**Securing Input and Output Data**

In order to protect your website's users and the website itself, you should protect your web applications from malicious input and operations. Application security is one of the pillars of a reliable application. This should not be overlooked; on the contrary, you must have security in mind all the time while developing an app.

While most of the focus (if not all) is directed toward the user input, it would be much better if the data was validated no matter the source. This is especially needed when, on a project, there is a team involved and not one single person. This can lead to lots of unpredictable events, such as code changes that may look inoffensive, but could trigger unexpected behavior in your application's flow. Imagine a class method that has been designed and is used for some internal logic process, but then ends up being used for processing *external* data (from the database, user input, or elsewhere). While the class' self-data may have some degree of trust, at least when it comes to the data type (depending on the design), the external data is not to be trusted. In some cases, working on a product in a small team, it is tempting to ask the application administrators to insert data in a specific format here and there, leaving data validation and sanitization for *later*, while you eagerly try to deliver more and more features (perhaps to meet a deadline). Then, imagine your product turns out to be so successful that management decides to extend the business and offer it as a SaaS solution. In this case, the application administrators are no longer your small team, and all clients' data will be at risk if you don't deal with the input validation and sanitization. This time, it will be pretty difficult to solve all the issues in a timely manner – you will have to find these security holes across the whole application.

In general, not taking care of data validation and sanitization will lead to a great technical debt in the future, as you will not only put your clients' data at risk, but application operations could return unpredictable results, which will require the developer to have to trace and debug the issues, which, again, takes time and money, while these bugs cause poor user experience.

**Best Practices**

Here are a few coding practices that will make your PHP code less prone to bugs and security issues:

* Use a single entry point for your web app:

This is about a single PHP file that is responsible for taking every HTTP request and processing it. This file would bootstrap all the dependencies, load the configuration files, initialize the request handlers (such as **Dispatcher**, **HttpKernel**, and others – note that each framework uses its own name), and will then route the request to the proper PHP script for this to produce the response. In our examples, we have used several input files to provide some examples; this is not the way to go for real-world applications. Later, we will look at an example of a simple bootstrap for the examples run through in this topic, inside a single input file, keeping each example file on disk.

* Separate the business logic from presentation logic:

It is always better to keep responsibilities separate from each other. Modern frameworks bring their own templating engines to help developers keep most (if not all) of the business logic in PHP files, rather than in presentation files. This helps to focus on only one part; that is, either gathering and/or processing data or displaying data (that is, through visuals). Additionally, it is easier to read business logic if it is not scattered all over the presentation markup. I will explain this in more detail in the bootstrap example later.

* Sanitize and validate your input early and escape it late:

Input data refers to data outside the application, be it user input, database data, filesystem file data, or other data. By sanitizing the data, you make sure you get the cleanest possible data for a given input, while by validating it, you make sure you allow the script to work with the accepted values or range of values. On the other hand, escaping the data for the output makes the application avoid some other issues such as **cross-site scripting** (**XSS**).

We'll see how this can be done in PHP shortly.

* Use type hinting whenever possible:

Using type hinting, you can be sure of the input and output type of a function, so this feature prevents code execution when the input or output data of a function is not the expected type. For example, if your function expects an iterable, but a string was passed, then the engine will throw a **TypeError** exception (which stops the script execution if it is not caught).

That's not all. By default, PHP will coerce the values of variables that do not match the expected type, when possible. This only applies to scalars. For example, if a function expects an integer but a numerical string is passed, then it will be converted to an integer. PHP features strict type checking as well, which I advise you to use in your application development. It can be added as per file use, and it's enough to add **declare(strict\_types=1);** and only apply it to function calls from the file on which the strict types were enforced. This means that a function call from non-strict type checking to a function from a file with strong type checking enabled, the caller's preference of weak typing will be respected, and the values will be coerced. Using strict type checking makes your application even less prone to bugs, and that's simply because **'123abc' == 123**, which leads me to the next point.

* Use strict comparison (**===**):

PHP supports two types of comparisons: loose comparisons (**==**) and strict comparisons (**===**). In the case of loose comparisons, PHP tries to align both operands' values to a common type, and then perform the comparison. That's why **0 == FALSE** evaluates to **TRUE**. While this is considered a feature of PHP, praised for being friendly to starter developers, I strongly advise you to avoid such a construct from the beginning. On the other hand, a string comparison will not try to coerce the operands' data, as it compares both values and types.

Generally speaking, you, as a developer looking at your code, should know what data you are dealing with in every line of your application.

In other words, the more magic you allow to drive your app, the more your app will be prone to *magic* bugs!

* Split your code into smaller pieces:

Try to avoid writing big long functions and instead try to split the code into pieces that you will be able to actually test. So, what granularity should you use to split your code? Well, just ask what you are trying to do with the data, then it will come down to functions with names such as **decorateComment**, **splitCollection**, **shouldTrim**, and others. If you end up with something such as **getCommentsByGroupingAndDecoratingLongOnes**, you'll probably find that function does too many operations, which could be split into shorter, more manageable and testable functions.

* Avoid using the error suppression operator, **@**:

This operator is pretty slow, as the PHP will turn off error reporting, and after the operation, it will restore the error reporting to the original value. Additionally, do not turn off error reporting at all, not even in production; instead, use a custom error handler and log the error in a preferred manner, so you can see whether something goes wrong during the code execution.

**Sanitizing and Validating the User Input**

As soon as the data arrives in a script, it should be sanitized, and it must always be validated. You want to make sure you don't receive harmful data and, therefore, you want to clean the user input, which means removing potentially malicious content from the provided input, or casting the data to a specific type such as an integer or Boolean. Additionally, you want to make sure the input data is a valid number, or an email address where expected, and so on.

The built-in **filter\_input()** function is used to process the data from the request and, if needed, will alter it to match the expected format.

The syntax is **filter\_input( int $type, string $variable\_name, int $filter = FILTER\_DEFAULT, mixed $options = null )**, so it takes as arguments the type of input to look into, the input parameter name to look for, the optional filter type, and any extra options if needed. What **FILTER\_SANITIZE\_\*** filters do is remove data that is not expected for specific formats. For example, **FILTER\_SANITIZE\_NUMBER\_INT** will remove everything except digits and plus and minus signs. This is the full list of [sanitizing options](http://php.net/manual/en/filter.filters.sanitize.php).

There are some more sanitization functions you should be aware of:

* **strip\_tags()**: This strips the HTML tags from a string; for example, **strip\_tags('Hello <script>alert(1)</script>!');** will remove the **<script>** opening and closing tags, resulting in the following output: **"Hello alert(1)!"**. This removes the HTML tags where they are not expected and removes potentially dangerous scripts from being stored in the application, which may be output further in the browser causing malicious actions.
* **trim()**: This strips whitespace characters by default, or other characters as specified, from the beginning and end of a string.

Here are some functions that you may want to use to validate your data:

* **is\_numeric()**: This tells us whether a variable is a number or a numeric string.
* **preg\_match()**: This performs a regular expression match.
* **in\_array()**: This checks whether the value exists in the list of values in the array that is given as an argument of the function.