www.phparch.com

July 2019
Volume 18 - Issue 7

php[architect]

Find The Way With Elasticsearch

Elasticsearch—There and Back Again

Add Location Based Searching to Your PHP App With Elasticsearch

Defensive Coding Crash Course

The Devilbox and Docker

ALSO INSIDE

PHP[WORLD] 2019

Washington D.C., Oct 23-24

Join us this fall for our annual php[world] conference. This year marks the 25th anniversary of PHP's creation and we plan to celebrate it in a big way!

Panels on the history & future of PHP and some amazing keynotes yet to be announced!

Save **$200** in our Early Bird sale through August 3rd

Register today!

world.phparch.com

We’re Hiring!

Learn more and apply at automattic.com/jobs.

Automattic wants to make the web a better place. Our family includes **WordPress.com**, **Jetpack**, **WooCommerce**, **Simplenote**, **WordPress.com VIP**, **Longreads**, and more. We’re a completely distributed company, working across the globe.

AUTOMATTIC

Defensive Coding Crash Course

Mark Niebergall

Ensuring software reliability, resiliency, and recoverability is best achieved by practicing effective defensive coding. Take a crash course in defensive coding with PHP as we cover attack surfaces, input validation, canonicalization, secure type checking, external library vetting, cryptographic agility, exception management, automatic code analysis, peer code reviews, and automated testing. Learn some helpful tips and tricks and review best practices to help defend your project.

In this article, we focus on coding techniques and PHP language features which can be used to help increase the defensive posture of an application. Increasing the defensiveness of the code increases the ability to mitigate offensive attacks and leads to higher application stability.

Defensive Coding

Historically, many people associate PHP with being insecure and exploitable. The language has shed a lot of insecure settings and practices. PHP affords a simpler syntax and has a reputation as easier to learn than some other languages. As a result, the code can be written and released without authors implementing or fully understanding defensive coding practices. While being simple to use and understand is a significant benefit for the language, it also means developers need to have a better understanding of proper techniques and approaches to creating code that is resistant to bad behaviors and bugs.

When referring to the term "defensive," the meaning is applied to code that assumes the worst is going to happen and correctly handles those scenarios. That assumption drives application development that makes no assumptions. The application fails safe and properly handles unexpected problems. Countermeasures are put in place to improve security and reduce the likelihood of a successful attack. This work takes a concerted effort across a broad range of development concepts, from how data is handled to logging users and how developers build and review code changes.

As we go through these different domains, we'll emphasize using built-in and readily available PHP language constructs.

Attack Surface

To determine where to focus on strengthening your defensive measures, it's crucial first to identify the attack surface. The attack surface is the sum of all points of a possible attack. The larger the attack surface, the harder it is to defend. For example, if an application has many public-facing APIs, then each of those APIs is included in the attack surface. If one of those public-facing APIs is to reset a password on a user, consider how that API could be leveraged to reset a password on an account by an unauthorized actor. If an application imports files that are provided by various third-parties or end-users, then those file importers are included as attack surfaces a malicious user could use to gain access to an application. Uploads could include imported CSV files or profile pictures, or any other file a user can upload. If an application accepts user input from forms, then each of those forms is a point to consider. Some common forms to start looking at include login, contact, comments, feedback, and user profiles.

Data Types

A core principle in PHP is correctly using data types. With PHP being a loosely-typed language where variables can change data types, this can be a bit harder to accomplish fully. With recent language changes, including method return types, the data types are more predictable than they used to be. With the incoming class property data types in PHP 7.4, this becomes even easier to achieve.

There are eight data types available. Can you name them all? They fall into three categories: 4 scalar, 2 compound, and 2 special. Scalar types, which include Boolean, integer, float, and string, along with the special type of null, are great for storing most simple values. The compound type array is handy, but be cautious with overuse since its structure is not rigid. Objects are ideal for a vast majority of logic and workflows. The special type of resource is commonly encountered when using system resources, such as a database connection or interacting with a system file. Within PHP internals, there is a slow and gradual move away from using resources and instead uses classes.

