Exalted Community Prototype

A study in web application security

Marie Hogebrandt

Sammanfattning

Detta är en rapport om ett system i PHP byggt på ramverket Laravel, med MySQL som databas. Det är en prototyp för ett större system och har ett grundläggande användarsystem samt funktioner för att kunna posta och se meddelanden. Det har byggts för att minimera de vanligaste säkerhetsriskerna.

Nyckelord: *Laravel, Webbapplikationssäkerhet, PHP, MySQL*

Abstract

This is a report about a system in PHP built on the framework Laravel, with MySQL as a database. It is a prototype for a large system and has a basic user system as well as functionality to post and view messages. It has been built to minimise the most common security risks.

Keywords: *Laravel, Web Application Security, PHP, MySQL*

Contents

1 Introduction

This project is a very basic prototype for a project that will be open source: A community tool for role players. The idea for it came out of a recent issue where one person's code was hosted by that person for several different role playing chats, and when he no longer had the time or energy to deal with the hosting, he forced the chats to close, as he was not interested in sharing it.

The current prototype only includes the beginnings of a user system and the ability to post on a communal wall. That communal wall is going to be expanded to several different kinds of messages, and the user system will become more full-fledged, with individualised profiles and e-mail reminders. One of the key requirements to the future implementations is the ability to easily export/import all of the data from one installation to another, as well as be open sourced so that others can pick it up if necessary.

1.1 Disposition

Chapter 1 describes the reasons behind the project and what is intended to be accomplished.

Chapter 2 describes separation of concerns in modern web development, followed by oWasp top-10 list of vulnerabilities for 2013, with several examples of the exploits in the wild.

Chapter 3 describes the technologies going into building the project, and how the different vulnerabilities are addressed, when relevant.

Chapter 4 describes the final discussions on how to think when starting a project security-first.

2 Development of Modern Web Applications

When developing in a modern context, in particular for the Internet, one needs to understand how to develop securely, efficiently and without repeating oneself. Though not as much is needed to understand the pertinent areas that this course covers, they are still important to understand the broader strokes.

2.1 Revision Control

Revision control – also known as version control or source control – is per [18] the concept of managing different versions of files, in the context of computer science most often a program of some kind.

In its' simplest form, it's backing up a file to another location before you do possible breaking changes on it, but in the more advanced forms it also allows for easy portability and encourages cooperation between developers.

Distributed vs Centralized Version Control Systems (VCS)

When it comes to software for version control, there are quite a few. Git [5] is the one I've used the most, which is a distributed VCS.

In both kinds, the developer works on local files that are then "checked into" the production repository. The main difference is that a distributed system takes a peer-to-peer approach (allowing the repository to be cloned from any other repository) whereas the centralized has one repository that all others answer to.

2.2 Dependency/Package Manager

"In software, a package management system, also called package manager, is a collection of software tools to automate the process of installing, upgrading, configuring, and removing software packages"[17]

Though the Wikipedia article mainly discusses package managers for operating systems, the same principles apply in web development. By stating packages that your project depends on – also known as dependencies – in an agreed-upon matter, the package manager can fetch them from where they are, rather than you needing to fetch them and put them in the right folder.

Ruby uses the gem Bundler, Frontend can use Bower, NodeJS uses Node Package Manager, Python uses pip (and/or easy_install), PHP has Composer and Pear, etc.

2.3 Separation of Concerns

Per [16] Multitier architecture is a "client–server architecture in which presentation, application processing, and data management functions are logically separated". In Web Development, that is most commonly generalized to three tiers:

Frontend

What is actually served to the browser, whether it's dynamically generated or static. It includes JavaScript, CSS, HTML, typography, graphics, etc. The user interface, or presentation layer. In this system, the least concerns have been put into the frontend, excepting details needed to show understanding of JavaScript-based security issues.

Middle

What is most commonly referred to as "web development", be it in Ruby on Rails, PHP, .NET or any other kind of server-side languages. The bulk of this system is written in PHP, using the framework Laravel.

Backend

Where the middle processing technologies generate the frontend consumed by the browser, in most dynamic systems it gets its' information – data – from the backend database or data store, whether through a MySQL database, a NoSQL MongoDB or any other kind of system, the backend provides access to the data, and should be properly secured.

