to the rim of a margarita to make it delicious. Ah margaritas. Anyway. So you take a random string (salt) and combine it with the plain-text password string to give you a unique value. This means that even with a lookup table of known password hashes an attacker can't match up your user's password hash with the database password hashes since a random salt has been used. Given two identical passwords, the resulting hashes will be unique. A random salt is one of the most important pieces of your password security.

Random Isn't Always Random

You're salt needs to be random to be effective. Random !== random though. You don't necessarily need truly random numbers but `rand()` won't cut it either.

The problem with using PHP's built-in `rand()` and `mt_rand()` functions is that they are seeded with data that can be manipulated and determined; they don't provide enough "randomness." The main ingredient needed to produce a truly random number is to include enough entropy into the source. Entropy is the amount of truly random information collected by the system that is generating your random number. Most noncryptographic random number generators, like `rand()` and `mt_rand()`, use algorithms to produce their numbers without enough outside sources of data to make them truly unique. This means that the data `rand()` produces can be manipulated and guessed by an attacker. After observing enough of the output of `rand()` an attacker can

reliably determine the future output. In fact after returning as few as 624 values from `rand()`,[4] it is possible to calculate all future values.

You have been warned, you SHOULD NOT use `rand()` for your salts. By buying this book you are hereby contractually obligated to not whine when I personally come beat you in the face with a first-generation iPad if you use `rand()` for your salts. That thing was solid as a brick.

Calling your server's `/dev/random` is your best bet for true random on most systems. The issue with using this for log in is that it blocks execution while collecting entropy from the system. Collecting entropy means that it will collect environmental data from your system, such as various hardware data, keyboard typing, mouse movements, disk access, etc., in order to generate a buffer of random bits. This means that it can take a long time to return, especially on servers that aren't busy. I actually ran into this problem recently. A developer on my team used `/dev/random` to generate an activation code. Everything seemed fine, it passed testing fine. Then after we deployed the project to its own production server, requests ended up taking over 60 seconds. The server wasn't busy because it was only hosting this one project, causing `/dev/random` to block for a good bit of time while it collected entropy. What was the solution?

```
/dev/urandom
```

The pseudorandom number generator `/dev/urandom` isn't considered true random, but it is cryptographically secure. This means that it might not be a truly random number, but it is regarded as secure enough for use in salts. It will return a very good pseudorandom number immediately with no blocking; and it uses the existing entropy pool to generate a pseudorandom number that is secure enough for the majority of authentication systems. If you're writing the login page for nuclear launch codes, it might be best to make the user wait on `/dev/random`, but for that social picture sharing site, `/dev/urandom` is good enough.

Hashing Algorithms

Let's discuss the popular hashing algorithms.

MD5

I see the MD5 hashing algorithm used incorrectly more often than anything else, usually because it is supported by most databases by default. MD5 has been mathematically proven for some time now to be insecure. The issue with MD5 is that it is trivially easy to produce collisions on modern hardware.

One of the most notable examples, in 2005 researchers were able to generate collisions in MD5 checksums using a laptop.[5] The significance of that is that it doesn't take a $200k beast of a server to break MD5, just any old laptop, and that was in 2005. In 2005 people, that was like 100 Internet years ago. No more MD5 for password hashing, please. Nonsecure hashes to verify data contents, sure. Just not for secure hashes that an attacker would be interested in breaking.

[4] http://eprint.iacr.org/2005/165.pdf.
[5] http://cryptography.hyperlink.cz/md5/MD5_collisions.pdf.

MD5 is not completely broken because it is still mostly secure when used with a proper salt. That doesn't mean that you shouldn't move on to a more future-proven solution though.

SHA-1

Ah good old SHA-1, trusty and secure for years. Those are IRL years too, in Internet years that's decades. In 2005 (2005 was a bad year for security), researchers from Shandong University released a research paper[6] proving that SHA-1 collisions could be reliably generated with less than 269 hash operations. Collisions at around 280 hashing operations are considered safe cryptographically. FYI 280 is about 1.20892x1024, so "cryptographically secure" means pretty darn secure. Since then, Moore's law[7] has ensured that SHA-1 is even less secure and should be avoided in any application that needs true security.

As I explained above for MD5, when used with a random salt, SHA-1 is still algorithmically secure.

SHA-256/SHA-512

The SHA-2 standard was introduced as a successor for SHA-1 in 2001. It's accepted and was accelerated a bit when SHA-1 was proven to be insecure in 2005. SHA-256 and SHA-512 are basically the same; SHA-256 uses 32-bit words, SHA-512 uses 64-bit words. They also have a different number of rounds. The core algorithm is practically identical though.

SHA-2 is currently considered cryptographically secure with no known vulnerabilities when used with a sufficient number of rounds (>64).

It has not seen the same amount of scrutiny as Blowfish though, the cypher that BCrypt uses internally. The Blowfish cypher has been around since 1991 and is still considered secure, yet using it with a weak key is a known weakness. Being based on a cypher gives BCrypt an additional layer of cost that makes it superior to a standard hashing algorithm, in other words, BCrypt is slower by design. Slow is a good thing here!

Even though I recommend BCrypt, SHA-256 and SHA-512 are currently valid and secure options for secure hashing when used as part of a derivation algorithm, like PBKDF2 or the algorithm implemented with PHP's `crypt()` function.

BCrypt

BCrypt is viewed by many as a newcomer and isn't as widely known. BCrypt was released in 1999, so it's not exactly new here. BCrypt is a key derivation function based on the Blowfish cypher, and it is iterative so it protects against brute forcing due to the cost associated with generating a hash.

