Features

HTTP authentication with PHP

The HTTP Authentication hooks in PHP are only available when it is running as an Apache module and is hence not available in the CGI version. In an Apache module PHP script, it is possible to use the header() function to send an "Authentication Required" message to the client browser causing it to pop up a Username/Password input window. Once the user has filled in a username and a password, the URL containing the PHP script will be called again with the predefined variables *PHP_AUTH_USER*, *PHP_AUTH_PW*, and *AUTH_TYPE* set to the user name, password and authentication type respectively. These predefined variables are found in the $_SERVER and *$HTTP_SERVER_VARS* arrays. Both "Basic" and "Digest" (since PHP 5.1.0) authentication methods are supported. See the header() function for more information.

Note

PHP Version Note

Superglobals, such as $_SERVER, became available in PHP » 4.1.0.

An example script fragment which would force client authentication on a page is as follows:

Example #1 - Basic HTTP Authentication example

```
<?php
if (!isset($_SERVER['PHP_AUTH_USER'])) {
   header('WWW-Authenticate: Basic realm="My Realm"');
   header('HTTP/1.0 401 Unauthorized');
   echo 'Text to send if user hits Cancel button';
   exit;
} else {
   echo "<p>Hello {$_SERVER['PHP_AUTH_USER']}.</p>";
   echo "<p>You entered {$_SERVER['PHP_AUTH_PW']} as your password.</p>";
}
?>
```

Example #2 - Digest HTTP Authentication example

This example shows you how to implement a simple Digest HTTP authentication script. For more information read the » RFC 2617.

```
<?php
$realm = 'Restricted area';
```

```
//user => password
$users = array('admin' => 'mypass', 'guest' => 'guest');


if (empty($_SERVER['PHP_AUTH_DIGEST'])) {
   header('HTTP/1.1 401 Unauthorized');
   header('WWW-Authenticate: Digest realm="'.$realm.
          '",qop="auth",nonce="'.uniqid().'",opaque="'.md5($realm).'"');

   die('Text to send if user hits Cancel button');
}


// analyze the PHP_AUTH_DIGEST variable
if (!($data = http_digest_parse($_SERVER['PHP_AUTH_DIGEST'])) ||
   !isset($users[$data['username']]))
   die('Wrong Credentials!');


// generate the valid response
$A1 = md5($data['username'] . ':' . $realm . ':' .
$users[$data['username']]);
$A2 = md5($_SERVER['REQUEST_METHOD'].':'.$data['uri']);
$valid_response =
md5($A1.':'.$data['nonce'].':'.$data['nc'].':'.$data['cnonce'].':'.$data['qo
p'].':'.$A2);

if ($data['response'] != $valid_response)
   die('Wrong Credentials!');

// ok, valid username & password
echo 'Your are logged in as: ' . $data['username'];


// function to parse the http auth header
function http_digest_parse($txt)
{
   // protect against missing data
   $needed_parts = array('nonce'=>1, 'nc'=>1, 'cnonce'=>1, 'qop'=>1,
'username'=>1, 'uri'=>1, 'response'=>1);
   $data = array();

   preg_match_all('@(\w+)=(?:([\'"])([^\2]+)\2|([^\s,]+))@', $txt, $matches,
PREG_SET_ORDER);

   foreach ($matches as $m) {
       $data[$m[1]] = $m[3] ? $m[3] : $m[4];
       unset($needed_parts[$m[1]]);
   }

   return $needed_parts ? false : $data;
}
?>
```

Note

Compatibility Note

Please be careful when coding the HTTP header lines. In order to guarantee maximum compatibility with all clients, the keyword "Basic" should be written with an uppercase "B", the realm string must be enclosed in double (not single) quotes, and exactly one space should precede the *401* code in the *HTTP/1.0 401* header line. Authentication parameters have to be comma-separated as seen in the digest example above.

Instead of simply printing out *PHP_AUTH_USER* and *PHP_AUTH_PW*, as done in the above example, you may want to check the username and password for validity. Perhaps by sending a query to a database, or by looking up the user in a dbm file.

Watch out for buggy Internet Explorer browsers out there. They seem very picky about the order of the headers. Sending the *WWW-Authenticate* header before the *HTTP/1.0 401* header seems to do the trick for now.

As of PHP 4.3.0, in order to prevent someone from writing a script which reveals the password for a page that was authenticated through a traditional external mechanism, the PHP_AUTH variables will not be set if external authentication is enabled for that particular page and safe mode is enabled. Regardless, *REMOTE_USER* can be used to identify the externally-authenticated user. So, you can use *$_SERVER['REMOTE_USER']*.

Note

Configuration Note

PHP uses the presence of an *AuthType* directive to determine whether external authentication is in effect.

Note, however, that the above does not prevent someone who controls a non-authenticated URL from stealing passwords from authenticated URLs on the same server.

Both Netscape Navigator and Internet Explorer will clear the local browser window's authentication cache for the realm upon receiving a server response of 401. This can effectively "log out" a user, forcing them to re-enter their username and password. Some people use this to "time out" logins, or provide a "log-out" button.

Example #3 - HTTP Authentication example forcing a new name/password

```
<?php
function authenticate() {
   header('WWW-Authenticate: Basic realm="Test Authentication System"');
   header('HTTP/1.0 401 Unauthorized');
```

```
   echo "You must enter a valid login ID and password to access this
resource\n";
   exit;
}

if (!isset($_SERVER['PHP_AUTH_USER']) ||
   ($_POST['SeenBefore'] == 1 && $_POST['OldAuth'] ==
$_SERVER['PHP_AUTH_USER'])) {
   authenticate();
} else {
   echo "<p>Welcome: {$_SERVER['PHP_AUTH_USER']}<br />";
   echo "Old: {$_REQUEST['OldAuth']}";
   echo "<form action='{$_SERVER['PHP_SELF']}' METHOD='post'>\n";
   echo "<input type='hidden' name='SeenBefore' value='1' />\n";
   echo "<input type='hidden' name='OldAuth'
value='{$_SERVER['PHP_AUTH_USER']}' />\n";
   echo "<input type='submit' value='Re Authenticate' />\n";
   echo "</form></p>\n";
}
?>
```

This behavior is not required by the HTTP Basic authentication standard, so you should never depend on this. Testing with Lynx has shown that Lynx does not clear the authentication credentials with a 401 server response, so pressing back and then forward again will open the resource as long as the credential requirements haven't changed. The user can press the '_' key to clear their authentication information, however.

Also note that until PHP 4.3.3, HTTP Authentication did not work using Microsoft's IIS server with the CGI version of PHP due to a limitation of IIS. In order to get it to work in PHP 4.3.3+, you must edit your IIS configuration "Directory Security". Click on "Edit" and only check "Anonymous Access", all other fields should be left unchecked.

Another limitation is if you're using the IIS module (ISAPI) and PHP 4, you may not use the *PHP_AUTH_** variables but instead, the variable *HTTP_AUTHORIZATION* is available. For example, consider the following code: *list($user, $pw) = explode(':', base64_decode(substr($_SERVER['HTTP_AUTHORIZATION'], 6)));*

Note

IIS Note:

For HTTP Authentication to work with IIS, the PHP directive cgi.rfc2616_headers must be set to *0* (the default value).

Note

If safe mode is enabled, the uid of the script is added to the *realm* part of the *WWW-Authenticate* header.

Cookies

PHP transparently supports HTTP cookies. Cookies are a mechanism for storing data in the remote browser and thus tracking or identifying return users. You can set cookies using the setcookie() or setrawcookie() function. Cookies are part of the HTTP header, so setcookie() must be called before any output is sent to the browser. This is the same limitation that header() has. You can use the output buffering functions to delay the script output until you have decided whether or not to set any cookies or send any headers.

