

Securing PHP: Step-by-Step

Artur Maj 2003-06-23

In my previous article ("[Securing Apache: Step-by-Step](#)") I described the method of securing the Apache web server against unauthorized access from the Internet. Thanks to the described method it was possible to achieve a high level of security, but only when static HTML pages were served. But how can one improve security when interaction with the user is necessary and the users' data must be saved into a local database?

This article shows the basic steps in securing PHP, one of the most popular scripting languages used to create dynamic web pages. In order to avoid repeating information covered in the previous article, only the main differences related to the process of securing Apache will be described.

Operating system

Like in the previous article, the target operating system is FreeBSD 4.7. However, the methods presented should also apply on most modern UNIX and UNIX-like systems. This article also assumes that a MySQL database is installed on the host, and is placed in the "/usr/local/mysql" directory.

Functionality

Generally, functionality will be very similar to the one described in the previous article. However, there are some changes:

- The web server must handle the PHP scripting language
- The PHP component must be able to read and write users' data in a locally installed MySQL database

Security assumptions

In case of security assumptions, the following have been added:

- The PHP configuration should take advantage of built-in security mechanisms
- PHP scripts must be executed in a chrooted environment
- The Apache server must reject all requests (GET and POST), which contain HTML tags

(possible Cross-Site-Scripting attack) or apostrophe/quotation marks (possible SQL Injection attack)

- No PHP warning or error messages should be shown to the web application's regular users
- It should be possible to store incoming GET and POST requests into a text file which will make it possible to use additional, host-based intruder detection system (HIDS), e.g. [swatch](#).

Preparing the software

First of all, we have to download the source code of the latest version of Apache, PHP and the [mod_security](#) module. The module will be used to implement the protection against CSS and SQL injection attacks. Next, the downloaded software must be unpacked and the content of the archive must be placed in the home directory. The source code of mod_security should be placed into the Apache's "src/modules/extra" directory:

```
gzip -dc apache_1.3.27.tar.gz | tar xvf -
gzip -dc php-4.3.2.tar.gz | tar xvf -
gzip -dc mod_security_1.5.tar.gz | tar xvf -
cp mod_security_1.5/apachel/mod_security.c apache_1.3.27/src/modules/extra/
```

If any available security patches to the above software exist, they should be downloaded and applied as well.

Before we start compiling the software, we must also decide, which of three methods of PHP installation we'll choose:

- as a web server's static module
- as a web server's dynamic module
- as a CGI interpreter

Each of above methods has its advantages and disadvantages. Compiling PHP as a static module will benefit from improved web server performance, but upgrading to a newer version of PHP later on will require recompilation of the whole web server. The second option, compiling as a dynamic module, hasn't got this disadvantage but the performance of the web server is decreased by approximately 5%. Additionally, one more module would be needed: mod_so. The

third method is to install PHP as a CGI interpreter - in conjunction with Apache's suEXEC mechanism this is quite an interesting solution. Unfortunately, when not properly installed it can pose a serious security threat.

From a security and performance point of view, the best choice seems to be compilation as a static module. That's why the rest of this article is based on that method of installation.

Installing PHP

Generally, the installation process of Apache with PHP is very similar to the process of installing Apache without PHP, as described in the previous article. The only difference is the use of two additional modules: `mod_php` and `mod_security`.

As in the previous article, we will start by creating an account and group called "apache". Then we must prepare the Apache web server as follows:

```
cd apache_1.3.27
./configure
```

and compile the PHP module:

```
cd ../php-4.3.2
./configure --with-mysql=/usr/local/mysql --with-apache=../apache_1.3.27 --
enable-safe-mode
make
su
make install
```

Next, we move to the directory with Apache source, and continue installation:

```
cd ../apache_1.3.27
./configure --prefix=/usr/local/apache --disable-module=all --server-uid=apache
--server-gid=apache --enable-module=access --enable-module=log_config --enable-
module=dir --enable-module=mime --enable-module=auth --activate-module=src/
modules/extra/mod_security --enable-module=security --activate-module=src/
modules/php4/libphp4.a
make
```

```
su
make install
chown -R root:sys /usr/local/apache
```

In the above `./configure` command, only those modules that are necessary to fulfill functionality and security assumptions are used. The choice of modules was discussed in details of the previous article. Now the next step is to return to the PHP directory and to copy the default PHP configuration file:

```
cd ../php-4.3.2
mkdir /usr/local/lib
chmod 755 /usr/local/lib
cp php.ini-recommended /usr/local/lib/php.ini
chown root:sys /usr/local/lib/php.ini
chmod 644 /usr/local/lib/php.ini
```

