**Youssef Kassab**
**Computer & Communications Engineer**
+961 3 38 41 00
me@youssefkassab.me – www.youssefkassab.me



# WEB APPLICATIONS SECURITY RECOMMENDATIONS

## VERSION 1.0

**CREATION DATE: 1 FEBRUARY 2009**

**REVISED DATE: 6 FEBRUARY 2009**

**Youssef Kassab**
**Computer & Communications Engineer**
+961 3 38 41 00
me@youssefkassab.me – www.youssefkassab.me



# Table of Contents

**Youssef Kassab**
**Computer & Communications Engineer**
+961 3 38 41 00
me@youssefkassab.me – www.youssefkassab.me



# I - Introduction

The Internet is filled with people trying to make a name for themselves by breaking your code, crashing your site, posting inappropriate content, and otherwise making your day interesting. It doesn't matter if you have a small or large site; you are a target by simply being online, by having a server that can be connected to. Many cracking programs do not discern by size, they simply trawl massive IP blocks looking for victims. Try not to become one [1].

When on a shared host, security simply isn't going to be as strong as when on a dedicated host. This is one of the tradeoffs for the inexpensive fee [2]. Shared hosting environments perhaps ought to be considered from the security mindset in the same fashion as a compromised system (that which has or may have been already cracked into) [3].

In order to secure our web applications I read some resources in order to identify different types of vulnerabilities and what are the measures that should be taken to protect our applications. In the first part I will talk about vulnerabilities and measures that should be taken for protection. In the second part you will find a checklist to be used when developing any application. In the appendixes you will find examples about filtering, validations and other security related issues.

[1] http://us3.php.net/manual/en/security.general.php

[2] http://phpsec.org/projects/guide/5.html

[3] http://www.phpwact.org/security/web_application_security

# II - Vulnerabilities & Measures

In this section I will list different types of vulnerabilities, explain them and define methods to protect our applications.

## 1- Form Processing

**Spoofed forms submission:**
The attacker can alter the form as desired—whether to eliminate a maxlength restriction, eliminate client-side data validation, alter the value of hidden form elements, or modify form element types to provide more flexibility. These modifications help an attacker to submit arbitrary data to the server, and the process is very easy and convenient—the attacker doesn't have to be an expert.

Although it might seem surprising, form spoofing isn't something you can prevent, nor is it something you should worry about. As long as you properly filter input, users have to abide by your rules.

**Spoofed HTTP requests:**
A more sophisticated attack than spoofing forms is spoofing a raw HTTP request. This gives an attacker complete control and flexibility, and it further proves how no data provided by the user should be blindly trusted.

How can an attacker modify the raw HTTP request? The process is simple. Using the telnet utility available on most platforms, you can communicate directly with a remote web server by connecting to the port on which the web server is listening (typically port 80).

As with spoofed forms, spoofed HTTP requests are not a concern as long as we reinforce the importance of input filtering and the fact that nothing provided in an HTTP request can be trusted.

## 2- Cross-Site Scripting

Cross-site scripting (XSS) is one of the best known types of attacks. It plagues web applications on all platforms, and PHP applications are certainly no exception. Any application that displays input is at risk—web-based email applications, forums, guestbooks, and even blog aggregators. In fact, most web applications display input of some type—this is what makes them interesting, but it is also what places them at risk. If this input is not properly filtered and escaped, a cross-site scripting vulnerability exists.

How can this happen? If you display content that comes from any external source without properly filtering it, you are vulnerable to XSS. Foreign data isn't limited to data that comes from the client. It also means email displayed in a web mail client, a banner advertisement, a syndicated blog, and the like. Any information that is not already in the code comes from an external source, and this generally means that most data is external data.

- By validating all external data as it enters and exits your application, you will mitigate a majority of XSS concerns.

- Letting PHP help with the filtering. Functions like htmlentities(), strip_tags(), and utf8_decode() can be useful. With the simple addition of htmlentities(), the page will become much safer. It should not be considered completely secure, but this is probably the easiest step you can take to provide an adequate level of protection. [$message = htmlentities($_GET['message']);]

- Verifying the length and also ensuring that only valid characters are allowed

- Using a naming convention to identify cleaned variables

## 3- Cross-Site Request Forgeries

A cross-site request forgery (CSRF) is a type of attack that allows an attacker to send arbitrary HTTP requests from a victim. The victim is an unknowing accomplice—the forged requests are sent by the victim, not the attacker. Thus, it is very difficult to determine when a request represents a CSRF attack. In fact, if you have not taken specific steps to mitigate the risk of CSRF attacks, your applications are most likely vulnerable.