There are currently no published exploits of BCrypt despite the fact that it has seen considerable attention from cryptographic researchers. It has also been around for a good length of time, so as of this writing, BCrypt is considered cryptographically secure.

---

[6] http://link.springer.com/chapter/10.1007/11535218_2#page-1.
[7] http://en.wikipedia.org/wiki/Moore's_law.

BCrypt does have a limitation of 72 characters for the plaintext password being encrypted. This is usually taken into consideration by either stripping the excess characters or by simply validating 72 as the maximum length.

BCrypt will be our choice for passwords in the following examples.

SCrypt

SCrypt is the new kid on the block. Released in 2012, it is a memory-intensive key derivation function. Theoretically, SCrypt is more secure than BCrypt due to the high cost inherent in the algorithm, but because it is so new, I don't personally recommend its use at this time.

New is a bad thing in the cryptographic world. It means that SCrypt hasn't received the same level of attacks and scrutiny as older algorithms. There have been a few exploits reported recently, which doesn't mean SCrypt isn't secure, but this does take away from the security advantage it theoretically has over BCrypt.

One big thing that SCrypt has going for it is that a few popular crypto-currencies are using it for their mining operations, most notably Litecoin and Dogecoin. This means that it will likely receive a large amount of attention sooner than its predecessors.

Storage

This section will be short, WAKE UP. In whatever system you store your password hashes, whether it's in a relational database, key store, lock box, sock drawer, or file system, use either an unlimited length text field or I recommend using a varchar (255). Your hashing algorithm will produce a maximum length string, so you don't have to worry about an attack overloading your database. Different hashing algorithms will produce different fixed length strings, so you could set your field length based on your algorithms. I instead prefer to use a larger-than-needed field length constraint to handle future hashing possibilities rather than try to save a few bytes.

Using BCrypt, your hash will always have a maximum character length of 60 characters. So in theory you could get away with a varchar(60) field in your database, but this doesn't account for future changes. It's better to future-proof your passwords now than to try to save a few bytes in your database. So just leave it as a text field or a varchar (255), you won't regret it.

Validation

The only validation that's needed on a password field is minimum length. You should allow any character, whitespace, phrases, etc. so your users can construct as complex of a password as they want. Pass-phrases should be endorsed, "correct horse battery staple" is a much better password than "myNewPassword."[8] Your only worry is that the password is not complex enough, hence the minimum length. For the love of all that is good, like cat

[8] http://xkcd.com/936/.

gifs, don't do stupid things like using JavaScript to restrict copy-paste. If the user wants to use a password management tool, do all you can to make that easy for them. If you do stupid things, you'll make your users, and the cats, very sad. Okay, that rant is over now.

The only caveat to this is that with BCrypt only the first 72 characters of the password will be used, so technically you could limit it to a maximum of 72 characters and not lose any data. That does put a limit on your users though, and it is not future-proof for your next hashing algorithm. If your user has a 74-character pass-phrase memorized, it's best to let them use that and only use the first 72 characters rather than make them think up a new pass-phrase. Some sources recommend hashing the passwords using a standard hashing algorithm (SHA-256, SHA-512, etc.) and then BCrypting the resulting hash to account for this length issue. That is a perfectly valid option. I'm not going to recommend it here simply because with a valid salt plus 72 characters of the password you will have enough data to keep your hashes secure according to current research.

Putting It All Together

Now that I've covered the basics, let's write a little code, finally.

First, I'm going to walk you through the traditional/deprecated way of doing this, and then I'll walk you through the newer way that is available with PHP 5.5 and up. The reasons for this are:

1. I want you to understand what is going on behind the scene instead of just seeing wrapper functions.
2. There is a decent chance that you're on an older version of PHP so I don't want you to be left behind.

The scenario here is that you are registering a new user. You will need to generate a random salt, generate their password hash, then store both of these to an imaginary database for authentication in the future.

Versions Older Than PHP 5.5

If you are using a PHP version less than 5.3.7, you need to upgrade to at least 5.3.7 to have a decent level of security. There is really no other sound recommendation I feel comfortable giving. If you are on an older version, there are many bug fixes that are patched with upgrades. Specifically to this case example, there was a BCrypt vulnerability patched in 5.3.7. I will be using the $2y$ prefix in this example, this is the "always to specification" prefix, meaning it has been updated with the vulnerability fix and is the most up-to-date logic.

First, let's generate a unique random salt. There are a few ways to do this in PHP depending on what extensions are compiled on your system. You can most likely just do this:

```
1   //generate the binary random salt
2   $saltLength = 22;
3   $binarySalt = mcrypt_create_iv(
4     $saltLength,
5     MCRYPT_DEV_URANDOM
```

```
6   );
7
8   //convert the binary salt into a safe string
9   $salt = substr(
10    strtr(
11      base64_encode($binarySalt),
12      '+',
13      '.'
14    ),
15    0,
16    $saltLength
17  );
```

You are calling `mcrypt_create_iv()` with the `MCRYPT_DEV_URANDOM` flag to tell it to buffer from `/dev/urandom`. Wrapping it in `bin2hex()` causes you to get a hexadecimal string back instead of the straight binary data.

