System program execution

Introduction

Those functions provide means to execute commands on the system itself, and means to secure such commands.

Installing/Configuring

Requirements

No external libraries are needed to build this extension.

Installation

There is no installation needed to use these functions; they are part of the PHP core.

Runtime Configuration

This extension has no configuration directives defined in *php.ini*.

Resource Types

This extension defines a *process* resource, returned by proc_open().

Predefined Constants

This extension has no constants defined.

Program execution Functions

Notes

Warning
Open files with lock (especially open sessions) should be closed before executing a program in the background.

See Also

These functions are also closely related to the backtick operator. Also, while in safe mode you must consider the safe_mode_exec_dir directive.

escapeshellarg

escapeshellarg -- Escape a string to be used as a shell argument

Description

string **escapeshellarg** (string `$arg`)

escapeshellarg() adds single quotes around a string and quotes/escapes any existing single quotes allowing you to pass a string directly to a shell function and having it be treated as a single safe argument. This function should be used to escape individual arguments to shell functions coming from user input. The shell functions include exec(), system() and the backtick operator.

Parameters

`arg`
: The argument that will be escaped.

Return Values

The escaped string.

Examples

Example #1 - escapeshellarg() example

```
<?php
system('ls '.escapeshellarg($dir));
?>
```

See Also

- escapeshellcmd()
- exec()
- popen()
- system()
- backtick operator

escapeshellcmd

escapeshellcmd -- Escape shell metacharacters

Description

string **escapeshellcmd** (string `$command`)

escapeshellcmd() escapes any characters in a string that might be used to trick a shell command into executing arbitrary commands. This function should be used to make sure that any data coming from user input is escaped before this data is passed to the exec() or system() functions, or to the backtick operator.

Following characters are preceded by a backslash: *#&;`|*?~<>^()[]{}$*, *\x0A* and *\xFF*. *'* and *"* are escaped only if they are not paired. In Windows, all these characters plus *%* are replaced by a space instead.

Parameters

`command`
: The command that will be escaped.

Return Values

The escaped string.

Examples

Example #2 - escapeshellcmd() example

```
<?php
$e = escapeshellcmd($userinput);

// here we don't care if $e has spaces
system("echo $e");
$f = escapeshellcmd($filename);

// and here we do, so we use quotes
system("touch \"/tmp/$f\"; ls -l \"/tmp/$f\"");
?>
```

See Also

- escapeshellarg()
- exec()
- popen()
- system()
- backtick operator

exec

exec -- Execute an external program

Description

string **exec** (string `$command` [, array `&$output` [, int `&$return_var`]])

exec() executes the given *`command`*.

Parameters

`command`
: The command that will be executed.

`output`
: If the *`output`* argument is present, then the specified array will be filled with every line of output from the command. Trailing whitespace, such as \n, is not included in this array. Note that if the array already contains some elements, exec() will append to the end of the array. If you do not want the function to append elements, call unset() on the array before passing it to exec().

`return_var`
: If the *`return_var`* argument is present along with the *`output`* argument, then the return status of the executed command will be written to this variable.

Return Values

The last line from the result of the command. If you need to execute a command and have all the data from the command passed directly back without any interference, use the passthru() function.

To get the output of the executed command, be sure to set and use the *`output`* parameter.

Examples

Example #3 - An exec() example

```
<?php
// outputs the username that owns the running php/httpd process
// (on a system with the "whoami" executable in the path)
echo exec('whoami');
?>
```

Notes

Warning

When allowing user-supplied data to be passed to this function, use escapeshellarg() or escapeshellcmd() to ensure that users cannot trick the system into executing arbitrary commands.

Note

If a program is started with this function, in order for it to continue running in the background, the output of the program must be redirected to a file or another output stream. Failing to do so will cause PHP to hang until the execution of the program ends.

Note

When safe mode is enabled, you can only execute files within the safe_mode_exec_dir. For practical reasons, it is currently not allowed to have.. components in the path to the executable.

Warning

With safe mode enabled, the command string is escaped with escapeshellcmd(). Thus, *echo y | echo x* becomes *echo y \| echo x*.

See Also

- system()
- passthru()
- escapeshellcmd()
- pcntl_exec()
- backtick operator

passthru

passthru -- Execute an external program and display raw output

Description

void **passthru** (string `$command` [, int `&$return_var`])

The passthru() function is similar to the exec() function in that it executes a *`command`*. This function should be used in place of exec() or system() when the output from the Unix command is binary data which needs to be passed directly back to the browser. A common use for this is to execute something like the pbmplus utilities that can output an image stream directly. By setting the Content-type to *image/gif* and then calling a pbmplus program to output a gif, you can create PHP scripts that output images directly.

Parameters

`command`
: The command that will be executed.

`return_var`
: If the *`return_var`* argument is present, the return status of the Unix command will be placed here.

Return Values

No value is returned.

Notes

Warning
When allowing user-supplied data to be passed to this function, use escapeshellarg() or escapeshellcmd() to ensure that users cannot trick the system into executing arbitrary commands.