There are a few things you can do to protect your applications against CSRF:

- Use POST rather than GET in forms. Specify POST in the method attribute of your forms. Of course, this isn't appropriate for all of your forms, but it is appropriate when a form is performing an action, such as buying stocks. In fact, the HTTP specification requires that GET be considered safe.

- Use $_POST rather than rely on register_globals. Using the POST method for form submissions is useless if you rely on register_globals and reference form variables like $symbol and $quantity. It is also useless if you use $_REQUEST.

- Force the use of your own forms.
  The biggest problem with CSRF is having requests that look like form submissions but aren't. If a user has not requested the page with the form, should you assume a request that looks like a submission of that form to be legitimate and intended?

Here is an improved message board:

```
<?php

session_start();

if (isset($_POST['message']))
{
if (isset($_SESSION['token']) && $_POST['token'] == $_SESSION['token'])
   {
      $message = htmlentities($_POST['message']);

      $fp = fopen('./messages.txt', 'a');
      fwrite($fp, "$message<br />");
      fclose($fp);
   }
}

$token = md5(uniqid(rand(), true));
```

```
$_SESSION['token'] = $token;

?>

<form method="POST">
<input type="hidden" name="token" value="<?php echo $token; ?>" />
<input type="text" name="message"><br />
<input type="submit">
</form>

<?php

readfile('./messages.txt');

?>

$ name = htmlentities($_POST['name'], ENT_QUOTES, 'UTF-8');
$ comment = htmlentities($_POST['comment'], ENT_QUOTES, 'UTF-8');
```

By including a token in your forms, you practically eliminate the risk of CSRF attacks. Take this approach for any form that performs an action.

## 4- Databases and SQL

Databases are used to store data in dynamic applications and this is a location we need to protect from attacks. One simple solution is to place all files containing the connection to the database outside of document root, and this is a good practice. Both include and require can accept a filesystem path, so there's no need to make modules accessible via URL.

It is not a good idea to your connection files processed by the PHP engine. This includes renaming your modules with a .php extension as well as using AddType to have .inc files treated as PHP files. Executing code out of context can be very dangerous, because it's unexpected and can lead to unknown results. However, if your modules consist of only variable assignments (as an example), this particular risk is mitigated.

Below is a method for protecting your database access credentials. Create a file, /path/to/secret-stuff, that only root can read (not nobody):

SetEnv DB_USER "myuser"
SetEnv DB_PASS "mypass"

Include this file within httpd.conf as follows:

Include "/path/to/secret-stuff"

Now you can use $_SERVER['DB_USER'] and $_SERVER['DB_PASS'] in your code. Not only do you never have to write your username and password in any of your scripts, the web server can't read the secret-stuff file, so no other users can write scripts to read your access credentials (regardless of language). Just be careful not to expose these variables with something like phpinfo() or print_r($_SERVER).

On another hand applications should never connect to the database as its owner or a superuser, because these users can execute any query at will, for example, modifying the

schema (e.g. dropping tables) or deleting its entire content. The most required privileges should be granted only.

### Protecting stored data in databases

The easiest way to work around this problem is to first create your own encryption package, and then use it from within your PHP scripts. PHP can assist you in this with several extensions, such as Mcrypt and Mhash, covering a wide variety of encryption algorithms. The script encrypts the data before inserting it into the database, and decrypts it when retrieving.

In case of truly hidden data, if its raw representation is not needed (i.e. not be displayed), hashing may also be taken into consideration. The well-known example for the hashing is storing the MD5 hash of a password in a database, instead of the password itself. See also crypt() and md5().

### SQL Injections

Protecting against SQL injections:

- Never trust any kind of input, especially that which comes from the client side, even though it comes from a select box, a hidden input field or a cookie.
- Never connect to the database as a superuser or as the database owner. Use always customized users with very limited privileges.
- Check if the given input has the expected data type. PHP has a wide range of input validating functions, from the simplest ones found in Variable Functions and in Character Type Functions (e.g. is_numeric(), ctype_digit() respectively) and onwards to the Perl compatible Regular Expressions support.
- If the application waits for numerical input, consider verifying data with is_numeric(), or silently change its type using settype(), or use its numeric representation by sprintf().
  *$query = sprintf("SELECT id, name FROM products ORDER BY name LIMIT 20 OFFSET **%d;**",$offset);*
- Quote each non numeric user supplied value that is passed to the database with the database-specific string escape function (e.g. mysql_real_escape_string(), **sql_escape_string()**, etc.). If a database-specific string escape mechanism is not available, the addslashes() and str_replace() functions may be useful (depending on database type). See the first example. As the example shows, adding quotes to the static part of the query is not enough, making this query easily crackable.
- Do not print out any database specific information, especially about the schema, by fair means or foul. See also Error Reporting and Error Handling and Logging Functions.
- You may use stored procedures and previously defined cursors to abstract data access so that users do not directly access tables or views, but this solution has other impacts.