Any cookies sent to you from the client will automatically be included into a $_COOKIE auto-global array if variables_order contains "C". If you wish to assign multiple values to a single cookie, just add *[]* to the cookie name.

Depending on register_globals, regular PHP variables can be created from cookies. However it's not recommended to rely on them as this feature is often turned off for the sake of security. *$HTTP_COOKIE_VARS* is also set in earlier versions of PHP when the track_vars configuration variable is set. (This setting is always on since PHP 4.0.3.)

For more details, including notes on browser bugs, see the setcookie() and setrawcookie() function.

Sessions

Session support in PHP consists of a way to preserve certain data across subsequent accesses. This enables you to build more customized applications and increase the appeal of your web site. All information is in the Session reference section.

Dealing with XForms

» XForms defines a variation on traditional webforms which allows them to be used on a wider variety of platforms and browsers or even non-traditional media such as PDF documents.

The first key difference in XForms is how the form is sent to the client. » XForms for HTML Authors contains a detailed description of how to create XForms, for the purpose of this tutorial we'll only be looking at a simple example.

Example #4 - A simple XForms search form

```
<h:html xmlns:h="http://www.w3.org/1999/xhtml"
       xmlns="http://www.w3.org/2002/xforms">
<h:head>
<h:title>Search</h:title>
<model>
 <submission action="http://example.com/search"
             method="post" id="s"/>
</model>
</h:head>
<h:body>
<h:p>
 <input ref="q"><label>Find</label></input>
 <submit submission="s"><label>Go</label></submit>
</h:p>
</h:body>
</h:html>
```

The above form displays a text input box (named `q`), and a submit button. When the submit button is clicked, the form will be sent to the page referred to by *action*.

Here's where it starts to look different from your web application's point of view. In a normal HTML form, the data would be sent as *application/x-www-form-urlencoded*, in the XForms world however, this information is sent as XML formatted data.

If you're choosing to work with XForms then you probably want that data as XML, in that case, look in *$HTTP_RAW_POST_DATA* where you'll find the XML document generated by the browser which you can pass into your favorite XSLT engine or document parser.

If you're not interested in formatting and just want your data to be loaded into the traditional *$_POST* variable, you can instruct the client browser to send it as *application/x-www-form-urlencoded* by changing the `method` attribute to *urlencoded-post*.

Example #5 - Using an XForm to populate *$_POST*

```
<h:html xmlns:h="http://www.w3.org/1999/xhtml"
       xmlns="http://www.w3.org/2002/xforms">
<h:head>
<h:title>Search</h:title>
```

```
<model>
 <submission action="http://example.com/search"
             method="urlencoded-post" id="s"/>
</model>
</h:head>
<h:body>
<h:p>
 <input ref="q"><label>Find</label></input>
 <submit submission="s"><label>Go</label></submit>
</h:p>
</h:body>
</h:html>
```

Note

As of this writing, many browsers do not support XForms. Check your browser version if the above examples fails.

Handling file uploads

POST method uploads

This feature lets people upload both text and binary files. With PHP's authentication and file manipulation functions, you have full control over who is allowed to upload and what is to be done with the file once it has been uploaded.

PHP is capable of receiving file uploads from any RFC-1867 compliant browser (which includes Netscape Navigator 3 or later, Microsoft Internet Explorer 3 with a patch from Microsoft, or later without a patch).

Note

Related Configurations Note

See also the file_uploads, upload_max_filesize, upload_tmp_dir, post_max_size and max_input_time directives in *php.ini*

PHP also supports PUT-method file uploads as used by Netscape Composer and W3C's Amaya clients. See the PUT Method Support for more details.

Example #6 - File Upload Form

A file upload screen can be built by creating a special form which looks something like this:

```
<!-- The data encoding type, enctype, MUST be specified as below -->
<form enctype="multipart/form-data" action="__URL__" method="POST">
   <!-- MAX_FILE_SIZE must precede the file input field -->
   <input type="hidden" name="MAX_FILE_SIZE" value="30000" />
   <!-- Name of input element determines name in $_FILES array -->
   Send this file: <input name="userfile" type="file" />
   <input type="submit" value="Send File" />
</form>
```

The *__URL__* in the above example should be replaced, and point to a PHP file.

The *MAX_FILE_SIZE* hidden field (measured in bytes) must precede the file input field, and its value is the maximum filesize accepted by PHP. Fooling this setting on the browser side is quite easy, so never rely on files with a greater size being blocked by this feature. The PHP settings for maximum-size, however, cannot be fooled. This form element should always be used as it saves users the trouble of waiting for a big file being transferred only to find that it was too big and the transfer failed.

Note

Be sure your file upload form has attribute *enctype="multipart/form-data"* otherwise the file upload will not work.

The global $_FILES exists as of PHP 4.1.0 (Use *$HTTP_POST_FILES* instead if using an earlier version). These arrays will contain all the uploaded file information.

The contents of $_FILES from the example form is as follows. Note that this assumes the use of the file upload name *userfile*, as used in the example script above. This can be any name.

$_FILES['userfile']['name']
: The original name of the file on the client machine.

$_FILES['userfile']['type']
: The mime type of the file, if the browser provided this information. An example would be *"image/gif"*. This mime type is however not checked on the PHP side and therefore don't take its value for granted.

$_FILES['userfile']['size']
: The size, in bytes, of the uploaded file.

$_FILES['userfile']['tmp_name']
: The temporary filename of the file in which the uploaded file was stored on the server.

$_FILES['userfile']['error']
: The error code associated with this file upload. This element was added in PHP 4.2.0

Files will, by default be stored in the server's default temporary directory, unless another location has been given with the upload_tmp_dir directive in *php.ini*. The server's default directory can be changed by setting the environment variable TMPDIR in the environment in which PHP runs. Setting it using putenv() from within a PHP script will not work. This environment variable can also be used to make sure that other operations are working on uploaded files, as well.

Example #7 - Validating file uploads

See also the function entries for is_uploaded_file() and move_uploaded_file() for further information. The following example will process the file upload that came from a form.

```
<?php
// In PHP versions earlier than 4.1.0, $HTTP_POST_FILES should be used
instead
// of $_FILES.

$uploaddir = '/var/www/uploads/';
$uploadfile = $uploaddir . basename($_FILES['userfile']['name']);

echo '<pre>';
```

```
if (move_uploaded_file($_FILES['userfile']['tmp_name'], $uploadfile)) {
   echo "File is valid, and was successfully uploaded.\n";
} else {
   echo "Possible file upload attack!\n";
}

echo 'Here is some more debugging info:';
print_r($_FILES);

print "</pre>";

?>
```

The PHP script which receives the uploaded file should implement whatever logic is necessary for determining what should be done with the uploaded file. You can, for example, use the *$_FILES['userfile']['size']* variable to throw away any files that are either too small or too big. You could use the *$_FILES['userfile']['type']* variable to throw away any files that didn't match a certain type criteria, but use this only as first of a series of checks, because this value is completely under the control of the client and not checked on the PHP side. As of PHP 4.2.0, you could use *$_FILES['userfile']['error']* and plan your logic according to the error codes. Whatever the logic, you should either delete the file from the temporary directory or move it elsewhere.

If no file is selected for upload in your form, PHP will return *$_FILES['userfile']['size']* as 0, and *$_FILES['userfile']['tmp_name']* as none.

The file will be deleted from the temporary directory at the end of the request if it has not been moved away or renamed.