Scalar	Compound	Special
boolean	array	null
integer	object	resource
float		
string		

Using the appropriate data type for data within an application is where defensive coding practices kick in. If a value is an `id` field that is always an integer, then an integer should be used

to represent that in PHP. Similar best practices apply for float, boolean, string, and null values.

Parameter type hinting should be used whenever possible. If a setter method exists for a value that is an integer, then hinting the type as integer should be used to coerce an integer to be passed in. If declare(strict_types=1); is used to enable strict typing, then that typing becomes enforced. Enable strict types whenever possible to improve data consistency and platform predictability (when used with proper exception management). Without strict types enabled, the value goes through typecasting, which may lead to unexpected values. Use ? to indicate a value that can be null. Typehint to classes whenever possible as well to enforce passing in a specific class instance as an argument. The class typehint can be the actual class, an abstract that it extends, an interface the class implements, or a trait the class uses.

```
public function setSomeId(int $id) {...}
public function setFinancialValue(float $value) {...}
public function setName(string $name) {...}
public function setOptionalProperty(string
?$nullableProperty) {...}
public function setService(Service $service) {...}
```

Likewise, the method return type should always be declared to maximize code predictability. Increasing predictability also increases the defensive rating.

```
public function getSomeId(): int  {...}
public function getFinancialValue(): float {...}
public function getName(): string {...}
public function getOptionalProperty(): ?string {...}
public function getService(): Service {...}
```

Use the value null correctly. An integer value of 0 has a very different meaning than a value of null. A string that is not set should be null because an empty string could mean the user supplied no value or left an optional field blank. Use null to identify a value that has not been set yet or the lack of a value.

Input Validation

Building on the core concepts of data types is validating all input. Input includes data coming into an application from end-users, file uploads, integration partners, APIs, and other resources. Treat all inbound data should as not trusted; therefore, validate all inbound data. A big part of data validation is filtering input. The saying "filter input, escape output" rings true and is a question to ask during development. If the data is input, has it been filtered? If the data is output, has it been escaped?

Filtering data encompasses only accepting data which passes specific criteria. If it doesn't match the criteria, then it gets filtered out. The age of a person is always an integer, but is there a filter for maximum value? If a person's age is inputted as 200, that is not okay and should be rejected, as should any negative value. Valid values would be a range of between 0 and 128, depending on how old you can accept as valid.

When creating filters, PHP provides filter_var to help make filtering easier. Using filter_var, a variable can be filtered out if it doesn't meet the passed in filtering rules and options. The signature for filter_var is:

```
filter_var ( mixed $variable [, int $filter = FILTER_DEFAULT
             [, mixed $options ]] ) : mixed
```

To filter a person's age, you could use filter_var($age, FILTER_VALIDATE_INT) to confirm the value is an integer. A common data point that is difficult to validate is an email address, but with filter_var($email, FILTER_VALIDATE_EMAIL) it becomes simple. If the return from filter_var is a Boolean false, then the input can be rejected as invalid. All the filters available are:

Filter	Description
FILTER_VALIDATE_BOOLEAN	Returns TRUE for 1, "true", "on", and "yes" and FALSE otherwise.
FILTER_VALIDATE_DOMAIN	Validates the domain against various RFCs.
FILTER_VALIDATE_EMAIL	Validates e-mail addresses against the syntax in RFC 822 with some exceptions.
FILTER_VALIDATE_FLOAT	Validates a value as float, and then converts to float if valid.
FILTER_VALIDATE_INT	Validates value as integer and you can specify a min and max range of allowed values.
FILTER_VALIDATE_IP	Validates value as IP address and allows excluding reserved ranges.
FILTER_VALIDATE_MAC	Validates value as MAC address.
FILTER_VALIDATE_REGEXP	Validates against a custom regular expression
FILTER_VALIDATE_URL	Validates a URL based on RFC 2396

Along with the minimum and maximum range of a value—such as the age of a person—the length and content should be further scrutinized. When appropriate, filter out data based on the length and the content. If a value should be between 10 and 12 characters, then apply a length check. The simplest way to check the length is with strlen()[1], which may be sufficient, but when character encoding and internationalization comes up, there are some edge cases that do not work. Use mb_strlen()[2] instead, which counts single and multibyte characters as one character of length.

