The Laravel Security Checklist

sqreen

INTRODUCTION

Damn, but security is hard.

It’s not always obvious what needs doing, and the payoffs of good security are at best obscure. Who is surprised when it falls off our priority lists?

We’d like to offer a little help if you don’t mind. And by « help » we don’t mean « pitch you our product »—we genuinely mean it.

Sqreen’s mission is to empower engineers to build secure web applications. Laravel already comes with great out-of-the-box security features but it’s often not enough. We’ve put our security knowledge to work in compiling an actionable list of best practices to help you get a grip on your security priorities. It’s all on the following pages.

We hope you find it useful. If you do, share it with your network. And if you don’t, please take to Twitter to complain loudly—it’s the best way to get our attention.

The Sqreen Team
@SqreenIO
hey@sqreen.com

CODE

✔ Use PHP 7!

Laravel 5.5 requires PHP 7 or above. PHP 7 includes a range of built-in security-specific improvements (such as libsodium in PHP 7.2) and deprecates older, insecure features and functionality. As a result, it is far easier to create more secure applications with PHP 7, than any previous version of PHP. Use it whenever possible.

Read more:

- Deprecated features in PHP 7.0.x
- Deprecated features in PHP 7.1.x
- Deprecated features in PHP 7.2.x
- Deprecated features in PHP 7.3.x
- Migrating a PHP 5 App to PHP 7

✔ Make sure you use the latest version of Laravel

New vulnerabilities appear on a regular basis. Make sure you use the latest version of Laravel for your projects. Versions up to 5.6.29 have known SQL injection or code execution vulnerabilities.

Read more:

- Read about new Laravel CVEs
- Upgrade guides
- Shift

✔ Use a SAST

- A SAST is a Static Application Security Tester (or testing service). A SAST scans source code looking for vulnerable code or potentially vulnerable code. Pre-production tools come with a number of false positives and false negatives, but still add some value.

Read more:

- 11 Best PHP Code Security Scanners to Find Vulnerabilities
- SAST, DAST, and RASP: A guide to the new security alphabet soup

---

✔ Use a DAST

A DAST is a Dynamic Application Security Tester (or testing service). A DAST searches for weaknesses and vulnerabilities in running applications. Pre-production tools come with a number of false positives and false negatives, but still add some value.

Read more:

- Common Approaches to Automated Application Security Testing - SAST and DAST
- Acunetix

✔ Filter and Validate All Data

Laravel's Eloquent ORM uses PDO parameter binding to limit SQL injections. But Laravel also offers other ways to craft SQL queries. Regardless of where the data comes from, whether that's a configuration file, server environment, GET and POST, or anywhere else, do not trust it. Filter and validate it!

Read more:

- Validation in Laravel

✔ Avoid using Raw Queries

In the item above, we saw the importance of validation. But we really recommend not to use raw SQL queries all-together. Using raw queries just increases the risk for SQL injection vulnerabilities.

Read more:

- Queries in Laravel

✔ Whitelist, Never Blacklist

Never attempt to filter out unacceptable input. Just filter for only what is acceptable. To attempt to filter out anything that is unacceptable leads to unnecessarily complicated code, which likely leads to defects and security flaws.

PHP Security - Never Blacklist; Only Whitelist

✔ Use Parameterized Queries

This should come out of the box if you're using Laravel's native features. But to avoid SQL injection attacks, never concatenate or interpolate SQL strings with external data. Use parameterized queries instead and prepared statements. These can be used with vendor-specific libraries or by using PDO.

Read more:

- Prepared statements and stored procedures in PDO
- Mysqli Prepared Statements
- The PostgreSQL pg_query_params function

✔ Use Eloquent ORM

Laravel comes embedded with a great ORM called Eloquent ORM. It is scrutinized and tested by a community of security-conscious developers.

Read more:

- Eloquent ORM Docs

✔ Use Laravel's built-in encryption

Laravel comes with a built-in encryption mechanism and we highly recommend you use that one instead of building your own. As of PHP 7.2, older encryption libraries have been deprecated, such as Mcrypt. However, PHP 7.2 supports the far better Libsodium library instead. If you want to use a different encryption library, take a look at Libsodium.

Read more:

- PHP 7.2: The First Programming Language to Add Modern Cryptography to its Standard Library
- Libsodium
- Laravel Encryption

✔ Set open_basedir

The `open_basedir` directive limits the files that PHP can access to the filesystem from the `open_basedir` directory and downward. No files or directories outside of that directory can be accessed. That way, if malicious users attempt to access sensitive files, such as `/etc/passwd`, access will be denied.

