Using PHP in administration scripts

Delivering Commands

Most admins tend to use the shell, Perl, or Python if they need a system administration script. But there is no need for web programmers to learn another language just to script a routine task. PHP gives admins the power to program command-line tools and even complete web interfaces. BY CARSTEN MÖHRKE

www.sxc.hu

PHP scripts written for the console are basically the same as scripts written for the web: programmers can use the whole range of PHP features. To allow simple, program name based execution of scripts on the command-line, you need to specify the PHP interpreter in the first line of the script following *#!*. If you do not know the path to the interpreter, you can type *which php* to find out. Also, the script file needs to be executable. Let's take a look at the ubiquitous "Hello, World" example in PHP:

```
#!/usr/bin/php -q
<?php
   echo "Hello, World\n";
?>
```

PHP tools are typically all you need for any kind of text manipulation, such as modifying configuration files, log files, user management entries, and the like. If you need to access operating system functions for tasks such as process management, you will need to add a few Linux commands.

Command Line

PHP has four functions designed for executing command lines: *exec()*, *system()*, *passthru()*, and *shell_exec()*:

- *exec()* executes the command that is passed to it as the first parameter and suppresses any output from the command. The last line of output is returned as a result. If the second parameter is the name of an array, *exec()* fills the array line by line with the output that would have appeared on screen. If you need to process the exit code for *exec()*, you can use the third parameter to do so.
- *system()* does not suppress output, in contrast to *exec()*. It has a second optional parameter for the exit code.
- *passthru()* works like *system()*, but in a binary compatible way. This allows the command to pass images generated as converter output to a web browser.
- *shell_exec()* works like the single parameter version of *exec()*, but instead of outputting the last line, it returns a string with the complete command output. The string contains newline characters, in contrast to the entries in the *exec()* return value array.

Both *exec()* and *shell_exec()* are useful in cases where you need to manipulate the text output from an external command. The following examples use *exec()*, which is more flexible than the other functions. There is a small glitch involved with this at present: PHP does not always return the external program's

Old and New Shell Interface

PHP version 4.3.0 or later has a new shell interface for programming command line tools: the CLI (Command Line Interface). Previous versions of PHP (and even some newer versions) had a program interface called Server Application Programmers Interface (SAPI), but SAPI was designed for use on web servers (CGI-SAPI) and added a HTTP header to output. (You can suppress the header by setting the *-q* option when you launch PHP, but this will not remove HTML tags from error messages.)

You can can check if you have the CGI version, which does not have the same feature-scope as the command-line interface, by typing *php -v*. The examples in this article will work with either version.

exit code correctly (for example when concatenating commands).

Process Control

The following PHP script uses *exec()* to output the number of processes running on a system:

```
unset ($out);
$cmd="ps axu | wc -l";
$erg=exec($cmd,$out);
$erg-=1; //minus 1 line header
echo "Number of active ➥
processes is $erg\n";
```

The script first deletes the content of the *$out* variable, which represents the array that will be storing the command output. *exec()* would not overwrite the array but simply append to it. The example avoids mixing up the results from multiple scripts by first flushing the array. The command discovers the number of lines in the process list and then subtracts one line for the column headings.

shell_exec() and *exec()* can only handle data from *stdout*. To catch standard error output from *stderr*, you need to redirect error output to *stdout*: *2>&1*.

Programmers often need to pass command-line arguments to a script. To allow this to happen, PHP enters the arguments that follow a filename in the command line into an array called *$argv*, keeping the original order. Additionally, the number of arguments is stored in the *$argc* variable.

Good Arguments

PHP does not support commands that read data directly from the keyboard. To provide support for keyboard entries, programmers can open the *stdin* stream as a file function and read from the file. This allows use of the related *stdout* and *stderr* streams:

```
$in=fopen("php://stdin","r");
$err=fopen("php://stdin","w");
$input=trim(fgets($fp, 100));
while (""==$input)
{
    fputs($err,"Please input a ➥
    value\n");
}
```

If you use *stdin* to read keyboard input in this way, your script should *trim()* the data to remove whitespace at the end of the string. There is no need to use *stdout* for data output, as PHP has the *echo* and *print* functions for this task. Output from both commands can be redirected.