## 5- Session Fixation

Session security is a sophisticated topic, and it's no surprise that sessions are a frequent target of attack. Most session attacks involve impersonation, where the attacker attempts to gain access to another user's session by posing as that user.

The most crucial piece of information for an attacker is the session identifier, because it is required for any impersonation attack. There are three common methods used to obtain a valid session identifier:

- Prediction
- Capture
- Fixation

Fixation is the simplest method of obtaining a valid session identifier. While it's not very difficult to defend against, if your session mechanism consists of nothing more than session_start(), you are vulnerable.

```
<?php

session_start();

if (isset($_SESSION['HTTP_USER_AGENT']))
{
    if ($_SESSION['HTTP_USER_AGENT'] != md5($_SERVER['HTTP_USER_AGENT']))
    {
        /* Prompt for password */
        exit;
    }
}
else
{
    $_SESSION['HTTP_USER_AGENT'] = md5($_SERVER['HTTP_USER_AGENT']);
}

?>
```

Just remember to make things difficult for the bad guys and easy for the good guys.

### Cookie Theft

One risk associated with the use of cookies is that a user's cookies can be stolen by an attacker. If the session identifier is kept in a cookie, cookie disclosure is a serious risk, because it can lead to session hijacking.

Protecting your users from cookie theft is therefore a combination of avoiding crosssite scripting vulnerabilities and detecting browsers with security vulnerabilities that can lead to cookie exposure. Because the latter is so uncommon (with any luck, these types of vulnerabilities will remain a rarity), it is not the primary concern but rather something to keep in mind.

Enabling SSL is a particularly useful way to minimize the exposure of data being sent between the client and the server, and this is very important for applications that exchange

sensitive data with the client. SSL provides a layer of security beneath HTTP, so that all data within HTTP requests and responses is protected.
If you are concerned about the security of the session data store itself, you can encrypt it so that session data cannot be read without the appropriate key. This is most easily achieved in PHP by using session_set_save_handler( ) and writing your own session storage and retrieval functions that encrypt session data being stored and decrypt session data being read. See Appendix C for more information about encrypting a session data store.

### Session Hijacking

The most common session attack is session hijacking. This refers to any method that an attacker can use to access another user's session. The first step for any attacker is to obtain a valid session identifier, and therefore the secrecy of the session identifier is paramount. The previous sections on exposure and fixation can help you to keep the session identifier a shared secret between the server and a legitimate user. The principle of Defense in Depth can be applied to sessions—some minor safeguards can offer some protection in the unfortunate case that the session identifier is known by an attacker. As a security-conscious developer, your goal is to complicate impersonation. Every obstacle, however minor, offers some protection.

The key to complicating impersonation is to strengthen identification.

## 6- Shared Hosts

When on a shared host, security simply isn't going to be as strong as when on a dedicated host. This is one of the tradeoffs for the inexpensive fee.

One particularly vulnerable aspect of shared hosting is having a shared session store. By default, PHP stores session data in /tmp, and this is true for everyone. You will find that most people stick with the default behavior for many things, and sessions are no exception

What's a better solution? Don't use the same session store as everyone else. Preferably, store them in a database where the access credentials are unique to your account. To do this, simply use the session_set_save_handler() function to override PHP's default session handling with your own PHP functions.

The following code shows a simplistic example for storing sessions in a database: (example here http://phpsec.org/projects/guide/5.html)

The best solution is to use a dedicated host.

## 7- Browsing the Filesystem

The script "browse.php" in APPENDIX I represents an example on how to browse the filesystem.

The safe_mode directive can prevent this particular script, but what about one written in another language?

A good solution is to store sensitive data in a database and use the technique mentioned earlier (where $_SERVER['DB_USER'] and $_SERVER['DB_PASS'] contain the access credentials) to protect your database access credentials.

