Hardening PHP - Some basic security measures.

Antonio Bonifati <http://ninuzzo.freehostia.com>

ABSTRACT

A summary of the most important PHP settings aimed at improving security of PHP web servers. Whenever you have to harden a PHP installation, this list will help you to assess what security measures can be applied without disrupting your application and to decide their scope of application. This is by no means exhaustive, but should serve as a good starting point to guarantee a minimum level of security and make the most simple kinds of attacks ineffective.

1. Fixing filesystem permissions of the Apache user

Aim: The Apache user should have write permissions only to the few directories he really needs to create new files in (typically directories used for file upload or application-level caching) and/or a few files that get overwritten by PHP code executed via HTTP (if any), not on the whole document root. This is a very basic security measure that should always be in place. Unix permissions matter!

Configuration-level solution

By default, make the document root and all files it contains belong to another user, not the Apache user. The Apache user should need to have only read-only access to the majority of files and directories served via web, except for the few ones that really needs to be written by web applications. Be selective and strict with Unix file permissions! Scout Apache log files for permission errors. They will also tell you what additional permissions your application needs if documentation about that is poor.

2. Chrooting PHP

Aim: You typically better execute all PHP processes blocked in the document root, so that no PHP script executed via the web will be able to access files outside that directory, e.g. the list of the system's accounts in /etc/passwd, which among other things, tells some of the valid usernames to login to the system.

Configuration-level solution

The open_basedir directive is a colon-separated list of file-system directories PHP access is restricted to. It can be set per-directory or per-virtualhost. All file operations will be limited to the defined directories and below. A single script can only tighten this value to subtrees below. E.g.:

```
open_basedir=/var/www
open_basedir=/web/vhosts:/var/www
```

3. File uploads

Aim: we want to prevent the execution of possibly uploaded PHP scripts in all URLs corresponding to paths the Apache user has write permission to.

Application-level solution

Do not trust $_FILES['...']['type']. E.g. simply let uploaded file types to be determined by its extension and only allow a few types depending on your users' needs (e.g. .doc, .rtf, .png). In any case, prevent uploading of files with a .php extension, or change their extension to another one, e.g. .txt, so that the source code of the uploaded PHP script is shown and it won't be executed.

Configuration-level solution

Disable file uploads altogether if not needed in php.ini:

```
file_uploads = off
```

or httpd.conf:

```
php_flag file_uploads off
```

If you cannot disable file uploads because your applications need this feature, then PHP scripts execution should be disabled in all those directories where the Apache user has write permission to. This can be done by putting in each directory a .htaccess file like this:

```
<Files *.php>
  Deny from all
</Files>
```

or, if your server also executes .php3 files:

```
<FilesMatch "*.(php|php3)$">
  Deny from all
</Files>
```

For added security you may want to forbid the use of .htaccess files globally, except perhaps a few directories where you want developers to make changes independently, and let the sysadmins centrally manage all PHP configuration overrides, in the Apache config files:

```
<Directory />
AllowOverride None
</Directory>
```

4. Disable unused PHP modules

Aim: Improve performance and security by disabling unused PHP modules.

Configuration-level solution

```
# php -m
```

lists all compiled-in PHP modules.

Outcomment the lines corresponding to modules not used by your applications. Remember to restart Apache afterwards. E.g. to disable the SQlite PHP module in Ubuntu just put a ; before the extension line in /etc/php5/apache2/conf.d/sqlite3.ini:

```
; configuration for php SQLite module
extension=sqlite3.so
```

A better approach is to rename the module configuration file:

```
# cd /etc/php5/apache2/conf.d/
# mv sqlite3.{ini,disable}
```

5. Disable fopen wrappers

Aim: Some PHP functions - e.g. include/require to include and evaluate files but also many others - also accept an HTTP or FTP URL and retrieve data from remote locations. If user input is passed as these functions' parameters without proper filtering, then code injection becomes very easy. This "fopen wrappers" facility is turned on by default in Ubuntu for many functions except include/require, but is rarely useful in legitimate code and most PHP programmers are often not even aware of it, do not know the list of all affected functions and thus they forget to implement filtering in order to make calls to these functions secure, whenever they are using variable parameters with them.

Configuration-level

As outlined before, it is really difficult to make sure all PHP code you run is secure with respect to this feature, but fortunately it is rarely used, so in most cases you can enforce security by disabling fopen wrappers completely in php.ini with no consequences:

```
allow_url_fopen = Off
allow_url_include = Off
```

6. Disable dangerous functions

Aim: Make server cracking more difficult by disabling certain PHP functions that provide an interface to the underlined OS (e.g. exec to execute an external program from PHP).