Table 1: Important ANSI Sequences

Meaning	Sequence
Clear screen	\033[2J
Cursor to position x,y	\033[x;yH
Cursor n characters left	\033[nC
Cursor n characters up	\033[nA
Font color red	\033[0;31m
Font color black	\033[0;30m
Font color green	\033[0;32m
Font color light gray	\033[0;37m

Colorful

To design a more comfortable application, programmers need functions to delete the screen, control the cursor, and add color to output. ANSI sequences [1] provide a useful solution, as the shell is capable of interpreting them. These sequences start with the Escape character and are better known as Escape sequences. For example, *echo "\033[2J"* clears the screen. Table 1 lists a few of the most important sequences.

The selected color stays valid until a different color is selected. The PHP *ncurses* function provides more portability and a much wider range of features. See the examples on the PHP or Zend homepages at [2], [3] for more details.

PHP-based User Management

The following example demonstrates the functions referred to thus far in the context of a script for user management on a server for training purposes. Figure 1 shows the menu this code creates. Listing 1 is just an excerpt, and the complete script is available for download at [4].

You may note that the script opens standard input with *fopen()* but does not use *fclose()* to close it. As PHP automatically closes open streams when the program finishes, the script avoids closing the stream for readability reasons. The execution time for command-line programs is not restricted.

PHP is ideal for creating web interfaces. This attractive option has a few hidden pitfalls. When designing the application, you need to consider the potential security risk. The smallest of security holes could have fatal consequences. An example:

```
01 <form method="POST">
02   Command: <input name=
     "cmd" /><br />
03   <input type=
     "submit"value=
     "Execute" />
04 </form>
05 <?php
06   if (isset ($_POST['cmd']))
07   {
08      $outp=shell_exec
       ($_POST[cmd]);
09      echo "<pre>$outp</pre>";
10     }
11 ?>
```

Solutions that will execute arbitrary commands (like this one) are out of the question. Administrative scripts should only be executable within a secure environment, and they should not allow any leeway for manipulation.

Special Privileges

Scripts often need root privileges. A web-based PHP script normally runs with the privileges of the web server user ID. It might seem obvious to give this user – typically *www*, *wwwrun*, or *nobody* – more wide-ranging privileges, but this is extremely dangerous. Commands like *sudo* or *su* provide an alternative approach. However, *su* requires the insecure practice of adding the clear text root password to your script.

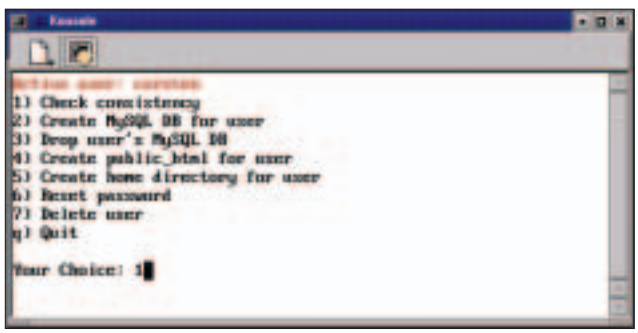

Figure 1: The menu of a PHP script for user management.

The *sudo* tool also can assign root privileges for specific actions to a user without that user needing to know the root password. Instead, users authenticate with their own passwords and are assigned the privileges specified by root in the */etc/sudoers* file. If an admin needs to assign the *webadmin* user the right to terminate processes with the *kill* command, he or she could add the following line to */etc/sudoers*:

```
webadmin ALL = /bin/kill, ➥
/usr/bin/killall
```

It is a good idea to use the special *visudo* editor for this. The editor ensures that only one user can edit the configuration file at any one time. It additionally ensures that entries comply with the rules and do not conflict.