## 8 – Email Header Injections

This attack consists of a change in the email header sent from a PHP script. This can be done if the input of the user is directly included in the FROM section. To secure the vulnerability to email headers injections we have many methods, one of them is by adding after the line

```
$from=$_POST["email"];
```

The following code:

```
if (eregi("\r",$from) || eregi("\n",$from)){
die("Why ?? :(");
}
```

You can see that the script is ended using the die() function if the email contains "\r" or "\n". "\n" corresponds to LF or 0x0A/%0A in hexadecimal, this means a new line and "\r" corresponds to CR or 0x0D/%0D in hexadecimal or "Carriage Return", this means a return to the beginning of the line.

The most important thing is to know that after the FROM statement is located the injection of code and this is the part to secure.

## 9- Data in Web Root Vulnerability

Sensitive information available in public web server document root:

- Store all sensitive data (all non-essential to page or application functionality preferably) outside of the web root (perhaps right below it in a subdirectory).
- Deny access to configuration or other data via directives in your web server.

Older software versions or backup copies of applications that exist in the publically-accessible web document root may be at risk of exploitation.

## 10- CGI Scanning Attack

Scanning and traversing URLs and web links in an attempt to find executable scripts or programs on a web server.

## 11- Session Files on Shared Server Vulnerability

Use session_set_save_handler() to redefine the way session data is stored.

## 12- Common File Name Vulnerability

Easily "guessable" file or application structures lead to possible compromises. For example, placing your admin section in an /admin/ directory (URI) makes your application an easier target for CGI or directory scanning.

Deny access to confidential file via directives in your web server, and include it in outer file (by php's include()) with public access

## 13- File Upload Attacks

Sometimes you want to give users the ability to upload files in addition to standard form data. Because files are not sent in the same way as other form data, you must specify a particular type of encoding—multipart/form-data:

```
<form action="upload.php" method="POST" enctype="multipart/form-data">
<p>Please choose a file to upload:
<input type="hidden" name="MAX_FILE_SIZE" value="1024" />
<input type="file" name="attachment" /><br />
<input type="submit" value="Upload Attachment" /></p>
</form>
```

PHP provides two convenient functions for mitigating these theoretical risks: is_uploaded_file( ) and move_uploaded_file( ). If you want to verify only that the file referenced in tmp_name is an uploaded file, you can use is_uploaded_file( ):

```
<?php
$filename = $_FILES['attachment']['tmp_name'];
if (is_uploaded_file($filename))
{
/* $_FILES['attachment']['tmp_name'] is an uploaded file. */
}
?>
```

If you want to move the file to a more permanent location, but only if it is an uploaded file, you can use move_uploaded_file( ):

```
<?php
$old_filename = $_FILES['attachment']['tmp_name'];
$new_filename = '/path/to/attachment.txt';
if (move_uploaded_file($old_filename, $new_filename))
{
/* $old_filename is an uploaded file, and the move was successful. */
}
?>
```

Lastly, you can use filesize( ) to verify the size of the file:

```
<?php
$filename = $_FILES['attachment']['tmp_name'];
if (is_uploaded_file($filename))
{
$size = filesize($filename);
```

```
}
?>
```

The purpose of these safeguards is to add an extra layer of security. A best practice is always to trust as little as possible.

# III - Development Checklist

- Set register_globals to OFF in the php.ini file

- Validate submitted data (all user inputs): Items on Forms, in URLs and so on. Anything that came in as data and needs to go out as a part of an HTML page should be HTML encoded, ideally so that only the alphanumerics are unencoded. Add htmlentities() to the input that will be displayed in an HTML page. Text items should have limits and validations on them (data types). Client side validation can be applied (JavaScript) but it's not enough since the attacker can create his own version of the file to escape from JavaScript validation, server side validation is crucial. Never run unescaped queries.

- Use reCaptcha when submitting registration information or any critical information

- Initialize variables ***$form_valid = false;***
  *Statement;*
  ***if ($form_valid)***
  *{ Statement; }*

- No customer ID numbers in URLs, no more GET method to be used unless really needed

- Don't derive the name or location of the file from the user-supplied data.

