Appendix E

Security and Validation

These topics were introduced in Chapter 8 in the context of learning the design and application of selection structures. Security protects against malicious attempts to inject scripts that will compromise users, while validation is concerned with ensuring that all input data meets the application requirements. These are both critical concerns for all application development. Once you start to develop your applications online, you must ensure that your code **always** contains the necessary safeguards and tests to protect your users, data sources, and requirements, from malicious attacks and invalid data. This appendix will describe some important safeguards (for Web input and output, and MySQL databases), introduce some important PHP security and validation functions, and demonstrate the use of regular expressions to validate specific string formats.

Security and Cross Site Scripting (XSS)

Cross Site Scripting (XSS) is an attack where a client can interact with a Web server in order to compromise other clients. It is important to protect your applications against XSS attacks. To understand the high degree of risk, let's begin with a harmless example where an XSS attack is carried out on an unprotected application (like those in your samples folder). The **city-weather** application in the **xss** folder under your **samples** folder allows you to either submit a city and today's weather, or view the weather of cities already submitted.

Open **city-weather.html** and click the link to view the cities. Now submit a city, and view the list again. When you submit the form, your input (the city and its weather) is sent to the **$_POST** array, and **city-weather.php** processes the form, sends back a confirmation page, and saves the city and weather to a file named **weather.txt**. If you click the **Weather Report** link, **weather-report.php** is opened, which reads the cities from the file and displays them on a Web page. As you can see this is a very simple application with no purpose other than to demonstrate how an insecure application can experience an XSS attack.

Open **city-weather.html** and this time paste the following script into the **City** input box (the **city-weather.html** page includes this text for your use, so you can just copy and paste it into the form):

```
<script>alert("XSS")</script>:Sunny
```

This time when you press the submit button an alert message will display on your screen with the message **"You have been hacked!!"**. What happened? In this case, when the form was submitted to **city-weather.php**, the "**city**" field contained a malicious program script, written in Javascript. The **$_POST** array received the script, which was then assigned to a variable named **$city**. The program then (assuming the script was a name of a city) appended the script and the weather to **weather.txt**, and sent a new page to your Web browser to confirm you that the city and weather had been recorded. However the city that was sent back to your browser was actually the script, which your browser automatically executed. The script delivered a "You are hacked!!" alert message but could have been coded to do something worse. This is an example of XSS **reflection**, where a program inadvertently **reflects** a malicious script back to a Web client by confirming the content that had been submitted from the form.

But note that the script was not **only** reflected back to the same client, it was also **stored**, in **weather.txt**. The **weather-report.php** program is designed to read the list of cities and weather from this file and send these as a Web page to any Web browser that requests it. One of the "cities" in weather.txt is now a malicious script, and when any Web browser renders the Web page that weather-report delivers, it will inadvertently execute the script. This is even more dangerous than the first example, because every user who opens **weather-report.php** will receive the script. This is an example of a **stored XSS**, where the script is stored on the Web server (in a file or database or other data source), and then inadvertently sent to any browser that requests a page that includes this data.

In this case the script simply generated a harmless message, but among other things, the script could have been designed to access the user's session information, or cookies, which can be used to compromise the user. As a Web developer you have a responsibility to ensure that XSS attacks do not succeed.

How can you protect against XSS attacks? The simplest option is to **entirely remove** any HTML that has been inserted into your program input, **before** the input is processed. This will remove any script tags; it also often a good idea to remove quotes and some other special characters that might be manipulated to modify strings.

The process of removing or modifying characters to protect against scripts or other attacks is known as **sanitizing**. The PHP's **filter_var**() function is a powerful multi-use function that will allow you to conduct a number of different sanitizing or validation operations depending on the parameter that you provide. Using this function is similar to using the trim() function, which you have already used to remove leading and training white space from an input string, for example:

```
$city = trim($_POST["city"]);
```

You can use the **filter_var**() function in a similar way. Here is a statement to **remove ALL** HTML tags (including script tags) from a $_POST array element, and return the sanitized string:

```
$city = filter_var($_POST["city"], FILTER_SANITIZE_STRING);
```

In this example, **two** arguments are sent to **filter_var**(): the first argument is the **string** that you want to be sanitized (in this case **$_POST**[**"city"**]), the second is a code that specifies what type of operation is to be performed. The code **FILTER_SANITIZE_STRING** specifies removal of all HTML tags, and by default will also convert quotes to HTML entities (see Chapter 17 for more about entities). In fact, the function will treat any character sequence as a tag if it begins with a < symbol, and will remove this symbol and the characters that follow, until a closing > is found. If the closing > is not found all remaining characters in the string will be removed.