1 strlen(): http://php.net/strlen

2 mb_strlen(): http://php.net/mb_strlen

Another filtering option is to blacklist values or to whitelist values. Blacklists explicitly deny or block particular values. Whitelisting refers to allowing only specific values. Whitelisting is the ideal filtering method, although usually not the most practical. Blacklists tend to need constant maintenance as users find ways around forbidden values.

Canonicalization

Standardizing on terminology and data formatting is covered with canonicalization. Within an organization, using a specific term instead of others is common, and that known term should be used. The same applies to using chosen formats for data.

Consider the format of a date. Internationally we see dates formatted as m/d/Y, d/m/Y, and Y/m/d. These dates need to be normalized, or canonicalized, into a standard format. For a date, this most likely would be formatted as the International Organization for Standardization (ISO) 8601 standardized format, which is Y-m-d (e.g., 2019-12-31 for December 31, 2019). The application filtering would need to know the inbound format and transform the value into the selected format. Use `date('c')` to get the current ISO-8601 date with time and timezone offset. The PHP documentation for the `date` function[3] provides a full list of available date and time formats.

The same rule applies when converting to data types. If users can submit various values that ultimately map to a Boolean `true` or `false`, then multiple terms should be accepted. For example, an input of `1`, `true`, `yes`, or `on` may all translate to `true` and anything else translates to `false`.

Library Vetting

External libraries should all be vetted to ensure they are secure and reliable. With PHP's dependency manager Composer making package inclusion simple, adding libraries to a project is easier than ever. With that ease of inclusion also comes the responsibility for the developer to consider the library's impact on the application's defensive posture.

Before including a library, research the project and see how it measures up to expected standards. You can go through and ask each of these questions to gauge the viability of the project:

1. How fast does the project address bugs?
2. How does the project address security issues?
3. Does the project have automated tests?
4. What level of code coverage do automated tests cover?
5. How popular is the project in terms of use and contributors?
6. Does the source code appear to be well-written and follow best practices?
7. When was the last time the project was updated?
8. With what versions of PHP is the project compatible?
9. Is the dependency free from known security vulnerabilities? SensioLabs Security Checker[4], which is included with Symfony, can check for this.

These are all important considerations to help choose useful libraries that do not open up exploits or unnecessarily introduce vulnerabilities in your application.

Cryptographic Agility

Cryptography is a continually changing realm. Both the attackers and the defenders are making improvements. Being able to adapt to these changes easily is cryptographic agility. Staying on newer PHP versions and current libraries is an easy way to help improve this agility rating. It is imperative that you do not use your algorithms or "roll your own crypto." Researchers dedicate entire careers to making secure algorithms.

Consider the routine task to securely validate passwords to authenticate users. Using a broken hashing algorithm or using a key-based encryption approach are insecure ways to address this problem. Old hashing algorithms, like MD5, SHA1, and Panama, can lead to cracking the passwords if the data is breached. Using encryption instead of hashing can expose the passwords if the keys can be found. PHP has made securely handling passwords straightforward with `password_hash()`[5] and `password_verify()`[6]. Use `password_hash` to hash a password, and use `password_verify` to check if a password entered by a user matches the hash of their actual password. It has a default hashing algorithm designed to change over time. In PHP 5.5+, it is blowfish (bcrypt), but since then other algorithms have been added, and the default could change. There is a `password_needs_rehash` to help identify if a password needs to be rehashed with the current algorithm. Then, store the hashed value in a database or whatever data storage tool is being used.

```
$hashedPassword = password_hash($plainTextPW, PASSWORD_DEFAULT);
// ...
$isValid = password_verify($plainTextPW, $hashedPassword);
```