- Use non standard file and folder names (no more folders named images or admin or Connections)

- Implement use of SSL when important data is being transferred

- Adding hidden variables containing a value that should be validated on submit is sometimes a good idea but not in all cases

- Add an empty hidden div and check if filled then reject page (if filled than the one who submitted this form is a robot)

- Hide all kinds of error information that can be useful for attackers. Use error messages that doesn't give important clues to attackers "incorrect username or password" instead of "invalid username" and hide all errors and replace them with "contact support" (error_reporting = E-ALL should be used in php.ini but error_display should be set to off)

- Change permissions on any configuration files containing private information such as database passwords or email accounts to 440 so they cannot be written to. If you need to edit them at a later time you will need to change it back to 640.

- Access Control: You don't want the user to have access to any Admin function or Clean up scripts. Protect the admin section using password on the directory if possible.

- The .htaccess file is used to deny access to your site or files. (you can also deny IPs using IP Deny Manager tool in the cpanel)

- Hide the programming language you are using. PHP can parse any valid script whatever its name is. In order to hide our programming language from attackers we can use .htm in place of .php when developing and in the .htaccess file we can tell the server to treat .htm files as php files. We can change our file extension by adding this line to the .htaccess or turn it on via the Apache Handlers in the cPanel (AddHandler application/x-httpd-php5 .html)

- To protect against SQL injection attacks we need to use this PHP function: mysql_real_escape_string(). This function escapes (makes safe) any special characters in a string for MySQL.
  *Example: $name = $_POST['name']; $safe_name = mysql_real_escape_string($name);*
  *Now you know the variable $safe_name, is safe to use with your SQL code.*
  We can also use parameterized queries or stored procedures. We should avoid building SQL commands through concatenation at almost any cost.

- Hide the PHP code: store your PHP files and the necessary passwords to access your MySQL databases in protected files or folders. Put the database access passwords in a file with a .inc.php extension (such as config.inc.php), and then place this file in a directory which is above the server's document root (public_html) (and thus not accessible to surfers of your site). Then, refer to the file in your PHP code with a require_once command. By doing things this way, your PHP code can read the included file easily but hackers will find it almost impossible to hack your site.

# IV - Conclusion

Securing web applications is very important for the businesses. It is the role of developers to ensure right measures are taken during the development of applications and along with the service providers inform customers about the importance of the investment in securing web applications and its implication on their businesses.

Offering high security standards represents a plus for a company creating websites for businesses especially that the number of web based attacks is increasing with the increase of critical and important data accessible through web applications.

# Appendix I: Miscellaneous

## 1- Browse the filesystem *(Browse.php)*

```
<?php

echo "<pre>\n";

if (ini_get('safe_mode'))
{
   echo "[safe_mode enabled]\n\n";
}
else
{
   echo "[safe_mode disabled]\n\n";
}

if (isset($_GET['dir']))
{
   ls($_GET['dir']);
}
elseif (isset($_GET['file']))
{
   cat($_GET['file']);
}
else
{
   ls('/');
}

echo "</pre>\n";

function ls($dir)
{
   $handle = dir($dir);

   while ($filename = $handle->read())
   {
      $size = filesize("$dir$filename");

      if (is_dir("$dir$filename"))
      {
         if (is_readable("$dir$filename"))
         {
            $line = str_pad($size, 15);
            $line .= "<a href=\"{$_SERVER['PHP_SE LF']}?dir=$dir$filename/\">$filename/</a>";
         }
         else
         {
            $line = str_pad($size, 15);
            $line .= "$filename/";
         }
      }
      else
      {
         if (is_readable("$dir$filename"))
         {
            $line = str_pad($size, 15);
            $line .= "<a href=\"{$_SERVER['PHP_SELF']}?file=$dir$filename\">$filename</a>";
```

```
            }
            else
            {
               $line = str_pad($size, 15);
               $line .= $filename;
            }
         }

         echo "$line\n";
      }

      $handle->close();
   }

   function cat($file)
   {
      ob_start();
      readfile($file);
      $contents = ob_get_contents();
      ob_clean();
      echo htmlentities($contents);

      return true;
   }

   ?>
```

## 2- Data Filtering

Data filtering is the cornerstone of web application security in any language and on any platform. By initializing your variables and filtering all data that comes from an external source, you will address a majority of security vulnerabilities with very little effort. A whitelist approach is better than a blacklist approach. This means that you should consider all data invalid unless it can be proven valid (rather than considering all data valid unless it can be proven invalid).

- Ensure that data filtering cannot be bypassed,
- Ensure that invalid data cannot be mistaken for valid data, and
- Identify the origin of data.

### Filtering Examples

It is important to take a whitelist approach to your data filtering, and while it is impossible to give examples for every type of form data you may encounter, a few examples can help to illustrate a sound approach.