If MCrypt isn't available on your system, you can always fall back to reading directly from `/dev/urandom`. Unless you are on Windows, in which case just install Linux before it's too late. Go ahead, I'll wait. You can thank me later; I'll take check, cash, or credit card for the thank you gift.

```
1   //generate the binary random salt
2   $saltLength = 22;
3   $binarySalt = file_get_contents(
4     '/dev/urandom',
5     false,
6     null,
7     0,
8     $saltLength
9   );
10
11  //convert the binary salt into a safe string
12  $salt = substr(
13    strtr(
14      base64_encode($binarySalt),
15    '+', '.'
16  ),
17  0,
18  $saltLength
19  );
```

Now that you have the salt, let's hash the password as well:

```
1   //generate the binary random salt
2   $saltLength = 22;
3   $binarySalt = mcrypt_create_iv(
4     $saltLength,
5     MCRYPT_DEV_URANDOM
```

```
);

//convert the binary salt into a safe string
$salt = substr(
  strtr(
    base64_encode($binarySalt),
    '+',
    '.'
  ),
  0,
  $saltLength
);

//set the cost of the bcrypt hashing
//remember to experiment on your server to find th\
e right value
$cost = 10;

//now we'll combine the algorithm code ($2y$) with\
  the cost and our salt
$bcryptSalt = '$2y$' . $cost . '$' . $salt;

//hash it, hash it good
$passwordHash = crypt(
  $_POST['password'],
  $bcryptSalt
);

//verify a secure hash was returned
//this could be an error code or insecure hash
if (strlen($passwordHash) === 60) {

  //this next part is just for demonstration
  $db = new  ImaginaryDatabase;
  $db->user()->create(array(
    'password' => $passwordHash
  ));

}
else  {

  //error   handling

}
```

So you've hashed the password and saved it to the database. Notice that you didn't save the salt? That's because `crypt()` will store the selected algorithm, hashed password, and salt all within the returned hash.

Well you have the user signed up now, so let's say they leave the site (how dare they!) and come back later. They will need a way to log in. Log in is nice and simple, you just need to check the plaintext password against the hash stored in the database:

```
1   //we are going to start with a default state
2   //of failure, always assume failure first
3   $valid = FALSE;
4
5   //grab the hash from our imaginary database
6   $user = $db->user()->where(array(
7     'email' => $_POST['email']
8   ))->row();
9
10  //now check to see if the login password matches
11  $pass = $_POST['password'];
12  if (crypt($pass, $user->pass) === TRUE) {
13    $valid = TRUE;
14  }
15
16  //other checks, error handling, etc...
17
18  if ($valid === TRUE) {
19    //valid auth stuff
20  }
```

If you pass a previously generated hash as the second parameter to the `crypt()` function, it will use that algorithm and salt to generate a new hash that you can then compare against the previously stored password hash.

Version PHP 5.5 or Higher

New password hashing functions were introduced in PHP 5.5[9] to greatly simplify the process of handling passwords. The purpose of this is to hide a lot of the complexities and make PHP users secure using the default functions built into PHP without relying on developers to know what they are doing. In other words, they are trying to steal my book content, jerks!

In all seriousness though, use the PHP password functions whenever possible. They provide built-in, up-to-date hashing with additional safety checks that I'm not even covering here, like protection against timing attacks.[10]

I'll walk you through the same exact steps here using PHP 5.5 syntax. You'll be using the `password_hash` function to generate the hashed password. This function will automatically create a random salt so you can skip that step completely. Let's just jump straight into it:

```
1   //FYI - the default cost is 10, it can be customiz\
2   ed though
3
```

[9] http://www.php.net/manual/en/book.password.php.
[10] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.65.9811.

```
4   //hash it, hash it good
5   $passwordHash = password_hash($_POST['password']);
6
7
8   //this next part is just for demonstration
9   $db = new  ImaginaryDatabase;
10  $db->user()->create(array(
11    'password' => $passwordHash
12  ));
```

Then to verify the user's password at log in, you will use:

```
1   //we are going to start with a default state of fa\
2   ilure
3   //always assume failure first
4   $valid = FALSE;
5
6   //grab the password hash from our imaginary databa\
7   se
8   $user = $db->user()-where([
9     'email' => $_POST['email']
10  ])->row();
11
12  //now check to see if the login password matches
13  $pass = $_POST['password'];
14  if (password_verify($pass, $user->pass) === TRUE) {
15    $valid = TRUE;
16  }
17
18  //other checks, error handling, etc...
19
20  if ($valid === TRUE) {
21    //valid auth stuff
22  }
```

As you can see, the new password functions make your life a lot easier in PHP 5.5. Another advantage to this functionality being baked into PHP is that you can rely on it remaining up to date in the future, instead of having to stay up to date on the latest password algorithms yourself. Even though I know this has been riveting, leave it to the pros like the PHP core team when you can, that way you have more time to make cool stuff.

When you implement your authentication scheme, make sure to reference the PHP docs to ensure your code is up to date. Also, be sure to take advantage of the other password functions available like `password_get_info()` and `password_needs_rehash()`.

Are you on PHP 5.3 still? Missing out on all the fun these users get using that new fangled 5.5? Not to worry, Anthony Ferrara has you covered. His `password_compat` library implements the new password functions in PHP 5.3.7 and up just for you. You can download it from Anthony's GitHub.[11] I **HIGHLY** recommend using this when possible instead of writing the code yourself. It has been tested so you can rely on it.

Brute Force Protection

You can have the best password hashes out there, like totally secure dude, but that doesn't do you any good if someone just hammers away at your login page until they find the correct password. Brute forcing is the process of someone using software to repeatedly try different passwords until they get in.

Securing yourself from this type of attack is pretty easy, sadly I see so many sites that are vulnerable to this. So how do you secure yourself from it? Just make it take longer than is feasible for someone to find a password this way.