In order for the PHP scripts to be properly handled by the Apache web server, the following line should be added to `/usr/local/apache/conf/httpd.conf`:

```
AddType application/x-httpd-php .php
```

At this point we can try to run Apache and check if PHP can properly communicate with MySQL. We can achieve this by using the sample `"test.php"` script with the following content (the `"user_name"` and `"password"` values should be changed in accordance with installed database):

```
<html><body>
<?php
    $link = mysql_connect("localhost", "user_name", "password")
        or die;
    print "Everything works OK!";
    mysql_close($link);
?>
</body></html>
```

The above web page can be viewed by using any Internet browser. If PHP instructions are properly interpreted and a connection to MySQL is established, we can start securing the

software. If not - we should analyze the Apache and MySQL log files and eliminate the cause of the problems.

Chrooting the server

The first step of securing the server is to prepare a chrooted environment for the Apache server with PHP module.

At this point, we should perform all the steps described in the "Chrooting the server" section of the previous article. In addition, before running Apache in chrooted environment for first time, we must also copy the following libraries (they are needed in order for PHP to run properly):

```
cp /usr/local/mysql/lib/mysql/libmysqlclient.so.12 /chroot/httpd/usr/lib/  
cp /usr/lib/libm.so.2 /chroot/httpd/usr/lib/  
cp /usr/lib/libz.so.2 /chroot/httpd/usr/lib/
```

Additionally, we have to copy the PHP default configuration file:

```
umask 022  
mkdir -p /chroot/httpd/usr/local/lib  
cp /usr/local/lib/php.ini /chroot/httpd/usr/local/lib/
```

and create a /chroot/httpd/tmp directory. The directory owner must be root, and access rights should be set to 1777. After creating the new environment we should test, if Apache runs correctly:

```
chroot /chroot/httpd /usr/local/apache/bin/httpd
```

Before we begin to configure PHP we must also take care of one more very important detail - communication between PHP and MySQL. Because PHP communicates locally with MySQL by using the /tmp/mysql.sock socket, placing PHP in the chrooted environment means that they cannot communicate with each other. To solve that problem, each time we run MySQL we must create hard link to the Apache chrooted environment:

```
ln /tmp/mysql.sock /chroot/httpd/tmp/
```

Note that in order to make communication between PHP and MySQL possible, the "/tmp/mysql.

sock" socket and the "/chroot/httpd/tmp" directory must be physically placed on the same filesystem (hard links don't work between filesystems).

Configuring PHP

The process of securing the Apache server was described in the previous article, so as a starting point we'll use the configuration file that was presented there. For Apache to handle PHP scripts, we must add the following lines to httpd.conf:

```
AddModule mod_php4.c
AddType application/x-httpd-php .php
AddType application/x-httpd-php .inc
AddType application/x-httpd-php .class
```

It is worth to note that besides "*.php", two extensions have been added as PHP scripts: "*.inc" and "*.class". Programmers often include additional files, with an extension like "*.inc", "*.class" or similar. Because by default those extensions are treated as regular files, the requests to download them will reveal the source code comprised in them. This can lead to revealing passwords or other sensitive information.

A few changes must also be made in the PHP configuration file (/chroot/httpd/usr/local/lib/php.ini). The most important changes that should be made to improve PHP security are as follows:

Parameter	Description
safe_mode = On	By enabling safe_mode parameter, PHP scripts are able to access files only when their owner is the owner of the PHP scripts. This is one of the most important security mechanisms built into the PHP. Effectively counteracts unauthorized attempts to access system files (e.g. /etc/paswd) and adds many restrictions that make unauthorized access more difficult.

<code>safe_mode_gid = Off</code>	When <code>safe_mode</code> is turned on and <code>safe_mode_gid</code> is turned off, PHP scripts are able to access files not only when UIDs are the same, but also when the group of the owner of the PHP script is the same as the group of the owner of the file.
<code>open_basedir = directory[:...]</code>	When the <code>open_basedir</code> parameter is enabled, PHP will be able to access only those files, which are placed in the specified directories (and subdirectories).
<code>safe_mode_exec_dir = directory[:...]</code>	When <code>safe_mode</code> is turned on, <code>system()</code> , <code>exec()</code> and other functions that execute system programs will refuse to start those programs, if they are not placed in the specified directory.
<code>expose_php = Off</code>	Turning off the "expose_php" parameter causes that PHP will not disclose information about itself in HTTP headers that are being sent to clients in responses to web requests.
<code>register_globals = Off</code>	When the <code>register_globals</code> parameter is turned on, all the EGPCS (Environment, GET, POST, Cookie and Server) variables are automatically registered as global variables. Because it can pose a serious security threat, it is strongly recommended to turn this parameter off (starting from the version 4.2.0, this parameter is turned off by default)

