

php *magazine*

Cutting-Edge Technologies for Web Professionals

The Truth about Sessions

NEW

Session Management Exposed

Doing Business the Open Source Way

Interview with MySQL AB and Zend

Bug Off

Eliminating Bugs from PHP Code

Writing PHP Extensions

Internals by Zeev Suraski

Clean Up Your Code

Refactoring Techniques

PHP at intelFLEET, LLC.

Data Acquisition



Table of Contents

Tools & Reviews

Locked! page 09

If you write PHP applications, for example a guestbook or an auction software and you distribute it you also know that your applications will be distributed by source. This article wants to analyze if and when it does make sense to encode your PHP applications and which products are therefore available.

Book Review page 16

Professional PHP Web Services

Business

Doing Business the Open Source Way page 17

Open Source is the way of the future, and now, even companies go for it. Meet the new entrepreneurs: MySQL AB and Zend Technologies.

Columns

Inside Wire page 21

Some useful and strange fixes for making URL tampering less inviting, how to get a little more strict on incoming data, overriding safe_mode with the CGI binary, running a PHP script, and more.

Start Up

Bug Off page 25

A tutorial on how to resolve and prevent bugs from impeding your PHP scripts.

Internals

Writing PHP Extensions page 31

One of the key factors of PHP's tremendous success was the very easy to use extensibility API. The simplicity of adding new functionality to the PHP engine, such as support for a new database or a new protocol, enabled a wide audience of developers to join in the project, and eventually resulted in one of the most powerful web platforms in use today. The purpose of this article is to explain the process of creating a new PHP extension, and to explain how to implement some of the features commonly used in extensions.

Cover Story



The Truth about Sessions page 39

Nearly every PHP application uses sessions. This article takes a detailed look at implementing a secure session management mechanism with PHP. Following a fundamental introduction to the Web's underlying architecture, the challenge of maintaining state, and the basic operation and intent of cookies, I will step you through some simple and effective methods that can be used to increase the security and reliability of your stateful PHP applications.

It is a common misconception that PHP provides a certain level of security with its native session management features. On the contrary, PHP simply provides a convenient mechanism. It is up to the developer to provide the complete solution, and as you will see, there is no one solution that is best for everyone.

Development

Clean Up Your Code page 46

This article describes a methodology to improve application design. It teaches us to build flexibility in our code when and where it is needed, and to avoid ending up with endless code clutter. The article also discusses when to refactor, and the things to keep in mind when applying this technique. Illustrated with real life examples in PHP, it explains a number of common refactorings. With these examples, the article proves that the methodology can be applied easily in a web development environment.

Enterprise

PHP at intelFLEET, LLC page 55

PHP is a well-known and commonly used server scripting language for the creation of dynamic web sites. Still many new users ask why PHP should be preferred over other technologies/languages and many also ask for references to companies who have used PHP with success. This is the story about how PHP was helpful in making a success of a small startup company located in Southern California with customers all over USA.

Departments

Editorial page 03

Advertising Index page 60

Imprint page 60

News & Trends page 04

Dear Readers,

Welcome to the first issue of the PHP Magazine. As with all 'first' editorials, we will reserve some space, without expounding too much, to discuss how we came to be.

The beginning of the year 2003 marked the release of the *International PHP Magazine* in print, establishing itself as the premier source of cutting-edge PHP Information. True to its name, the magazine gained international reputé with its stunning technical content, fostered and nurtured by the likes of Derrick Rethans and Jan Lehnardt, with extensive inputs from core members of the PHP team. From that point, it took us over a year to realize that we had to bring out an electronic version to satiate the ever-burgeoning amount of information-demand that we receive from avid PHP enthusiasts around the world.

You asked for it, and here we are!!

The *PHP Magazine* is your monthly dose of PHP, containing an assortment of carefully handpicked articles from the vast resource pool of the PHP Magazine editorial. This issue also features, a brand-new Cover Story based on PHP Security along with some articles centered around that theme. Most of the articles are written by authors who deal with PHP in their daily work, so feel free to administer yourself with doses in large quantities.

To start with, the *News & Trends* section chronicles the 'goings-on' in the PHP arena.

In the *Tools&Reviews* track, we do an under-the-hood analysis of PHP encoding solutions – with the PHP bytecode encoders of Zend and ionCube, and a review of a PHP book as well.

For those of you with a *Business* bent of mind, we profile MySQL AB and Zend Technologies – two companies whose success stories demonstrate that making money and working for Open Source projects at the same time is very much compatible. In this interview David Axmark and Doron Gerstel talk about the links both companies have with Open Source, PHP, and associated licensing issues.

The *Inside Wire* column documents the work of PHP programmers who come up with useful and strange ways to fix things that may or may not be broken. From the weird to the simple – the *Start Up* corner houses an article on debugging PHP scripts for newbie PHP users; it'll be interesting for more advanced readers as well. To move on to higher things, the *Internals* section focuses on extending PHP – this series will put you on your way to becoming a hardcore extension writer.

In this issue, we chose to run a cover story on Session Security, since there is a definitive void for information in this area. Our author agrees that our community has been harmed, by a lack of good security-related documentation. The cover story takes a detailed look at implementing a secure session management mechanism with PHP.

For those of you who are trying to cope with constant changes in code design, we get down to some hands-on *Development* with refactoring – a way to change your code design without changing the inherent functionality. As a parting shot, for the *Enterprising* lot, we record how PHP helped turn a small startup company in Southern California into a big-time player with customers all over USA – enjoy the case study on *intelleFLEET, LLC*.

We hope you enjoy reading all that we have lined up for you.

We look forward to hearing your questions, suggestions, and guidance, concerning the content and detail in the magazine. We would also like to hear about any other topics that you think are interesting and can be helpful to the PHP community at large. Feel free to write to us at ibritto@php-mag.net.

Before we sign off, it's the season of peace and joy – we wish you a Merry Christmas and a Peaceful & Prosperous New Year ahead. Let's raise a toast to our monthly dose of PHP.



Indu Britto



Zend/Win Enabler - Running PHP on Windows

Zend has announced the beta release of ZPS for windows - a solution for running PHP on Windows with increased performance and assured stability. Here are some highlights of ZPS from the Zend web site:

- The Enabler that marries PHP and Windows with no limits, is produced and supported by the designers of PHP themselves.
- Finally, a Windows - PHP Enabler that has stability and scalability built-in
- Provide your customers with multi-platform PHP applications, running Linux and/or Windows seamlessly
- Keep you boss and your customers happy - performance up to 3x better than ISAPI and up to 10x better than CGI, with none of IS-API's instability.
- No more wondering about unstable, experimental or mysterious IIS and Apache connectivity methods

<http://www.zend.com/store/products/zend-win-enabler.php#1>

Dumping PHP Data Structures to/from XML

PHP_XML_Dumper 0.50 released - PHP_XML_Dumper is a class designed to dump PHP data structures to and from XML, using a DTD compatible with the Perl module XML::Dumper. This is useful for transferring data structures on the fly from PHP to Perl and vice-versa.

<http://www.avitable.org/>

SAXY XML Parser

Alternative to Expat, written purely in PHP. SAXY is a Simple API for XML (SAX) XML parser for PHP 4. It is lightweight, fast, and modeled on the methods of the Expat parser for compatibility. SAXY is non-validating, and recognizes – but does not attempt to handle – document types, comments, notations, and processing instructions. One of the major advantages of using SAXY is it is not an extension and is not subject to restrictions by your hosting provider.

<http://www.engageinteractive.com/saxy/>

PHP Live! 2.5 Released

Using only PHP and MySQL, PHP Live! is a powerful web-based live chat support software for your web site. Functions include unlimited operators and departments, the ability to initiate chat, the ability to push URLs, a real-time visitor traffic monitor, a proactive survey, a chat icon for each department, and more.

<http://www.osicodes.com/demos/phplive/c.php?k=1.6.8>

Finding Bottlenecks in PHP Code

DBG 2.11.0 released - Php Debugger DBG is a comprehensive software tool that helps you to debug your PHP script. It may work with your production or development web server or locally without any other computers. DBG is equipped with the ability to backtrace errors. It shows local and global variables as well as parameters that have been passed to all nested function calls at any point of execution. Among other things, it allows you to execute scripts in a step-by-step manner, set breakpoints (including conditional ones), evaluate expressions, and watch variables. The profiler allows you to find bottlenecks in PHP code at the functions level as well as the modules level and even the source lines level. DBG 2.11.0 brings with it, the addition of the PCRE and getopt library to the source tree.

<http://dd.cron.ru/>

Zend Performance Suite 3.6.0 Released

Zend Performance Suite (ZPS) is the complete performance management solution for delivering PHP-based dynamic content cost-effectively. ZPS, based on Zend's state-of-the-art Dynamic content caching, Code acceleration and File compression technologies, is a single solution that will dramatically improve the number of customers your server will be able to handle. Some of the highlights of the Zend Performance Suite include:

- Unparalleled server performance increase - up to 25X increase in server throughput
- No code intervention necessary
- Flexible configuration of caching conditions
- Dramatic cost savings, with fast ROI payback
- See the results your self with the built-in testing capability
- Ease of use; Straightforward deployment;
- API functions for personalization

<http://www.zend.com/>

Managing Water Supply Networks

DC Maintenance Management System 1.0.0 released - DC Maintenance Management System is a Web-based application to record and analyze customer complaints and repairs in water supply networks. It uses PHP, mapserver, and PostGIS. DC Maintenance Management System 1.0.0 brings with it updated and extended documentation, improved installation process, and a new tool to update landmarks. Icons and more colorful user interface. A clearer work order form and Web-based backup and restore.