Example #8 - Uploading array of files

PHP supports HTML array feature even with files.

```
<form action="" method="post" enctype="multipart/form-data">
<p>Pictures:
<input type="file" name="pictures[]" />
<input type="file" name="pictures[]" />
<input type="file" name="pictures[]" />
<input type="submit" value="Send" />
</p>
</form>

<?php
foreach ($_FILES["pictures"]["error"] as $key => $error) {
   if ($error == UPLOAD_ERR_OK) {
       $tmp_name = $_FILES["pictures"]["tmp_name"][$key];
       $name = $_FILES["pictures"]["name"][$key];
       move_uploaded_file($tmp_name, "data/$name");
   }
}
?>
```

Error Messages Explained

Since PHP 4.2.0, PHP returns an appropriate error code along with the file array. The error code can be found in the *error* segment of the file array that is created during the file upload by PHP. In other words, the error might be found in *$_FILES['userfile']['error']*.

UPLOAD_ERR_OK
Value: 0; There is no error, the file uploaded with success.

UPLOAD_ERR_INI_SIZE
Value: 1; The uploaded file exceeds the upload_max_filesize directive in *php.ini*.

UPLOAD_ERR_FORM_SIZE
Value: 2; The uploaded file exceeds the *MAX_FILE_SIZE* directive that was specified in the HTML form.

UPLOAD_ERR_PARTIAL
Value: 3; The uploaded file was only partially uploaded.

UPLOAD_ERR_NO_FILE
Value: 4; No file was uploaded.

UPLOAD_ERR_NO_TMP_DIR
Value: 6; Missing a temporary folder. Introduced in PHP 4.3.10 and PHP 5.0.3.

UPLOAD_ERR_CANT_WRITE
Value: 7; Failed to write file to disk. Introduced in PHP 5.1.0.

UPLOAD_ERR_EXTENSION
Value: 8; File upload stopped by extension. Introduced in PHP 5.2.0.

Note
These became PHP constants in PHP 4.3.0.

Common Pitfalls

The *MAX_FILE_SIZE* item cannot specify a file size greater than the file size that has been set in the upload_max_filesize ini-setting. The default is 2 Megabytes.

If a memory limit is enabled, a larger memory_limit may be needed. Make sure you set memory_limit large enough.

If max_execution_time is set too small, script execution may be exceeded by the value. Make sure you set *max_execution_time* large enough.

Note

max_execution_time only affects the execution time of the script itself. Any time spent on activity that happens outside the execution of the script such as system calls using system(), the sleep() function, database queries, time taken by the file upload process, etc. is not included when determining the maximum time that the script has been running.

Warning

max_input_time sets the maximum time, in seconds, the script is allowed to receive input; this includes file uploads. For large or multiple files, or users on slower connections, the default of *60 seconds* may be exceeded.

If post_max_size is set too small, large files cannot be uploaded. Make sure you set *post_max_size* large enough.

Not validating which file you operate on may mean that users can access sensitive information in other directories.

Please note that the CERN httpd seems to strip off everything starting at the first whitespace in the content-type mime header it gets from the client. As long as this is the case, CERN httpd will not support the file upload feature.

Due to the large amount of directory listing styles we cannot guarantee that files with exotic names (like containing spaces) are handled properly.

A developer may not mix normal input fields and file upload fields in the same form variable (by using an input name like *foo[]*).

Uploading multiple files

Multiple files can be uploaded using different *name* for *input*.

It is also possible to upload multiple files simultaneously and have the information organized automatically in arrays for you. To do so, you need to use the same array submission syntax in the HTML form as you do with multiple selects and checkboxes:

Example #9 - Uploading multiple files

```
<form action="file-upload.php" method="post" enctype="multipart/form-data">
 Send these files:<br />
 <input name="userfile[]" type="file" /><br />
 <input name="userfile[]" type="file" /><br />
 <input type="submit" value="Send files" />
```

```
</form>
```

When the above form is submitted, the arrays *$_FILES['userfile']*, *$_FILES['userfile']['name']*, and *$_FILES['userfile']['size']* will be initialized (as well as in *$HTTP_POST_FILES* for PHP versions prior to 4.1.0). When register_globals is on, globals for uploaded files are also initialized. Each of these will be a numerically indexed array of the appropriate values for the submitted files.

For instance, assume that the filenames */home/test/review.html* and */home/test/xwp.out* are submitted. In this case, *$_FILES['userfile']['name'][0]* would contain the value *review.html*, and *$_FILES['userfile']['name'][1]* would contain the value *xwp.out*. Similarly, *$_FILES['userfile']['size'][0]* would contain *review.html* 's file size, and so forth.

$_FILES['userfile']['name'][0], *$_FILES['userfile']['tmp_name'][0]*, *$_FILES['userfile']['size'][0]*, and *$_FILES['userfile']['type'][0]* are also set.

PUT method support

PHP provides support for the HTTP PUT method used by some clients to store files on a server. PUT requests are much simpler than a file upload using POST requests and they look something like this:

```
PUT /path/filename.html HTTP/1.1
```

This would normally mean that the remote client would like to save the content that follows as: */path/filename.html* in your web tree. It is obviously not a good idea for Apache or PHP to automatically let everybody overwrite any files in your web tree. So, to handle such a request you have to first tell your web server that you want a certain PHP script to handle the request. In Apache you do this with the *Script* directive. It can be placed almost anywhere in your Apache configuration file. A common place is inside a <Directory> block or perhaps inside a <VirtualHost> block. A line like this would do the trick:

```
Script PUT /put.php
```

This tells Apache to send all PUT requests for URIs that match the context in which you put this line to the put.php script. This assumes, of course, that you have PHP enabled for the .php extension and PHP is active. The destination resource for all PUT requests to this script has to be the script itself, not a filename the uploaded file should have.

With PHP you would then do something like the following in your put.php. This would copy the contents of the uploaded file to the file *myputfile.ext* on the server. You would probably want to perform some checks and/or authenticate the user before performing this file copy.

Example #10 - Saving HTTP PUT files

```
<?php
/* PUT data comes in on the stdin stream */
$putdata = fopen("php://input", "r");

/* Open a file for writing */
$fp = fopen("myputfile.ext", "w");

/* Read the data 1 KB at a time
  and write to the file */
while ($data = fread($putdata, 1024))
 fwrite($fp, $data);

/* Close the streams */
fclose($fp);
fclose($putdata);
?>
```

Using remote files

As long as *allow_url_fopen* is enabled in *php.ini*, you can use HTTP and FTP URLs with most of the functions that take a filename as a parameter. In addition, URLs can be used with the **include()**, **include_once()**, **require()** and **require_once()** statements (since PHP 5.2.0, *allow_url_include* must be enabled for these). See List of Supported Protocols/Wrappers for more information about the protocols supported by PHP.

Note
In PHP 4.0.3 and older, in order to use URL wrappers, you were required to configure PHP using the configure option *--enable-url-fopen-wrapper*.

Note
The Windows versions of PHP earlier than PHP 4.3 did not support remote file accessing for the following functions: **include()**, **include_once()**, **require()**, **require_once()**, and the imagecreatefromXXX functions in the GD Functions extension.

For example, you can use this to open a file on a remote web server, parse the output for the data you want, and then use that data in a database query, or simply to output it in a style matching the rest of your website.

Example #11 - Getting the title of a remote page

```
<?php
$file = fopen ("http://www.example.com/", "r");
if (!$file) {
   echo "<p>Unable to open remote file.\n";
   exit;
}
while (!feof ($file)) {
   $line = fgets ($file, 1024);
   /* This only works if the title and its tags are on one line */
   if (eregi ("<title>(.*)</title>", $line, $out)) {
       $title = $out[1];
       break;
   }
}
fclose($file);
?>
```

You can also write to files on an FTP server (provided that you have connected as a user with the correct access rights). You can only create new files using this method; if you try to overwrite a file that already exists, the fopen() call will fail.

To connect as a user other than 'anonymous', you need to specify the username (and possibly password) within the URL, such as 'ftp://user:password@ftp.example.com/path/to/file'. (You can use the same sort of syntax to access files via HTTP when they require Basic authentication.)

Example #12 - Storing data on a remote server

```
<?php
$file = fopen ("ftp://ftp.example.com/incoming/outputfile", "w");
if (!$file) {
   echo "<p>Unable to open remote file for writing.\n";
   exit;
}
/* Write the data here. */
fwrite ($file, $_SERVER['HTTP_USER_AGENT'] . "\n");
fclose ($file);
?>
```

Note

You might get the idea from the example above that you can use this technique to write to a remote log file. Unfortunately that would not work because the fopen() call will fail if the remote file already exists. To do distributed logging like that, you should take a look at syslog().

Connection handling

Internally in PHP a connection status is maintained. There are 3 possible states:

- 0 - NORMAL
- 1 - ABORTED
- 2 - TIMEOUT

When a PHP script is running normally the NORMAL state, is active. If the remote client disconnects the ABORTED state flag is turned on. A remote client disconnect is usually caused by the user hitting his STOP button. If the PHP-imposed time limit (see set_time_limit()) is hit, the TIMEOUT state flag is turned on.

You can decide whether or not you want a client disconnect to cause your script to be aborted. Sometimes it is handy to always have your scripts run to completion even if there is no remote browser receiving the output. The default behaviour is however for your script to be aborted when the remote client disconnects. This behaviour can be set via the ignore_user_abort *php.ini* directive as well as through the corresponding *php_value ignore_user_abort* Apache .conf directive or with the ignore_user_abort() function. If you do not tell PHP to ignore a user abort and the user aborts, your script will terminate. The one exception is if you have registered a shutdown function using register_shutdown_function(). With a shutdown function, when the remote user hits his STOP button, the next time your script tries to output something PHP will detect that the connection has been aborted and the shutdown function is called. This shutdown function will also get called at the end of your script terminating normally, so to do something different in case of a client disconnect you can use the connection_aborted() function. This function will return **TRUE** if the connection was aborted.

Your script can also be terminated by the built-in script timer. The default timeout is 30 seconds. It can be changed using the *max_execution_time php.ini* directive or the corresponding *php_value max_execution_time* Apache .conf directive as well as with the set_time_limit() function. When the timer expires the script will be aborted and as with the above client disconnect case, if a shutdown function has been registered it will be called. Within this shutdown function you can check to see if a timeout caused the shutdown function to be called by calling the connection_status() function. This function will return 2 if a timeout caused the shutdown function to be called.

One thing to note is that both the ABORTED and the TIMEOUT states can be active at the same time. This is possible if you tell PHP to ignore user aborts. PHP will still note the fact that a user may have broken the connection, but the script will keep running. If it then hits the time limit it will be aborted and your shutdown function, if any, will be called. At this point you will find that connection_status() returns 3.

Persistent Database Connections

Persistent connections are links that do not close when the execution of your script ends. When a persistent connection is requested, PHP checks if there's already an identical persistent connection (that remained open from earlier) - and if it exists, it uses it. If it does not exist, it creates the link. An 'identical' connection is a connection that was opened to the same host, with the same username and the same password (where applicable).

People who aren't thoroughly familiar with the way web servers work and distribute the load may mistake persistent connects for what they're not. In particular, they do *not* give you an ability to open 'user sessions' on the same link, they do *not* give you an ability to build up a transaction efficiently, and they don't do a whole lot of other things. In fact, to be extremely clear about the subject, persistent connections don't give you *any* functionality that wasn't possible with their non-persistent brothers.

Why?

This has to do with the way web servers work. There are three ways in which your web server can utilize PHP to generate web pages.

The first method is to use PHP as a CGI "wrapper". When run this way, an instance of the PHP interpreter is created and destroyed for every page request (for a PHP page) to your web server. Because it is destroyed after every request, any resources that it acquires (such as a link to an SQL database server) are closed when it is destroyed. In this case, you do not gain anything from trying to use persistent connections -- they simply don't persist.

The second, and most popular, method is to run PHP as a module in a multiprocess web server, which currently only includes Apache. A multiprocess server typically has one process (the parent) which coordinates a set of processes (its children) who actually do the work of serving up web pages. When a request comes in from a client, it is handed off to one of the children that is not already serving another client. This means that when the same client makes a second request to the server, it may be served by a different child process than the first time. When opening a persistent connection, every following page requesting SQL services can reuse the same established connection to the SQL server.

The last method is to use PHP as a plug-in for a multithreaded web server. Currently PHP 4 has support for ISAPI, WSAPI, and NSAPI (on Windows), which all allow PHP to be used as a plug-in on multithreaded servers like Netscape FastTrack (iPlanet), Microsoft's Internet Information Server (IIS), and O'Reilly's WebSite Pro. The behavior is essentially the same as for the multiprocess model described before.

If persistent connections don't have any added functionality, what are they good for?

The answer here is extremely simple -- efficiency. Persistent connections are good if the overhead to create a link to your SQL server is high. Whether or not this overhead is really high depends on many factors. Like, what kind of database it is, whether or not it sits on the same computer on which your web server sits, how loaded the machine the SQL server sits on is and so forth. The bottom line is that if that connection overhead is high, persistent connections help you considerably. They cause the child process to simply

connect only once for its entire lifespan, instead of every time it processes a page that requires connecting to the SQL server. This means that for every child that opened a persistent connection will have its own open persistent connection to the server. For example, if you had 20 different child processes that ran a script that made a persistent connection to your SQL server, you'd have 20 different connections to the SQL server, one from each child.

Note, however, that this can have some drawbacks if you are using a database with connection limits that are exceeded by persistent child connections. If your database has a limit of 16 simultaneous connections, and in the course of a busy server session, 17 child threads attempt to connect, one will not be able to. If there are bugs in your scripts which do not allow the connections to shut down (such as infinite loops), the database with only 16 connections may be rapidly swamped. Check your database documentation for information on handling abandoned or idle connections.

Warning
There are a couple of additional caveats to keep in mind when using persistent connections. One is that when using table locking on a persistent connection, if the script for whatever reason cannot release the lock, then subsequent scripts using the same connection will block indefinitely and may require that you either restart the httpd server or the database server. Another is that when using transactions, a transaction block will also carry over to the next script which uses that connection if script execution ends before the transaction block does. In either case, you can use register_shutdown_function() to register a simple cleanup function to unlock your tables or roll back your transactions. Better yet, avoid the problem entirely by not using persistent connections in scripts which use table locks or transactions (you can still use them elsewhere).

An important summary. Persistent connections were designed to have one-to-one mapping to regular connections. That means that you should *always* be able to replace persistent connections with non-persistent connections, and it won't change the way your script behaves. It *may* (and probably will) change the efficiency of the script, but not its behavior!

See also fbsql_pconnect(), ibase_pconnect(), ifx_pconnect(), ingres_pconnect(), msql_pconnect(), mssql_pconnect(), mysql_pconnect(), ociplogon(), odbc_pconnect(), **ora_plogon()**, pfsockopen(), pg_pconnect(), and sybase_pconnect().

Safe Mode

The PHP safe mode is an attempt to solve the shared-server security problem. It is architecturally incorrect to try to solve this problem at the PHP level, but since the alternatives at the web server and OS levels aren't very realistic, many people, especially ISP's, use safe mode for now.

Warning
Safe Mode was removed in PHP 6.0.0.

Security and Safe Mode

Security and Safe Mode Configuration Directives

Name	**Default**	**Changeable**	**Changelog**
safe_mode	"0"	PHP_INI_SYSTEM	Removed in PHP 6.0.0.
safe_mode_gid	"0"	PHP_INI_SYSTEM	Available since PHP 4.1.0. Removed in PHP 6.0.0.
safe_mode_include_dir	NULL	PHP_INI_SYSTEM	Available since PHP 4.1.0. Removed in PHP 6.0.0.
safe_mode_exec_dir	""	PHP_INI_SYSTEM	Removed in PHP 6.0.0.
safe_mode_allowed_env_vars	"PHP_"	PHP_INI_SYSTEM	Removed in PHP 6.0.0.
safe_mode_protected_env_vars	"LD_LIBRARY_PATH"	PHP_INI_SYSTEM	Removed in PHP 6.0.0.
open_basedir	NULL	PHP_INI_ALL	PHP_INI_SYSTEM in PHP < 6.
disable_functions	""	*php.ini* only	Available since PHP 4.0.1.
disable_classes	""	*php.ini* only	Available since PHP

			4.3.2.

For further details and definition of the PHP_INI_* constants see ini_set().

Here's a short explanation of the configuration directives.

safe_mode boolean
Whether to enable PHP's safe mode.

safe_mode_gid boolean
By default, Safe Mode does a UID compare check when opening files. If you want to relax this to a GID compare, then turn on safe_mode_gid. Whether to use *UID* (**FALSE**) or *GID* (**TRUE**) checking upon file access.

safe_mode_include_dir string
UID / *GID* checks are bypassed when including files from this directory and its subdirectories (directory must also be in include_path or full path must including). As of PHP 4.2.0, this directive can take a colon (semi-colon on Windows) separated path in a fashion similar to the include_path directive, rather than just a single directory. The restriction specified is actually a prefix, not a directory name. This means that "safe_mode_include_dir = /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: "safe_mode_include_dir = /dir/incl/" If the value of this directive is empty, no files with different *UID* / *GID* can be included in PHP 4.2.3 and as of PHP 4.3.3. In earlier versions, all files could be included.

safe_mode_exec_dir string
If PHP is used in safe mode, system() and the other functions executing system programs refuse to start programs that are not in this directory. You have to use / as directory separator on all environments including Windows.

safe_mode_allowed_env_vars string
Setting certain environment variables may be a potential security breach. This directive contains a comma-delimited list of prefixes. In Safe Mode, the user may only alter environment variables whose names begin with the prefixes supplied here. By default, users will only be able to set environment variables that begin with PHP_ (e.g. PHP_FOO=BAR).

Note
If this directive is empty, PHP will let the user modify ANY environment variable!

safe_mode_protected_env_vars string
This directive contains a comma-delimited list of environment variables that the end user won't be able to change using putenv(). These variables will be protected even if safe_mode_allowed_env_vars is set to allow to change them.

open_basedir string
Limit the files that can be opened by PHP to the specified directory-tree, including the

file itself. This directive is *NOT* affected by whether Safe Mode is turned On or Off. 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`. The special value . indicates that the working directory of the script will be used as the base-directory. This is, however, a little dangerous as the working directory of the script can easily be changed with chdir(). In *httpd.conf*, open_basedir can be turned off (e.g. for some virtual hosts) the same way as any other configuration directive with "php_admin_value open_basedir none". 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.

`disable_functions` string

This directive allows you to disable certain functions for security reasons. It takes on a comma-delimited list of function names. disable_functions is not affected by Safe Mode. This directive must be set in *php.ini* For example, you cannot set this in *httpd.conf*.

`disable_classes` string

This directive allows you to disable certain classes for security reasons. It takes on a comma-delimited list of class names. disable_classes is not affected by Safe Mode. This directive must be set in *php.ini* For example, you cannot set this in *httpd.conf*.

Note
Availability note This directive became available in PHP 4.3.2

See also: register_globals, display_errors, and log_errors.

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       1116 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
```

However, there may be environments where a strict *UID* check is not appropriate and a relaxed *GID* check is sufficient. This is supported by means of the safe_mode_gid switch. Setting it to *On* performs the relaxed *GID* checking, setting it to *Off* (the default) performs *UID* checking.

If instead of safe_mode, you set an open_basedir directory then all file operations will be limited to files under the specified directory. For example (Apache *httpd.conf* example):

```
<Directory /docroot>
 php_admin_value open_basedir /docroot
</Directory>
```

If you run the same script.php with this open_basedir setting then this is the result:

```
Warning: open_basedir restriction in effect. File is in wrong directory in
/docroot/script.php on line 2
```

You can also disable individual functions. Note that the disable_functions directive can not be used outside of the *php.ini* file which means that you cannot disable functions on a per-virtualhost or per-directory basis in your *httpd.conf* file. If we add this to our *php.ini* file:

```
disable_functions = readfile,system
```

Then we get this output:

```
Warning: readfile() has been disabled for security reasons in
/docroot/script.php on line 2
```

Warning
These PHP restrictions are not valid in executed binaries, of course.

Functions restricted/disabled by safe mode

This is a still probably incomplete and possibly incorrect listing of the functions limited by safe mode.

Safe mode limited functions

Function	Limitations
dbmopen()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed.
dbase_open()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed.
filepro()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed.
filepro_rowcount()	Checks whether the files or directories being

	operated upon have the same UID (owner) as the script that is being executed.
filepro_retrieve()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed.
ifx_*	sql_safe_mode restrictions, (!= safe mode)
ingres_*	sql_safe_mode restrictions, (!= safe mode)
mysql_*	sql_safe_mode restrictions, (!= safe mode)
pg_lo_import()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed.
posix_mkfifo()	Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed.
putenv()	Obeys the safe_mode_protected_env_vars and safe_mode_allowed_env_vars ini-directives. See also the documentation on putenv()
move_uploaded_file()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed.
chdir()	Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed.
dl()	This function is disabled when PHP is running in safe mode.
backtick operator	This function is disabled when PHP is running in safe mode.
shell_exec() (functional equivalent of backticks)	This function is disabled when PHP is running in safe mode.
exec()	You can only execute executables within the safe_mode_exec_dir. For practical reasons it's currently not allowed to have.. components in the path to the executable. escapeshellcmd() is executed on the argument of this function.
system()	You can only execute executables within the safe_mode_exec_dir. For practical reasons

	it's currently not allowed to have.. components in the path to the executable. escapeshellcmd() is executed on the argument of this function.
passthru()	You can only execute executables within the safe_mode_exec_dir. For practical reasons it's currently not allowed to have.. components in the path to the executable. escapeshellcmd() is executed on the argument of this function.
popen()	You can only execute executables within the safe_mode_exec_dir. For practical reasons it's currently not allowed to have.. components in the path to the executable. escapeshellcmd() is executed on the argument of this function.
fopen()	Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed.
mkdir()	Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed.
rmdir()	Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed.
rename()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed. Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed.
unlink()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed. Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed.
copy()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed. Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed. (on `source` and `target`)

chgrp()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed.
chown()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed.
chmod()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed. In addition, you cannot set the SUID, SGID and sticky bits
touch()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed. Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed.
symlink()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed. Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed. (note: only the target is checked)
link()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed. Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed. (note: only the target is checked)
apache_request_headers()	In safe mode, headers beginning with 'authorization' (case-insensitive) will not be returned.
header()	In safe mode, the uid of the script is added to the *realm* part of the *WWW-Authenticate* header if you set this header (used for HTTP Authentication).
PHP_AUTH variables	In safe mode, the variables *PHP_AUTH_USER*, *PHP_AUTH_PW*, and *AUTH_TYPE* are not available in *$_SERVER*. Regardless, you can still use *REMOTE_USER* for the USER. (note: only affected since PHP 4.3.0)

highlight_file(), show_source()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed. Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed. (note: only affected since PHP 4.2.1)
parse_ini_file()	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed. Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed. (note: only affected since PHP 4.2.1)
set_time_limit()	Has no effect when PHP is running in safe mode.
max_execution_time	Has no effect when PHP is running in safe mode.
mail()	In safe mode, the fifth parameter is disabled. (note: only affected since PHP 4.2.3)
All filesystem and stream functions.	Checks whether the files or directories being operated upon have the same UID (owner) as the script that is being executed. Checks whether the directory in which the script is operating has the same UID (owner) as the script that is being executed. (see the safe_mode_include_dir *php.ini* option.

Using PHP from the command line

As of version 4.3.0, PHP supports a new *SAPI* type (Server Application Programming Interface) named *CLI* which means *Command Line Interface*. As the name implies, this *SAPI* type main focus is on developing shell (or desktop as well) applications with PHP. There are quite a few differences between the *CLI SAPI* and other *SAPI* s which are explained in this chapter. It's worth mentioning that *CLI* and *CGI* are different SAPI's although they do share many of the same behaviors.

The *CLI SAPI* was released for the first time with PHP 4.2.0, but was still experimental and had to be explicitly enabled with *--enable-cli* when running*./configure*. Since PHP 4.3.0 the *CLI SAPI* is no longer experimental and the option *--enable-cli* is on by default. You may use *--disable-cli* to disable it.

As of PHP 4.3.0, the name, location and existence of the CLI/CGI binaries will differ depending on how PHP is installed on your system. By default when executing *make*, both the CGI and CLI are built and placed as *sapi/cgi/php* and *sapi/cli/php* respectively, in your PHP source directory. You will note that both are named php. What happens during *make install* depends on your configure line. If a module SAPI is chosen during configure, such as apxs, or the *--disable-cgi* option is used, the CLI is copied to *{PREFIX}/bin/php* during *make install* otherwise the CGI is placed there. So, for example, if *--with--apxs* is in your configure line then the CLI is copied to *{PREFIX}/bin/php* during *make install*. If you want to override the installation of the CGI binary, use *make install-cli* after *make install*. Alternatively you can specify *--disable-cgi* in your configure line.

Note
Because both *--enable-cli* and *--enable-cgi* are enabled by default, simply having *--enable-cli* in your configure line does not necessarily mean the CLI will be copied as *{PREFIX}/bin/php* during *make install*.

The Windows packages between PHP 4.2.0 and PHP 4.2.3 distributed the CLI as *php-cli.exe*, living in the same folder as the CGI *php.exe*. Starting with PHP 4.3.0 the Windows package distributes the CLI as *php.exe* in a separate folder named *cli*, so *cli/php.exe*. Starting with PHP 5, the CLI is distributed in the main folder, named *php.exe*. The CGI version is distributed as *php-cgi.exe*.

As of PHP 5, a new *php-win.exe* file is distributed. This is equal to the CLI version, except that php-win doesn't output anything and thus provides no console (no "dos box" appears on the screen). This behavior is similar to php-gtk. You should configure with *--enable-cli-win32*.

Note
What SAPI do I have? From a shell, typing *php -v* will tell you whether *php* is CGI or CLI. See also the

function php_sapi_name() and the constant **PHP_SAPI**.

Note
A Unix *man* ual page was added in PHP 4.3.2. You may view this by typing *man php* in your shell environment.

Remarkable differences of the *CLI SAPI* compared to other *SAPI* s:

- Unlike the *CGI SAPI*, no headers are written to the output. Though the *CGI SAPI* provides a way to suppress HTTP headers, there's no equivalent switch to enable them in the *CLI SAPI*. CLI is started up in quiet mode by default, though the *-q* and *--no-header* switches are kept for compatibility so that you can use older CGI scripts. It does not change the working directory to that of the script. (*-C* and *--no-chdir* switches kept for compatibility) Plain text error messages (no HTML formatting).
- There are certain *php.ini* directives which are overridden by the *CLI SAPI* because they do not make sense in shell environments:

  Overridden *php.ini* **directives**

Directive	*CLI SAPI* **default value**	**Comment**
html_errors	**FALSE**	It can be quite hard to read the error message in your shell when it's cluttered with all those meaningless *HTML* tags, therefore this directive defaults to **FALSE**.
implicit_flush	**TRUE**	It is desired that any output coming from print(), echo() and friends is immediately written to the output and not cached in any buffer. You still can use output buffering if you want to defer or manipulate standard output.
max_execution_time	0 (unlimited)	Due to endless possibilities of using PHP in shell environments, the maximum execution time has been set to unlimited. Whereas applications written for the web are often executed very quickly, shell application tend to have a much longer execution time.

register_argc_argv	**TRUE**	Because this setting is **TRUE** you will always have access to *argc* (number of arguments passed to the application) and *argv* (array of the actual arguments) in the *CLI SAPI*. As of PHP 4.3.0, the PHP variables *$argc* and *$argv* are registered and filled in with the appropriate values when using the *CLI SAPI*. Prior to this version, the creation of these variables behaved as they do in *CGI* and *MODULE* versions which requires the PHP directive register_globals to be *on*. Regardless of version or register_globals setting, you can always go through either $_SERVER or *$HTTP_SERVER_VARS*. Example: *$_SERVER['argv']*

Note
These directives cannot be initialized with another value from the configuration file *php.ini* or a custom one (if specified). This is a limitation because those default values are applied after all configuration files have been parsed. However, their value can be changed during runtime (which does not make sense for all of those directives, e.g. register_argc_argv).

- To ease working in the shell environment, the following constants are defined:

  CLI specific Constants

Constant	**Description**
STDIN	An already opened stream to *stdin*. This saves opening it with `<?php` `$stdin = fopen('php://stdin', 'r');` `?>`

	If you want to read single line from *stdin*, you can use `<?php` `$line = trim(fgets(STDIN)); // reads one line from STDIN` `fscanf(STDIN, "%d\n", $number); // reads number from STDIN` `?>`
STDOUT	An already opened stream to *stdout*. This saves opening it with `<?php` `$stdout = fopen('php://stdout', 'w');` `?>`
STDERR	An already opened stream to *stderr*. This saves opening it with `<?php` `$stderr = fopen('php://stderr', 'w');` `?>`

Given the above, you don't need to open e.g. a stream for *stderr* yourself but simply use the constant instead of the stream resource:

```
php -r 'fwrite(STDERR, "stderr\n");'
```

You do not need to explicitly close these streams, as they are closed automatically by PHP when your script ends.

Note
These constants are not available in case of reading PHP script from *stdin*.

- The *CLI SAPI* does *not* change the current directory to the directory of the executed script!

  Example showing the difference to the *CGI SAPI*:

  ```
  <?php
  // Our simple test application named test.php
  echo getcwd(), "\n";
  ?>
  ```

  When using the *CGI* version, the output is:

```
$ pwd
/tmp

$ php -q another_directory/test.php
/tmp/another_directory
```

This clearly shows that PHP changes its current directory to the one of the executed script.

Using the *CLI SAPI* yields:

```
$ pwd
/tmp

$ php -f another_directory/test.php
/tmp
```

This allows greater flexibility when writing shell tools in PHP.

Note
The *CGI SAPI* supports this *CLI SAPI* behaviour by means of the *-C* switch when run from the command line.

The list of command line options provided by the PHP binary can be queried anytime by running PHP with the *-h* switch:

```
Usage: php [options] [-f] <file> [--] [args...]
       php [options] -r <code> [--] [args...]
       php [options] [-B <begin_code>] -R <code> [-E <end_code>] [--] [args...]
       php [options] [-B <begin_code>] -F <file> [-E <end_code>] [--] [args...]
       php [options] -- [args...]
       php [options] -a