The generic syntax for assigning privileges with sudo is: WHO WHERE = WHAT. WHO can be a user name, like in our example, or a group of users, which needs to be defined using *User_Alias*. WHERE means the host where the user will have these privileges.

Changing Identity

Thanks to the entry in the sudo configuration file, *webadmin* can now *kill* arbitrary processes after entering the password for the user account. An automated script would stop and prompt for the password before continuing. There is a workaround for this that involves using sudo's *-S* parameter to tell sudo to read the password from standard input. The command line so far is as follows: *echo geheim | sudo -S kill 13221*

If the *webadmin* user is not the root user of the web server from the viewpoint of the Unix processes – this would mean scripts inheriting the user's privileges, which is probably undesirable in most cases – you will probably need a combination of *su* and *sudo*. The web server user first assumes the identity of *webadmin* temporarily by running *su*, and is then assigned root privileges for specific commands via *sudo*. A script that uses this approach might look like this:

```
$user="webadmin";  //sudoer
$pwd="geheim"; ➥
// Webadmin password
$befehl="kill -9§ §
".escapeshellarg($_GET
["pid"]);
$cmd_line="echo $pwd | ➥
su -l $user -c
\"echo $pwd | sudo -S ➥
$command 2>&1\"";
$ret=shell_exec($cmd_line);
```

The script first uses *escape-shellargs()* to manipulate the *pid*, which it parses from a form. This removes the potential for shell injection attacks, which might otherwise insert malevolent code. ■

INFO

[1] ANSI codes: *http://en.wikipedia.org/wiki/ANSI_escape_code*

[2] Ncurses function manual: *http://www.php.net/ncurses*

[3] Ncurses tutorial: *http://www.zend.com/zend/tut/tut-degan.php*

[4] Listing: *http://www.linux-magazine.com/Magazine/Downloads/50*

THE AUTHOR

Carsten Möhrke is a freelance consultant and trainer, the author of "Better PHP Programming", and the CEO of Netviser. You can contact Carsten at cmoehrke@netviser.de.

Listing 1: PHP Menu

```
01 #!/usr/bin/php -q
02 <?php
03 function cls()
04 {
05     echo "\033[2J"; //Clear
   screen
06     echo "\033[0;0H"; // Cursor
   to 0,0
07 }
08
09 function text_red()
10 {
11     echo "\033[0;31m";
12 }
13
14 function text_black()
15 {
16     echo "\033[0;30m";
17 }
18
19 // Additional functions
20
21 // Has a user name been input?
22 $in=fopen("php://stdin","r");
23 $err=fopen("php://stderr","w");
24 if (3==$argc &&
25     "-u"==$argv[1] &&
26     isset($argv[2]))
27 {
28     $user=$argv[2]; // read
   username
29 }
30 else
31 {   // no username input
32     fputs($err,"Please use -u
   to input a user name\n");
33     exit(1);
34 }
35 // Function to check if user
   exists
36 // with etc/passwd
37 if (false ===
   user_exists($user))
38 {
39     fputs($err,"User name does
   not exist\n");
40     exit(2);
41 }
42
43 while (1) //Endless processing
   loops
44 {
45     cls();
46     text_red();
47     echo "Administration for
   user $user\n";
48     text_black();
49     echo "1) Check
   consistency\n";
50     echo "2) Create MySQL DB
   for user\n";
51     // More Menu Items
52     echo "q) Quit
   Program\n\n";
53     echo "Please select: ";
54     // Remove whitespace from
   input
55
   $input=trim(fgets($in,255));
56
57     switch ($input)
58     {
59         case "1":
   consistency_check($in);
60                     break;
61      // More cases
62      case "q": exit(0);
63      // beep for invalid input
64         default:  echo chr(7);
65     }
66 }
67 ?>
```