Note
If a program is started with this function, in order for it to continue running in the background, the output of the program must be redirected to a file or another output stream. Failing to do so will cause PHP to hang until the execution of the program ends.

Note

When safe mode is enabled, you can only execute files within the safe_mode_exec_dir . For practical reasons, it is currently not allowed to have.. components in the path to the executable.

Warning

With safe mode enabled, the command string is escaped with escapeshellcmd(). Thus, *echo y | echo x* becomes *echo y \| echo x*.

See Also

- exec()
- system()
- popen()
- escapeshellcmd()
- backtick operator

proc_close

proc_close -- Close a process opened by proc_open() and return the exit code of that process.

Description

int **proc_close** (resource `$process`)

proc_close() is similar to pclose() except that it only works on processes opened by proc_open(). proc_close() waits for the process to terminate, and returns its exit code. If you have open pipes to that process, you should fclose() them prior to calling this function in order to avoid a deadlock - the child process may not be able to exit while the pipes are open.

Parameters

`process`
The proc_open() resource that will be closed.

Return Values

Returns the termination status of the process that was run.

proc_get_status

proc_get_status -- Get information about a process opened by proc_open()

Description

array **proc_get_status** (resource `$process`)

proc_get_status() fetches data about a process opened using proc_open().

Parameters

`process`
The proc_open() resource that will be evaluated.

Return Values

An array of collected information on success, and **FALSE** on failure. The returned array contains the following elements:

element	type	description
command	string	The command string that was passed to proc_open().
pid	int	process id
running	bool	**TRUE** if the process is still running, **FALSE** if it has terminated.
signaled	bool	**TRUE** if the child process has been terminated by an uncaught signal. Always set to **FALSE** on Windows.
stopped	bool	**TRUE** if the child process has been stopped by a signal. Always set to **FALSE** on Windows.
exitcode	int	The exit code returned by the process (which is only meaningful if *running* is **FALSE**). Only first call of

		this function return real value, next calls return *-1*.
termsig	int	The number of the signal that caused the child process to terminate its execution (only meaningful if *signaled* is **TRUE**).
stopsig	int	The number of the signal that caused the child process to stop its execution (only meaningful if *stopped* is **TRUE**).

See Also

- proc_open()

proc_nice

proc_nice -- Change the priority of the current process

Description

bool **proc_nice** (int `$increment`)

proc_nice() changes the priority of the current process by the amount specified in *`increment`*. A positive *`increment`* will lower the priority of the current process, whereas a negative *`increment`* will raise the priority.

proc_nice() is not related to proc_open() and its associated functions in any way.

Parameters

`increment`
The increment value of the priority change.

Return Values

Returns **TRUE** on success or **FALSE** on failure. If an error occurs, like the user lacks permission to change the priority, an error of level **E_WARNING** is also generated.

Notes

Note
Availability proc_nice() will only exist if your system has 'nice' capabilities. 'nice' conforms to: SVr4, SVID EXT, AT&T, X/OPEN, BSD 4.3. This means that proc_nice() is not available on Windows.

proc_open

proc_open -- Execute a command and open file pointers for input/output

Description

resource **proc_open** (string `$cmd`, array `$descriptorspec`, array `&$pipes` [, string `$cwd` [, array `$env` [, array `$other_options`]]])

proc_open() is similar to popen() but provides a much greater degree of control over the program execution.

Parameters

`cmd`
: The command to execute

`descriptorspec`
: An indexed array where the key represents the descriptor number and the value represents how PHP will pass that descriptor to the child process. 0 is stdin, 1 is stdout, while 2 is stderr. The currently supported pipe types are *file* and *pipe*. The file descriptor numbers are not limited to 0, 1 and 2 - you may specify any valid file descriptor number and it will be passed to the child process. This allows your script to interoperate with other scripts that run as "co-processes". In particular, this is useful for passing passphrases to programs like PGP, GPG and openssl in a more secure manner. It is also useful for reading status information provided by those programs on auxiliary file descriptors.

`pipes`
: Will be set to an indexed array of file pointers that correspond to PHP's end of any pipes that are created.

`cwd`
: The initial working dir for the command. This must be an *absolute* directory path, or **NULL** if you want to use the default value (the working dir of the current PHP process)

`env`
: An array with the environment variables for the command that will be run, or **NULL** to use the same environment as the current PHP process

`other_options`
: Allows you to specify additional options. Currently supported options include:

- *suppress_errors* (windows only): suppresses errors generated by this function when it's set to **TRUE**
- *bypass_shell* (windows only): bypass *cmd.exe* shell when set to **TRUE**
- *context*: stream context used when opening files (created with stream_context_create())

- *binary_pipes*: open pipes in binary mode, instead of using the usual *stream_encoding*

Return Values

Returns a resource representing the process, which should be freed using proc_close() when you are finished with it. On failure returns **FALSE**.