Another common need during development is for cryptographically secure pseudo-random generated (CSPRNG) values. PHP has `random_bytes` and `random_int`; if a random token string is needed, this can be generated with:

```
$token = bin2hex(random_bytes($tokenLength));
```

For a CSPRNG integer value, the below will work to generate that value:

```
$number = random_int($minLength, $maxLength);
```

3 `date` *function:* https://php.net/function.date

4 *SensioLabs Security Checker:* https://github.com/sensiolabs/security-checker

5 `password_hash()`: http://php.net/password_hash

6 `password_verify()`: http://php.net/password_verify

If more advanced cryptography is needed, PHP has a couple of great ways to securely do this. PHP 7.2 added the Sodium extension to PHP core. The OpenSSL extension has been available for quite some time and is also actively maintained. A common use of both of these is for symmetric and asymmetric cryptography, which involves creating keys and using those keys to encrypt and decrypt data. Symmetric-key encryption uses a shared private key to both encrypt and decrypt data and relies on keeping the key secret from others. Asymmetric-key encryption uses a private key and a public key. The private key is used by the data owner to encrypt the data, and the public key is then used to decrypt the data. The process can also go the other way, with the public key being used to encrypt data, and the private key being able to decrypt the data.

A practical example of this is a secure website that has HTTPS using TLS, where the server has the private key, and website users receive the public key to decrypt data coming to them. The website users also use the public key to encrypt data sent back to the website server. This way, the data transferred back and forth is encrypted.

> *Be aware the* `mcrypt` *extension no longer ships with PHP core as of 7.2. If your applications still rely on the functionality it provides, you need to update it to use another crypto library.*

Exception Management

In software development, never assume unexpected behaviors won't happen. Writing code that handles these problems helps provide a stable platform. Exception management deals with how these problems are handled when they occur. The use of `try` and `catch` in PHP provides a clean way to do this.

Consider a function that makes an API call to an external entity. Many things could go wrong, ranging from the server having downtime to a programming error to an unparsable response. Without exception handling, the code may look something like Listing 1.

Listing 1

```
1. public function callApi($uri, Request $request): ?Response
2. {
3.     $response = $this->getApiClient()
4.                      ->makeRequest($uri, $request);
5.
6.     if ($response === null || $response->getCode() === 500
7.         || $response->getCode() !== 200) {
8.
9.         return null;
10.    }
11.
12.    return $response;
13. }
```

Listing 2

```
1. public function callApi($uri, Request $request): Response {
2.     $response = $this->getApiClient()
3.                      ->makeRequest($uri, $request);
4.
5.     if ($response === null) {
6.         throw new NullResponseException('Response was null');
7.     } elseif ($response->getCode() !== 200) {
8.         throw new NotOkayResponseCodeException(
9.             'Response code was ' . $response->getCode()
10.        );
11.    }
12.    return $response;
13. }
```

The problem with this is we don't know what went wrong given the response. Perhaps it is useful to know if there was a different response status code, or there is an alternative URI a request can be sent to if there is a problem. Exception handling can be used to make this better. This practice includes using different custom exceptions so the handling code can know what went wrong. See Listing 2.

Automated Code Analysis

Automated code analysis tools can be set up to run against a code repository. Analysis types can range from code statistics to code styling to development patterns to code dependencies. These tools can be run locally during the software development process and are commonly coupled with Continuous Integration (CI). Running these tools gives insights into code health and alignment with project goals.