Read more:

- open_basedir configuration directive
- PHP Filesystem Security
- Isolated Execution Environments by DigitalOcean

✔ Make sure permissions on filesystem are limited

PHP scripts should only be able to write in places you need to upload files of specifically write files. This places should not be anywhere a PHP script can be executed by the server. Else, it open the way for an attacker to write a PHP file somewhere and to run arbitrary PHP code.

Learn more:

- OWASP filesystem guide
- File visibility on Laravel
- StackOverflow: How to set up File Permissions for Laravel

✔ Perform Strict Type Comparisons

If weak type checking is used, such as with the `==` operator, vulnerabilities can occur due to the often peculiar ways that PHP converts types. These include 1.14352 being converted to 1, strings converting to 1, "1is this true" converts to true, and so on. This is because according to the manual:

> By default, PHP will coerce values of the wrong type into the expected scalar type if possible.

Use strict type checking to ensure that when comparing two items that they are of the same type. And in PHP 7.1, use `declare (strict_types=1);`.

Read more:

- PHP 7 type hinting: inconsistencies and pitfalls
- PHP strict typing

✔ Use libxml_disable_entity_loader(true)

To avoid XML External Entity Injections, when working with XML content, use `libxml_disable_entity_loader` to disable external entity resolution.

Read more:

- XML external entity attack
- XML External Entity (XXE) Prevention Cheat Sheet
- libxml_disable_entity_loader
- libxml

✔ Don't Implement Your Own Crypto

Unless you're a security expert—and even if you are—never implement your own crypto. This is a common cause of security errors, as too few eyes have had a chance to review the code. Instead, use a publicly reviewed, critiqued, and tested library, such as Libsodium in PHP 7.2 or Laravel's built-in encryption.

Read more:

- Why is writing your own encryption discouraged?
- Why shouldn't we roll our own (cryptography)?
- Why You Don't Roll Your Own Crypto

✔ Integrate Security Scanners Into Your CI Pipeline

Security scanners can help to detect questionable code and code that contains obvious security defects. Continuous Integration (CI) tools can use these scanners to test your code and fail the build if the scanner meets or surpasses acceptable thresholds.

Read more:

- ircmaxell/php-security-scanner
- PHP Quality Assurance

✔ Keep All Dependencies Up to Date

Most PHP code relies on external, third-party dependencies. However, these need to be kept up to date, wherever possible, to ensure that any bug and security fixes are available to your code. Ensure you're using Composer as your dependency manager and keep up to date with all of your dependencies.

Composer basic usage

✔ Invalidate Sessions When Required

After any significant application state change, such as a password change, password update, or security errors, expire and destroy the session.

Read more:

- Session management in Laravel
- PHP Session Security Best Practices

✔ Never Store Sensitive Data in Session

No sensitive data—ideally only a minimum of data or the session id—should ever be stored in a session.

Read more:

- Session hijacking attack

- Session fixation attack
- OWASP Session Management Cheat Sheet

✔ Never Store Sessions in a Shared Area

It has been common, when using shared hosting providers, for PHP to be automatically configured to store sessions on the filesystem, in the same directory. Never do this. Always check your configuration and store session information in a private location, accessible only by your application.

Read more:

- Session management in Laravel
- Shared Hosting: PHP Session Security
- Custom Session Handlers
- Storing sessions in Memcache

✔ Use Secure Session Settings

When using sessions make sure that you configure them to be as secure as possible to prevent as many attacks as you practically can. This includes locking a session to a domain or IP address, don't permanently store cookies, use secure cookies (sent over HTTPS), use large session `sid_length` and `sid_bits_per_character` values. Laravel comes with secure configurations out-of-the-box so make sure you read the documentation and don't try to innovate on that part.

Read more:

- Session configurations
- https://secure.php.net/manual/en/session.security.ini.php
- Session Management Basics

✔ Don't Cache Sensitive Data

When you cache data to speed up your application, such as database requests, ensure that sensitive data isn't cached.

Read more:

- Best practice for caching sensitive data

✔ Store Passwords Using Strong Hashing Functions

Ensure that all passwords and other potentially sensitive data are hashed, using robust hashing functions such as bcrypt. Don't use weak hashing functions, such as MD5 and SHA1. Laravel comes with a native hash mechanism using Bcrypt and Argon2. Use them!