So someone tries to log in, fail. They try again, fail. One more, fail. Now you make them wait 60 seconds before trying again. Done. That's really the simplest, yet a still super effective method of preventing brute force attacks. If an attacker can only try three passwords per minute, then it's going to take them forever to find the user's password, so they move on. Your users are safe. You don't have to limit it at three exactly, three or five or ten are pretty popular numbers; personally I wouldn't go over ten per minute.

Next up you can expand on that a bit: block based on IP. Only allow X number of login attempts from a certain IP across all users instead of just each individual user. Just be sure to include good error messages and reasonable access reinstatement times; corporate users often share the same public IP address, so you don't want to punish thousands of valid users due to one bad user if you can help it.

Also note that the same logic applies to your API authentication. Don't leave an opening to attackers to simply take a different path to get the same result. If you secure something on the front end of your application and then expose that same functionality through an API, make sure to secure it just as well there.

Upgrading Legacy Systems

Now to the elephpant in the room. Get it? ElePHPant. Yeah, it's been a long night, sorry. So how to you upgrade your existing system that has MD5 passwords with no salt?

I'm going to give you two options:

- Path 1: As each user logs in, you will silently upgrade their hash to use BCrypt. They won't even know the difference. Soon enough you will have a database of well-secured passwords.
- Path 2: Use BCrypt to hash the existing MD5 hashes that are in the database. New passwords will be hashed with MD5 first and then BCrypt.

[11] https://github.com/ircmaxell/password_compat/.

Upgrade Path 1

Path 1 is the traditional advice for migrating to new authentication schemes and is by far the best option in the majority of circumstances. To implement this, you would do something similar to this:

```
1   $valid = FALSE;
2
3   //grab the password hash from our imaginary databa\
4   se
5   $user = $db->user()-where([
6     'email' => $_POST['email']
7   ])->row();
8
9   if ($user->pass_hash_algo === 'bcrypt') {
10    //they have previously logged in and
11    //upgraded  their  hash  so  they're  good
12    //proceed with verifying login as usual
13  }
14  else  {
15    //this is modified from our old hash check
16    $oldHash = md5($_POST['password']);
17
18    if ($oldHash === $user->pass) {
19      //generate the new hash with the plaintext
20      //password they just gave us
21      $newHash = password_hash($_POST['password']);
22
23      //save the new hash and flag to our database
24      $user->pass = $newHash;
25      $user->pass_hash_upgraded = 'bcrypt';
26      $user->save();
27
28      $valid = TRUE;
29    }
30
31    //other checks, error handling, etc...
32  }
```

With this modified login system in place, your users will upgrade to the new hash automatically as they log in. This upgrade path can easily be modified throughout the years as the recommended hashing algorithms change. In a few years, SCrypt could be the accepted standard and it would be trivial to upgrade users to that.

Upgrade Path 2

Path 2 isn't as clean of a solution but is immediately secure. Where Path 1 retains the insecure hashes until each user logs in next, this will secure your data right now so you don't have to lose any sleep tonight. With this path you will immediately rehash all current password hashes using BCrypt. Your new hashes are just BCrypt hashes of MD5 hashes of the passwords. This complicates the process since you will need to use both BCrypt and MD5 basically forever. This could become cumbersome years from now when you're SCrypting BCrypts of SHA1'd MD5 hashes.

This is a two-step process. First, write a script to do basically the following and run it against your current database. Please tell me you're doing this with a database dump and not on the prod server. Right? Cause that would make me sad. Also you should add some type of status to each record while you do this. Basically, just use this as an example, no copy-pasting mmkay.

```
$users = $db->users()->result();

foreach ($users as $user) {

  $newHash = password_hash($user->pass);

  $user->pass = $newHash;
  $user->save();

}
```

Now that your database has been updated, change your registration method to do this:

```
//hash it with MD5  then BCrypt
$passHash = password_hash(md5($_POST['password']));

//this next part is just for demonstration
$db = new  ImaginaryDatabase;
$db->user()->create([
  'password' => $passHash
]);
```

The login method will need to check against an MD5 hashed password instead of the plaintext as well:

```
//grab the hash from our imaginary database
$user = $db->user()-where([
  'email' => $_POST['email']
])->row();

//md5  hash  the  password  before  we  bcrypt  it
$md5Pass = md5($_POST['password']);

```

```
9   //now check to see if the login password matches
10  if(password_verify($md5Pass,$user->pass)===TRU\
11  E) {
12    $valid = TRUE;
13  }
14
15  //other checks, error handling, etc...
```

It's Over. We're Safe

That's all folks. There's a ton more on this subject to cover, as always, but this is the need-to-know info. Keep the pages turning for the next chapter! We'll cover restricting URLs, safely handling files, and ensuring that no one sees data they shouldn't.

Resources

If all of this talk about cryptography has interested you, I strongly suggest you check out Bruce Schneider's work. You should definitely read his book, *Applied Cryptography*,[12] it is the bible of crypto.

I would also like to bring attention to the work that a fellow member of the PHP community has been doing. Anthony Ferrara[13] championed the password_hash functionality integrated into the PHP core starting with version 5.5. This was a huge step toward ensuring we have secure PHP applications by default in the future.

[12] https://www.schneier.com/book-applied.html.
[13] http://blog.ircmaxell.com/.

CHAPTER 4

Authentication, Access Control, and Safe File Handling

Erica works for a small, local manufacturing company as their programmer and general IT person. She was recently tasked with developing software for the office staff to upgrade them to the digital age instead of using paper records.

The first step was to have staff members scan documents when they were finished with them. Instead of filing the paper forms, they would be stored electronically.