<http://dcmms.sourceforge.net/>

RC4 Encryption in PHP

RC4 is fairly fast, secure and symmetric encryption algorithm. Developed by Ron Rivest in 1987 was kept trade secret until 9th September 1994 when it was posted on a Cypherpunks mailing list. Generally the key it uses is limited to 40 bits for various legal reasons but 128bits is the more common forms these days. To prove its strength products like Oracle Secure SQL are examples. It's symmetric meaning it uses the same key and steps as to encrypt when decrypting.

<http://www.devhome.org/php/tutorials/rc4crypt.html>

Let PHP Manage your DVDs, VCDs, and Video Tapes

phpVideoPro 0.5.5 released - If you've got too many DVDs and video tapes to handle, then you need a better system! That's exactly why phpVideoPro was created. This program is all you need to get your huge collection under control. It puts your information at your fingertips. phpVideoPro manages your collection of DVDs, Video CDs, and video tapes. It stores all data in a database, and provides you with features for adding/changing entries, displaying lists, printing labels and lists, and more. An online help system is built-in to guide you when necessary. Support for multiple languages is provided (English, German, French, Polish, Bulgarian, Swedish, etc.), and supported databases include MySQL and PostgreSQL. The new release adds some bug fixes and updates to the Spanish and Russian language support files.

<http://www.izzysoft.de/>

JpGraph 1.14 Released

Major feature enhancements release. JpGraph is an OO Graph drawing library for PHP 4.0.2 and above. Highlights of the available features are: text, linear, and log scales for both the X and Y axes, anti-aliasing of lines, color-gradient fills, support for GIF, JPG, and PNG formats, support for two Y axes, spider plots (a.k.a Web plots), pie-charts, lineplots, filled line plots, impulse plots, bar plots, and error plots, support for multiple plot types in one graph, intelligent autoscaling, and extensive documentation (145 pages).

In JpGraph 1.14 more internal error checking was added to better handle abnormal data. Support for BIG5 Chinese fonts was added. Support for icons in backgrounds was added. Various minor bug fixes were made, as well as an important correction to Gantt charts to properly handle Daylight Savings Time.

<http://www.aditus.nu/jpgraph/>

Group-Office 1.94 Released

Group-Office is Web-based office suite written in PHP that is extensible with modules. It features user management (optionally syn-

chronized with system and Samba), module management, an email client, a file manager, a scheduler, project management, and Web site management. The new release is a minor feature enhancement release that doesn't need the „register_globals“ PHP setting to be enabled anymore. This allows Group-Office to work on any Linux setup and makes it more secure.

<http://www.group-office.com>

General Purpose PHP Component Framework

Anticipating the availability of PHP 5, RefleXiveCMS has adopted a purely object approach. RefleXiveCMS is a general purpose PHP component framework. An easy to understand architecture allows independently developed components to work together. It comes with lots of ready-to-use goodies, and code generators will get you started immediately. Given the explosion of freely available PHP classes, a component framework was needed to make lego-like reuse possible. This is the chief goal of RefleXiveCMS. RefleXiveCMS 0.2.6 includes work done on the „calendar“ plugin. Calendar and seminar (weekly calendar view) objects are now usable in many languages and look good. Other parts of the code have had cosmetic work done. PHP has been switched to E_ALL, and all encountered warnings are suppressed.

<http://www.virtualmice.net/reflexivcms/>

Organizing your Homework Assignments

PHP Student Center 0.1 released - Student Center is an effort of the students of Westbrook High School to make a student web portal. It contains homework assignments, news, and even a daily lunch display. It shares its authentication with a windows NT/AD domain so students need only remember one username and password.

<http://studentcenter.sourceforge.net/>

Meshing your Web Page Content Together

PHP-Mesh 0.5 (Major Feature enhancement release) -PHP-Mesh was developed to use the combination of PHP, with the extremely clean nature of Sitemesh. It is a basic framework for meshing together content of web pages with the style in which they appear on the user's screen. In short, it is a PHP mini-port of the SiteMesh system that is popular with Java Web developers. With PHP-Mesh 0.5, the last major feature from SiteMesh was added, specifically the ability to decorate pages within another decorator. This enables any page which works standalone to work as a portal in another page (actually, in the decorator), and thus you should no longer need to use standard includes anywhere on the site.

<http://xaoza.net/software/phpmesh/>

The PHP Benchmark Project

Sebastian Bergmann has been working on developing an interesting tool, PHP_Benchmark, which aims to provide a set of PHP scripts to track performance regressions between PHP versions.

http://www.sebastian-bergmann.de/PHP_Benchmark/

A PHP Servlet

phplet 0.0.3 released - PHplet is similar to Java Servlet as it implements the init(), service(), destroy() methods and runs through a container. The lifecycle of PHplet is the same of servlet. It can run PHP classes that extend the HttpPhplet interface with the same methods of javax.http.HttpServlet. The first releases of the Phplet Application Server are already available for download via the project page.

<http://sourceforge.net/projects/phplet/>

My PHP FAQ

phpMyFAQ 1.3.9-RC1 released - phpMyFAQ is a multilingual, completely database-driven FAQ system. It also offers a content management system, flexible multi-user support, a news system, user tracking, language modules, templates, extensive XML support, PDF support, a backup system, and an easy to use installation script.

<http://www.phpmyfaq.de/>

How About a Game of Chess?

OCC 1.0.4 released - Online Chess Club is a PHP chess game that allows you to play any number of games simultaneously against your friends online using only a web browser, provided you own some PHP-ready Web space. It recognizes checkmate, stalemate, and allows you to draw a game. Additionally, finished games can be either be archived or deleted. With this release, OCC works with PHP 4.3 and higher. Also, game data is now wrapped in a directory, which allows you to prevent any other scripts from sneaking. A server-wide user ranking is now available, and games may be deleted in the very first turn without affecting the statistics.

<http://lgames.sourceforge.net/>

New Module for the phpWebSite CMS

phpwRSSFeeds 0.1.0 released - phpwRSSFeeds is a module for the phpWebSite CMS (and higher) that provides the ability to display syndicated news feeds in RSS format. It uses the PEAR XML_RSS Parser. Its features include the ability to show a list of headlines in a block or the full summaries on any page, and support for all existing RSS schemas.

<http://sourceforge.net/projects/phpwrsfeeds>

A PHP WikiWikiWeb Clone

PhpWiki 1.3.5 Major BugFix release - PhpWiki is a WikiWikiWeb clone written in PHP. PhpWiki works right out of the box with zero configuration, and comes with a set of default pages. It's useful for collaborating on documentation on a project, having freeform discussions, and easy editing and searching. In the latest very stabilized release, there are many behind-the-scenes server side changes regarding content handling, caching, headers, etc. Flat file database support has returned. There are translation updates, a plugin to list available plugins, a PhotoAlbum plugin, a Comment plugin, a RedirectTo plugin, a RawHtml plugin, a WikiBlog page type, numerous layout fixes, numerous bugfixes, and minor improvements.

<http://www.phpwiki.org/>

Statistics Prove PHP's Increasing Dominance

InformationWeek has a note about PHP's increasing popularity, based on a NetCraft survey that says PHP is found on 52% of the 14.5 million Apache-based web sites that it inspected, compared with 19.4% using Perl. PHP is not widely known outside Web-development communities, but the number of PHP developers is probably 400,000 to 500,000, says Shane Caraveo (senior developer with ActiveState). „It's dominant on Linux, Sun's Solaris, and Unix. The exception is Windows sites using ASP,“ he says.

<http://www.informationweek.com/>

PEAR-compliant Template System for PHP

phpSavant 1.1 released - Savant is a powerful but lightweight PEAR-compliant template system for PHP. It is non-compiling, and uses PHP itself as its template language so you don't need to learn a new markup system. It has an object-oriented system of template plugins and output filters, so it sports almost all of the power of Smarty with almost none of the overhead. phpSavant 1.1 allows you to get back a specific token with getToken() instead of the whole array, and adds a new output filter to colorize text between „code“ tags.

<http://phpsavant.com/>

The DotPHP Framework

DotPHP 0.5 released - DotPHP is framework similar to ASP.NET. It contains FormForge, Web components, NuSOAP, and PHPBaseClasses. DotPHP is next step in Web Components project. DotPHP contain web components ver 3.00. Developers can make web site by using components alone, similar to making an application with DELPHI or C++ with some limitations. DotPHP doesn't warrant knowledge about HTML, CSS or JavaScripts, save the components. Download DotPHP 0.5.

<http://webcomp.sourceforge.net/>

Creating Modules for Documentation Elements

PHP Doc System 1.2 released - PHP Doc System allows developers to create modules for documentation elements (installation steps, buttons, screens, etc.) and then refer to them instead of having to copy/paste information they'd want to have in two or more places. It can run as dynamic PHP, including everything on the fly or it can output static HTML that can be included in a software distribution. PHP Doc System 1.2 adds Previous/Next links to each page using the TOC data. There is now an option to show the module summary on Table of Contents page. The code has been changed to use long PHP tags and other miscellaneous code cleanups.