  -a               Run interactively
  -c <path>|<file> Look for php.ini file in this directory
  -n               No php.ini file will be used
  -d foo[=bar]     Define INI entry foo with value 'bar'
  -e               Generate extended information for debugger/profiler
  -f <file>        Parse and execute <file>.
  -h               This help
  -i               PHP information
  -l               Syntax check only (lint)
  -m               Show compiled in modules
  -r <code>        Run PHP <code> without using script tags <?..?>
  -B <begin_code>  Run PHP <begin_code> before processing input lines
  -R <code>        Run PHP <code> for every input line
  -F <file>        Parse and execute <file> for every input line
  -E <end_code>    Run PHP <end_code> after processing all input lines
  -H               Hide any passed arguments from external tools.
  -s               Display colour syntax highlighted source.
  -v               Version number
  -w               Display source with stripped comments and whitespace.
  -z <file>        Load Zend extension <file>.
```

```
args...          Arguments passed to script. Use -- args when first argument
                 starts with - or script is read from stdin

--ini            Show configuration file names

--rf <name>      Show information about function <name>.
--rc <name>      Show information about class <name>.
--re <name>      Show information about extension <name>.
--ri <name>      Show configuration for extension <name>.
```

The *CLI SAPI* has three different ways of getting the PHP code you want to execute:

- Telling PHP to execute a certain file.

  ```
  php my_script.php

  php -f my_script.php
  ```

  Both ways (whether using the *-f* switch or not) execute the file *my_script.php*. You can choose any file to execute - your PHP scripts do not have to end with the*.php* extension but can have any name or extension you wish.

Note
If you need to pass arguments to your scripts you need to pass -- as the first argument when using the *-f* switch.

- Pass the PHP code to execute directly on the command line.

  ```
  php -r 'print_r(get_defined_constants());'
  ```

  Special care has to be taken in regards of shell variable substitution and quoting usage.

Note
Read the example carefully, there are no beginning or ending tags! The *-r* switch simply does not need them. Using them will lead to a parser error.

- Provide the PHP code to execute via standard input (*stdin*). This gives the powerful ability to dynamically create PHP code and feed it to the binary, as shown in this (fictional) example:

  ```
  $ some_application | some_filter | php | sort -u >final_output.txt
  ```

You cannot combine any of the three ways to execute code.

Like every shell application, the PHP binary accepts a number of arguments but your PHP script can also receive arguments. The number of arguments which can be passed to your

script is not limited by PHP (the shell has a certain size limit in the number of characters which can be passed; usually you won't hit this limit). The arguments passed to your script are available in the global array *$argv*. The zero index always contains the script name (which is - in case the PHP code is coming from either standard input or from the command line switch *-r*). The second registered global variable is *$argc* which contains the number of elements in the *$argv* array (*not* the number of arguments passed to the script).

As long as the arguments you want to pass to your script do not start with the - character, there's nothing special to watch out for. Passing an argument to your script which starts with a - will cause trouble because PHP itself thinks it has to handle it. To prevent this, use the argument list separator --. After this separator has been parsed by PHP, every argument following it is passed untouched to your script.

```
# This will not execute the given code but will show the PHP usage
$ php -r 'var_dump($argv);' -h
Usage: php [options] [-f] <file> [args...]
[...]

# This will pass the '-h' argument to your script and prevent PHP from showing it's
usage
$ php -r 'var_dump($argv);' -- -h
array(2) {
 [0]=>
 string(1) "-"
 [1]=>
 string(2) "-h"
}
```

However, there's another way of using PHP for shell scripting. You can write a script where the first line starts with *#!/usr/bin/php*. Following this you can place normal PHP code included within the PHP starting and end tags. Once you have set the execution attributes of the file appropriately (e.g. *chmod +x test*) your script can be executed like a normal shell or perl script:

Example #13 - Execute PHP script as shell script

```
#!/usr/bin/php
<?php
var_dump($argv);
?>
```

Assuming this file is named *test* in the current directory, we can now do the following:

```
$ chmod +x test
$ ./test -h -- foo
array(4) {
 [0]=>
 string(6) "./test"
 [1]=>
 string(2) "-h"
 [2]=>
 string(2) "--"
 [3]=>
```

```
  string(3) "foo"
}
```

As you see, in this case no care needs to be taken when passing parameters which start with - to your script.

Long options are available since PHP 4.3.3.

Command line options

Option	Long Option	Description
-a	--interactive	Runs PHP interactively. If you compile PHP with the Readline extension (which is not available on Windows), you'll have a nice shell, including a completion feature (e.g. you can start typing a variable name, hit the TAB key and PHP completes its name) and a typing history that can be accessed using the arrow keys. The history is saved in the *~/.php_history* file. **Note** Files included through auto_prepend_file and auto_append_file are parsed in this mode but with some restrictions - e.g. functions have to be defined before called. **Note** Autoloading is not available if using PHP in CLI interactive mode.
-c	--php-ini	This option can either specify

		a directory where to look for *php.ini* or specify a custom *INI* file (which does not need to be named *php.ini*), e.g.: `$ php -c /custom/directory/ my_script.php` `$ php -c /custom/directory/custom-file.ini my_script.php` If you don't specify this option, file is searched in default locations.
-n	--no-php-ini	Ignore *php.ini* at all. This switch is available since PHP 4.3.0.
-d	--define	This option allows you to set a custom value for any of the configuration directives allowed in *php.ini*. The syntax is: `-d configuration_directive[=value]` Examples (lines are wrapped for layout reasons): `# Omitting the value part will set the given configuration directive to "1"` `$ php -d max_execution_time -r '$foo = ini_get("max_execution_time"); var_dump($foo);'` `string(1) "1"`

		`# Passing an empty value part will set the configuration directive to ""` `php -d max_execution_time= -r '$foo = ini_get("max_execution_time"); var_dump($foo);'` `string(0) ""` `# The configuration directive will be set to anything passed after the '=' character` `$ php -d max_execution_time=20 -r '$foo = ini_get("max_execution_time"); var_dump($foo);'` `string(2) "20"` `$ php -d max_execution_time=doesntmakesense -r '$foo = ini_get("max_execution_time"); var_dump($foo);'` `string(15) "doesntmakesense"`
-e	--profile-info	Activate the extended information mode, to be used by a debugger/profiler.
-f	--file	Parses and executes the given filename to the *-f* option. This switch is optional and can be left out. Only providing the filename to execute is sufficient. **Note** To pass arguments to scripts the first argument needs to be --, otherwise PHP will interperate them as PHP options.

-h and -?	--help and --usage	With this option, you can get information about the actual list of command line options and some one line descriptions about what they do.
-i	--info	This command line option calls phpinfo(), and prints out the results. If PHP is not working correctly, it is advisable to use *php -i* and see whether any error messages are printed out before or in place of the information tables. Beware that when using the CGI mode the output is in *HTML* and therefore quite huge.
-l	--syntax-check	This option provides a convenient way to only perform a syntax check on the given PHP code. On success, the text *No syntax errors detected in <filename>* is written to standard output and the shell return code is *0* . On failure, the text *Errors parsing <filename>* in addition to the internal parser error message is written to standard output and the shell return code is set to *255*. This option won't find fatal errors (like undefined functions). Use *-f* if you would like to test for fatal errors too. **Note** This option does not work together with the *-r* option.
-m	--modules	

		Using this option, PHP prints out the built in (and loaded) PHP and Zend modules: `$ php -m` `[PHP Modules]` `xml` `tokenizer` `standard` `session` `posix` `pcre` `overload` `mysql` `mbstring` `ctype` `[Zend Modules]`
-r	--run	This option allows execution of PHP right from within the command line. The PHP start and end tags (*<?php* and *?>*) are *not needed* and will cause a parser error if present. **Note** Care has to be taken when using this form of PHP to not collide with command line variable substitution done by the shell. Example showing a parser error `$ php -r "$foo = get_defined_constants( );"` `Command line code(1) : Parse error - parse error, unexpected '='` The problem here is that the sh/bash performs

variable substitution even when using double quotes ". Since the variable *$foo* is unlikely to be defined, it expands to nothing which results in the code passed to PHP for execution actually reading:

```
$ php -r " =
get_defined_constants(
);"
```

The correct way would be to use single quotes '. Variables in single-quoted strings are not expanded by sh/bash.