2.4 oWasp top-10 Vulnerabilities 2013

oWasp.org [10] details the 10 most common and/or dangerous vulnerabilities. Most of these require a dynamic system, but having a static HTML site does not remove all the need to think in security.

A1-Injection

Injection is still one of the most common vulnerabilities despite the ease with which one can harden ones' system against that kind of intrusion. The most common case is that a system is too trusting, so it allows users to input data without ensuring that it is the right type, or has likely values. A humorous take on the issues comes from this old (in)famous comic [12], where the mother has named her son "Robert'); DROP TABLES Students;–", which led to a school losing its' student table.

When it comes to protecting against SQL injection in particular, the best way is to parametrise the connection and not put data into the database without ensuring that it can at least do no harm.

Injection is mainly protected against in the middle and backend.

A2-Broken Authentication and Session Management

The best example I can use here is a site I have worked with on occasion. It is written in PHP and MySQL and is going on ten years by now. Sessions are using only the native PHP drivers, and all the information on the user is sent from page to page in a combination of global variables, session variables and hidden form input(!). To my knowledge, there were at least three bugs related to these, where one I solved, the other two are too closely merged with the rest of the system.

1. There were two different login systems sharing the same space. One was for admin abilities, the other for basic user abilities. The variable `$username` was used in both but different values, so if one logged in as basic user and then did a particular admin action, the `$username` was overwritten.
2. Sessions times out at regular intervals, and as the system does not take that into account, whatever was being worked on ended up lost
3. By using FireBug or something similar to change the values on one of the login pages, a person could easily hijack another person's session and items just through manipulating the hidden form input.

Depending on how the authentication and session management is handled, this needs to be dealt with either in the frontend or middle. Most traditional web applications covers this in the middle layer, but if the application is written with a large focus on client side – IE. EmberJS –, as much care needs to be taken on the frontend.

A3-Cross-Site Scripting (XSS)

In a way, XSS is a frontend variant of SQL-injection, as the base problem is the same: An application is too trusting of whatever data it is being supplied, and because of this an attacker might be able to execute scripts on that page that will hijack users in various ways. A very common way is to exploit a comment system, where a very recent instance was with two of the most common cache plugins on WordPress, W3 Total Cache and WP Super Cache [3] and the way they interact with the `mfunc` function. If the blog allowed HTML in the comment system, it would by default also allow PHP.

A4-Insecure Direct Object References

To quote oWASP:

> A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key.

Generally, this is a failure in the middle and backend. The server shouldn't allow for directories to be browsed (at the very least, ensure that there is an index.html file in each directory), and configuration should not be accessible publicly. Several very common systems – PHPBB, MediaWiki, to name two – by default stores their config file in the root of the `public_html` folder, that is: the same folder that can be accessed through any browser. Of course, assuming that the config files does not echo anything out, it isn't necessarily that big a risk, unless a misconfiguration of the server leaves the site not recognizing the format, and thus prompting a download.

A5-Security Misconfiguration

Security misconfiguration comes in two large areas, where one is easier to deal with than the other, depending on your skill and to a certain extent budget.

- Framework/Middle configurations are the easiest ones for a web developer to secure. You should always have access to the newest versions of the components you use, and the ability to update them. Ensure that you understand the various options.
- Server/backend security misconfigurations are reasonably easy to use *if this is your server, or a server you have full access and rights to update software.* If this is shared hosting, you may end up being hit by a vulnerability in the system, such as this recent Ruby on Rails vulnerability[2].

A6-Sensitive Data Exposure

Any data that is sensitive needs to be protected, both through ensuring that the network can't be sniffed while its' in transit, and ensure that it's at least difficult to use any information stolen through other means, such as by getting a dump of the database.

A7-Missing Function Level Access Control

This can be generalised out a bit more than just Access Control, as the same misunderstanding can lead to other problems. The general idea of the misunderstanding is often two-folded:

1. All users use JavaScript
2. The only way an URL can be found is through a link on the site

Examples of the first leads to, for instance, only validating data using JavaScript, with the assumption that the data will now be correctly validated. Examples of the second leads to admin areas where as long as they know the URL, they can do anything. The better way, obviously, is to validate both clientside and serverside, and to have fail-safes that redirects the user back (or reports a 404) if they're straying.