The following validates an email address:

```
<?php

$clean = array();

$email_pattern = '/^[^@\s<&>]+@([-a-z0-9]+\.)+[a-z]{2,}$/i';

if (preg_match($email_pattern, $_POST['email']))
{
```

```
   $clean['email'] = $_POST['email'];
}

?>
```

The following ensures that $_POST['color'] is red, green, or blue:

```
<?php

$clean = array();

switch ($_POST['color'])
{
   case 'red':
   case 'green':
   case 'blue':
      $clean['color'] = $_POST['color'];
      break;
}

?>
```

The following example ensures that $_POST['num'] is an integer:

```
<?php

$clean = array();

if ($_POST['num'] == strval(intval($_POST['num'])))
{
   $clean['num'] = $_POST['num'];
}

?>
```

The following example ensures that $_POST['num'] is a float:

```
<?php

$clean = array();

if ($_POST['num'] == strval(floatval($_POST['num'])))
{
   $clean['num'] = $_POST['num'];
}

?>
```

You should never make a practice of validating data and leaving it in $_POST or $_GET, because it is important for developers to always be suspicious of data within these superglobal arrays.

Initializing your variables is such a good practice.

## 3- Tips & Tricks

- If you use a directory index file such as index.php (instead of dispatch.php), you can use URLs such as http://example.org/?task=print_form. This way we will have only one php file and this file will access other files

- Having error_reporting set to E_ALL will help to enforce the initialization of variables, because a reference to an undefined variable generates a notice.

- Most of directives in the php.ini file can be set with ini_set(), in case you do not have access to php.ini or another method of setting these directives.

- By using **open_basedir** you can control and restrict what directories are allowed to be used for PHP. You can also set up apache-only areas, to restrict all web based activity to non-user, or non-system files.
  When a script tries to open a file with, for example, fopen() or gzopen(), the location of the file is checked. When the file is outside the specified directory-tree, PHP will refuse to open it. All symbolic links are resolved, so it's not possible to avoid this restriction with a symlink. If the file doesn't exist then the symlink couldn't be resolved and the filename is compared to (a resolved) *open_basedir*.
  Under Windows, separate the directories with a semicolon. On all other systems, separate the directories with a colon. As an Apache module, open_basedir paths from parent directories are now automatically inherited.
  The restriction specified with open_basedir is actually a prefix, not a directory name. This means that "open_basedir = /dir/incl" also allows access to "/dir/include" and "/dir/incls" if they exist. When you want to restrict access to only the specified directory, end with a slash. For example: "open_basedir = /dir/incl/"
  The default is to allow all files to be opened.

## 4- Safe mode

When safe_mode is on, PHP checks to see if the owner of the current script matches the owner of the file to be operated on by a file function or its directory. For example:

```
rw-rw-r--   1 rasmus   rasmus      33 Jul  1 19:20   script.php
rw-r--r--   1 root     root        16 May 26 18:01 /etc/passwd
```

Running script.php:

```
<?php
 readfile('/etc/passwd');
?>
```

Results in this error when safe mode is enabled:

*Warning: SAFE MODE Restriction in effect. The script whose uid is 500 is not allowed to access /etc/passwd owned by uid 0 in /docroot/script.php on line 2*

## 5- Validating input

### Example #1:

Example when uploading files to the directory of a certain user

```
<?php
$username     = $_SERVER['REMOTE_USER']; // using an authentication mechanisim
$userfile     = $_POST['user_submitted_filename'];
$homedir      = "/home/$username";

$filepath     = "$homedir/$userfile";

if (!ctype_alnum($username) || !preg_match('/^(?:[a-z0-9_-]|\.(?!\.))+$/iD', $userfile)) {
   die("Bad username/filename");
}

//etc...
?>
```

Don't use the input of the user directly, we should read the username from the authentication mechanism, but since a user can register with a username like "../etc" we should validate the username using the preg_match

### Example #2:

Correctly validating the input

```
<?php
$file = $_GET['file'];
// Whitelisting possible values
switch ($file) {
   case 'main':
   case 'foo':
   case 'bar':
      include '/home/wwwrun/include/'.$file.'.php';
      break;
   default:
      include '/home/wwwrun/include/main.php';
}
?>
```

You should always carefully examine your code to make sure that any variables being submitted from a web browser are being properly checked, and ask yourself the following questions:

- Will this script only affect the intended files?
- Can unusual or undesirable data be acted upon?
- Can this script be used in unintended ways?
- Can this be used in conjunction with other scripts in a negative manner?
- Will any transactions be adequately logged?