Development of the archiving system proceeded well. The system consisted of a basic database listing all the documents, with various filters such as department, date, tags, etc. Nothing too complex; Erica wanted to keep it simple.

The documents displayed to users are filtered based on the user's department and their position (e.g., user, manager, director). Everything worked well for several months, and then it happened. There was a breach of a secure document. The subsequent investigation determined a disgruntled employee had gained access to a file meant only for senior management. The employee leaked the document to a competitor to secure a winning bid for a cushy position for himself at said company.

So how did it happen? Erica has been working all weekend trying to figure it out, when boom, she finds it. It's so obvious, but she never thought to secure it.

In Erica's system, files are uploaded and named after the ID fields of their respective database records. An uploaded PDF might be named "13424.pdf", while the next file, a DOC for example, would be named "13425.doc", and so on. Erica had secured the document listing that each user sees, but had not considered securing the files themselves when opened or downloaded by the browser. There was no protection to stop a user from gaining access to sensitive documents by simply guessing file names by incrementing the ID.

The stories are getting a little long, but if you're committed enough, you can make any story work. I once told a woman I was Kevin Costner, and it worked because I believed it.

B. Edmunds, *Securing PHP Apps*, DOI 10.1007/978-1-4842-2120-4_4

Authentication

The first step to properly handling sensitive data is authentication. You need to ensure that a user is who they say they are.

Here you will expand on the code samples from Chapter 3 to validate a password, then denote for later use that the user has a valid log in:

```
1   class UserModel
2   {
3     //... other methods
4
5     function login($user, $password)
6     {
7       if  (password_verify($_POST['password'], $user-\
8       >pass) === TRUE) {
9         $valid = TRUE;
10      }
11
12      //other checks, error handling, etc...
13
14      if ($valid === TRUE) {
15        //successfully logged in
16
17        //just example session class
18        //your own code or framework will likely
19        //have it's own session wrapper
20        Session::put('user', $user->id);
21        return TRUE;
22      }
23
24      //failed to login
25      Session::put('user', NULL);
26      return FALSE;
27    }
28  }
```

This had saved the user's successful log in. A key point here is noting their user ID. Just because they're logged in doesn't mean they have access to do anything they want in your application.

Role-Based Access Control

In addition to determining if the user has a valid log in, you need to determine if they have access to the page/section/feature they are requesting.

Let's start with a basic use case. You have two types of users: Muggles and Admins. Muggles are your basic users with very restricted access. Admins are the site administrators who have access to do everything. An Admin can edit or delete users, manage posts, etc. It is very important that a normal user—a Muggle—cannot perform these actions.

Different frameworks have differing conventions for when and where to check for proper access. In Laravel, filters are preferred. Symfony has voters. In most systems, you may simply use the constructors of your controllers. The following is a hypothetical example; your exact implementation will vary, and is up to you.

First, let's add a method to the `UserModel` for querying the current user:

```
1   class UserModel
2   {
3     /... other methods
4
5     //... login method
6
7     function current()
8     {
9       $user = FALSE;
10
11      if (isset($_SESSION['user']) === TRUE) {
12        $user = DB::findById($_SESSION['user']);
13      }
14
15      return $user;
16    }
17
18    //let's imagine there is a method for retrieving
19    //the user's groups.
20  }
```

Now let's use these methods to verify that this user has appropriate access:

```
1   class AdminController
2   {
3     function   construct()
4     {
5       $userModel = new  UserModel;
6       $user = $userModel->current();
7
8       //check to make sure the user is logged in and
9       //is a member of the admin group
10      if ($user === FALSE || in_array('admin', $user\
11      ->groups) === FALSE) {
12
13        //please add a legit error message
14        //with headers and all that fancy stuff
15        die('Not Authorized');
16      }
17    }
18
19    //other admin only methods
20  }
```

You will usually use some type of abstracted access control layer on top of your regular controllers/routes/etc. This layer should map your routes to the access level required to view that route. For example, `/admin/*` might only be accessible to users in the Admin group. `POST` and `PUT` requests might only be accessible to editors. `DELETE` requests might only be accessible by Admins.

The point is, each page should check the user's access levels and determine if they are authorized for the requested action(s). You don't want to assume that, if a user can see a form and `POST` it to the correct endpoint, the user is authorized to perform that action. This will help ensure protection against malicious users, as well as unexpected changes to your data by users who should not have been able to access that data in the first place.

You never want to breeze through authentication security. Take it seriously! You will save yourself or other developers from unnecessary development time and headaches in the future.

Validating Redirects

In the flow of your application, oftentimes you will `POST` a form to an endpoint, validate that data, perform some action, then redirect the user to the next step in the application flow.

If the page you are redirecting to contains sensitive data, someone can bypass your expected flow by simply sending a request straight to this final page, bypassing your validation step.

There are a few ways to handle this. The first way is to simply not redirect at all. Most of the time when you think you need a redirect, you could simply call the next method directly. So instead of:

```
1   if ($valid === TRUE && $dataSaved === TRUE) {
2     URL::redirect('/blog/1/edit');
3   }
```

you could call the edit method without a redirect:

```
1   if ($valid === TRUE && $dataSaved === TRUE) {
2     $this->_edit(1);
3   }
```

In this example, the `_edit()` method would be a protected or private method that can't be accessed from a URL without going through the previous step.