<http://www.alexking.org/software/phpdocsystem/>

New Zend Studio Released

Zend.com has announced the release of Zend Studio 3.0.1a Client and 3.0.1 Server. The products have been released with Mac OS X support and bug fixes. The general changes in ZDE 3.0.1 include:

- Stopping a Search operation could take a very long time
- Presence of very large content on the clipboard could result in degraded performance
- Renaming a directory could sometimes result in an internal error
- Refresh problem in 'Project Inspector'

And, the changes in the appearance include:

- Under certain situations, the ZDE could launch with all of the toolbar icons disabled.
- Shortcut keys were not always visible under Windows.
- Docking and undocking Profiler windows didn't restore the same location and size.
- Focus was sometimes lost during Alt-Tab under Windows.
- Improved default keymaps under OS X
- Room for the line number in the status bar was sometimes too small under Linux.

Also, there are other changes in areas such as the debugger, profiler, and editor.

<http://www.zend.com/>

"Free" UserLinux For The Enterprise

Bruce Perens, co-founder of the Open Source Initiative and long time leader of the Debian Linux community has announced that he is planning to release a new Linux distribution to „challenge Red Hat's enterprise version“ of Linux. Naming the distribution UserLinux, Perens says that the distribution will be free for unlimited use and certified by large computer makers. UserLinux will be based on Debian and possibly available within six months. „The people who develop open-source code,“ Perens said, „are getting tired of being told that they have to pay to use it.“

<http://www.wired.com/news/infrastructure/0,1377,61166,00.html>

Net_LDAP 0.6.3 Released

Net_LDAP is a clone of Perl's Net::LDAP object interface to ldap servers. It does not contain all of Net::LDAP's features, but has:

- A simple OO-interface to connections, searches and entries.
- Support for tls and ldap v3.
- Simple modification, deletion and creation of ldap entries.
- Support for schema handling.

Net_LDAP layers itself on top of PHP's existing LDAP extensions.

<http://pear.php.net/>

Zend Studio Reviewed

phpbuilder has a neat article that offers a complete review of Zend Studio. It takes a close look at the Zend Studio, and compares it to the several freely available PHP IDEs.

The final summary of the review reads thus „If you like WYSIWYG IDEs such as Dreamweaver, then Zend Studio is not for you. Also, the system requirements of ZDE recommend at least 192MB of RAM (although most new computers come with that and more anyway). I found it a little memory-hungry and it sometimes took a little time to load up, so it's not ideal when you want to „quickly fix that one line. Apart from that, I like that it didn't „bloat“ my code like DW has a habit of doing and I loved the code completion, especially when using my own functions.“ „I have now stopped using Dreamweaver when coding in PHP. The functions that it provides may be all very well if you are relatively new to PHP, but it doesn't come close to the functionality of Zend Studio.“

<http://phpbuilder.com/columns/karsenbarg20031104.php3>

Turck MMCache for PHP 2.4.6 Released

Turck MMCache is a free PHP accelerator, optimizer, encoder, and dynamic content cache. It increases performance of PHP scripts by caching them in a compiled state, so that the overhead of compiling is almost completely eliminated. It also uses some optimizations for speeding up PHP scripts' execution. It typically reduces server load and increases the speed of PHP code by 1-10 times. It is tested with PHP 4.1.0-4.3.3, and Apache 1.3 and 2.0 under Linux and Windows. Some of the changes associated with the latest release of Turck MMCache includes the fixing of some PHP5 specific optimization bugs. Also compatibility with „pcntl“ extension was fixed. This release has been tested with php-4.3.4.

<http://turck-mmcache.sourceforge.net/>

PHP 4.3.4 Released

PHP 4.3.4 has been released, after a long QA process. This is a medium size maintenance release, with a fair number of bug fixes. All users are encouraged to upgrade to 4.3.4.

PHP 4.3.4 includes the following important fixes, additions and improvements in a list of over 60 various bug fixes:

- Fixed disk_total_space() and disk_free_space() under FreeBSD
- Fixed FastCGI being unable to bind to a specific IP
- Fixed several bugs in mail() implementation on win32
- Fixed crashes in a number of functions
- Fixed compile failure on MacOSX 10.3 Panther

http://www.php.net/release_4_3_4.php

phpQLAdmin 2.0.17 Released

phpQLAdmin is designed primarily for administration of a Qmail-LDAP user database, but also has EZMLM and QmailLDAP/Controls

management ability. Some of the changes associated with the latest release of phpQLAdmin include support for Opera in the folding branches. PHP parsing errors were fixed. The crypt function now really uses DES. The Bind9-LDAP manager was finished and enabled. Account expiration times can now be set. Basic Web server management was partially implemented. For the entire list of changes, please refer to the ChangeLog. Download phpQLAdmin 2.0.17.

<http://phpqladmin.bayour.com/>

MySQL 4.1.1 Released

A new version of the popular Open Source/Free Software database management system, MySQL, has been released. It is now available in source and binary form for a number of platforms. This is the second Alpha development release of the 4.1 tree, adding many new features and fixing recently discovered bugs.

<http://lists.mysql.com/announce/175>

Fresh news - every day:



Locked!

Why you should (or should not) encode your PHP sources

by Björn Schotte

If you know PHP you know that PHP is distributed by (C-)Source. If you write PHP applications, for example a guestbook or an auction software and you distribute it, you also know that your applications will be distributed by source. On the other hand, there is proprietary software, i.e. software that is only available as an executable binary and not with its source, for example the Microsoft Office Suite. In the last months, there was a big change from proprietary software to OpenSource software. Of course, not the whole software industry will follow this way. A big part of it will continue to distribute their software as a proprietary product. This article wants to analyze if and when it does make sense to encode your PHP applications and which products are therefore available.

The idea of encoding is very easy: you have to ensure that your source or parts of it will be compiled, optimized and encoded. The result of it will be distributed to the customer. The PHP installation of the customer that wants to run your encoded application has to decode the compiled bytecode and has to execute it without the ZendEngine. In order to do this, the PHP installation has to be extended with a ZendExtension that cares for decrypting and executing. After the installation the bytecode will go its own way: because the sourcecode should not be available to hijackers the extension has to use and execute the bytecode without the ZendEngine. With the optimizing process that was done before encoding both products that were tested could gain a bit performance compared to the non-encoded versions.

Of course we can argue about the use and sense of such encoding products for your PHP applications. At first view, it may be senseless because more and more customers and especially the government want to have the products as Open Source. In this case, distributing an encoded application would be counterproductive and could lead to losing the pitch. The customer's wish is obvious: he wants to save his investment and he wants to fix bugs himself or continue developing (if allowed in the license of the product) the application if you get into insolvency. So, you should really, really think about if it does make sense to encode your application or parts of it.

Another point for encoding your source could be the avoidance of support requests. You all know the typical situation that a customer buys your application and thinks he is Rasmus, Zeev and Andi himself in one person, grabs the source and puts his own code into your application. The result is that he has changed the core of the application so much that it does not run anymore and that he calls the support hotline every day. So, the foolishness of the customer could have been avoided by encoding the core of your application so that the customer could not change important parts of the code carelessly. The encoding of the source code conduces the safety of the customer himself. If you do not want to use an encoder for this typical situation you can avoid support requests by showing the customer the *md5* sums of your PHP files and thus proving him that he changed the application and that you are not responsible for the damage.

Protecting your intellectual property could be another classical reason. The vendor who thinks that his 3 mio. *loc* PHP application should be protected would probably encode the whole source and distribute the encoded application to his customers. This does make sense if the customer only wants to use the product but does not want to change the source of the application. Typical customer segments are the old economy, customers without their own PHP developer department and cus-

tomers without third party PHP software houses. The five-man joiner's workshop who bought the encoded CMS only would like to use the product. They do not want to change the source of the application.

This could lead into a bad situation if you have customers who do have their own PHP development department or who do have a third-party PHP software house: the customer may want to buy the product but he also wants to extend it (if the license allows it) with his own PHP department or his PHP software house. So, it could be that you loose the pitch because he wants to get the product as Open Source. This may also be a big concern in the very data sensitive areas like the health area. So, you could loose an important potential customer.

Or imagine you may want to distribute demo versions of your commercial PHP applications on the PHP magazine CD: it is important that your application will be encoded (and, for example, has an expire) and will not be distributed by source.

Often it is senseless to encode your PHP applications. You could use your license to prevent the customer from changing the sourcecode. If you catch him while changing the source, he will have a problem. After the big *dot.bombs* it is important for the customers to save their investments. The saving should not be the fact that only the vendor's consultants who cost USD \$10,000/day may change the application. It should be better to give the customer the opportunity to change the source himself (for example with his own PHP development department). Therefore, it is important to create a ring of trust between you and your customer (of course there are customers that are black sheeps) in order to decide if the customer gets the application encrypted or as Open Source.