A8-Cross-Site Request Forgery (CSRF)

A CSRF attack is one where the user's browser is tricked/forced into taking an action using its' credentials. As an example, a user posting an image with the source of an URL. As the browser fetches the page, it will also fetch that URL.

The most common – and quite easy to implement – is to use a so-called CSRF-token: a unique value sent along with a form submission so that it in the backend can be verified that it is a legitimate request.

A9-Using Components with Known Vulnerabilities

A good example of a component with known vulnerabilities – and a good case on why you should know what you're using! – is the plugin Timthumb for WordPress (the vulnerability is fixed in the latest versions, but it's important to be aware of this). Back in late 2011, Mark Maunder [8] had his blog hacked. The short of it is that 'Timthumb.php' allows site visitors to load images, and a caching mechanism to not need to re-process images. It will fetch a remote file and put it in a publicly accessible directory, where it was supposed to limit which domains one could fetch images from, but that was part of what was broken.

So, this was 2011. I've already mentioned that it's been fixed, why mention it more? Because there are still themes – free and paid for – that includes 'timthumb.php' and have not been update. I have personally dealt with a site that was hacked using an inactive theme *with that particular file*.

A10-Unvalidated Redirects and Forwards

The usecase I have found on unvalidated redirects and forwards is generally related to various affiliate sites, such as wanting all links to Amazon from your blog to go through a particular page for analytics reasons, and then spammers/phishers find the page and use it for their own nefarious purposes.

A better way to deal with that usecase is to either whitelist which domains are redirected to, or write a system where the url takes parameters that in the backend gets rewritten to the appropriate url.

3 Implementation

3.1 Technology Stack

Frontend

CSS

The base typographic styles comes from Typeplate[15], with headings in Quattrocento Sans and the body type being Libre Baskerville. The intention is to create a more pleasing design as a future step, but it is functional with a max-width of 600px.

The CSS is pre-processed using Sass and Compass, as well as the gem Breakpoint.

JavaScript

Very little JavaScript is required in this prototype. Modernizr, jQuery and Selectivizr are all concatenated and minified into the head script. The footer script has Magnific, a lightbox script that will be implemented to allow login on the same page.

HTML

There are two different templating engines in play on the site. The prototype of frontend functionality is built using Assemble.io and Handlebars templates, and Laravel serves Blade templates.

Grunt

GruntJS is a taskrunner like Make, Rake and similar ones, written in JS. It has several different tasks used for raising the quality of both code and the finished product.

- JavaScript tasks: Validates and JShints to ensure that the code style is fulfilled and that there are no syntax errors. Modernizr builds a lean, mean feature detecting machine.
- PHP tasks: Lints, sniffs the code and tests using PHPUnit
- Build tasks: Assembles pages, concatenates and minifies JS and CSS
- Test tasks: Runs Mocha and PHPUnit tests on both the frontend and middle

Bower

Bower is a package manager written in NodeJS, fetching components for the frontend, such as jQuery, Modernizr (for development, not production), Typeplate and Magnific.

Middle and backend

The site is written in Laravel, a "PHP Framework for Web Artisans"[14], using a MySQL database. Laravel is a MVC framework, using Blade templating language per default. It is fully PSR-0[11] compatible for class loading, and uses Composer as a dependency manager. In fact, cloning the "Laravel" project gives a starting point for a project, where you need to install the dependencies using Composer.

The relevant files for the backend are stored in `/app`. The directory `/app/src` contains the entirety of the frontend codebase, and any CSS-, image and JavaScript files used by Laravel are compiled into `/public_html/assets`. The directory `/app/storage` needs to be writable by the server, as that is where the cache, logs and views are written to. Sessions are stored in the database.

Controllers

There are five controllers currently, where only three of them are connected to routes. `Base` and `Authorized` are both used as parents, where Base sets up the CSRF-filter to be ran before all `post`-actions, and Authorized inherits Base, and also sets up actions that should have the `Auth::check()`-filter ran before them, IE when a user either *must be* or *must not be* logged into the application.