### Example #3:

Avoiding attacks with links sent to the user's email in the "forgot your password" emails. If sessions are being used to keep track of things, this can be avoided easily:

```
<?php
session_start();
```

```
$clean = array();
$email_pattern = '/^[^@\s<&>]+@([-a-z0-9]+\.)+[a-z]{2,}$/i';
if (preg_match($email_pattern, $_POST['email']))
{
$clean['email'] = $_POST['email'];
$user = $_SESSION['user'];
$new_password = md5(uniqid(rand(), TRUE));
if ($_SESSION['verified'])
{
/* Update Password */
mail($clean['email'], 'Your New Password', $new_password);
}
}

?>
```

## 6- Includes Bad Practice

A common beginners mistake with the include / require functions looks like this;

```
echo '<a href="'.$_SERVER['PHP_SELF'].'?page=home">Home</a>';
echo '<a href="'.$_SERVER['PHP_SELF'].'?page=links">Links</a>';
echo '<a href="'.$_SERVER['PHP_SELF'].'?page=contact">Contact</a>';

include 'pages/'.$_GET['page'];
```

## 7- $_FILES

This array contains files uploaded from a form. This is a prime location for security holes. Use PEAR::HTTP_Upload to upload files - see http://pear.php.net/package/HTTP_Upload.

## 8- Error Reporting

With PHP security, there are two sides to error reporting. One is beneficial to increasing security, the other is detrimental.

A standard attack tactic involves profiling a system by feeding it improper data, and checking for the kinds, and contexts, of the errors which are returned. This allows the system cracker to probe for information about the server, to determine possible weaknesses. For example, if an attacker had gleaned information about a page based on a prior form submission, they may attempt to override variables, or modify them.

Errors shouldn't show that we are using php or mysql database as well as the structure and organization of files on the web server.

There are three major solutions to this issue. The first is to scrutinize all functions, and attempt to compensate for the bulk of the errors. The second is to disable error reporting entirely on the running code. The third is to use PHP's custom error handling functions to create your own error handler. Depending on your security policy, you may find all three to be applicable to your situation.

One way of catching this issue ahead of time is to make use of PHP's own error_reporting(), to help you secure your code and find variable usage that may be dangerous. By testing your code, prior to deployment, with E_ALL, you can quickly find areas where your variables may be open to poisoning or modification in other ways. Once you are ready for deployment, you should either disable error reporting completely by setting error_reporting() to 0, or turn off the error display using the *php.ini* option *display_errors*, to insulate your code from probing. If you choose to do the latter, you should also define the path to your log file using the *error_log* ini directive, and turn *log_errors* on.

## 9- Magic quotes

Magic Quotes is a process that automatically escapes incoming data to the PHP script. It's preferred to code with magic quotes off and to instead escape the data at runtime, as needed.

Why use Magic Quotes

- Useful for beginners Magic quotes are implemented in PHP to help code written by beginners from being dangerous. Although SQL Injection is still possible with magic quotes on, the risk is reduced.
- Convenience For inserting data into a database, magic quotes essentially runs addslashes() on all Get, Post, and Cookie data, and does so automagically.

Why not to use Magic Quotes

- Portability Assuming it to be on, or off, affects portability. Use get_magic_quotes_gpc() to check for this, and code accordingly.
- Performance Because not every piece of escaped data is inserted into a database, there is a performance loss for escaping all this data. Simply calling on the escaping functions (like addslashes()) at runtime is more efficient. Although *php.ini-dist* enables these directives by default, *php.ini-recommended* disables it. This recommendation is mainly due to performance reasons.
- Inconvenience Because not all data needs escaping, it's often annoying to see escaped data where it shouldn't be. For example, emailing from a form, and seeing a bunch of \' within the email. To fix, this may require excessive use of stripslashes().

## 10 - Hiding PHP

In general, security by obscurity is one of the weakest forms of security. But in some cases, every little bit of extra security is desirable.

A few simple techniques can help to hide PHP, possibly slowing down an attacker who is attempting to discover weaknesses in your system. By setting expose_php to *off* in your *php.ini* file, you reduce the amount of information available to them.