Read more:

- Hashing in Laravel
- Use Bcrypt or Scrypt Instead of SHA* for Your Passwords, Please!
- The Dangers of Weak Hashes

✔ Use a Reputable ACL or RBAC Library

To ensure that access to sensitive data is both authenticated and authorized, use mature ACL (Access Control Library) and RBAC (Role Based Access Control) packages.

Read more:

- Two Best Laravel Packages to Manage Roles/Permissions
- Laravel Authentication

✔ Use a Package Vulnerability Scanner

As modern PHP applications use a wide variety of external dependencies, you need to be sure that you're not using any with known vulnerabilities. To do that, ensure that you're regularly scanning your source code with a vulnerability scanner.

Read more:

- Roave Security Advisories
- FriendsOfPHP/security-advisories
- SensioLabs Security Advisories Checker
- retire.js

- AuditJS

✔ Always Perform Context-Aware Content Escaping

Whether your outputting information in an HTML template, in CSS, or in JavaScript, avoid exposing your users to CSRF (Cross Site Request Forgery) and XSS (Cross Site Scripting) attacks by performing context-aware content escaping.

Read more:

- Blade Templates
- Safer PHP output
- Twig escape method
- XSS (Cross Site Scripting) Prevention Cheat Sheet

✔ Deprecate Features and Libraries When They're No Longer Used

Stay abreast of the packages and language features that you're using. If a language feature planned to be deprecated, or a third-party package becomes abandoned, then start planning to replace it with an alternative.

Read more:

- sensiolabs-de/deprecation-detector
- samsonasik/is-deprecated
- How to Deprecate PHP Package Without Leaving Anyone Behind

✔ Never Display PHP Errors and Exceptions in Production

While errors, warnings, and exceptions are helpful during development, if displayed in production or any other public-facing environment, they may expose sensitive information or intellectual property. Ensure that this information is logged internally, and not exposed publicly.

Read more:

- Error Handling in Laravel

✔ Disable Unsafe and Unrequired Functionality

Some PHP installations can be preconfigured with unsafe and unrequired functionality already enabled. Ensure that you review your PHP configuration and `phpinfo()` output for any unsafe settings and disable or limit them.

Read more:

- OWASP PHP Configuration Cheat Sheet

✔ Filter File Uploads

If malicious files can be uploaded and executed by users, then the application, its data, or the supporting server(s) can be compromised. Ensure that PHP's file upload configuration is correctly configured to avoid these attacks from occurring.

Read more:

- OWASP Unrestricted File Upload
- Laravel File Uploads
- How to securely upload files with PHP
- How to Securely Allow Users to Upload Files

✔ Disable or Limit Program Execution Functionality

Program execution functionality, such as exec, passthru, shell_exec, and system, can leave open the possibility for users to be able to execute arbitrary code on your system and shell injection attacks. Disable this functionality if it's not explicitly needed.

Read more:

- Program execution Functions
- Shell injection attacks
- PHP disable_functions

INFRASTRUCTURE

✔ Connect to Remote Services With TLS or Public Keys

When accessing any database, server, or remote services, such as _Redis_, _Beanstalkd_, or _Memcached_, always do so using TLS or public keys. Doing so ensures that only authenticated access is allowed and that requests and responses are encrypted, and data is not transmitted in the clear. By default Laravel is served over plain HTTP, so make sure you use the secure command in Valet.

Read more:

- Securing sites with TLS
- Public Key Infrastructure and SSL/TLS Encryption
- What is SSL, TLS and HTTPS?
- SSL vs. TLS - What's the Difference?

✔ Check Your SSL / TLS Configurations

Ensure that your server's SSL/TLS configuration is up to date and correctly configured, and isn't using weak ciphers, outdated versions of TLS, valid security certificates without weak keys, etc, by scanning it regularly.

Read more:

- SSL Labs
- Observatory by Mozilla

✔ Renew Your Certificates On Time

Using SSL certificates is essential to encrypting your web site or application's traffic with HTTPS. However, they do expire. Ensure that you're updating your certificates before they expire.

Read more:

- Get Alerts For Expiring SSL Certificates

- Free and Auto-Renewing SSL Certificates: Letsencrypt Quick Setup (2017 Edition)

✔ Rate Limit Requests to Prevent DDoS Attacks

To stop users attempting to perform brute force login attacks and overwhelm your forms, use tools such as Fail2Ban to throttle requests to acceptable levels.