`Home` is the index page and also the page of policies. It will be further generalized to serve any kind of more "static" content, such as what would be referred to as "pages" in other contexts.

`Users` deals with all user actions. Users are seen as a resource in this system, with the login/logout routes added on top of that.

`Messages` is also a resource, though it only handles listing of messages and storing new messages currently.

Models

There are two different models used: `User` and Message. A third model, `Base`, sets up validation when an attempt to save the model occurs. The `User` model has a skeleton of a `boot` function that will perform actions after a User is created, such as sending an e-mail with login information, and logging the creation.

Every `User` can have several `Messages`, which is made easy using Eloquent, the default ORM in Laravel. An expansion on the system will use messages inheriting the base message model to set different types.

Config

All the configuration options of Laravel are stored as arrays in the `/app/config` directory. They can be fetched using the syntax `Config::('filename.keyname')`, and a similar syntax allows you to temporarly set a config option. Another large advantage of Laravel is that by adding new directories under the config directory, one can define different environments, such as testing (pre-defined), production, staging or similar.

Server

The current deployment server is a virtual host setup using Vagrant. It has Ubuntu 12.04LTS 32-bits, with MySQL 5.5 and PHP 5.4 under an Apache2 server. Laravel requires Mcrypt as part of its' core encryption functionalities, as well as PHP 5.4+.

3.2 Vulnerabilities

A1-Injection

The Database layer uses Eloquent. One of the large advantages to how Laravel is built is that it allows to change what kind of database is used. This system uses MySQL, but it is easy to change it to using SQLite or other drivers. This is doable because it uses PDO and prepared statements. Because it uses those, it is by default more secure against SQL Injection, since the prepared statements will only accept the proper arguments.

For instance, if the `username` column in a table is a string, and the data supplied from the user is "test); DROP TABLE users", that is exactly what the username column will recieve. It will not be interpreted as a command.

A2-Broken Authentication and Session Management

As the system is implemented (using Laravel default session management), the session expires when the browser is closed. It will also close the session if the user goes to the route `users/logout`, after which the user will be redirected back to the main page. However, it is important to know that there is a known feature/bug (per [6] in Chrome where if you have the setting "on startup" set to "continue where I left off", the session *will not* be destroyed.

The sessions are stored in the database, in a table called `sessions`, created by Laravel.

A3-Cross-Site Scripting (XSS)

There are two different routes that the system takes to protect against XSS. Fields that should not contain HTML (and which won't be hashed/encrypted) get `htmlspecialchars` used on them, whereas fields that are expected to contain HTML are stored as-is in the database.

Fields that are expected to contain a certain amount of HTML are then treated right before they're outputted. I have defined an HTML macro (a functionality of Laravel) called `markdown` to clean and transform the specified string. The macro is called in the blade template where it is relevant, so the original data is not transformed.

Note that users are discouraged to use HTML in their posts and such, and instead encouraged to use Markdown, a "text-to-HTML conversion tool for web writers"[7], using the original syntax to be able to work with client side scripts that output a preview of the markdown (the client side scripts are not implemented in this version).

The following steps are taken to protect against XSS:

1. Using a quite permissive list of allowed HTML elements (see `app/config/purifier.php`), it cleans the data, in particular removing "dangerous" attributes, using the HTMLPurifier, "a standards-compliant HTML filter library"[9]. The only allowed attributes are `href`, `class`, `width`, `height`, `alt` and `src`. Style is forbidden not because it is particularly dangerous, but because it risks messing up the design, and all the `on*` attributes are forbidden as JavaScript should not be allowed for authors. An important HTML element that is of course removed is `<script>`, for the same reasons that `on*` being forbidden.

2. PHP Markdown is then used to transform markdown into HTML. This includes some encoding into HTML entities, in particular code contained in codeblocks.

3. Final step of the `HTML::markdown` macro is unrelevant to protecting against XSS, but quite relevant for a more pleasing style, namely using PHP Typography to create "curly quotes" and similar things.

A4-Insecure Direct Object References

At current there are few places where this would be a concern. The system does not use the traditional definition of directories (being an MVC framework), with the exception of the `assets` directory. However, all directories are protected using Apache2 from being listed.

In a future implementation, there will be files that can be accessed, but they will all be stored above the public folder and served using routes to protect them from being exposed to the public.

A5-Security Misconfiguration

The weakest link at the moment is this one. Laravel supports setting up different configs depending on environment, which will be done before the system goes live but is not yet. One of the key components is that rather than showing an error trace in production, it will log the errors and report more cleaned-up ones to the user.

The beginnings of ensuring a repeatable development/deployment/testing/production environment has been set up using Vagrant and shell recipes, but not finalized.

A6-Sensitive Data Exposure

In this system, the e-mail is protected by being encrypted in the database, and using SSL throughout the more sensitive areas. The e-mail is considered "sensitive" as it is the only way to really identify the user, as any other personal information is limited and opt-in only. Another reason can be found in an article by Daniel Cid [4] from 2013 which revisits a hack of PHPBB.com in the beginning of 2009.

The hacker revealed several things, but the key for why I want the e-mails can be found in the following quote:

> So I login and see what I can come across, wow 400,000 registered emails, I'm sure that will go quick on the black market, sorry people but expect a lot of spam

A7-Missing Function Level Access Control

The current iteration of the system does not have more granulated user roles than logged in/guest, but the principles that are applied currently will apply once it does.

The controller `Authorized` uses filters defined in Laravel (the details of individual filters can be found in `app/filters.php`) to determine whether a request should be let through or not. Controllers inheriting `Authorized` (for instance the `Users` controller) defines a whitelist of actions that *does not* require a user to be logged in, and may optionally define a guestlist of actions that requires the user to *not* be logged in.

That is, unless a specified action is in the array of whitelisted actions, before the HTTP-request is served, Laravel checks whether the person is logged in. If they are not, they will be redirected to the `login` route. Once they've logged in, the system will attempt to return them to the page they originally attempted to reach. Examples of whitelisted routes are `login` and `users.create`.

On the other hand, if the action is whitelisted and in the guestlist, the system will check that they are not logged in before giving them access to it. After all, why should someone access the login route when they are already logged in?

A8-Cross-Site Request Forgery (CSRF)

Another of Laravel's builtin filters is a CSRF-token that is made automatically on each form created using Laravel HTML functions. The `Base` controller runs a similar filter to the `Authorized` controller, but where `Authorized` checks the user capabilities, `Base` runs the `csrf` filter on all post-actions. If the token does not match, it throws a TokenMismatchException.

The CSRF-token is also used in the logout process, where the logout links are appended with the token, which is then checked against the value of the user.

A9-Using Components with Known Vulnerabilities

To make it easier to keep track of which components are used where and to in the extension be able to write tasks to check everything against at least the more comprehensive databases, all components in PHP use Composer, and all frontend components use Bower.

One of the development components for PHP is the Security Checker[13] from SensioLabs, which using a gist by Barry vd. Heuvel[1] was made into a service for Laravel. To check the Composer components against Sensiolabs database, run `php artisan security:check`.

Obviously, running automated checks may not catch all vulnerabilities, but they're a good, automated tool to supplement occasional manual checks, and as all components are documented in either `composer.json` or `bower.json`, it is easier to track them. There is no mentioning of `package.json`, as the NPM modules are only used in development, not in a production environment.

A10-Unvalidated Redirects and Forwards

Not applicable, as no redirects use data supplied from the user to indicate destination.

4 Discussion

This was an interesting project to work on. Many of the vulnerabilities are ones I've been aware of for a long time, though others were new. In particular A10 never even occured to me, as its' base assumption (that I want to redirect to an external site through a page on my own) is almost alien to me.

The vulnerability that always surprises me the most is A1, as its' one I've personally been aware of for many years and have been migating in my code since at least 2006.

The largest change from how I've written code in earlier years comes from using Yeoman, Bower, Composer and Grunt, starting the project with not an empty file – or even a couple of empty files with a Plan –, but with a project of several files to be merged and implemented as I worked. It's also the first project I've been using code sniffers as extensively as I have been here, enforcing PSR-2 coding style on all but a few (in particular the generated classes for database migrations) subsets of classes.