If you do in fact need a separate endpoint, which you may if there are multiple entry points to this destination, then you need to verify that the proper steps have been executed. Passing any appropriate data along each request in an expiring session variable (commonly called flash data) is usually the safest way.

```
1   if ($valid === TRUE && $dataSaved === TRUE) {
2     Session::flash('blogEditValid', TRUE);
3     URL::redirect('/blog/1/edit');
4   }
```

In your edit endpoint, you would verify the passed-along data:

```
1   public function edit($id)
2   {
3     if (Session::get('blogEditValid') !== TRUE) {
4       //again, this should be a proper error
5       //in a real world app
6       die('Not Authorized');
7     }
8   }
```

Obfuscation

Have you heard the phrase "security through obscurity"? It's rarely true, but in some cases it is helpful. Most applications will use an ID field in each table as a primary key. This ID is then used throughout the system to access data. It is passed through URLs, forms, and APIs to denote which piece of data is needed.

Sometimes, though, you don't want to expose the user to the actual row ID. Maybe you are launching a new product and don't want the user to know that they are only the 13th user. Maybe you have public data but don't want your site to be easily crawled by scraping bots.

In these cases, you can obfuscate the ID to something that isn't incremental (such as 1, 2, 3, etc.) but can be translated to your ID field. Rather than doing this:

```
1   Route::get('/edit/{id}', function($id)
2   {
3     //id =  4321234
4     $post = $db->post()->where([
5       'id' => $id
6     ])->row();
7   });
```

you could do this:

```
1   Route::get('/edit/{hash}', function($hash)
2   {
3     //hash =  BaPjae
4     $id = HashId::decrypt($hash);
5
6     $post = $db->post()->where([
7       'id' => $id
8     ])->row();
9   });
```

In this example, the `HashId::decrypt()` call is simply taking some preexisting security hash from your server and applying it against the passed hash to determine the ID. This can also be called in reverse to generate a hash:

```
1   $hash = HashId::encrypt(4321234);
2   echo $hash; //outputs BaPjae
```

It is rather easy to write these hashing methods yourself, but I recommend using the tried and true HashIDs[1] libraries. These libraries are not only easy to use, but are also well maintained and available across many different programming languages to ensure interoperability throughout your entire infrastructure.

There are cases where obfuscation can be argued as needed or it could even be required in your particular use case. For example, I've seen people use obscurity as an additional safeguard for HIPAA (Health Insurance Portability and Accountability Act) data. One use case was a specification that required keeping HIPAA-sensitive data in a separate database, in this case it was a three-tier server architecture. Primary keys used by the web application were stored in one database (along with nonsensitive data), sensitive data was stored in another, and an intermediary database stored the relations. This was designed so that if any one of these databases were compromised, the data obtained would be anonymous or nonsensitive.

I'm going to reiterate this point because it is extremely important: in most cases, obscurity doesn't protect you from any legitimate attacks. You shouldn't rely on it for security. It is simply a means of making things a little harder to find.

Safe File Handling

Circling back to our original story: if you have documents that are served to your users for viewing or downloading, you can't simply set access control on the `*.pdf` files. *Why not?* See, I knew you were going to ask that. It helps that I'm the narrator here.

What you need to do is store the file on your server where it can't be accessed from your web server. Here's one example of a recommended directory structure:

```
application/
composer.json
composer.lock
htdocs/
    .htaccess assets/
    templates/
    index.php
    robots.txt
uploads/
workers/
```

[1] http://www.hashids.org/php/.

Here `htdocs/` is the location that Apache (or whatever web server you're using) would serve for your domain. And `uploads/` is where you would store uploaded documents. You can also do this using your server's configuration. In Apache you could return an error 404 to any request to the uploads directory if it were in the `htdocs` tree.

To implement this in your application, you would have an endpoint for accessing the documents depending on their type. So maybe you need to serve up monthly financial statements, but only to users in the accounting group.

Here you will define the endpoint to access this. With this endpoint you will verify the user's access level, read the file, and finally output it to the browser with the appropriate headers:

```
1   Route::get(
2     '/accounting/statements/{year}/{month}',
3     function($year, $month) {
4
5           //check user access
6           $userModel = new  UserModel;
7           $user = $userModel->current();
8
9           //check to make sure the user is logged in
10          //and a member of accounting
11          if ($user === FALSE || in_array('accounting', $us\
12          er->groups) === FALSE) {
13          //please add a legit error message
14          die('Not Authorized');
15        }
16
17        //the user has access, let's read the file
18        $directory = __DIR__. '../uploads/acct/stmnts/';
19        $filename = (int) substr($year, 0, 4)  .
20                    (int) substr($month, 0, 2) .
21                    '.pdf';
22
23        //error handling for invalid files
24
25        //open the file
26          $fileHandle = fopen($directory . $filename, 'r');
27
28          //read the file and close it
29          $fileContents = fread($fileHandle);
30          fclose($fileHandle);
31
32          //send the appropriate headers
33          header('Content-type: application/pdf');
34          header('Content-Disposition: inline; filename="' \
35          . $filename . '"');
36          header('Content-Length: ' . filesize($directory .\
37          $filename));
```

```
38          header('Expires: 0');
39          header('Cache-Control: must-revalidate');
40          header('Pragma: public');
41
42          //echo the file contents to the browser
43          echo $fileContents;
44
45  });
```

This prevents someone that shouldn't see this file from accessing it, since it is outside the web server tree and it is being checked against the proper access controls.

Recap

I've covered basic authentication procedures, proper access control through never trusting without validation, and safe file handling procedures. I hope you find this helpful while developing your next awesome idea.

Keep reading, 'cause I'm not done yet. Remember, you're done when I say you're done!