I want to mention a really bad real-life example: a customer wanted to have an application that should use an already developed class library for generating form elements that was already used in-house in other projects. As the developer looked

	ionCube Encoder	Zend SafeGuard Suite
Company	IonCube Ltd.	Zend Inc.
Headquarters located at	London, UK	Israel
Website	www.ioncube.com/	www.zend.com/
Languages	English	German, English, French, Hebrew, Japanese, Russian
Supported OSes for the Encoder	Linux, FreeBSD	Linux glibc 2.1/2.2, Windows 98/NT4.0/2000/XP, HP/UX and AIX on demand (only command line)
Supported OSes for deployment	Linux, FreeBSD, Windows, OpenBSD + BSDi on request	Linux glibc 2.1/2.2, Windows 98/ME/NT4.0/2000/XP, Solaris 2.6 or later, FreeBSD 3.4 or later, MacOS, HP/UX and AIX, OpenBSD/NetBSD on demand
Supported OSes in the future	Solaris, perhaps MacOS, PowerPC/Alpha	for the encoder: Solaris, FreeBSD, MacOS X
Supported Webservers	Apache 1&2, IIS. (Others likely to work. Apache2 was reported to work by a customer during beta testing of the windows loader)	Apache 1.3.x, Apache 2.0.x (since 11/2002), IIS 4 or later, Zeus (via FastCGI), every CGI-Webserver
Supported PHP versions	4.06 (Unix only), 4.1x, 4.2x. 4.3x loader available	from PHP V 4.05
Features		
GUI for encoding	no	yes
Encoding via Shell	yes	yes
Support?	yes	yes
24/7 Support?	no, but 12/7 + enhanced support times	on demand
Phone Support?	yes	yes
E-Mail Support?	yes	yes
Other support levels with guaranteed reaction times	no	on demand
Prices		
Price for the encoder	USD \$349 for the encoder V1/2	Perpetual: USD \$2.400 1-Year-License: USD \$960
Price for encoder plus license manager	USD \$1000 for the license manager „Cerberus“ including encoder	Perpetual: USD \$7.300 1-Year-License: USD \$2.920
Upgrade costs	Free for small upgrades including upgrade to V2 of the encoder	For the 1st year all major and minor upgrades and bugfixes free; after that 20% of the product price fee for upgrades, support and enhancements.

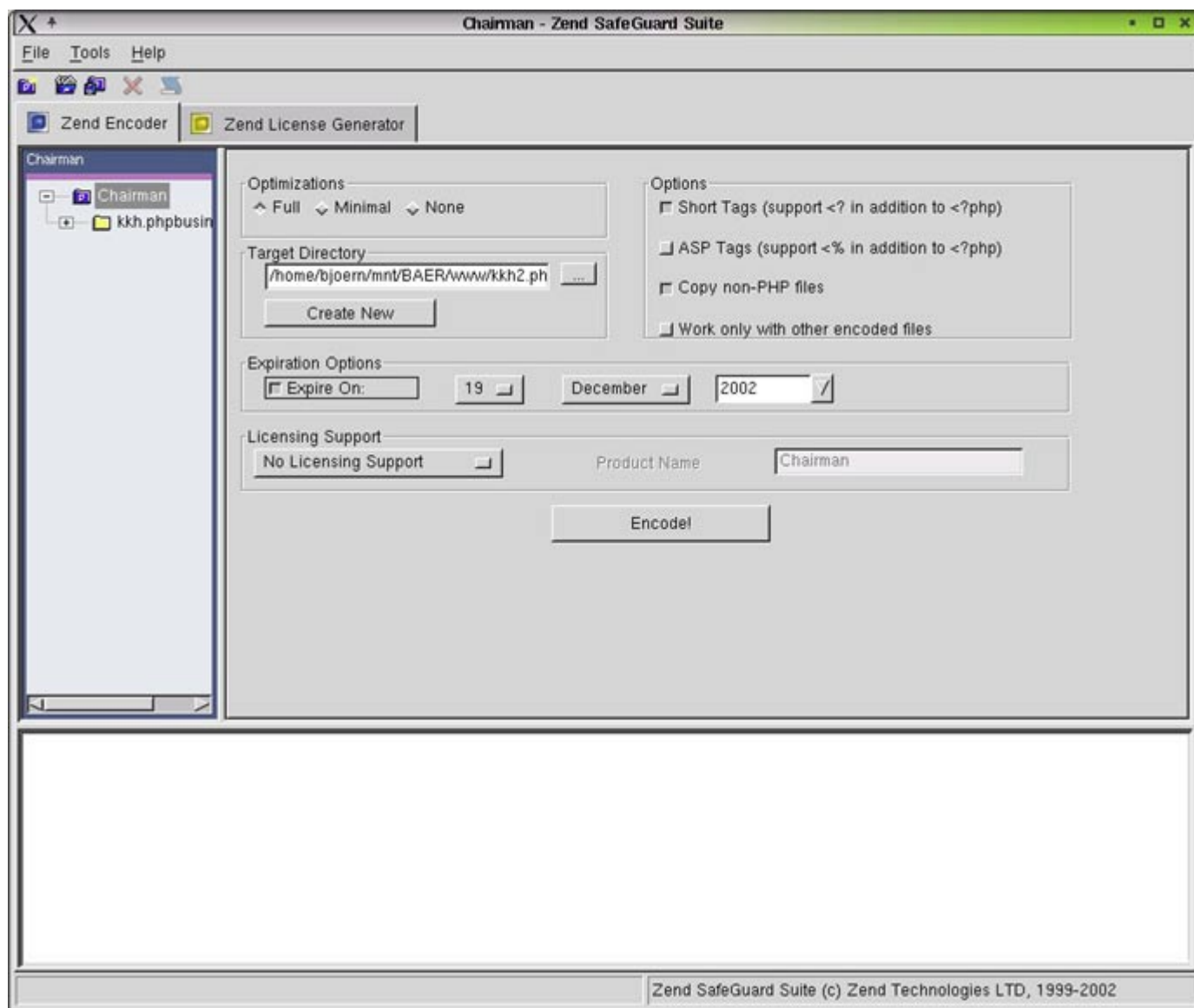


Fig. 1: The SafeGuard GUI at Linux

at the class library he found out that it was encoded and that only an API documentation was available. Unfortunately, the project required to create some more flexible form elements that the API was not able to create. The result was that the developer had to invest more time to circumvent the functionality of the class library in order to get the required result. So, the customer had to invest more money to launch the project.

This example shows that in many cases you should never encode class libraries that could be a part of an application. You will not do a favour to yourself nor to your customer.

So let us start. Now, you have some examples at hand to decide yourself if it does make sense to encode your application or at least parts of it. If you look at the market of encoding tools there are currently two products able to safely encode PHP

code: the ZendEncoder resp. the Zend SafeGuard Suite from Zend Technologies and the newcomer ionCube Encoder from ionCube Inc. A rough comparison will be shown in the textbox *Product overview*. For the sake of fairness in this comparison I will test the new version 2 of the ionCube encoder that has not been released yet at the time of writing this article against the Zend SafeGuard Suite. The SafeGuard Suite consists of the ZendEncoder plus a license manager. The new version of ionCube Encoder, codename *Cerberus*, should also include a license manager.

Zend SafeGuard Suite

Like all Zend products the SafeGuard Suite installs itself very comfortable with a dialog(1) based shell script. The Zend Safe-