There are many PHP code analysis tools available. A good list can be found at https://github.com/exakat/php-static-analysis-tools. Some highlights include:

- **PHPStan:** https://github.com/phpstan/phpstan automatically finds bugs within code.
- **PHPCS:** https://github.com/squizlabs/PHP_CodeSniffer a variety of code sniffers to check code styling and compatibility for upgrading PHP versions.
- **PHPLOC:**https://github.com/sebastianbergmann/phploc for measuring project code structure.
- **SensioLabs Security Checker:** https://github.com/sensiolabs/security-checker to check dependencies for known security vulnerabilities.

Try out a few tools, see what works for your projects, and what fits your needs.

Peer Code Reviews

Peer code reviews provide a formalized way to share knowledge and catch hard-to-find problems with code. These types of reviews occur when a peer reviews a pull request. The code changes are examined, and the peer might manually test the

code. Peer code reviews should be used to complement, not replace, automatic code analysis, and automated tests. Most source code repository hosts have a code review tool to facilitate peer code reviews before the code being merged in.

Having a peer who understands an application do a thoughtful review of code changes can identify business logic problems, security vulnerabilities, cascading impacts of the changes, scalability concerns, and best practices missed. Meaningful comments and constructive critiques can improve a whole team or community. Sharing knowledge on reviews helps lift everyone to higher levels that cannot be achieved individually and helps spread project knowledge, so there is not one person who knows a part of a project. Conversations can be started on how to improve an application. There are many benefits to having a good peer code review process in place.

There are some drawbacks of peer code reviews. Syntax problems may not be identified, so automated analysis is important. Comments made in writing tend to be interpreted in the worst possible way, so take time to have conversations instead, and be aware that the reviewer most likely has good intentions. Peer reviews are not a replacement for automatic tests.

Automated Testing

Having automated tests, both unit and behavioral, can significantly improve code quality, decrease defects, and allow for refactoring with confidence. For unit tests, good choices include PHPUnit and Codeception. Behavioral tests can be written using Behat, PHPSpec, Codeception, or others. The testing framework depends on project needs and may be influenced by the PHP framework being used.

To achieve Continuous Integration (CI), and Continuous Deployment (CD), automated tests are an absolute must. Having CI and CD positions a project to better respond defensively to any emerging threats. Code changes can be made, tests executed to validate the changes, and code then deployed into production.

Pulling Everything Together

With so many domains within defensive coding practices, there is a lot to take in. Improving your defensive posture takes time and energy. Experience running into these problems and finding suitable solutions helps make addressing future problems easier and more efficient. From cryptographic agility to automated testing to using the right data type, using these concepts together builds up the defenses. The saying "defense in depth" refers to having multiple components coming together to improve the defense.

Similarly, using the concepts covered in this article, along with the many other defensive concepts that were not covered, come together to defend against the inevitable problems. This quick crash course in defensive coding is intended to refresh what you may have already learned or experienced, put some terminology to concepts you may already be familiar with, and introduce you to a variety of defensive domains. My hope is you consistently implement changes over time to improve not only the projects you are involved with but also to improve your professional skill set as a developer.

Mark Niebergall is a security-minded PHP software engineer with over a decade of hands-on experience with PHP projects. He is the Utah PHP User Group Co-Organizer and often speaks at user groups and conferences. Mark has a Masters degree in MIS, is CSSLP and SSCP cybersecurity certified, and volunteers for (ISC)2 exam development. Mark enjoys going for long runs and teaching his five boys how to push buttons and use technology. @mbniebergall

Related Reading

- *Strong Security Stance in the New Year* by Eric Man. Jan. 2019. https://phpa.me/security-corner-jan-2019
- *Securing Your Site in Development and Beyond* by Michael Akopov. Jan. 2018 https://phpa.me/jan18-securing-dev
- *The Life-Changing Magic of Tidying Your Code* by Bryce Embry. May 2019. https://phpa.me/embry-tidying-code

The Devilbox and Docker

Gunnard Engebreth

Starting a new project has its highs and lows; setting up your dev environment should not be one of them. Back in the day, a dev could spend a couple of days just getting a *nix environment up and running on a machine and even then, it probably did not mirror the production environment in the least! The advent of virtual machines (VMs) helped. Now, we could use something like Vagrant to manage several environments on one machine. With Docker and the Devilbox, we have a fantastic source for spinning up a development environment rapidly while not skimping out on complete customization.