In general you can sanitize most of your program input by applying **filter_var**() with the **FILTER_SANITIZE_STRING** filter to each input value (for example to all elements in the $_POST array). Since it is also a good idea to remove any leading or trailing white space from all input values, you can combine the use of **filter_var**() with your use of the trim() function. For example to trim **and** sanitize **$_POST**[**"city"**]:

```
$city = filter_var(trim($_POST["city"]), FILTER_SANITIZE_
    STRING);
```

This should be your standard approach to sanitize most program input from untrusted sources.

However there are cases where your sanitizing procedure must be adapted to meet your application requirements. For example, by default, the filter **FILTER_SANITIZE_STRING** will not only cause all HTML tags to be removed from a string, as mentioned above, it will **also convert single and double quotes** to HTML entities. For example my last name **O'Kane** contains a single quote and would be converted to **O'Kane**. If this was then displayed on a Web page the browser would correctly convert **&#39**; back to a single quote, but if the value was stored in a file or database, it would be stored as **O'Kane**. A flag can be added to **FILTER_SANITIZE_STRING** to specify that quotes should be left unchanged:

```
$city = filter_var($_POST["city"],
  FILTER_SANITIZE_STRING,FILTER_FLAG_NO_ENCODE_QUOTES);
```

Other flags can be applied to make other adjustments. However before altering the default behavior of this operation, do your own research and be sure that the modifications you are making will still satisfy your security and application requirements.

Furthermore, it is not **always** possible to use the filter **FILTER_SANITIZE_STRING** to remove all HTML from an input string: what if your application requirements **allow** the user to enter HTML (for example to submit a blog entry, or update a profile). In these cases, there are other ways to use filter_var(), to **convert** dangerous characters, such as the '<' and '>' symbols, to HTML entities, for example by using the **FILTER_SANITIZE_SPECIAL_CHARS** filter:

```
$city = filter_var($_POST["city"], FILTER_SANITIZE_
    SPECIAL_CHARS);
```

The **filter_var**() function also provides other sanitizing options. In all cases do careful research before deciding on your best course of action. If you are not sure what you are doing, follow the basic procedure described above, or ask for help.

Here is a list of filters that can be applied when using filter_var():

```
https://www.php.net/manual/en/filter.filters.sanitize.php
```

Protecting MySQL Databases

Stripping input of HTML tags provides protection against XSS, but a different approach must be taken to protect your MySQL databases against injections. If data from a Web form (or any untrusted source) is to be included in a MySQL query, the best protection will be to use MySQL **prepared statements**. A prepared statement separates the structure of a query from the data that is to be included, which makes it impossible for an attacker to inject malicious SQL. Prepared statements are easy to create and are also the best approach for coding queries since they are very efficient. The **w3schools** site provides a good introduction to SQL injection and prepared statements, with code examples to get you started:

```
https://www.w3schools.com/sql/sql_injection.asp
https://www.w3schools.com/php/php_mysql_prepared_statements.asp
```

Validating Input

You now know how to **sanitize** your program input, the next step is to **validate** the input. Validation rules are an important part of every application's requirements document and should specify what tests and associated actions should be applied to each input. Some of these rules will apply to most, or all, inputs and applications, for example: any **required fields** must have in fact been submitted and contain a value; **numeric** fields must contain a valid number; special-purpose strings such as **email addresses**, **phone numbers**, **social security numbers**, **state abbreviations**, **zip codes, and URLs** must conform to the correct formats. Other validation rules will be specific to a particular input, for example: a numeric input, such as an age, must be within a certain **range; a** sales code or date must conform to a prescribed **format**; or an input value description must be of, or below, a certain **length**.

PHP provides a number of standard functions designed to handle common validation needs but you must also develop your own code to handle more specific tests. While you have done some validation as you worked through this book, validation tests on a working application must be thorough and complete, and every input value must be tested in all ways necessary to ensure that the application will work correctly. Any validation errors will usually need to be reported, either directly to the user, or in an error log of some kind.