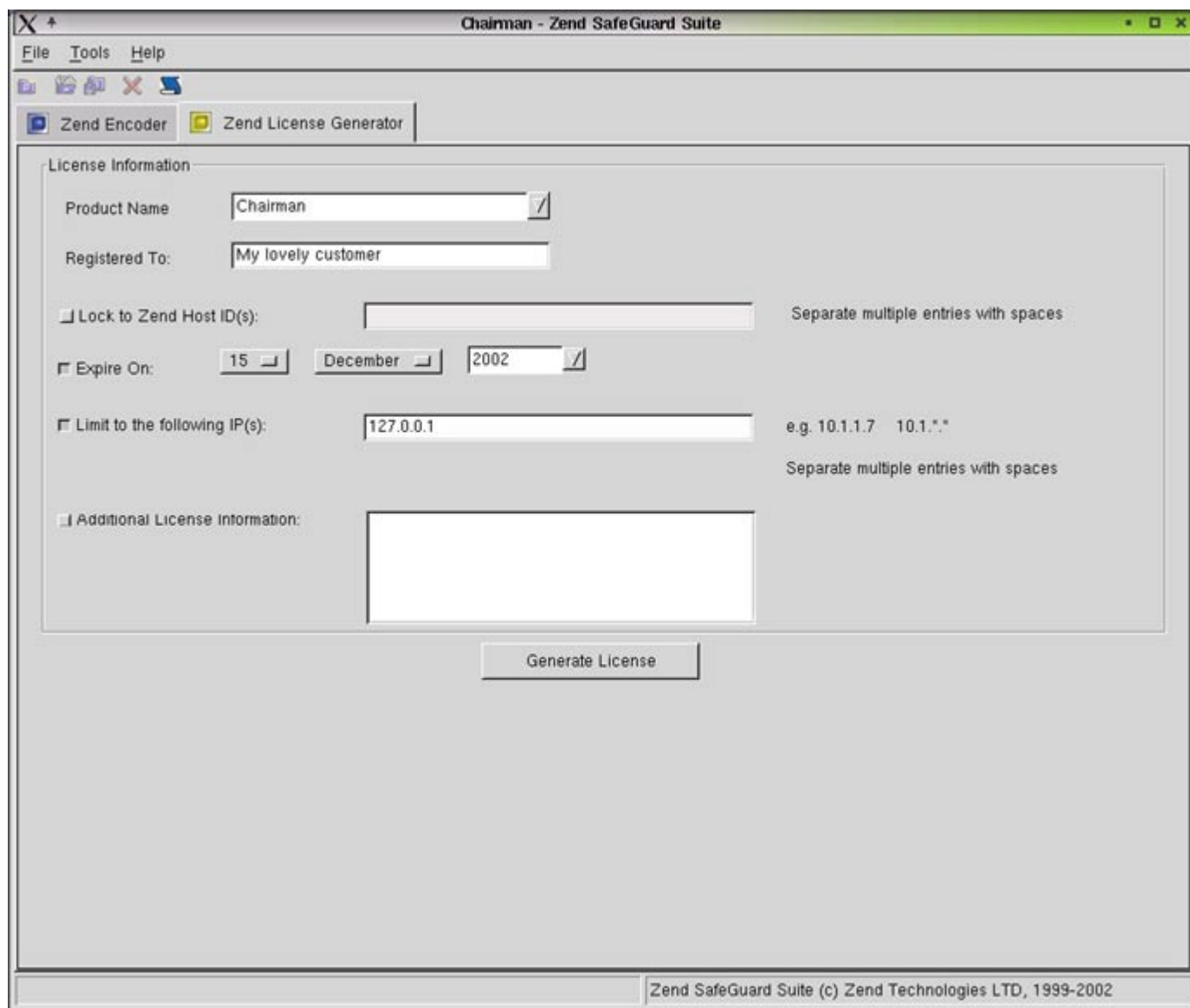


Fig. 2: License manager of the SafeGuard suite for creating license files

Guard Suite consists of the ZendEncoder and the license manager merged under a very handy GUI. Those of you who can abandon a license manager that can bind applications to specific MAC addresses or license files that can use the cheaper stand-alone ZendEncoder (also including GUI). The GUI of the SafeGuard Suite is available under Linux and Windows via GTK. For loading/executing the encoded scripts you need, similar to the ionCube encoder, the ZendOptimizer. The installation of the ZendOptimizer is also very unspectacular with a dialog(1)-based shell script which also restarts the web server.

The GUI is by default at `/usr/local/Zend/bin/ZendSafeGuard` and can be easily executed. After the execution a window with a tidy and thoughtful GTK GUI will appear (figure 1).

With the rudimentary project management functionality you are able to define projects and bind one or more files or whole directories to them. With the buttons that are shown in the figure you can say if you want to have ASP or short open tag support, if the encoder should copy non-php files to the target directory where all the encoded files will also go into. Furthermore, you can set an expire on your application, i.e. the user can run the application only until a specific date.

With the tab *Zend License Generator* you get to the license manager. Here, you can create license files (`.zl`), see figure 2. You can bind the license to a specific date, specific IPs or Zend HostIDs. Additionally, you can enter license information (in the format “element = value”) that can be extracted by a PHP call

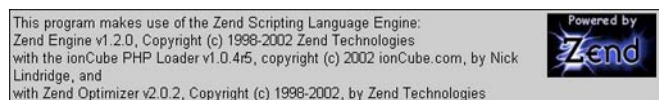


Fig. 4: ZendOptimizer together with the loader of the ionCube encoder

`zend_loader_file_licensed()`. It returns an array with all elements. You specify the location of your license files in the `php.ini` with the `zend_optimizer_license_path`. By using `zend_loader_file_licensed()` you can display additional licensing information of your product in your PHP script.

After a click on the *Encode!* button the SafeGuard suite encodes the appropriate scripts and shows at the bottom which script it is currently encoding. It also tests the scripts for parse errors (in combination with the ZendIDE it is possible to jump to the line of the code where i.e. the parse error has happened). If you do not have the ZendIDE like in this test you have to exhaustingly scroll through the list to look up the errors, especially when you are encoding a huge application with hundreds of thousands of scripts. Here, Zend really has to improve and to list the files with errors in an extra field.

If you have encoded your application by using a license file and no license file is present, PHP will throw an error when starting the script via Browser. The encoding of the PHP applications runs very fast in both the SafeGuard Suite and the ionCube encoder. Even huge projects with hundreds of thousands of PHP files should get encoded very fast without problems. The example application with about 45,000 lines of code was encoded in very few seconds and the encoded files were put into a separate directory. If you want you can use the encoder on the shell but the GUI is very comfortable so that normally you do not want to use the shell interface. But I discovered two grave errors with the command line interface of the ZendEncoder that are not visible on the first hand: the command line version does not preserve file permissions and it does not copy non-PHP files like shell-scripts, READMEs etc. into the target directory. So, the command line version of the ZendEncoder is very useless since you have to grab all the pieces of your application that was splitted while encoding – the READMEs, shell-scripts and non-PHP files residing in the source directory and only the encoded files in the target directory.

Because the encoded scripts will be optimized the way the ionCube encoder does, encoded scripts do have a slight performance gain.

The installation and use under Windows is the same. The installation under Windows comes with an InstallShield Installer that installs itself very comfortably. The installation mechanism of the ZendOptimizer tries to detect your PHP version. As you can see in figure 3, the GUI is nearly the same as under Linux:

Zend's competitor: the ionCube Encoder

The ionCube encoder is currently being actively developed in version 2 and will contain a license manager, code name *Cerberus*, too. The installation of the encoder is also very down-to-earth but does not include such a comfortable dialog(1) bases shell script as the Zend products have. In the package, you also find a user's guide, a quick reference and a quick start readme in ASCII format. For using the encoded scripts you also need a so-called loader which decodes and executes the encoded scripts. The loader can be downloaded for free from the homepage of ionCube and is available for Linux, FreeBSD and Windows. The encoder itself does not have a GUI. You can only use it via the command line. If you are used to command lines you find yourself very comfortable with it. A project manager or license manager who sits in another department and has to encode and license the product will have problems with the command-line since he is used to GUIs. IonCube is currently thinking about providing a GUI, at the latest when the encoder will be available under Windows. Furthermore, ionCube provides you with a commercial online encoding service where you can upload your scripts or script packages that get encoded. This can be seen as a cheap alternative to a stand-alone encoder but in real life you hardly want to upload your intellectual property to a website. So, it can be seen as a nice-to-have feature. Those of you who want to use encoders in a more serious way would like to buy a stand-alone encoder.

After downloading the loader you have to install it. It is sufficient to add one line into the `php.ini` and to restart the server.

```
zend_extension = /pfad/zum/ioncube_loader_1.0.4rc5.so
```

With Windows it is nearly the same: you also have to add this line into `php.ini` and restart your web server. Please be careful to install the appropriate `.dll` and not the `.so` file for Linux/FreeBSD. If you do not have access to `php.ini`, it is possible to use the loader as a PHP extension. Instructions for installing it this way can be found on the homepage of ionCube.

The vender says that the loader also works when using the ZendOptimizer. You have to make sure that you load the ionCube loader extension before loading the ZendOptimizer:

```
zend_extension = /pfad/zum/ioncube_loader_1.0.4rc5.so
zend_optimizer.optimization_level=15
zend_extension = /usr/local/Zend/lib/ZendOptimizer.so
```

If the loader will work with the ZendAccelerator, it is out of my control because at zend.com/store the product ZendAccelerator is not listed anymore and therefore it was not possible for me to test an evaluation version of the ZendAccelerator

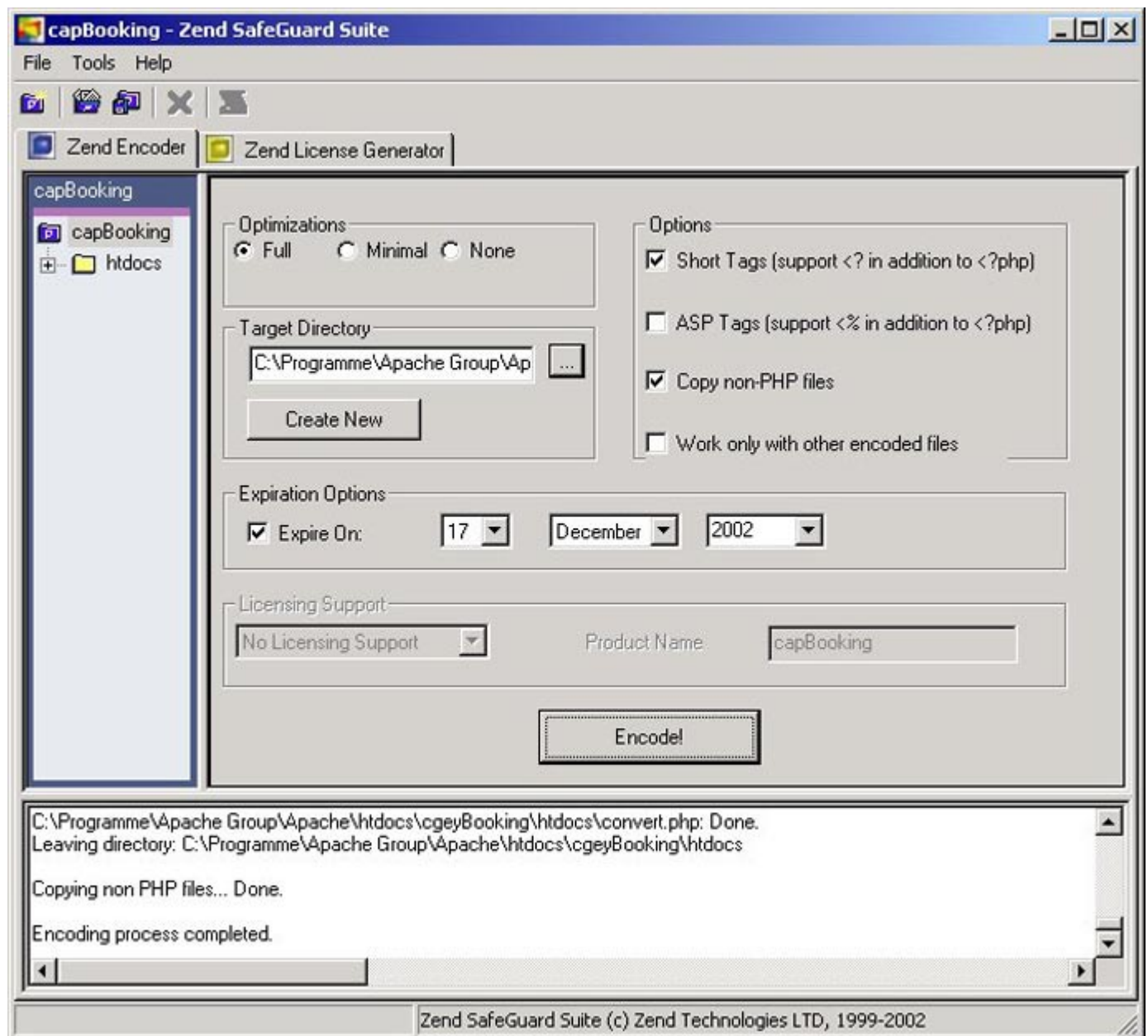


Fig. 3: SafeGuard Suite on Windows

with the ionCube loader. If the loader works properly you can encode your PHP applications. As already mentioned the encoder only exists as a command line version and carries a lot of command-line options with it (printed out with `a2ps` 2 pages on 1 DIN A4 page). With the options you decide which directory should be encoded, into which directory the encoded files should be saved to, if the encoded files should be analysed and compressed etc.

For example, I have encoded a `phpmyadmin/` directory, “call time pass by reference” enabled, compressing encoded files and verifying with `--verify` if every encoded file is a valid

PHP file that can be read by PHP systems who do not have the loader installed:

```
~/ioncube_encoder_evaluation_2.0.0_21 --key=YOURKEY phpmyadmin -o
phpmyadminenc --exclude=config.inc.php --allow-call-time-pass-reference
--compress --verify
```

If everything is done, you will get a directory `phpmyadminenc/` after a short time containing the encoded PHP files. All non-PHP-files were copied into this directory and the file `config.inc.php` (keyword `--exclude`) was not encoded to be able to

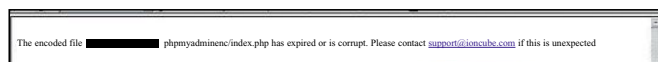


Fig. 5: File has expired or corrupted

configure database specific configurations in this file. Encoded files do have two lines of PHP code to test if the loader is installed and if not, trying to load it dynamically. You can change this behaviour by using the option *--without-loader-check*.

If you have installed the loader and try to load *phpmyadminenc/* via your browser you should get a normal phpMyAdmin Web-GUI. You can use it like you are used to and you do not have the feeling that this application was encoded.

Perhaps you want that your applications should not run after some specific date or after X days or that they will only run on specific IPs. The ionCube encoder knows the options *--expire-on*, *--expire-in*, *--allowed-ip-addr* and *--allowed-ip-mask*. The evaluation version I had for testing gives you the possibility to set *--expire-on* to the current day (i.e. 2002-11-15) so that the message the file is expired or corrupt appears when trying to start the application. Furthermore, you can set dates in the past with *--expire-on*. Both does not make sense to me but the vendor says that they will revisit the validation and warning routines in the future (figure 5).

It is similar and easy to encode your scripts for one or more specific IPs. With *--allowed-ip-addr=127.0.0.2* and another run on the local server you get the error that the script was not encoded for this server. A restriction to MAC addresses like the Zend SafeGuard Suite provides is only possible with the upcoming license generator. A combination of the options is also possible so that the application, for example, can only run on the IP 123.456.789.1 (*--allowed-ip-addr=123.456.789.1*) and expires at 2002/12/31 (*--expire-on=2002-12-31*).

The test field

Both products were tested with Linux and Windows. The Linux system is an old SuSE 6.2 with PHP 4.2.3 and the newest 1.3.x Apache. The GUI of the Zend SafeGuard Suite ran under a newer SuSE Linux 7.3 because it required a more recent glibc version. The PHP scripts were exported via Samba to the SuSE 7.3 box. The Linux system had a AMD K6-II with 300 MHz and 392 MB RAM, the 7.3 box was an AMD 1,2 GHz with 1 GB RAM.

The Windows machine is a mPentiumIII 700 MHz and 320 MB RAM with Windows 2000, also PHP 4.2.3 and the newest Apache 1.3.x.

The encoders have been tested with a relatively complex software, the ThinkPHP Chairman portal toolkit in a minimal version with about 45,000 lines of code (normal version >100,000 lines of code). Furthermore, the freely available software phpMyAdmin was encoded and tested.

It would be too much to list all the options of the ionCube encoder. The missing GUI was harmful at the test because you have to learn the options since there are so many on the command line. The use of the license manager was refused in the evaluation version but it should have the possibility to create a license with *--license-req* and to distribute it to the end user.

The vendor, at the moment a small company compared to Zend, says that he responds to support requests very fast and often within an hour.

Conclusion

A final recommendation cannot be given here because it depends on your requirements and the amount of your budget. Positive aspects of the Zend SafeGuard Suite include the GTK-GUI which makes a clear and comfortable impression and integrates seamlessly into the ZendIDE. Furthermore, the existing infrastructure of a company, i.e. the support, is another positive point. If you are a small company and you want to earn money with encoded applications you will wonder about the price of the SafeGuard Suite; perhaps you can make some deductions from the license manager and therefore only use the smaller version: the ZendEncoder. Bigger companies who set value on support and backing should buy the SafeGuard Suite although the price may seem a bit high.

For all others you should take a look at the ionCube encoder which brings you a very good power compared to the price. Negative aspects are the missing GUI so the ionCube encoder probably will not get used in bigger companies where the product manager is responsible for the creation and controlling of the licenses of the product. It may be possible that this will not be a negative point anymore when the ionCube encoder gets a GUI in the near future – then the encoder will have the same comfortability as the Zend SafeGuard Suite.

Furthermore, you do not have an encoder under windows and the support infrastructure which is currently evolving may be a negative point although the vendor stresses that the fewest support requests are dedicated to the product itself. For smaller companies or people who definitively like the command-line and do not need a GUI should definitively have a look at the ionCube encoder. Both products do have an easy installation part. The change in your company from Open Source to encoded scripts can be managed in minutes. At last, I warn you that you should really think about if you do have to encode your scripts. Also, incompatibilities after encoding did not stand out. ■

Björn Schotte is editor in chief of the German PHP Magazine and CEO of ThinkPHP, a company that works in the enterprise PHP market and deploys PHP and PHP support for big companies. You can reach him at bjoern@thinkphp.de.

Professional PHP Web Services

James Fuller, Harry Fuecks, et al.



I was looking forward to receiving this book for my first review, I mean, wow... free book! Imagine my disappointment when a surprisingly slim package falls through my letterbox. The book is a mere 480 pages long, that's very little for US\$50 compared to some of Wrox's other offerings. A large por-

tion of this book is available online in the form of 7 appendices.

It is worth mentioning that although Wrox's parent company has gone out of business seemingly Apress has bought this title. Wiley Publishing, now the owner of wrox.com has pledged to keep the online resources for all of the original Wrox titles online regardless of whether they bought the titles themselves or not, so all the online appendices and code examples are still available at wrox.com for the foreseeable future.

Eagerly I started reading my way through Chapter 1 of this book, having done so, I promptly did again. The start of this book has more acronyms than an entire season of Star Trek and unfortunately the definitions for these acronyms are either far too brief or in a few cases non-existent. I would have liked a simple table summary at the end just to re-cap these.

Unfortunately, my experience did not improve with Chapter 2. It soon becomes clear that perhaps instead of 7 online appendices, the book would have benefited from one or two extra printed chapters. Chapter 2 tries to cover XML Basics, all the XML Schema needed for the book and HTTP in just 48 pages. Whilst the XML basics was definitely enough to get started, XML Schema was skipped through in just 3 pages. I don't believe anyone could learn XML Schema in so brief an introduction, I certainly didn't. I think this book would have benefited by the addition of an entire chapter on XML Schema and expanding the HTTP section a bit more.

Despite the very bad start to this book, by the middle of Chapter 3, it becomes clear that indeed the authors do know what they are talking about, with the introduction to XML-

RPC being very thorough and concise you are quickly though not too soon thrown into the deep end developing an XML-RPC client for O'Reilly's Meerkat news service. I can safely say that by the time I done with this chapter I was convinced that Web Services are God's gifts to programmers... well maybe I wouldn't go that far. My only complaint at this point where is that the examples were not printed in the book, and whilst this isn't usually a problem, as with all Wrox books, there is no CD, you must download all examples and such from the website. Whilst this book is small enough to carry around for travel reading, be prepared to need your laptop with WiFi capabilities. There is only one chapter dedicated to XML-RPC, the reason for this being, as is said in the book, that the focus for the book is SOAP based Web Services.

The book really came into its own with its SOAP based chapters, with a basic introduction to SOAP quickly followed by a look deeper in to the technology. Whereas the first chapters suffered from being too brief, you really start to get a feeling that the authors actually know what they are talking about... finally. This book really does start from the basics, for someone with no knowledge of SOAP or namespaces, the first chapter on SOAP will bring you right up to speed. Be warned, if you are not a Star Trek fan, most examples in this chapter are Star Trek based.

The book just continues to excel with the remainder of its chapters, with chapters on WSDL, UDDI and application integration it covers most everything you will need. There is also a chapter devoted to security and another which covers the best practices when creating Web Services. All in all, this book whilst weak to start is a great read and I will certainly recommend that you buy the revision which I'm told will address all of the issues brought to light by this review.

Davey Shafik

James Fuller, Harry Fuecks, et al.

Professional PHP Web Services

478 pages, \$49.99

Apress LP, 2003

ISBN: 1-861008-07-4

Doing Business the Open Source Way

Open Source is the way of the future

by Damien Seguy

Running a business is complex enough. But it seems that running an Open Source business adds even more challenge: all the sources are made available. This means that your users may have a look at it, to correct any bug, or adapt it to their need, but so does your competition. Major software companies keep their source code jealously hidden, and are reluctant to disclose it to anyone. Even employees have to agree with a complex non-disclosure agreement before getting their hand on the real work. Does Open Source leave you unprotected? Meet the new entrepreneurs: MySQL AB and Zend Technologies.

Since Linux, Open Source softwares have demonstrated that they are viable solutions for both maintaining and developing software. With open source software, bugs are being tracked and eradicated by a large number of users. Contributions are gathered and benefit to everyone. And above all, the project itself cannot be sunk by the company bankruptcy: there is no commercial environment nor market to satisfy, which could eventually drive the project to its end.

Nowadays, we see a new kind of companies emerging: Open Source software companies. They are using a new strategy: develop software the way Linux does, backed by significant commercial force, to support the product and bring it to the whole market. MySQL AB and Zend Technologies are such companies, whose success demonstrates that making money and working for Open Source project at the same time is compatible.

MySQL AB and Zend Technologies

MySQL AB is a Swedish company, started by David Axmark, Allan Larsson and Michael "Monty" Widenius. MySQL AB develops and maintains the MySQL database server, the worlds most popular database. MySQL is dual licensed: users may choose between the GNU General Public License, with open source released directly on MySQL.com web site. On the other hand, they may purchase any commercial licenses offered by

MySQL AB, giving them the right to include MySQL in their own product, and sell it packaged. Since sources are free, MySQL AB sells support. Of course, free support is also offered from the forum and mailing list, but customer support ensures that problems are addressed faster, and requests for new functionalities are considered with higher priority. MySQL AB also collects royalties from their commercial licenses, from training sessions settled all around the world, and consulting to big companies. MySQL AB has 55 employees and is posting record sales level for the 3rd quarter of 2002. David Axmark is MySQL AB co-founder and now he works in relation with the community.

Zend Technologies is an Israeli company, started and named by Zeev Suraski and Andi Gutmans. Zeev and Andi rewrote the PHP core from scratch: the Zend Engine. This piece of software is the underlying layer of every PHP-driven web site since PHP 3. PHP and the included Zend Engine is freely downloadable from php.net and zend.com under the PHP license, which is a derivate of the BSD style Apache licence. Nowadays, Zend Technologies continues to develop the Zend Engine, and publishes it at no cost. Indeed, they even chose to change the Zend Engine licence to match the PHP license. Zend's business model is to develop and sell PHP tools that help developing, protecting and scaling PHP web sites, thanks to their excellent knowledge of the internal of the language. Zend Studio, the Zend Safe-

guard suite and the Zend Accelerator are solutions that lead on the PHP market. Zend technologies is headed by Doron Gershtel, CEO and co-founder.

Starting the business: from idea to reality

Open Source projects usually start as a technological project. The first aim of the original author is to solve a need he encountered. Releasing it as Open Source is usually no more than an obvious step. Then, the project takes larger proportion, as early enthusiasts adopt it. Eventually, founders have to answer the question: “is it worth building a company?”. This is especially the case with Doron. He was approached by Zeev and Andi, who offered him a job as CEO.

Doron Gershtel: “It was summer of 1999 when Zeev Suraski asked my former boss, Dr. Shimon Eckhouse, to review their business plan. Dr. Eckhouse introduced me to Zeev and Andi Gutmans and it didn’t take long to realize that they were bright, intelligent, ambitious young guys, but more importantly, that they had a great vision and held the key to the scripting language that could “ignite the revelation”. There were (and still are) a few driving forces that push PHP in the enterprise world. I have no doubt that an Open Source language, as good as it is, (and in fact it is very good) requires commercial backing. The business concept of Zend is to help companies that use PHP to be more efficient, more profitable and more competitive.”

David Axmark relates the same start for MySQL. In fact, MySQL AB was not formed before he and Monty could find a CEO. David Axmark: “[The opportunity for business] was obvious. Especially when you compared [this opportunity] with selling our software (well mostly services in practice) to a few local customers. And the commercial forerunner was Aladdin Ghostscript, which also had a dual licensing scheme. So, the idea was to get a product spread by distributing it freely. And then to make some money from people who wanted to put it inside a product. And we are still using that idea with a few modifications.

[...] Well a “normal” company did not appear until we got a professional CEO a year ago. And to fill that position we approached an old friend who has been the CEO of a few technical growth companies before. After a bit of thinking he said yes. At the same time we got some investment money to scale the company up to a more normal mix of technical versus non technical people.”

Fuelling the growth

Indeed, one of the greatest strength of Open Source is to spread products with a lot of ease. Costless to acquire for testing, those products are also well accepted by the technical community.

Open Source ensures users that they may tweak the software to their need. This gives a great power to end-users. This results in great brand image to the product, at no cost.

It may even generate spontaneous contributions to the projects. Open Sources are known to be built from contributions, may it be patches for corrections or brand new functionalities. And this trend does not disappear when the company shows up. David says:

“I have discussed this with some people who liked to make contributions. And for them it was a very good deal that we provided lots of work as GPL software that they used. So they had no problems signing over copyright on their contributions to us. I think this would be much harder if we had a business model like “90% is GPL but we also have these proprietary add-ons that you have to pay for”. We hope to get more contributions to the server when we have better internal documentation and some internal API that makes it possible to do more modular features. It is more fun to write something that becomes basically workable in a weekend or so. But on the other hand we have had very very few contributions to the server code since it is very hard to get into it. The client side on the other hand is based almost only on contributions. Like the PHP, Perl, Ruby ... interfaces.”

However, the acceptance of the software, and the contributions from the community are not sufficient to fuel a company, as Doron reminds:

“Absolutely, [we get contributions] both in core architecture and developer support. The Zend Engine 2 activity generates tremendous interest. And zend.com is more popular than ever, with 150,000 users, thousands of postings in code galleries and forums, and tutorials from PHP experts such as Jason Gilmore or Thomas Oertli.

[...] When building a business around a technological breakthrough, the first issue to consider is the real market needs – is there someone out there who will benefit from what we have, and be willing to pay for that value?

We have had the wonderful good fortune to find an existing PHP community willing to talk to us, share their perspectives with us, explain their needs to us and in all possible ways, help identify the true market needs. More than that, they enjoy the opportunity to do so, because they see it as an opportunity to strengthen their own future. Because of this, our customer base – both paying customers of our commercial products and non-paying customers of the Zend Engine and of the zend.com Developer Zone resource site – remain fiercely loyal to Zend.”

Customers are the ones that will pay the experts to push the limits of the product much further. Among a large user base, they may not be the most numerous, but they will always show up. MySQL AB has now over 3 million users, and PHP ac-

counts for over 25% of market share in web scripting languages, and over 1.2 million servers. Zend Technologies claims 3000 customers worldwide, less than a percent of the total user base. But those users are the most demanding ones, and the ones that will pay for such demands, as Doron states:

“Any commercial enterprise, whether they use Open Source or closed source technologies, has to focus on the bottom line, economically. PHP users are constantly seeking to make their operations as efficient, effective and profitable as possible. In some cases, this can be done by utilizing Open Source PHP add-ons. But for the business solutions on which Zend focuses – development, protection and performance management – customers achieve greater results by investing in technology that is based on strong and continuous innovation, thorough multi-platform QA and support, and fast time-to-market.

That is our philosophy, and the results back it up: we see no distinction between ISV's who use PHP and others who use closed source-scripting languages as far as their willingness to invest in commercial products. Everyone wants the tools that will help them win in this hyper-competitive world.”

Taking care of community and customers

So, Open Source businesses have to deal with two different groups: the community, who gets most of the work done for free, and the customers, who need solutions for their money. Indeed, those two groups have to be clearly identified, and treated differently. Open Source and commercial spirit may mix without problem.

“For our business model I saw no conflict. I still do not. You just need to understand the value of the model.” says David.

In fact, one may even mix closed and Open Sources, as long as they are clearly oriented:

“Zend is involved in both Open Source and closed source endeavours. The Zend Engine, which is one of the core technologies of the company, is indeed Open Source. However, the development tools and performance management applications Zend makes are not. There's a very clear distinction between the PHP infrastructure work that we do, and the commercial applications that we create”.