display_errors = Off	If the display_errors parameter is turned off, PHP errors and warnings are not being displayed. Because such warnings often reveal precious information like path names, SQL queries etc., it is strongly recommended to turn this parameter off on production servers.
log_errors = On	When log_errors is turned on, all the warnings and errors are logged into the file that is specified by the error_log parameter. If this file is not accessible, information about warnings and errors are logged by the Apache server.
error_log = filename	This parameter specifies the name of the file, which will be used to store information about warnings and errors (attention: this file must be writeable by the user or group apache)

In addition, changing the file extension can be taken into account. For example *.php to *.asp, *.dhtml or even *.html. Such a change will make it difficult for any potential intruders to recognize the server-side technology that is being used. In order to change the extensions, all the *.php files should be renamed to *.dhtml (for example), and the following line should be changed in /chroot/httpd/usr/local/apache/conf/httpd.conf:

```
AddType application/x-httpd-php .php
```

to the new one:

```
AddType application/x-httpd-php .dhtml
```

Thanks to that, web users will not see *.php extension in the URL address which is what immediately suggests that the PHP technology is being used at the server side.

Defending against CSS and SQL Injection attacks

The last step of securing the server is implementing the logging of the GET and POST payloads, and implementing protection against Cross-Site-Scripting and SQL Injection attacks. In order to perform that, we will use the `mod_security` module, which we enable by adding the following line into `httpd.conf`:

```
AddModule mod_security.c
```

To enable logging of the GET and POST requests, it suffices to add the following section to `httpd.conf`:

```
<IfModule mod_security.c>
    AddHandler application/x-httpd-php .php

    SecAuditEngine On
    SecAuditLog logs/audit_log
    SecFilterScanPOST On
    SecFilterEngine On
</IfModule>
```

The above commands will enable the Audit Engine, which is responsible for logging requests, and the Filtering POST Engine, which will make it possible to log POST requests. In order to protect web application against CSS attacks, the following lines should also be inserted before "`</IfModule>`":

```
SecFilterDefaultAction "deny,log,status:500"
SecFilter "<(.\|\\n)+>"
```

The first line causes that the server to return the "Internal Server Error" message when the request contains the search phrase from any `SecFilter` variable. The second line sets up the filter to search for HTML tags in the GET and POST requests.

One of the typical signatures of SQL Injection attack is the appearance of an apostrophe (') or quotation mark (") in the GET or POST request. By rejecting all the requests containing those characters, we can make the use of SQL Injection technique very difficult:

```
SecFilter "'"
SecFilter "\""
```

Note, that although filtering the <, >, ', " characters lets us defend against CSS and SQL Injection attacks, it can lead to the improper functioning of the PHP application. It happens, because regular users cannot use those characters in the HTML forms. To solve that problem, the JavaScript language can be used on the client side, which should replace the prohibited characters with special tags, e.g. < > " etc.

Summary

Achieving a high level of a web server's security using server-side technologies (PHP, ASP, JSP etc.) is a very difficult task in practice. The very nature of interactions with a web server in any significant way decreases the web server's security. That is why server-side scripts should only be used where it is absolutely necessary.

The methods described in this article let us mitigate the risk of a successful break-in when new vulnerabilities in Apache, PHP or even the web application itself are found. Of course, the article doesn't exhaust the subject of securing the PHP technology - only the basic outlines were presented. And although applying them can increase the level of security, we cannot forget that the security of the whole environment depends not only on Apache's or PHP's configuration, but also and foremost - on the web application itself.

Relevant Links

[Securing Apache: Step-by-Step](#)

[Sample httpd.conf with PHP support](#)

[Apache HTTP Server Project](#)

[PHP](#)

[mod_security](#)

About the author

[Artur Maj](#) works as a Principal Software Engineer for Oracle Corporation, in the EMEA Mobile, Wireless & Voice Center of Expertise. He is experienced in designing computer systems,

performing security audits as well as providing security training. He is also author of many articles and publications devoted to securing computer systems and software against intruders.

View [all articles](#) by Artur Maj on SecurityFocus.

Comments or reprint requests can be sent to the [editor](#).

[Privacy Statement](#)

Copyright 2006, SecurityFocus