Configuration-level

If all applications on a php installation do no use a subset of these functions, then they can be disabled:

```
exec,passthru,shell_exec,system,proc_open,popen,curl_exec,
curl_multi_exec,parse_ini_file,show_source,
```

Just grep all your code for calls to these functions and list the largest subset of the above functions you can safely disable without affecting any legitimate application you are running. Then add this comma-delimited list to the value of the disable_functions directive in php.ini, which by default contains only some pcntl_* functions in Ubuntu. E.g., for the whole list above you will write:

```
disable_functions = pcntl_alarm,pcntl_fork,pcntl_waitpid,pcntl_wait,
pcntl_wifexited,pcntl_wifstopped,pcntl_wifsignaled,pcntl_wexitstatus,
pcntl_wtermsig,pcntl_wstopsig,pcntl_signal,pcntl_signal_dispatch,
pcntl_get_last_error,pcntl_strerror,pcntl_sigprocmask,
pcntl_sigwaitinfo,pcntl_sigtimedwait,pcntl_exec,pcntl_getpriority,
pcntl_setpriority,exec,passthru,shell_exec,system,proc_open,popen,
curl_exec,curl_multi_exec,parse_ini_file,show_source,
```

A Nagious plugin that alerts sysadmins whenever someone tries to use a forbidden function can be installed, but it won't be very effective because most PHP backdoors disable error logging or first check for a function to exist before trying to call it, in order to not raise suspicion. Unfortunately there is no way to prevent PHP scripts from disabling error logging, so at the script level errors can be silenced at will.

7. Protect all your servers from bruteforce SSH attacks

Aim: We want to limit the rate of ssh connections in order to block DoS attacks and password cracking attempts. Both threats can even come from one of your servers in the same network segment, e.g. if the latter has been initially compromized via PHP or other service and an SSH brute forcer has been installed there, trying to gain access to other local machines.

Configuration-level

The following IPTABLES rules will allow only X port 22 connections from any given IP address within an S second period, and require S seconds of no subsequent connection attempts before it will resume allowing connections again. The --rttl option also takes into account the TTL of the datagram when matching packets, so as to endeavour to mitigate against spoofed source addresses.

```
# iptables -A INPUT -p tcp --dport 22 -m state --state NEW -m recent --set --name SSH -j ACCEPT
# iptables -A INPUT -p tcp --dport 22 -m recent --update --seconds S --hitcount X+1 --rttl \
   --name SSH -j LOG --log-prefix "SSH_brute_force "
# iptables -A INPUT -p tcp --dport 22 -m recent --update --seconds S --hitcount X+1 --rttl \
   --name SSH -j DROP
```

E.g. set X=3 and S=60 to allow only 3 connections within one minute period.

Since attacks may come not only from the Internet, but also from your same machines - e.g. one compromised machine of yours - it is not enough to have a central firewall limiting the rate of SSH connections: in additional to a boundary network protection, each host should also run IPTABLES and use the previous rules.

Optionally, a white list can be made for trusted users (e.g. VPN users, but be careful to not white list any IP belonging to internal hosts that may be compromised and become source of attacks):

```
# iptables -N SSH_WHITELIST
# iptables -A INPUT -p tcp --dport 22 -m state --state NEW -m recent --set --name SSH
# iptables -A INPUT -p tcp --dport 22 -m state --state NEW -j SSH_WHITELIST
# iptables -A INPUT -p tcp --dport 22 -m state --state NEW -m recent --update --seconds S \
   --hitcount X --rttl --name SSH -j ULOG --ulog-prefix SSH_brute_force
# iptables -A INPUT -p tcp --dport 22 -m state --state NEW -m recent --update --seconds S \
   --hitcount X --rttl --name SSH -j DROP
```

8. Install a rootkit/backdoor/exploits detector on every server

Aim: To be able to scan for suspect programs and ideally be automatically alerted if there is a suspicious change in the system.

Configuration-level

Install http://rkhunter.sourceforge.net/ also available as the rkhunter Ubuntu package. Rkhunter compares SHA-1 hashes of important system files on your system against an online database to make sure the executables have not been tampered with.

The first check should preferably be run after a clean install:

```
# sudo rkhunter -c
```

Add --skip-keypress if you do not want output to be paged. Thereafter, you can look for more details, especially about failing tests, in the log file /var/log/rkhunter.log

When you update the system, you should run rkhunter with the --propupd option in order for it to accept the modifications made during the system updating:

1 - rkhunter --check
2 - system update/upgrade
3 - sudo rkhunter --propupd
4 - rkhunter --check

A Nagios plugin is also available: http://exchange.nagios.org/directory/Plugins/Operating-Systems/Linux/check_rootkit/details

Note: Many publicy available UNIX penetration rootkits, both in PHP and other languages, can be found

here: http://packetstormsecurity.com/UNIX/penetration/rootkits If you are a PHP programmer, you may want to study their source code, to see exactly what PHP features they abuse.