4.1 Frameworks

I chose to implement it in Laravel, as re-writing the wheel is more likely to introduce bugs due to misunderstandings about basic demands. That said, when using a framework as the basis for a project, it is very important that one a) understands the code behind it and b) use it as it is intended to be used. If a framework stifles your workflow and you find yourself fighting against it more than it helps, it is not the framework to use.

I have personal experience with both CodeIgniter, Kohana and Laravel, where CI is too compatible with older installations, and breaks the promises of the query string in a way I am not comfortable with. Kohana is nice with a broad range of functions, but in working with it I've often had to work despite it, rather than with it. Laravel is a more comfortable fit as it encourages modularisation (as evidenced by encouraging the use of Composer), doesn't break the promise of the query string functioning despite its' pretty URLs, and the syntactic sugar is pretty.

Bibliography

[1] Barry vd. Heuvel. *Symfony Security Checker in Laravel.* Fetched the 19 October 2013. URL: https://gist.github.com/barryvdh/6696739.

[2] Bogdan Calin. *Hackers exploit Ruby on Rails vulnerability to compromise servers, create botnet.* Fetched the 19 October 2013. URL: http://www.pcworld.com/article/2040062/hackers-exploit-ruby-on-rails-vulnerability-to-compromise-servers-create-botnet.html.

[3] Bogdan Calin. *WordPress Caching Plugins Remote PHP Code Execution.* Fetched the 19 October 2013. URL: http://www.acunetix.com/blog/web-security-zone/wp-plugins-remote-code-execution/.

[4] Daniel Cid. *Security Archive – Case Study: phpbb.com Compromised.* Fetched the 19 October 2013. URL: http://blog.sucuri.net/2013/09/security-case-study-archive-phpbb-com.html.

[5] Git Community. *Git –distributed-even-if-your-workflow-isnt.* Fetched the 19 October 2013. URL: http://git-scm.com/.

[6] Google Groups User. *Chrome is not deleting temporary cookies.* Fetched the 19 October 2013. URL: https://productforums.google.com/d/msg/chrome/9l-gKYIUg50/HOvdZbPiuXAJ.

[7] John Gruber. *Daring Fireball: Markdown.* Fetched the 19 October 2013. URL: http://daringfireball.net/projects/markdown/.

[8] Mark Maunder. *Technical details and scripts of the WordPress Timthumb.php hack.* Fetched the 19 October 2013. URL: http://markmaunder.com/2011/08/02/technical-details-and-scripts-of-the-wordpress-timthumb-php-hack/.

[9] MeWebStudio. *HTML Purifier.* Fetched the 19 October 2013. URL: http://htmlpurifier.org/.

[10] oWasp. *Top 10 2013-Top 10.* Fetched the 19 October 2013. URL: https://www.owasp.org/index.php/Top_10_2013-Top_10.

[11] PHP Framework Interoperability Group. *PHP FIG-standards accepted.* Fetched the 19 October 2013. URL: https://github.com/php-fig/fig-standards/tree/master/accepted.

[12] Randall Munroe. *Exploits of a mom.* Fetched the 19 October 2013. URL: http://xkcd.com/327/.

[13] SensioLabs. *Security Checker.* Fetched the 19 October 2013. URL: https://packagist.org/packages/sensiolabs/security-checker.

[14] Taylor Otwell. *Laravel PHP Framework*. Fetched the 19 October 2013. URL: `http://laravel.com/`.

[15] Typeplate contributors. *Typeplate » A typographic starter kit encouraging great type on the Web*. Fetched the 19 October 2013. URL: `http://typeplate.com`.

[16] Wikipedia. *H.264/MPEG-4 AVC*. Fetched the 19 October 2013. URL: `http://en.wikipedia.org/wiki/Multitier_architecture`.

[17] Wikipedia. *Package Management System*. Fetched the 19 October 2013. URL: `http://en.wikipedia.org/wiki/Package_management_system`.

[18] Wikipedia. *Revision Control*. Fetched the 19 October 2013. URL: `http://en.wikipedia.org/wiki/Revision_control`.