Read more:

- Fail2Ban for Laravel
- Fail2ban
- How To Protect SSH with Fail2Ban on Ubuntu 14.04

✔ Log All The Things

Regardless of whether you're logging failed login attempts, password resets, or debugging information, make sure that you're logging, and with an easy to use, and mature package, such as Monolog.

Read more:

- Monolog
- Laravel Logging Basics

✔ Do not send sensitive information in headers

By default Laravel will send its version number in HTTP headers. Double check your headers to ensure that you're not sending sensitive information.

Read more:

- Hide PHP and Apache informations from HTTP headers

✔ Do Not Store Sensitive Data In Configuration Files

Just like you shouldn't store sensitive data in cache entries, you also should not store sensitive data in configuration files. This includes ssh keys, access credentials, and API tokens. Store them in environment variables instead.

Read more:

- The Twelve-Factor App
- PHP dotenv

✔ Make Requests Over HTTPS Wherever Possible

To avoid Man in the Middle attacks, to protect the integrity of your site, and privacy of your users, you need to make all requests over HTTPS—especially requests involving sensitive data, such as logins, password and account changes.

Read more:

- Why HTTPS Matters
- Let's Encrypt
- Apache SSL/TLS Strong Encryption: How-To
- Configuring NGINX HTTPS servers
- Docker, Traefik, and Let's Encrypt

PROTECTION

✔ Send All Available Security Headers

There are several security headers that you can use to make your websites and web-based applications more secure, for minimal effort. These include HSTS, X-XSS-Protection, X-Frame-Options, X-Content-Type-Options, and a Content Security Policy. Ensure that they're being configured correctly and sent in your request responses.

Read more:

- Use these Five Security Headers To Create More Secure Applications
- SecurityHeaders
- PHP header function
- Hardening Your HTTP Security Headers

✔ Have a Content Security Policy

Whether you have a one page, static website, a large static website, or a sophisticated web-based application, implement a Content Security Policy (CSP). It helps to mitigate a range of common attack vectors, such as XSS.

Read more:

- Content Security Policy (CSP) via MDN web docs
- Content Security Policy (CSP) via the Google Chrome extensions documentation
- CSP Evaluator
- Content Security Policy (CSP) Validator
- Easily add and manage a CSP with Sqreen

✔ Protect your users against account takeovers

Credential stuffing or brute force attacks are easy to setup. You should make sure your users are protected against account takeovers.

Learn more:

- [Sqreen]
- [Blocking Bruteforce attacks - OWASP]

✔ Monitor your application security

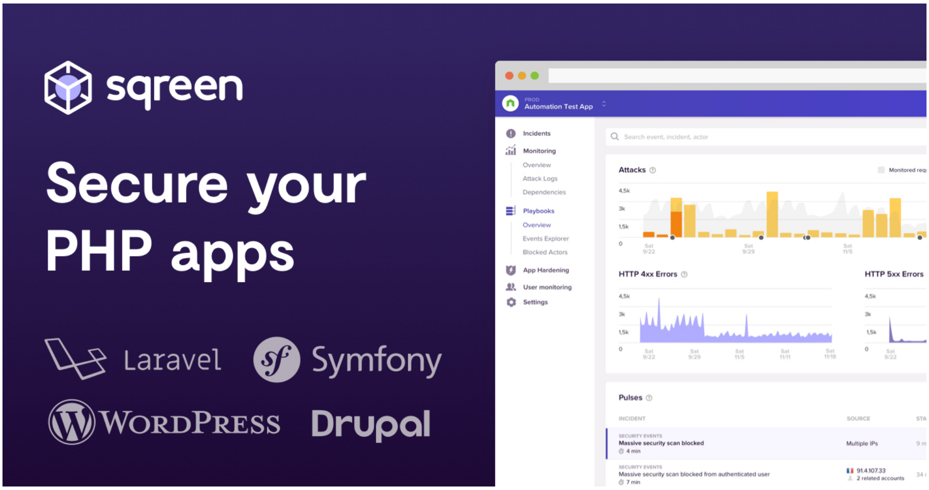
Monitor your application security for suspicious behaviors and attacks. Knowing when your application is starting to get attacked is key to protect it before it's too late.

- Monitor your Laravel App security

✔ Protect your sensitive data in real-time

Code vulnerabilities will always exist. Make sure you have a security solution in place that detects and blocks OWASP attacks but also business logic threats.

- Protect your PHP app



www.sqreen.com