CHAPTER 5

Safe Defaults, Cross-Site Scripting, and Other Popular Hacks

No story this time. This chapter is a catch-all for a couple other attacks you need to protect against, so there isn't an overarching narrative. Try to contain your disappointment.

Never Trust Yourself: Use Safe Defaults

One of the core concepts of a secure system is safe defaults. Whenever possible (and it's usually possible), you should define variables, properties, and so forth early with a safe default.

A safe default usually means a NULL, empty, or FALSE state. When determining logic flow, the default should always be a failure. For example, in the authentication examples in the previous chapters, you checked if the password is correct. If it was, you proceed to the positive application logic. If it failed, the function executed the default logic for a nonpositive result.

Let's look at a basic example with form validation:

```
1   Route::post('/signup', function()
2   {
3     //check for a valid form
4     if (Form::validate('signup') === TRUE) {
5       //process the form
6     }
7
8     //the default logic is failure
9     die('Invalid Form Data');
10  });
```


B. Edmunds, *Securing PHP Apps*, DOI 10.1007/978-1-4842-2120-4_5

Never Trust Dynamic Typing: It's Not Your Friend

Dynamic typing is a feature loved by newer programmers, because it seems to make development easier. Dynamic typing means you don't have to be so picky about typing; you just get close enough and it'll work. The problem with this is it doesn't always work the way you'd expect.

Let's look at a classic example with a native PHP function, `stripos()`. The `stripos()` function finds the position of the first occurrence of a string inside another string. The scenario is that you're trying to figure out if the letter "i" exists in the phrase "I am the one who knocks."

```
1   $phrase = 'I am the one who knocks';
2
3   $letterExists = stripos($phrase, 'i');
4
5   if ($letterExists != FALSE) {
6     echo 'we should be here';
7     return TRUE;
8   }
9
10  echo "we shouldn't be here, yet we are";
11  return FALSE;
```

The `stripos()` function returns the index of the match in the haystack string. Because "i" is the first letter of the string, `strpos()` is returning `0`. Due to dynamic typing, `!= FALSE` evaluates the same as `!= 0`. Now let's fix this with an explicit check using `!==` instead:

```
1   $phrase = 'I am the one who knocks';
2
3   $letterExists = stripos($phrase, 'i');
4
5   if ($letterExists !== FALSE) {
6     echo 'we should be here';
7     return TRUE;
8   }
9
10  echo "we shouldn't be here, annnnnnd we aren't";
11  return FALSE;
```

Another lesser-known advantage to explicit checks is performance. Most of the time an explicit check will be faster, because it doesn't need typecasting. If you're ever curious about performance differences in PHP, I recommend checking out PHP Bench.[1]

[1] http://www.phpbench.com/.

Cross-Site Scripting

Cross-site scripting (XSS) is the process of injecting malicious code into the target web site. This can be done in several ways, but the end result is the user's browser runs unauthorized code as themselves, within their current session.

Nonpersistent XSS

This is the traditional type of XSS exploit. It involves injecting data into a site and then guiding users to the malicious content.

Say a page on your site takes `?page_num=2&per_page=50` as query string parameters. If you do not escape these parameters, an attacker could change these values to malicious code. This code could take the user to a delete page, run JavaScript in their browser, or any number of client side attacks.

After injecting their malicious code, the attacker somehow gets a user to visit the page. When the user arrives, the application will verify their valid user session and execute the malicious code. A user could even end up deleting their own account!

Persistent XSS

A persistent XSS exploit is an exploit stored permanently on the server. For example, a social sharing site like Facebook allows users to save messages and display them to other users. An attacker could store malicious code in a Facebook post. If Facebook was not properly escaping this data when displaying it back to other users, that code would be executed.

So anyone who sees the attacker's status would be running this malicious code.

Attack Entry Points

I just mentioned a couple examples of XSS exploit entry points, but those are not the only ones. Basically anywhere you take input from a user and display it back on a web page is an opportunity for an attacker to exploit your site.

Be sure to think about **all** places data enters your system, not just input fields. For example, maybe you allow users to upload images and then redisplay the image XIFF data. Maybe you parse uploaded CSV files of various data exports from external programs. Anywhere that you redisplay information, it needs to be protected.

How to Protect Yourself

The fix for this is not very difficult. First, never take data directly from a URL and echo it back to the browser. The same goes for data from other sources, like your database or uploaded files. To protect yourself, you simply need to escape data going into your database and escape data being displayed back to your users.

I've already discussed proper database escaping in Chapter 1, so I'll focus on displaying data here. PHP makes this very simple with the built-in function `htmlentities()`. This function will properly handle the majority of your data.

Let's see what this looks like implemented. Your view file used to look like this:

```
1   <h1>Title</h1>
2
3   Hello <?=$name?>,
```

Now with the protection applied, it looks like this:

```
1   <h1>Title</h1>
2
3   Hello <?=htmlentities($name)?>
```

If you want to stay DRY,[2] you would probably abstract this away into a view helper library. Most frameworks already include this in their view or template libraries.

Cross-Site Request Forgery

Cross-site request forgery (CSRF) is basically the opposite of an XSS exploit. Where XSS takes advantage of the user by means of a trusted web site, CSRF takes advantage of the web site by means of a trusted user.

An example of this is an attacker sending out fake e-mails with a link to delete a blog post, an e-mail, or whatever. The target user then clicks the link and arrives at a delete page. Because the user is an administrator with a valid session, your application goes ahead and deletes the record as requested. The user had no idea that the link was taking them to it and now their account has been deleted without their consent. Not cool.

This doesn't have to be a text link either; it is often attached to an image or a button. This might sound like a small risk because most critical web site functions are behind forms that expect POSTed data. But this can just as easily be expanded upon to use a button or JavaScript to submit hidden forms.