ChangeLog

Version	Description
6.0.0	Added the *context* and *binary_pipes* options to the `other_options` parameter.
5.2.1	Added the *bypass_shell* option to the `other_options` parameter.
5.0.0	Added the `cwd`, `env` and `other_options` parameters.

Examples

Example #4 - A proc_open() example

```
<?php
$descriptorspec = array(
  0 => array("pipe", "r"),  // stdin is a pipe that the child will read from
  1 => array("pipe", "w"),  // stdout is a pipe that the child will write to
  2 => array("file", "/tmp/error-output.txt", "a") // stderr is a file to
write to
);

$cwd = '/tmp';
$env = array('some_option' => 'aeiou');

$process = proc_open('php', $descriptorspec, $pipes, $cwd, $env);

if (is_resource($process)) {
   // $pipes now looks like this:
   // 0 => writeable handle connected to child stdin
   // 1 => readable handle connected to child stdout
   // Any error output will be appended to /tmp/error-output.txt

   fwrite($pipes[0], '<?php print_r($_ENV); ?>');
   fclose($pipes[0]);
```

```
    echo stream_get_contents($pipes[1]);
    fclose($pipes[1]);

    // It is important that you close any pipes before calling
    // proc_close in order to avoid a deadlock
    $return_value = proc_close($process);

    echo "command returned $return_value\n";
}
?>
```

The above example will output something similar to:

```
Array
(
    [some_option] => aeiou
    [PWD] => /tmp
    [SHLVL] => 1
    [_] => /usr/local/bin/php
)
command returned 0
```

Notes

Note

Windows compatibility: Descriptors beyond 2 (stderr) are made available to the child process as inheritable handles, but since the Windows architecture does not associate file descriptor numbers with low-level handles, the child process does not (yet) have a means of accessing those handles. Stdin, stdout and stderr work as expected.

Note

If you only need a uni-directional (one-way) process pipe, use popen() instead, as it is much easier to use.

See Also

- popen()
- exec()
- system()
- passthru()
- stream_select()
- The backtick operator

proc_terminate

proc_terminate -- Kills a process opened by proc_open

Description

bool **proc_terminate** (resource `$process` [, int `$signal`])

Signals a *process* (created using proc_open()) that it should terminate. proc_terminate() returns immediately and does not wait for the process to terminate.

proc_terminate() allows you terminate the process and continue with other tasks. You may poll the process (to see if it has stopped yet) by using the proc_get_status() function. However this is only possible with PHP 5.2.2 or newer, as previous versions destroyed the given process *resource*.

Parameters

process
: The proc_open() resource that will be closed.

signal
: This optional parameter is only useful on POSIX operating systems; you may specify a signal to send to the process using the *kill(2)* system call. The default is *SIGTERM*.

Return Values

Returns the termination status of the process that was run.

See Also

- proc_open()
- proc_close()
- proc_get_status()

shell_exec

shell_exec -- Execute command via shell and return the complete output as a string

Description

string **shell_exec** (string `$cmd`)

This function is identical to the backtick operator.

Parameters

`cmd`
The command that will be executed.

Return Values

The output from the executed command.

Examples

Example #5 - A shell_exec() example

```
<?php
$output = shell_exec('ls -lart');
echo "<pre>$output</pre>";
?>
```

Notes

Note

This function is disabled when PHP is running in safe mode.

See Also

- exec()
- escapeshellcmd()

system

system -- Execute an external program and display the output

Description

string **system** (string `$command` [, int `&$return_var`])

system() is just like the C version of the function in that it executes the given *`command`* and outputs the result.

The system() call also tries to automatically flush the web server's output buffer after each line of output if PHP is running as a server module.

If you need to execute a command and have all the data from the command passed directly back without any interference, use the passthru() function.

Parameters

`command`
: The command that will be executed.

`return_var`
: If the *`return_var`* argument is present, then the return status of the executed command will be written to this variable.

Return Values

Returns the last line of the command output on success, and **FALSE** on failure.

Examples

Example #6 - system() example

```
<?php
echo '<pre>';

// Outputs all the result of shellcommand "ls", and returns
// the last output line into $last_line. Stores the return value
// of the shell command in $retval.
$last_line = system('ls', $retval);

// Printing additional info
echo '
</pre>
<hr />Last line of the output: ' . $last_line . '
```

```
<hr />Return value: ' . $retval;
?>
```

Notes

Warning

When allowing user-supplied data to be passed to this function, use escapeshellarg() or escapeshellcmd() to ensure that users cannot trick the system into executing arbitrary commands.

Note

If a program is started with this function, in order for it to continue running in the background, the output of the program must be redirected to a file or another output stream. Failing to do so will cause PHP to hang until the execution of the program ends.

Note

When safe mode is enabled, you can only execute files within the safe_mode_exec_dir. For practical reasons, it is currently not allowed to have.. components in the path to the executable.

Warning

With safe mode enabled, the command string is escaped with escapeshellcmd(). Thus, *echo y | echo x* becomes *echo y \| echo x*.

See Also

- exec()
- passthru()
- popen()
- escapeshellcmd()
- pcntl_exec()
- backtick operator