127.0.0.1 is where the heart is.

As humans, we feel comfortable in places we know. It could be your house or apartment, the local coffee shop you can bike to with those gluten-free, vegan, scones, or maybe even at your desk at work (no judgment). What makes these places comfortable, no matter how busy or crowded, is the fact that your expectations about the general atmosphere, experience, and result rarely rock your world. The occasional new employee or "fill-in-the-blank" that is missing is just something to adjust to. Now, as developers, we don't just want comfort, we want control.

When our team is assigned a new project, we tend to overlook a significant step. We can look at the scope, desired delivery date, technologies involved, etc. and come up with a reasonable timeline with milestones and update meetings, but the hidden possibility of technical debt is right in front of us, the dev environment. Unless you have a bulletproof(ish) system in place to set up, develop, test, collaborate, version control, and deploy—there could be a day or two of work ahead of you. In this article, I show how to use the Devilbox, Docker, and Git to go from zero to 100 percent as a team resource for developing your next project.

What's in the Box?

The Devilbox[1], at first glance, could be looked at as just another MAMP or XAMP, but don't let that fool you. The DevilBox is a modern and highly customizable dockerized PHP stack supporting LAMP and MEAN across all major platforms. With Devilbox you can develop standard PHP (Laravel, Drupal, legacy code, WordPress) as well as more frontend (Node.js, Angular, Vue.js) apps with the ability to easily share this dev environment or container with your team. This methodology benefits your workflow as well as compartmentalizing your code and services (Apache, NGINX, MySQL, and others.) for each specific project. Instead of only working on one project on your host machine via standard install of web and database services, you can specify each configuration within the respective containers and safely move between them without compromising the infrastructure. Another benefit is that when you are working on a project in a team environment, you can distribute the exact project container to collaborators to ensure everyone works within the same setup. This practice takes out the "it works on my machine" fallacy that creeps up so often when hunting down bugs.

Off the shelf, or Git repo rather, Devilbox comes with most daemons that a majority of developers would need. Of course, you can install any "specific-to-your-project" software needed once you have the basics up and running.

Apache (2.2–2.4) and NGINX (stable, mainline) are the go-to web servers offered. PHP (5.2–7.3) comes along as well as a majority of the most used modules. Having multiple versions of PHP at your command allows for the ability to safely test and fix your code, especially when moving legacy code to a newer version of PHP. This can separate the patching time when your team can access `brokenwebsite.dev` to see all the errors while keeping your main repo and dev server clean of new fixed code on `workingwebsite.dev`. But we will get into that a bit later.

As far as databases are concerned we have:

- MariaDB (5.5–10.3)
- PerconaDB (5.5–5.7)
- PgSQL (9.1–10.0)
- Redis (2.8–4.0)
- Memcached (1.4.21–latest)
- MongoDB (2.8–3.5)

MailHog (1.0–latest) and RabbitMQ (3.6–latest) are included as additional services if need be. Like previously stated, this is the out-of-the-box configuration, and while it contains a powerhouse of tools, typically these are all not needed together. Devilbox has a solution for that and allows you to pick and choose which ones you want to enable when you sping up your containers.

If you want to get down to the nitty-gritty and take charge of Devilbox, you need to step on over to the `.env` file. It is where you get the play god… err, the devil. Let us start at the "images" section. This section is located around line #240. Here you can comment/uncomment out which version of each service you would like to run on your environment.

1 *Devilbox:* http://www.thedevilbox.org

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[**architect**]

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe