CHAPTER 3

PHP

PHP is the most popular web scripting language and an essential part of the Apache platform. Consequently, it is likely most web application installations will require PHP's presence. However, if your PHP needs are moderate, consider replacing the functionality you need using plain-old CGI scripts. The PHP module is a complex one and one that had many problems in the past.

This chapter will help you use PHP securely. In addition to the information provided here, you may find the following resources useful:

- Security section of the PHP manual (*http://www.php.net/manual/en/security.php*)
- PHP Security Consortium (*http://www.phpsec.org*)

Installation

In this section, I will present the installation and configuration procedures for two different options: using PHP as a module and using it as a CGI. Using PHP as a module is suitable for systems that are dedicated to a single purpose or for sites run by trusted groups of administrators and developers. Using PHP as a CGI (possibly with an execution wrapper) is a better option when users cannot be fully trusted, in spite of its worse performance. (Chapter 6 discusses running PHP over FastCGI which is an alternative approach that can, in some circumstances, provide the speed of the module combined with the privilege separation of a CGI.) To begin with the installation process, download the PHP source code from *http://www.php.net*.

Using PHP as a Module

When PHP is installed as a module, it becomes a part of Apache and performs all operations as the Apache user (usually *httpd*). The configuration process is similar to that of Apache itself. You need to prepare PHP source code for compilation by calling the *configure* script (in the directory where you unpacked the distribution), at a minimum

letting it know where Apache's *apxs* tool resides. The *apxs* tool is used as the interface between Apache and third-party modules:

```
$ ./configure --with-apxs=/usr/local/apache/bin/apxs
$ make
# make install
```

Replace `--with-apxs` with `--with-apxs2` if you are running Apache 2. If you plan to use PHP only from within the web server, it may be useful to put the installation together with Apache. Use the `--prefix` configuration parameter for that:

```
$ ./configure \
> --with-apxs=/usr/local/apache/bin/apxs \
> --prefix=/usr/local/apache/php
```

In addition to making PHP work with Apache, a command-line version of PHP will be compiled and copied to */usr/local/apache/php/bin/php*. The command-line version is useful if you want to use PHP for general scripting, unrelated to web servers.

The following configuration data makes Apache load PHP when it starts and allows Apache to identify which pages contain PHP code:

```
# Load the PHP module (the module is in
# subdirectory modules/ in Apache 2)
LoadModule php5_module libexec/libphp5.so
# Activate the module (not needed with Apache 2)
AddModule mod_php5.c

# Associate file extensions with PHP
AddHandler application/x-httpd-php .php
AddHandler application/x-httpd-php .php3
AddHandler application/x-httpd-php .inc
AddHandler application/x-httpd-php .class
AddHandler application/x-httpd-php .module
```

I choose to associate several extensions with the PHP module. One of the extensions (*.php3*) is there for backward compatibility. Java class files end in *.class* but there is little chance of clash because these files should never be accessed directly by Apache. The others are there to increase security. Many developers use extensions other than *.php* for their PHP code. These files are not meant to be accessed directly but through an `include( )` statement. Unfortunately, these files are often stored under the web server tree for convenience and anyone who knows their names can request them from the web server. This often leads to a security problem. (This issue is discussed in more detail in Chapters 10 and 11.)

Next, update the `DirectoryIndex` directive:

```
DirectoryIndex index.html index.php
```

Finally, place a version of *php.ini* in */usr/local/apache/php/lib/*. A frequent installation error occurs when the configuration file is placed at a wrong location, where it fails to have any effect on the PHP engine. To make sure a configuration file is active, create a

page with a single call to the `phpinfo( )` function and compare the output with the settings configured in your *php.ini* file.

Using PHP as a CGI

Compiling PHP as a CGI is similar to compiling it for the situation where you are going to use it as a module. This mode of operation is the default for PHP, so there is no need to specify an option on the *configure* line. There are two ways to configure and compile PHP depending on the approach you want to use to invoke PHP scripts from Apache.

One approach is to treat PHP scripts like other CGI scripts, in which case the execution will be carried out through the operating system. The same rules as for other CGI scripts apply: the file must be marked as executable, and CGI execution must be enabled with an appropriate `ExecCGI` option in the configuration. To compile PHP for this approach, configure it with the `--enable-discard-path` option:

```
$ ./configure \
> --enable-discard-path \
> --prefix=/usr/local/apache/php
$ make
# make install
```

The operating system must have a way of determining how to execute the script. Some systems use file extensions for this purpose. On most Unix systems, the first line, called the shebang line, in the script must tell the system how to execute it. Here's a sample script that includes such a line:

```
#!/usr/local/apache/php/bin/php
<? echo "Hello world"; ?>
```

This method of execution is not popular. When PHP is operating as an Apache module, PHP scripts do not require the shebang line at the top. Migrating from a module to CGI operation, therefore, requires modifying every script. Not only is that time consuming but also confusing for programmers.

The second approach to running PHP as a CGI is Apache-specific and relies on Apache's ability to have a CGI script post-process static files. First, configure, compile, and install PHP, this time specifying the `--enable-force-cgi-redirect` option:

```
$ ./configure \
> --enable-force-cgi-redirect \
> --prefix=/usr/local/apache/php
$ make
# make install
```

Place a copy of the PHP interpreter (*/usr/local/apache/php/bin/php*) into the web server's *cgi-bin/* directory. Configure Apache to use the interpreter to post-process all PHP files. In the example below, I am using one extension (*.php*), but you can add

more by adding multiple `AddHandler` directives (as shown in the section "Using PHP as a Module"):

```
Action application/x-httpd-php /cgi-bin/php
AddHandler application/x-httpd-php .php
```

I have used the same MIME type (*application/x-httpd-php*) as before, when configuring PHP to work as a module. This is not necessary but it makes it easier to switch from PHP working as a module to PHP working as a CGI. Any name (e.g., *php-script*) can be used provided it is used in both directives. If you do that, you can have PHP working as a module and as a script at the same time without a conflict.

Placing an interpreter (of any kind) into a *cgi-bin/* directory can be dangerous. If this directory is public, then anyone can invoke the interpreter directly and essentially ask it to process any file on disk as a script. This would result in an information leak or command execution vulnerability. Unfortunately, there is no other way since this is how Apache's `Action` execution mechanism works. However, a defense against this type of attack is built into PHP, and that's what the `--enable-force-cgi-redirect` switch we used to compile PHP is for. With this defense enabled, attempts to access the PHP interpreter directly will always fail. I recommend that you test the protection works by attempting to invoke the interpreter directly yourself. The *configure* script silently ignores unrecognized directives, so the system can be open to attack if you make a typing error when specifying the `--enable-force-cgi-redirect` option.

To ensure no one can exploit the PHP interpreter by calling it directly, create a separate folder, for example *php-cgi-bin/*, put only the interpreter there, and deny all access to it using `Deny from all`. Network access controls are not applied to internal redirections (which is how the `Action` directive works), so PHP will continue to work but all attack attempts will fail.

Choosing Modules

PHP has its own extension mechanism that breaks functionality into modules, and it equally applies when it is running as an Apache module or as a CGI. As was the case with Apache, some PHP modules are more dangerous than others. Looking at the *configure* script, it is not easy to tell which modules are loaded by default. The command line and CGI versions of PHP can be invoked with a `-m` switch to produce a list of compiled-in modules (the output in the example below is from PHP 5.0.2):

```
$ ./php -m
[PHP Modules]
ctype
iconv
pcre
posix
session
SPL
```

```
SQLite
standard
tokenizer
xml

[Zend Modules]
```

If you have PHP running as an Apache module, you must run the following simple script as a web page, which will provide a similar output:

```
<pre>
<?
$extension_list = get_loaded_extensions( );
foreach($extension_list as $id => $extension) {
    echo($id . ". " . $extension . "\n");
}
?>
</pre>
```

For the purpose of our discussion, the list of default modules in the PHP 4.x branch is practically identical. From a security point of view, only the *posix* module is of interest. According to the documentation (*http://www.php.net/manual/en/ref.posix.php*), it can be used to access sensitive information. I *have* seen PHP-based exploit scripts use POSIX calls for reconnaissance. To disable this module, use the `--disable-posix` switch when configuring PHP for compilation.

In your job as system administrator, you will likely receive requests from your users to add various PHP modules to the installation (a wealth of modules is one of PHP's strengths). You should evaluate the impact of a new PHP module every time you make a change to the configuration.

Configuration

Configuring PHP can be a time-consuming task since it offers a large number of configuration options. The distribution comes with a recommended configuration file *php.ini-recommended*, but I suggest that you just use this file as a starting point and create your own recommended configuration.

Disabling Undesirable Options

Working with PHP you will discover it is a powerful tool, often too powerful. It also has a history of loose default configuration options. Though the PHP core developers have paid more attention to security in recent years, PHP is still not as secure as it could be.

register_globals and allow_url_fopen

One PHP configuration option strikes fear into the hearts of system administrators everywhere, and it is called `register_globals`. This option is off by default as of PHP 4.2.0, but I am mentioning it here because:

- It is dangerous.
- You will sometimes be in a position to audit an existing Apache installation, so you will want to look for this option.
- Sooner or later, you will get a request from a user to turn it on. Do not do this.

I am sure it seemed like a great idea when people were not as aware of web security issues. This option, when enabled, automatically transforms request parameters directly into PHP global parameters. Suppose you had a URL with a `name` parameter:

```
http://www.apachesecurity.net/sayhello.php?name=Ivan
```

The PHP code to process the request could be this simple:

```
<? echo "Hello $name!"; ?>
```

With web programming being as easy as this, it is no wonder the popularity of PHP exploded. Unfortunately, this kind of functionality led to all sorts of unwanted side effects, which people discovered after writing tons of insecure code. Look at the following code fragment, placed on the top of an administration page:

```
<?
if (isset($admin) == false) {
    die "This page is for the administrator only!";
}
?>
```

In theory, the software would set the `$admin` variable to `true` when it authenticates the user and figures out the user has administration privileges. In practice, appending `?admin=1` to the URL would cause PHP to create the `$admin` variable where one is absent. And it gets worse.

Another PHP option, `allow_url_fopen`, allows programmers to treat URLs as files. (This option is still on by default.) People often use data from a request to determine the name of a file to read, as in the following example of an application that expects a parameter to specify the name of the file to execute:

```
http://www.example.com/view.php?what=index.php
```

The application then uses the value of the parameter `what` directly in a call to the `include( )` language construct:

```
<? include($what) ?>
```

As a result, an attacker can, by sending a path to any file on the system as parameter (for example `/etc/passwd`), read any file on the server. The `include( )` puts the contents of the file into the resulting web page. So, what does this have to do with `allow_url_fopen`?

Well, if this option is enabled and you supply a URL in the `what` parameter, PHP will read and *execute* arbitrary code from wherever on the Internet you tell it to!

Because of all this, we turn off these options in the *php.ini* file:

```
allow_url_fopen = Off
register_globals = Off
```

Dynamic module loading

I have mentioned that, like Apache, PHP uses modules to extend its functionality dynamically. Unlike Apache, PHP can load modules programmatically using the `dl()` function from a script. When a dynamic module is loaded, it integrates into PHP and runs with its full permissions. Someone could write a custom extension to get around the limitations we impose in the configuration. This type of attack has recently been described in a Phrack article: "Attacking Apache with builtin Modules in Multihomed Environments" by andi@void (*http://www.phrack.org/phrack/62/p62-0x0a_Attacking_Apache_Modules.txt*).

The attack described in the article uses a custom PHP extension to load malicious code into the Apache process and take over the web server. As you would expect, we want this functionality turned off. Modules can still be used but only when referenced from *php.ini*:

```
enable_dl = Off
```

Display of information about PHP

I mentioned in Chapter 2 that Apache allows modules to add their signatures to the signature of the web server, and told why that is undesirable. PHP will take advantage of this feature by default, making the PHP version appear in the `Server` response header. (This allows the PHP Group to publish the PHP usage statistics shown at *http://www.php.net/usage.php*.) Here is an example:

```
Server: Apache/1.3.31 (Unix) PHP/4.3.7
```

We turned this feature off on the Apache level, so you may think further action would be unnecessary. However, there is another way PHP makes its presence known: through special Easter egg URLs. The following URL will, on a site with PHP configured to make its presence known, show the PHP credits page:

```
http://www.example.com/index.php?=PHPB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000
```

There are three more special addresses, one for the PHP logo, the Zend logo, and the real Easter egg logo, respectively:

```
PHPE9568F34-D428-11d2-A769-00AA001ACF42
PHPE9568F35-D428-11d2-A769-00AA001ACF42
PHPE9568F36-D428-11d2-A769-00AA001ACF42
```

The Easter egg logo will be shown instead of the official PHP logo every year on April 1. Use the `expose_php` configuration directive to tell PHP to keep quiet. Setting this

directive to `Off` will prevent the version number from reaching the `Server` response header and special URLs from being processed:

```
expose_php = Off
```

Disabling Functions and Classes

The PHP configuration directives `disable_functions` and `disable_classes` allow arbitrary functions and classes to be disabled.

One good candidate function is `openlog( )`. This function, with `syslog( )`, allows PHP scripts to send messages to the syslog. Unfortunately, the function allows the script to change the name under which the process is visible to the syslog. Someone malicious could change this name on purpose and have the Apache messages appear in the syslog under a different name. The name of the logging process is often used for sorting syslog messages, so the name change could force the messages to be missed. Fortunately, the use of `openlog( )` is optional, and it can be disabled.

```
disable_functions = openlog
```

Some PHP/Apache integration functions (listed below and available only when PHP is used as an Apache module) can be dangerous. If none of your scripts require this functionality, consider disabling them using the `disable_functions` directive:

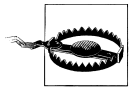
```
apache_child_terminate
apache_get_modules
apache_get_version
apache_getenv
apache_note
apache_setenv
virtual
```

Restricting Filesystem Access

The most useful security-related PHP directive is `open_basedir`. It tells PHP which files it can access. The value for the directive consists of a list of file prefixes, separated by a colon on Unix or a semicolon on Windows. The restrictions imposed by this directive apply to PHP scripts and (data) files. This option should be used even on servers with only one web site, and it should be configured to point one folder up from the web server root, which for the purposes of this book we set to */var/www/htdocs*. Given that web server root, here is how `open_basedir` should be set:

```
open_basedir = /var/www/
```

The setting above will allow the PHP engine to run the scripts that are under the web server root (*/var/www/htdocs*) and to access the data files that are stored in a private area (*/var/www/data*). If you do not need nonpublic files, allow PHP to access the web server tree only by restricting PHP to */var/www/htdocs* instead.



Know the difference between restrictions to a folder and restrictions to a prefix. For example, if were we to set the value of the directive to */var/www*, scripts would be able to access the files in */var/www* and */var/www2*. By having the slash at the end (as in the example above), the scripts are prevented from going outside */var/www*.

In Chapter 2, I described a method of restricting Apache into its own filesystem. That type of protection uses the operating system features and results in robust protection, so a process cannot access outside files even when it wants to. In contrast, the `open_basedir` restrictions in PHP are a form of self-discipline. The developers of PHP have attempted to add special checks wherever files are accessed in the source code. This is a difficult task, and ways to trick PHP are published online from time to time. Controlling third-party modules is nearly impossible. A good example is this Bugtraq message:

> "PHP4 cURL functions bypass open_basedir" (*http://www.securityfocus.com/archive/1/379657/2004-10-26/2004-11-01/0*)

In the message, the author describes how the cURL PHP extension can be used to bypass `open_basedir` restrictions.

Another directive, `doc_root`, sounds suspiciously like a synonym for `open_basedir`, but it isn't. This one only works when PHP is used as a CGI script and only to limit which scripts will be executed. (Details are available at *http://www.php.net/security.cgi-bin*.)

Setting Logging Options

Not all PHP errors are logged by default. Many useful messages are tagged with the level `E_NOTICE` and overlooked. Always set error logging to the maximum:

```
error_reporting = E_ALL
log_errors = On
```

To see any errors, you need to turn error logging on. This is done using the `error_log` configuration option. If this option is left unspecified, the errors go to the standard error output, typically the Apache error log. Otherwise, `error_log` accepts the following values:

`syslog`
: Errors are sent to the system's syslog.

`<filename>`
: By putting an actual filename as the parameter, you tell PHP to write all errors to the specified separate log file.

When using a separate file for PHP logging, you need to configure permissions securely. Unlike the Apache logs, which are opened at the beginning when Apache is still running as *root*, PHP logs are created and written to later, while the process is

running as the web server user. This means you cannot place the PHP error log into the same folder where other logs are. Instead, create a subfolder and give write access to the subfolder to the web server user (*httpd*):

```
# cd /var/www/logs
# mkdir php
# chown httpd php
```

In the *php.ini* file, configure the `error_log` option:

```
error_log = /var/www/logs/php/php_error_log
```

The option to display errors in the HTML page as they occur can be very useful during development but dangerous on a production server. It is recommended that you install your own error handler to handle messages and turn off this option. The same applies to PHP startup errors:

```
display_errors = Off
display_startup_errors = Off
```

Setting Limits

When PHP is compiled with a `--enable-memory-limit` (I recommend it), it becomes possible to put a limit on the amount of memory a script consumes. Consider using this option to prevent badly written scripts from using too much memory. The limit is set via the `memory_limit` option in the configuration file:

```
memory_limit = 8M
```

You can limit the size of each `POST` request. Other request methods can have a body, and this option applies to all of them. You will need to increase this value from the default value specified below if you plan to allow large file uploads:

```
post_max_size = 8M
```

The `max_input_time` option limits the time a PHP script can spend processing input. The default limit (60 seconds) is likely to be a problem if clients are on a slow link uploading files. Assuming a speed of 5 KBps, they can upload only 300 KB before being cut off, so consider increasing this limit:

```
max_input_time = 60
```

The `max_execution_time` option limits the time a PHP script spends running (excluding any external system calls). The default allowance of 30 seconds is too long, but you should not decrease it immediately. Instead, measure the performance of the application over its lifetime and decrease this value if it is safe to do so (e.g., all scripts finish way before 30 seconds expire):

```
max_execution_time = 30
```

Controlling File Uploads

File uploads can be turned on and off using the `file_uploads` directive. If you do not intend to use file uploads on the web site, turn the feature off. The code that supports file uploads can be complex and a place where frequent programming errors occur. PHP has suffered from vulnerability in the file upload code in the past; you can disable file uploading via the following:

```
file_uploads = Off
```

If you need the file upload functionality, you need to be aware of a parameter limiting the size of a file uploaded. More than one file can be uploaded to the server in one request. The name of the option may lead you to believe the limit applies to each separate file, but that is not the case. The option value applies to the sum of the sizes of all files uploaded in one go. Here is the default value:

```
upload_max_filesize = 2M
```

Remember to set the option `post_max_size` to a value that is slightly higher than your `upload_max_filesize` value.

As a file is uploaded through the web server before it is processed by a script, it is stored on a temporary location on disk. Unless you specify otherwise, the system default (normally */tmp* on Unix systems) will be used. Consider changing this location in the *php.ini* configuration file:

```
upload_tmp_dir = /var/www/tmp
```

Remember to create the folder:

```
# cd /var/www
# mkdir tmp
# chown httpd tmp
```

Increasing Session Security

HTTP is a stateless protocol. This means that the web server treats each user request on its own and does not take into account what happened before. The web server does not even remember what happened before. Stateless operation is inconvenient to web application programmers, who invented sessions to group requests together.

Sessions work by assigning a unique piece of information to the user when she arrives at the site for the first time. This piece of information is called a *session identifier* (*sessionid* for short) The mechanism used for this assignment is devised to have the user (more specifically, the user's browser) return the information back to the server on every subsequent request. The server uses the *sessionid* information to find its notes on the user and remember the past. Since a session identifier is all it takes for someone to be recognized as a previous user, it behaves like a temporary password. If you knew someone's session identifier, you could connect to the application she was using and assume the same privileges she has.

Session support in PHP enables an application to remember a user, keeping some information between requests. By default, the filesystem is used to store the information, usually in the */tmp* folder. If you take a look at the folder where PHP keeps its session information, you will see a list of files with names similar to this one:

```
sess_ed62a322c949ea7cf92c4d985a9e2629
```

Closer analysis will reveal that PHP uses session identifiers when it constructs file names for session data (the session identifier is the part after *sess_*). As a consequence, any system user who can list the contents of the */tmp* folder can learn all the active session identifiers and hijack sessions of any of the active users. To prevent this, you need to instruct PHP to store session data in a separate folder, which only the Apache user (*httpd*) can access. Create the folder first:

```
# cd /var/www
# mkdir sessions
# chown httpd sessions
```

Then configure PHP to store session data at the new location:

```
session.save_path = /var/www/sessions
```

This configuration change does not solve all problems though. System users will not be able to learn about session identifiers if the permissions for the folder */var/www/sessions* are configured to deny them access. Still, for any user that can write and execute a PHP script on the server, it will be trivial to write a program to retrieve the list of sessions because the script will run as the web server user.

Multiple applications, user groups, or web sites should never share the same session directory. If they do, they might be able to hijack each other's sessions. Create a separate session directory for each different purpose.

Casual session ID leaks and hijacking attempts can be prevented with the help of the `session.referer_check` option. When enabled, PHP will check the contents of the `Referer` request header for the string you provide. You should supply a part of the site domain name:

```
# comment
session.referer_check = apachesecurity.net
```

Since the `Referer` request header contains the URL of the user's previous page, it will contain the site's domain name for all legitimate requests. But if someone follows a link from somewhere else and arrives at your site with a valid session ID, PHP will reject it. You should not take this protection seriously. This option was designed to invalidate sessions that were compromised by users accidentally posting links that contained session IDs. However, it will also protect from simple cross-site request forgery (CSRF) attacks, where a malicious site creates requests to another site using

the existing user session. When the attacker completely controls the request, he also controls the contents of the Referer header, making this feature ineffective.

When this option is enabled, then even users whose browsers support cookies (and are thus using cookies for session management) will have their sessions invalidated if they follow a link from somewhere else back to your site. Therefore, since `session.referer_check` does not solve any problem in its entirety, I recommend that a proper session hijack defense be built into the software, as described in Chapter 10.

Setting Safe Mode Options

Safe mode (*http://www.php.net/manual/en/features.safe-mode.php*) is an attempt of PHP developers to enhance security of PHP deployments. Once this mode is enabled, the PHP engine imposes a series of restrictions, making script execution more secure. Many developers argue that it is not the job of PHP to fix security problems caused by the flawed architecture of server-side programming. (This subject is discussed in detail in Chapter 6.) However, since there is no indication this model will be changed any time soon, the only choice is to go ahead and do what can be done now.

Safe mode is implemented as a set of special checks in the PHP source code, and checks are not guaranteed to exist in all places. Occasionally, someone reports a hole in the safe mode and PHP developers fix it. Furthermore, there may be ways to exploit the functionality of PHP modules included in the installation to gain unrestricted access.

That being said, the PHP safe mode is a useful tool. We start by turning on the safe mode:

```
safe_mode = On
```

File access restrictions

The biggest impact of safe mode is on file access. When in safe mode, an additional check is performed before each filesystem operation. For the operation to proceed, PHP will insist that the *uid* of the file owner matches the *uid* of the user account owning the script. This is similar to how Unix permissions work.

You can expect problems in the following cases:

- If more than one user has write access for the web server tree. Sooner or later, a script owned by one user will want to access a file owned by another.
- If applications create files at runtime.

This second case is the reason programmers hate the safe mode. Most PHP applications are content management systems (no surprise there since PHP is probably the best solution for web site construction), and they all create files. (These issues are covered in Chapter 6.)

The easiest solution is to have the developer and Apache accounts in the same group, and relax *uid* checking, using *gid* checking instead:

```
safe_mode_gid = On
```

Since all PHP scripts include other scripts (libraries), special provisions can be made for this operation. If a directory is in the include path and specified in the `safe_mode_include_dir` directive, the *uid*/*gid* check will be bypassed.

Environment variable restrictions

Write access to environment variables (using the `putenv( )` function) is restricted in safe mode. The first of the following two directives, `safe_mode_allowed_env_vars`, contains a comma-delimited list of prefixes indicating which environment variables may be modified. The second directive, `safe_mode_protected_env_vars`, forbids certain variables (again, comma-delimited if more than one) from being altered.

```
# allow modification of variables beginning with PHP_
safe_mode_allowed_env_vars = PHP_
# no one is allowed to modify LD_LIBRARY_PATH
safe_mode_protected_env_vars = LD_LIBRARY_PATH
```

External process execution restrictions

Safe mode puts restrictions on external process execution. Only binaries in the safe directory can be executed from PHP scripts:

```
safe_mode_exec_dir = /var/www/bin
```

The following functions are affected:

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

Some methods of program execution do not work in safe mode:

`shell_exec( )`
: Disabled in safe mode

backtick operator
: Disabled in safe mode

Other safe mode restrictions

The behavior of many other less significant functions, parameters, and variables is subtly changed in safe mode. I mention the changes likely to affect many people in

the following list, but the full list of (constantly changing) safe mode restrictions can be accessed at *http://www.php.net/manual/en/features.safe-mode.functions.php*:

`dl()`
: Disabled in safe mode.

`set_time_limit()`
: Has no effect in safe mode. The other way to change the maximum execution time, through the use of the `max_execution_time` directive, also does not work in safe mode.

`header()`
: In safe mode, the *`uid`* of the script is appended to the `WWW-Authenticate` HTTP header.

`apache_request_headers()`
: In safe mode, headers beginning with `Authorization` are not returned.

`mail()`
: The fifth parameter (`additional_parameters`) is disabled. This parameter is normally submitted on the command line to the program that sends mail (e.g., *sendmail*).

`PHP_AUTH` *variables*
: The variables `PHP_AUTH_USER`, `PHP_AUTH_PW`, and `AUTH_TYPE` are unavailable in safe mode.

Advanced PHP Hardening

When every little bit of additional security counts, you can resort to modifying PHP. In this section, I present two approaches: one that uses PHP extension capabilities to change its behavior without changing the source code, and another that goes all the way and modifies the PHP source code to add an additional security layer.

PHP 5 SAPI Input Hooks

In PHP, SAPI stands for *Server Abstraction Application Programming Interface* and is a part of PHP that connects the engine with the environment it is running in. One SAPI is used when PHP is running as an Apache module, a second when running as a CGI script, and a third when running from the command line. Of interest to us are the

three input callback hooks that allow changes to be made to the way PHP handles script input data:

`input_filter`
: Called before each script parameter is added to the list of parameters. The hook is given an opportunity to modify the value of the parameter and to accept or refuse its addition to the list.

`treat_data`
: Called to parse and transform script parameters from their raw format into individual parameters with names and values.

`default_post_reader`
: Called to handle a POST request that does not have a handler associated with it.

The `input_filter` hook is the most useful of all three. A new implementation of this hook can be added through a custom PHP extension and registered with the engine using the `sapi_register_input_filter()` function. The PHP 5 distribution comes with an input filter example (the file *README.input_filter* also available at *http://cvs.php.net/co.php/php-src/README.input_filter*), which is designed to strip all HTML markup (using the `strip_tags()` function) from script parameters. You can use this file as a starting point for your own extension.

A similar solution can be implemented without resorting to writing native PHP extensions. Using the `auto_prepend_file` configuration option to prepend input sanitization code for every script that is executed will have similar results in most cases. However, only the direct, native-code approach works in the following situations:

- If you want to enforce a strong site-wide policy that cannot be avoided
- If the operations you want to perform are too slow to be implemented in PHP itself
- When the operations simply require direct access to the PHP engine

Hardened-PHP

Hardened-PHP (*http://www.hardened-php.net*) is a project that has a goal of remedying some of the shortcomings present in the mainstream PHP distribution. It's a young and promising project led by Stefan Esser. At the time of this writing the author was offering support for the latest releases in both PHP branches (4.x and 5.x). Here are some of the features this patch offers:

- An input filter hook ported to 4.x from PHP 5
- An extension (called *varfilter*) that takes advantage of the input filter hook and performs checks and enforces limits on script variables: maximum variable name length, maximum variable value length, maximum number of variables, and maximum number of dimensions in array variables

- Increased resistance to buffer overflow attacks
- Increased resistance to format string attacks
- Support for syslog (to report detected attacks)
- Prevention of code execution exploits by detecting and rejecting cases where attempts are made to include remote files (via `include()` or `require()`) or files that have just been uploaded
- Prevention of null byte attacks in include operations

Patches to the mainstream distributions can be difficult to justify. Unlike the real thing, which is tested by many users, patched versions may contain not widely known flaws. To be safe, you should at least read the patch code casually to see if you are confident in applying it to your system. Hopefully, some of the features provided by this patch will make it back into the main branch. The best feature of the patch is the additional protection against remote code execution. If you are in a situation where you cannot disable remote code inclusion (via `allow_url_fopen`), consider using this patch.