How to Protect Against Forgeries

The first step is to ensure no data-altering actions are performed by GET requests. Anything that performs an action on data should require a POST, PUT, or DELETE request. If the user clicks a delete button, they should then be taken to a form used to confirm the action. If data-altering actions need to be performed over GET (maybe for a RESTful API), you can require a unique token in the query string. In the following examples I will be using POST data, but the exact same concepts apply when dealing with GET requests. Just set the token in the query string instead of in the POST parameters.

[2] http://en.wikipedia.org/wiki/Don't_repeat_yourself.

Now that you are submitting forms for your data manipulations, you will need to add CSRF tokens to your forms. The CSRF token will be a standard Nonce (Number used Once). You will generate a random token, store it in the user's session, then add it as a hidden field to your form. Once the form is POSTed, you can check the CSRF token against the one in the session to validate the request.

First, let's create a function to generate the token. This will usually be placed in a universally callable place, maybe as a route filter, voter, or a helper library:

```
1   //assuming the rest of the form class here
2
3   static function generateCsrf()
4   {
5     $token = mcrypt_create_iv(
6       16,
7       MCRYPT_DEV_URANDOM
8     );
9
10    Session::flash(
11      'csrfToken',
12      $token
13    );
14
15    return $token;
16  }
```

Note that you are using session flash data here. Flash data will be stored to the session but can only be accessed on one request before it is destroyed. This concept is supported in most session wrapping classes. This keeps the token from being valid on more than one request.

Next let's call this within the route closure and pass the token to the view that is generating the form:

```
1   Route::get('/signup', function()
2   {
3     $data['token'] = Form::generateCsrf();
4
5     return  View::render('signup.form', $data);
6   });
```

And now for our view, `signup/form.php`:

```
1   <form method="POST" action="/signup">
2
3     <label>
4           First Name:
5           <input type="text" name="first_name" />
6     </label>
7
```

```
8     <label>
9           Last Name:
10          <input type="text" name="last_name" />
11    </label>
12
13    <label>
14          Email:
15          <input type="text" name="email" />
16    </label>
17
18    <input type="hidden" name="token" value="<?=$tok\
19  en?>" />
20
21    <input type="submit" name="submit" value="Signup\
22  " />
23
24  </form>
```

When this form is POSTed, the token can now be validated:

```
1   Route::post('/signup', function()
2   {
3     //this would probably be abstracted away into
4     //a route filter or your form validation
5     if  ($_POST['token']  === Session::get('csrfToken'\
6   )) {
7           //process the form
8     }
9
10    //like earlier, you should add a
11    //legit error message here
12    die('Invalid Form Data');
13  });
```

Now that this token-checking process is in place, if an attacker tricks a user into submitting a fake form, the request will fail. The user will not have a matching CSRF token in their session data for your web site.

Multiple Form Submits

Another prolific issue in PHP applications is multiple form submissions. A user submits a form to perform some action, let's say transferring 20 Bitcoins from one wallet to another. The user clicks submit, but they don't notice a change immediately, so they click again. Now the user has inadvertently transferred 40 Bitcoins instead of 20. Luckily for us, the CSRF token logic I just discussed will handle this without any extra work. To prevent this in most situations, you just need to pass a unique token, validate it, then clear it once it has been processed.

Race Conditions

Race conditions are not super common in PHP, but are very hard to debug once they occur. It is best to handle them before they happen. A race condition is when multiple things are happening at once, causing unexpected logic flow. The issue is when Block B executes before Block A, due to Block A taking longer to perform.

A basic example is two processes writing to the same file. Without a transactional locking mechanism in place, data is susceptible to corruption. If you expect Process A to be finished writing to a file before Process B starts, data could be overwritten, written on top of other data, or many other types of data corruption. With transactional locking in place, you ensure Process A is finished writing to the file before allowing Process B to write to the same file.

This example is very process-specific, but the main concept of preventing race conditions is to make logic transactional where it should be.

Another example is with database writes. To prevent race conditions in the database, use transactions to apply certain database changes only if all statements are successful.

The general concept is that if only one thing is supposed to happen at a time, you should check to ensure that each step has finished before proceeding to the next step.

Outdated Libraries/External Programs

Another quick item to bring to your attention is outdated libraries. The best way to ensure that your code stays safe is to keep all of your dependencies up to date. No matter how secure the code you personally write is, all it takes is a single security vulnerability in any library you use and your site can be exploited.

Another thing to keep in mind is external programs you use on your server. For example, PhpMyAdmin has had several security flaws throughout its lifetime that have left servers vulnerable. Outdated Wordpress installations are also well known in development circles as a back door for hackers. Any program that exposes critical functions on your server is a possible attack entry point.

Try to keep your external dependencies to a minimum and always keep them up to date with their latest security releases.

Destructor

I had a lot of fun during the process of writing this. I truly hope that you learned something new and enjoyed reading this.

You've learned a bit about protecting your applications from users, and users from themselves; securing your communication via HTTPS; safe password encryption techniques; role-based access control; and about popular attack vectors.

Please get in contact[1] if you want to learn more about any of the subjects covered here or just argue over the meaning of life.

Thanks for reading.

[1] feedback@buildsecurephpapps.com.

B. Edmunds, *Securing PHP Apps*, DOI 10.1007/978-1-4842-2120-4

Index

B. Edmunds, *Securing PHP Apps*, DOI 10.1007/978-1-4842-2120-4

S

T, U

V, W, X, Y, Z