Here is a sequence of tests that you are likely to need in order to validate each input value:

If the input is **required** (not optional), the PHP **isset**() function will test if a variable has been set, for example **if** (**isset**(**$_POST**[**"city"**]). If the variable has either not been declared or has been assigned **NULL** this function will return **false**, otherwise **true**. The PHP **empty**() method should then be used to test whether or not a variable contains a value, for example **if** (**!empty**(**$_POST**[**"city"**]).

The PHP **is_numeric**() function will test if a value is numeric, for example if (**is_numeric**(**$_POST**[**"age"**]).

You have already seen how the PHP **filter_var**() function can be used to **sanitize** data; it can also be used to **validate** certain types of data, for example **filter_var**(**$email, FILTER_VALIDATE_EMAIL**) will test if a variable named $mail contains a **valid email** address. The **filter_var**() function will also validate URLs, integer values, floating point values and more. For a complete list see:

```
https://www.php.net/manual/en/filter.filters.validate.php
```

Beyond these more standard tests, using available PHP functions, you will also need to develop code of your own to perform customized validation that are more specific to your application requirements, such as checking that a number is within a certain range, or that a character sequence contains one of a number of allowed codes. In some of these cases, **regular expressions** can provide another useful validation tool.

Regular Expressions

Your application may work with input that must conform to a special format of some kind. For example: dates might be required to be in a certain format, such as "year-month-day"; passwords might be restricted to letters and numbers and a few special characters, and might also be required to not exceed a maximum length; employee IDs might be all numeric digits and 8 characters long; similarly product codes might be constructed of 3 upper-case letters, followed by 6 numbers followed by an optional lower-case letter. In all these cases, **regular expressions** can be applied to ensure that the input value meets the requirements. A regular expression (**or regex**) is constructed using a formal language to define a **search pattern** that can then be applied to any character string. Regular expressions are utilized for a wide range of search operations, including input validation.

As a simple example, here is a regex search pattern that will accept product codes that meet the requirement listed above and reject any strings that fail to fit the pattern:

```
[A-Z]{3}[0-9]{6}[a-z]?
```

This pattern is defined as follows: exactly 3 upper-case letters, followed by exactly 6 numeric digits, followed 0 or 1 lower-case letters.

The PHP **preg_match**() function can be used to apply a regular expression to a value and will return 1 if the value meets the requirements of the search patter, 0 otherwise. For example:

```
$pattern = "#[A-Z]{3}[0-9]{6}[a-z]?#";

$value1 = "TEC672452w"
$value2 = "TEC672452"
$value3 = "TEC67w452"
$value4 = "TEC67w"

preg_match($pattern, $value1) will return 1
preg_match($pattern, $value2) will return 1
preg_match($pattern, $value3) will return 0
preg_match($pattern, $value4) will return 0
```

Note that the expression stored in **$pattern** begins and ends with '#'. PHP uses "Perl-Compatible Regular Expressions" (**PCRE**) syntax, which requires a delimiter at the start and end of the expression. A delimiter can be any non-alphanumeric, non-backslash, non-whitespace character. Commonly used delimiters are forward slashes (/), hash signs (#) and tildes (~).

This is just a simple example; regular expressions are very sophisticated and skilled. Many excellent tutorials and reference are available online.

Useful Code Samples

The w3schools site provides a very helpful walk through of a basic Web form validation process:

```
https://www.w3schools.com/php/php_form_validation.asp
```

The validation code for this example will provide a useful template for your own form-processing code: find it under the "Try It" link, or go to (the complete URL should be a single line):

```
https://tryphp.w3schools.com/showphp.php?filename=demo_form
    _validation_complete
```

NOTE: PHP also provides two other functions that perform similarly to certain configurations of ***filter_var****() to* ***sanitize*** *data: the* ***htmlspecialchars****() function will take any string as an argument and, by default, will return the same string, except that any occurrences of the following characters will have been replaced with their HTML entity equivalents: double quotes (****"****), the "less than" symbol '<' (****<****), the "greater than '>' symbol (****>****), and the ampersand '&' character (****&****); a companion function,* ***htmlentities****(), will convert* ***ALL*** *characters that have equivalent HTML entities to their entity designations.*