```
$ php -r '$foo =
get_defined_constants(
); var_dump($foo);'
array(370) {
 ["E_ERROR"]=>
 int(1)
 ["E_WARNING"]=>
 int(2)
 ["E_PARSE"]=>
 int(4)
 ["E_NOTICE"]=>
 int(8)
 ["E_CORE_ERROR"]=>
 [...]
```

If you are using a shell different from sh/bash, you might experience further issues. Feel free to open a bug report at » http://bugs.php.net/. One can still easily run into troubles when trying to get shell variables into the code or using backslashes for escaping. You've been warned.

Note

-r is available in the *CLI* SAPI and not in the *CGI* SAPI.

Note

This option is meant for a very basic stuff. Thus some configuration directives (e.g. auto_prepend_file and auto_append_file) are ignored in this mode.

-B	--process-begin	PHP code to execute before processing stdin. Added in PHP 5.
-R	--process-code	PHP code to execute for every input line. Added in PHP 5. There are two special variables available in this mode: *$argn* and *$argi*. *$argn* will contain the line PHP is processing at that moment, while *$argi* will contain the line number.
-F	--process-file	PHP file to execute for every input line. Added in PHP 5.
-E	--process-end	PHP code to execute after processing the input. Added in PHP 5.

Example #14 - Using the *-B*, *-R* and *-E* options to count the number of lines of a project.

```
$ find my_proj | php
-B '$l=0;' -R '$l +=
count(@file($argn));'
-E 'echo "Total Lines:
$l\n";'
Total Lines: 37328
```

-s	--syntax-highlight and --syntax-highlight	Display colour syntax highlighted source.

		This option uses the internal mechanism to parse the file and produces a *HTML* highlighted version of it and writes it to standard output. Note that all it does it to generate a block of *<code> [...] </code> HTML* tags, no *HTML* headers. **Note** This option does not work together with the *-r* option.
-v	--version	Writes the PHP, PHP SAPI, and Zend version to standard output, e.g. `$ php -v` `PHP 4.3.0 (cli), Copyright (c) 1997-2002 The PHP Group` `Zend Engine v1.3.0, Copyright (c) 1998-2002 Zend Technologies`
-w	--strip	Display source with stripped comments and whitespace. **Note** This option does not work together with the *-r* option.
-z	--zend-extension	Load Zend extension. If only a filename is given, PHP

		tries to load this extension from the current default library path on your system (usually specified */etc/ld.so.conf* on Linux systems). Passing a filename with an absolute path information will not use the systems library search path. A relative filename with a directory information will tell PHP only to try to load the extension relative to the current directory.
	--ini	Shows configuration file names and scanned directories. Available as of PHP 5.2.3. **Example #15 -** *--ini* **example** `$ php --ini` `Configuration File (php.ini) Path: /usr/dev/php/5.2/lib` `Loaded Configuration File: /usr/dev/php/5.2/lib/php.ini` `Scan for additional .ini files in: (none)` `Additional .ini files parsed: (none)`
--rf	--rfunction	Shows information about the given function or class method (e.g. number and name of the parameters). Available as of PHP 5.1.2. This option is only available if PHP was compiled with Reflection support.

Example #16 - basic *--rf* usage

```
$ php --rf var_dump
Function [ <internal>
public function
var_dump ] {

 - Parameters [2] {
   Parameter #0 [
<required> $var ]
   Parameter #1 [
<optional> $... ]
 }
}
```

--rc	--rclass	Show information about the given class (list of constants, properties and methods). Available as of PHP 5.1.2. This option is only available if PHP was compiled with Reflection support.

Example #17 - *--rc* **example**

```
$ php --rc Directory
Class [
<internal:standard>
class Directory ] {

 - Constants [0] {
 }

 - Static properties
[0] {
 }

 - Static methods [0]
{
 }

 - Properties [0] {
 }

 - Methods [3] {
   Method [ <internal>
public method close ]
{
   }

   Method [ <internal>
public method rewind ]
{
   }

   Method [ <internal>
public method read ] {
   }
 }
}
```

--re	--rextension	Show information about the given extension (list of *php.ini* options, defined functions, constants and classes). Available as of PHP 5.1.2. This option is only available if

PHP was compiled with Reflection support.

Example #18 - *--re* **example**

```
$ php --re json
Extension [
<persistent> extension
#19 json version 1.2.1
] {

 - Functions {
   Function [
<internal> function
json_encode ] {
   }
   Function [
<internal> function
json_decode ] {
   }
 }
}
```

--ri	--rextinfo

Shows the configuration information for the given extension (the same information that is returned by phpinfo()). Available as of PHP 5.2.2. The core configuration information are available using "main" as extension name.

Example #19 - *--ri* **example**

```
$ php --ri date

date

date/time support =>
enabled
"Olson" Timezone
Database Version =>
2007.5
Timezone Database =>
internal
```

```
Default timezone =>
Europe/Oslo

Directive => Local
Value => Master Value
date.timezone =>
Europe/Oslo =>
Europe/Oslo
date.default_latitude
=> 59.22482 =>
59.22482
date.default_longitude
=> 11.018084 =>
11.018084
date.sunset_zenith =>
90.583333 => 90.583333
date.sunrise_zenith =>
90.583333 => 90.583333
```

The PHP executable can be used to run PHP scripts absolutely independent from the web server. If you are on a Unix system, you should add a special first line to your PHP script, and make it executable, so the system will know, what program should run the script. On a Windows platform you can associate *php.exe* with the double click option of the *.php* files, or you can make a batch file to run the script through PHP. The first line added to the script to work on Unix won't hurt on Windows, so you can write cross platform programs this way. A simple example of writing a command line PHP program can be found below.

Example #20 - Script intended to be run from command line (script.php)

```
#!/usr/bin/php
<?php

if ($argc != 2 || in_array($argv[1], array('--help', '-help', '-h', '-?')))
{
?>

This is a command line PHP script with one option.

 Usage:
 <?php echo $argv[0]; ?> <option>
```

```
 <option> can be some word you would like
 to print out. With the --help, -help, -h,
 or -? options, you can get this help.

<?php
} else {
   echo $argv[1];
}
?>
```

In the script above, we used the special first line to indicate that this file should be run by PHP. We work with a CLI version here, so there will be no HTTP header printouts. There are two variables you can use while writing command line applications with PHP: *$argc* and *$argv*. The first is the number of arguments plus one (the name of the script running). The second is an array containing the arguments, starting with the script name as number zero (*$argv[0]*).

In the program above we checked if there are less or more than one arguments. Also if the argument was *--help*, *-help*, *-h* or *-?*, we printed out the help message, printing the script name dynamically. If we received some other argument we echoed that out.

If you would like to run the above script on Unix, you need to make it executable, and simply call it as *script.php echothis* or *script.php -h*. On Windows, you can make a batch file for this task:

Example #21 - Batch file to run a command line PHP script (script.bat)

```
@C:\php\php.exe script.php %1 %2 %3 %4
```

Assuming you named the above program *script.php*, and you have your CLI *php.exe* in *C:\php\php.exe* this batch file will run it for you with your added options: *script.bat echothis* or *script.bat -h*.

See also the Readline extension documentation for more functions you can use to enhance your command line applications in PHP.