

Configure a Safe Environment for PHP Web Apps

Zend Core for i5/OS provides options to safeguard your system

by Alan Seiden

WITH ITS SUPPORT OF THE POPULAR web programming language PHP, the System i runs a large variety of PHP-based software for the web. Given the Internet's open nature, prudent system administrators who deploy web applications in IBM's Zend Core for i5/OS environment will want to ensure security. What precautions are needed?

Although System i's architecture automatically protects against buffer overruns, viruses, and worms, and Zend Core PHP provides additional safeguards beyond those that exist in generic PHP, you should add safeguards against other dangers, such as

- propagating viruses to browsers (even if the site itself is immune)
- password "sniffing" (stealing)
- unauthorized running of applications
- disclosure or alteration of private data

Some of these safeguards require specific PHP programming techniques, which I'll discuss in a future article. Here, I discuss how you can obtain broad protection within the PHP environment itself.

Note: Web security is a rapidly evolving field. All the recommendations I discuss here are believed to be accurate as of Zend Core for i5/OS version 2.0.1 (i.e., PHP 5.2.1).

The First Line of Defense

No matter what applications you run, a secure run-time environment reduces your risk of known and unknown problems. PHP security expert Chris Snyder, co-author with Michael Southwell of *Pro PHP Security* (Apress 2005), recommends multiple levels of protection — so-called "defense in depth" — because "you don't know what will go wrong." Snyder, a web developer at the Fund for the City of New York, considers a web application to be only as safe as the environment in which it runs.

The PHP execution environment is the outer perim-

eter of application security; you want to stop attacks here if at all possible. This article deals with that outer ring of defense, including elements such as PHP and application patch maintenance, encryption, directory structures, configuration files, and the regular updating of PHP.

Keep PHP and Apps Up to Date

Each new release of PHP improves security by eliminating vulnerabilities reported by the PHP community. Between releases, Zend issues "hot fixes" — temporary patches that correct serious bugs that could compromise security until the next release becomes available.

Zend's Shlomo Vanunu recommends that administrators keep current by configuring automatic Zend Network updates, which will apply patches and release updates as needed to maintain security. Vanunu, a senior consultant in Zend's lab in Ramat-Gan, Israel, and a team leader of Zend's i5 Global Services department, notes that IBM has arranged for all System i customers to get these updates through a free subscription to Zend Network Silver Support.

Here are the steps to configure automatic updates:

1. Register at Zend Network (zend.com/network). You may have done this already if you downloaded Zend Core from the web.
2. From a System i command line, type in the following:

```
GO ZENDCORE/ZCMENU
```

3. Choose menu option 2 — Update via Zend Network.
4. From the Update menu, choose to update immediately or on a schedule.

Besides PHP's own updates, popular PHP-based web applications issue their own patches and regularly release upgrades. According to Chris Snyder, even the best projects end up with some bugs that can be dangerous. So it behooves everyone to stay informed about vulnerabilities and corresponding updates using these methods:

- SecurityFocus summaries, led by Daniel Convissor (phpsec.org/projects/vulnerabilities/securityfocus.html)
- e-mail newsletters provided by development teams
- web-based forums, which are a source of news and a good way to get to know the developers

Also, I should include a word about PHP's configuration file, `php.ini`. PHP's global settings — including the security values shown in this article — are stored in `/usr/local/Zend/Core/etc/php.ini`. It goes without saying that you must control access to this file to maintain a secure system. This means limiting who has access to the file and ensuring that no web application has write-enabled access to it. This file can be edited with the `WRKLNK` command and Windows-based text editors.

The `php.ini` file format follows the popular `variable=value` syntax: `setting = value`. For example:

```
error_log = /usr/local/Zend/Core/logs/php_error_log
```

When the value is boolean (an on or off value), the line should read:

```
setting = On or setting = Off
```

For example:

```
log_errors = On
```

Note: You can use the numerals 1 and 0 interchangeably with On and Off. You can also access the settings of `php.ini` graphically through the web-based Zend Core control center with these steps:

1. Go to `http://hostname:8000/ZendCore`.
2. Click the Configuration menu.
3. Click the PHP Configuration option.

No matter how you edit `php.ini`, updates will not take effect until Zend's Apache web server is restarted. To restart Apache, follow these steps:

1. Type in `GO ZENDCORE/ZCMENU`.
2. Choose option 5: Service Management menu.
3. Choose option 6: Restart Apache server instances.

Beware of Uninitialized Variables

For efficiency, PHP does not check whether variables have been initialized before they are used. When programmers neglect to initialize variables, "unintended consequences" may occur. Normally, uninitialized variables don't present a security problem, even though they

can cause application failure. However, one scenario makes uninitialized variables a loaded gun: PHP's `register_globals` option.

This setting controls whether or not variables can be set directly from the querystring arguments included in the URL. The default value for `register_globals` is Off, which prevents URL assignment of variables. If it's set to On, however, any user can manipulate script variables directly by including assignments on the URL. For example, if your script uses the variable `authenticationPassed` to indicate that the user has been authenticated, an interloper could simply add the string "authenticationPassed=1" to a URL, neatly bypassing your authentication code.

The obvious safeguard against such abuse is to ensure that `register_globals` stays at its default Off value. For more examples of `register_globals` vulnerabilities, see php.net/register_globals.

Program Error Messages

Where would we be without program error messages? I'm not talking about messages to end users such as "zip code required," but errors that describe internal programming problems such as "array index out of bounds."

We do need to get these messages when developing our applications. Nevertheless, once a system is deployed to end users, displaying such messages not only looks amateurish, but it could reveal sensitive information such as file locations and internal API parameters. What's more, if search engines index your error messages, hackers could search for their favorite delicious exploitable errors. This technique is actually widespread enough to have a name: "Google hacking."

You can ensure that error messages accrue in your log file (where you can inspect them) but remain out of public view with the following `php.ini` settings:

```
display_errors = Off
log_errors = On
error_log = /usr/local/Zend/
  Core/logs/php_error_log (
  the default)
error_reporting =
  E_ALL | E_STRICT
```

When `E_ALL | E_STRICT` is specified, PHP will log all errors, no matter how small, including warnings and notices (such as of uninitialized variables). For more information about secure error reporting, see php.net/error_reporting.

Conceal `phpinfo()`

The function `phpinfo()` (manual page: php.net)