Another tactic is to configure web servers such as apache to parse different filetypes through PHP, either with an *.htaccess* directive, or in the apache configuration file itself. You can then use misleading file extensions:

**Example #1 Hiding PHP as another language**

```
# Make PHP code look like other code types
AddType application/x-httpd-php .asp .py .pl
```

Or obscure it completely:

**Example #2 Using unknown types for PHP extensions**

```
# Make PHP code look like unknown types
AddType application/x-httpd-php .bop .foo .133t
```

Or hide it as HTML code, which has a slight performance hit because all HTML will be parsed through the PHP engine:

**Example #3 Using HTML types for PHP extensions**

```
# Make all PHP code look like HTML
AddType application/x-httpd-php .htm .html
```

For this to work effectively, you must rename your PHP files with the above extensions. While it is a form of security through obscurity, it's a minor preventative measure with few drawbacks.

## 11- Useful PHP functions:

| | |
|---|---|
| escapeshellcmd() | http://www.php.net/escapeshellcmd |
| escapeshellarg() | http://www.php.net/escapeshellarg |
| realpath() | http://www.php.net/realpath |
| addslashes() | http://www.php.net/addslashes |
| mysql_real_escape_string() | http://www.php.net/mysql_real_escape_string |
| mysql_escape_string() | http://www.php.net/mysql_escape_string |

## 12- A Catalog of Security Sensitive PHP Functions

These PHP functions are common targets of various security attacks. These functions should probably not be used directly, but instead be wrapped by libraries that enforce security considerations.

The purpose of this list is to highlight security vulnerabilities on a PHP function basis.

- Security Sensitive Eval Functions - These functions can allow an arbitrary string or file to be executed as PHP code. (http://www.phpwact.org/security/functions/eval_functions )
- Security Sensitive Shell Functions - These functions allow shell commands to be run on the server. (http://www.phpwact.org/security/functions/shell_functions )

- Security Sensitive File Functions - These functions allow files to be read written, or permissions changed (http://www.phpwact.org/security/functions/file_functions ).
- Security Sensitive Database Functions (http://www.phpwact.org/security/functions/database_functions)
- Security Sensitive HTTP Request Functions - These functions read information from the HTTP request. (http://www.phpwact.org/security/functions/request_functions)
- Security Sensitive HTTP Response Functions - Output functions (http://www.phpwact.org/security/functions/response_functions)
- Security Sensitive Network Functions - Functions which allow remote computers to be accessed (http://www.phpwact.org/security/functions/network_functions)
- Security Sensitive Mail Functions - Functions which allow mail to be sent. (http://www.phpwact.org/security/functions/mail_functions)

## 13- A Catalog of Security Attacks

You will find below Methods of attacking a web application from the attackers' perspective and how to prevent each attack from the application developers' perspective [4].

### Information Gathering Attacks

- Directory Scanning Attack - An attempt to discover the file structure of a web site in preparation for further attacks
- Link Crawl Attack - Traversing application links attempting to discover the structure of the application
- Path Truncation Attack - Examining directory listings by removing the filename portion of the URL
- CGI Scanning Attack
- File System Scanning Attack - Scan the local file system to match its structure and detect vulnerable files.

### Injection Attacks

- Global Variable Injection Attack - Use parameters to inject arbitary values into uninitialized global variables in a PHP script.
- Remote File Injection Attack - Convince a PHP script to use a remote file instead of a presumably trusted file from the local file system.
- SQL Injection Attack - Attempt to get the database server to execute arbitrary SQL.
- Email Injection Attack - Attempt to get the program to send arbitrary emails.
- Command Injection Attack - Attempt to execute shell commands.
- Code Injection Attack - Attempt to execute arbitrary PHP code.
- Cross Site Scripting Attack - Attempt to coerce the program to outputing third party javascript.
- Cookie Tampering Attack - Attempt to manipulate an application's cookie values.
- Parameter Manipulation Attack - Attempt to manipulate input to application validation and filtering.
- LDAP Injection Attack

[4] http://www.phpwact.org/security/attack/catalog

- Globally Writable File Attack - File based input can be injected into other applications.

### Misc Attacks

- Password Cracking Attack - Brute force password guessing
- Denial of Service Attack - If you can't beat'em, shut them down.

# Appendix II: Resources

- http://phpsec.org/projects/guide/1.html
- http://php-ids.org/
- http://us3.php.net/manual/en/security.general.php
- http://us3.php.net/manual/en/security.filesystem.php
- http://unixwiz.net/techtips/sql-injection.html
- http://www.phpwact.org/security/web_application_security
- http://httpd.apache.org/docs/1.3/howto/htaccess.html