One of the major concerns about Open Source is the availability of the code. As soon as the sources are released, competitors will be able not only to grab the concept, but also to exploit the technology that was used to build it. David explains “Well, they would have to rewrite it. So, basically they just need the idea/specification. And we can take the same input from them. So, we see this as no problem at all.”

In fact, the best protection of the code is its own complexity. Understanding a SQL server or the internal of Zend is not an

easy task. It may require too much of reverse engineering to prove its viability. So, when it comes to add extra protection, here is MySQL's solution:

“Basically nothing. We keep source internal until it has consistent MySQL alpha quality. Like the 4.1 version that will hopefully come out in weeks. That has been developed parallel to the 4.0 version since last year. With more and more developers shifting new development to the 4.1 tree as 4.0 gets more stable. But we do open our BK tree (Note: BitKeeper tree) as soon as it has a public alpha state, so from there you can see all codes as soon as it is pushed into the tree.

The things we do protect is the copyright of the code (so we can do dual licensing) and the trademark. I would say that the trademark is the thing that will cause some problems with the community since we need to step up its protection. And that can create problems when someone's headline on a web page must be removed because it contains MySQL in the wrong context. Or when someone uses our logo in a non-agreed way.”

The Zend Engine is really similar: Zeev and Andi added themselves the hooks for internal add-ons. Zend Technologies used those hooks to create successful products like Zend Encoder or Zend Accelerator. Indeed, it also created the opportunity for programmers to build their own Zend Engine add on, some even competing with Zend products. Yet, Doron sees no problems there: “In the software world, especially when it comes to Open Source, barriers to entry for commercial players as well as other developing freeware is common. The fact that Zend Engine is Open Source gives us much more benefit (tremendous brand recognition) than damage when others use this technology and develop add-on's based on the Zend Engine.

In most cases these imitations increase the awareness in the market, and eventually most customers are interested in buying from the “source”, for reliability reasons. It is no coincidence that Zend's products are always the ones that serve as a benchmark for comparison.”.

New schemes

One new concept that companies introduce to Open Source projects is deadlines. Often, Open Source software doesn't feel compelled by market needs. They tend to keep technical excellence as their first priority. Apache 2.0 has been en route for over two years, and it has not reached enough maturity to go beyond alpha phase. While this behaviour makes sense when dealing with such a large market share, this is not a way to run a business. Technical excellence and critical development have to be balanced, as Doron explains:

“I believe that they complement each other, both on a technology perspective as well as a methodology perspective. Open Source, in its modular development methodology, is great at

covering a broad cross-section of platform support and functionality, over time. Take, for example, GTK, or PEAR, or even applications such as postNuke. However, when there is a need for complex integration of sub-systems in a single bullet-proof application, such as Zend Studio or Zend Performance Suite, modular development doesn't work as well. When development tasks are on a voluntary basis, it isn't possible to conform to fixed timetables or work with Gantt charts, a critical need for software project integration.

In the end, they feed on each other. Commercial applications strengthen the base of Open Source participants, and Open Source growth strengthens the need and the opportunity for commercial ventures."

David adds: "[...] We are trying to find a good compromise between a normal commercial and an Open Source development model. And we have the added benefit that we have a fixed team who works on MySQL every day."

Looking to the future

When it comes to looking ahead to the future, Open Source businesses face the same challenge any software publisher faces. If technical lead is confirmed every day, one of the next battlegrounds will be legal aspects. David has experienced those threats:

"A few. One is Software Patents that can be used against any free software or proprietary company. The problem with those is that you cannot protect yourself. And that it does not matter if you invented something internally 20 years ago if someone else got a patent. And avoiding mines. Like a certain partner that we still have a court case with (Editor's note: That case has been resolved in the meantime). That totally changed my view on trusting larger business partners. It takes a bit of changing to start thinking about that people may actually be lying you right in your face."

The other major battle for both PHP and MySQL is the adoption by corporations. This is a common objective for the Open Source world, now that its viability as products has been settled. And it is important for companies to remain focused on the most important thing:

"My main focus is on growing the PHP market. By this, I mean ensuring PHP's growth and adoption by corporate enterprises. Zend invests 20% of its R&D budget into Open Source development, not to mention other community building efforts such as zend.com. In addition to this, we work hard at Zend to consolidate our leadership position in the PHP marketplace. With more than 3000 customers, I believe that we are in a position to do so."

Zend Technologies and MySQL AB started when they could find a CEO. Staffing such a company also means introducing new profiles, where only engineers and experts once reigned.

This means a large shift in the direction of the group, and the recruitment. But it also brings nice surprises: "Well it is hard to get people experienced in writing code for the MySQL server. But we do get many applicants to both technical jobs and business jobs. From the beginning we only got developer applications so things have changed a bit. And we do have a strength here since we are a totally global company so we have people in about 14 countries.

[...] We are very well integrated and will work hard to stay that way. So, our developers are helping the sales people daily. And we do not have a marketing department, yet. Except yours truly (Note: David himself) and Zak for the "technical" marketing. And we are not the normal marketing persons. But since Monty and I are technical we are still and plan to stay very technically driven. We still do not publish release dates like the media wants..."

At Zend's, the shift toward commercial action is much clearer. It is a way to show one's objective and determination:

"Zend is quite unique in that we are achieving growth exactly at a time when much of the industry is faltering. Zend's core team was working well together even before the market turned, and the current situation has allowed us to augment this to create a tremendous team.

Finally, started as technological project, Open Source business often meet shifts in direction. Yet, technology is still at the core of the business, and if the company has to be customer driven, it is still technology driven. It is sometimes difficult to know who is really leading the group.

[...] Zend is a market-driven organization, in the true sense of the word. Note that I say 'market' driven, not 'marketing' driven. We succeed at getting our entire organization, from marketing to sales to R&D, focused on market activities and customer needs. In most companies, that's a tough thing to do. But with Zeev and Andi being the central figures in a community of 500,000 developers, we see trends before they start."

Running an Open Source business is possible. Both MySQL AB and Zend Technology are highly successful. Zend signed contracts with industry giants; MySQL is now being integrated in long term strategies by significant software editors. Open Source brings robustness and wide spread to a corporate product that would otherwise stay hidden. It also adds transparency to the code, and keeps the development team on the cutting edge. Anyone will see any of their flaws, they must stay the best. Just like the usual business credo. ■

Links & Literature

- MySQL AB: www.mysql.com/
- Zend Technologies: www.zend.com/

Inside Wire

by Leendert Brouwer

In this article we're going to look at a few things that might not be something you intuitively think of when approaching certain problems, or you might not even see the problem in the first place. As we all know, PHP has a huge userbase. If a lot of people use a technology, then there's a lot of experience out there. Some programmers invent neat solutions to solve certain things, and sharing them with peers is generally the next logical step in the PHP culture.

Making URL tampering less inviting

The fact that you should never trust a user should be an extension of the programmer's brain. When programming, a decent amount of paranoia is often needed to avoid having your application cracked. Visitors can be downright mean, and we should punish them for that as soon as we can. Ideally, even before they're tempted to mess with our URL's. How? One way is to encode the parameters in the URL so that it is less obvious what's in them. Say, you need to pass a username along with the URL. First, we might choose to not call our parameter "username". Instead, we could use a name that does not expose the nature of our parameter, so that Mister "oh-im-so-cool" Cracker doesn't really have a clue about what the parameter is supposed to represent. To keep our example simple, we'll just use "u" for the name. Listing 1 shows how we can send the encoded value along with the URL, and decode it at the other end. To encode the string we use `base64_encode()`, which is a function that is normally used to encode binary data for safe transport, but it works fine for our purpose too. To keep things nice we encode the base64-encoded string with `rawurlencode()` to comply with RFC 1738 and pass the parameter that way using a HTTP Location header. In the receiving script we simply `rawurldecode()` the incoming GET-parameter "u" and use `base64_decode()` to get our original string

back. Now the visitor will see a somewhat strange URL like `http://www.yourdomain.com/letsgohere.php?u=SG9seUd-vYXQ%3D` and will be confused, as we intended. Of course this is not meant to be used for actually securing your data, but it's a nice trick to scare off potential script kiddies or leechers.

Requiring authentication codes

Many times, I have gotten mailinglists that had a URL to unsubscribe from the list through a url like this: `http://www.somesite.com/unsubscribe.php?email=myemail@mydomain.com`, and by clicking the link you're unsubscribing yourself. It's just too tempting to play with that. Guess what happens when you launch `http://www.thedomain.com/unsubscribe.php?email=info@somesite.com`. It is likely that the people behind `somesite.com` have subscribed themselves to the mailinglist to receive their own mailinglists just to confirm it has been sent. The next time they might just be a little puzzled because they're not receiving any mail. There are of course a lot of variations on this particular kind of prank inviting situations. To avoid this, when setting up the subscription system for the mailinglist, you could store some unique code that goes with the email address. That way you can include both in the link you use for unsubscribing, and the email address will only be unsubscribed when the com-

bination of both the email address and the unique code is a valid match. Code that could be used to generate a unique string is visible in Listing 2 (I've used `substr()` to limit the length of the code because it looks ugly). Now the link to remove yourself from the list could look like this: `http://www.thedomain.com/unsubscribe.php?email=info@somesite.com&code=78c7c1`. That will take some guessing before someone can do some annoying things, because without the match of email and code, removal is not possible. This is an easy fix in case you're writing applications that use information to trigger certain actions that can easily be tricked.

A little more strict on incoming data

A lot of programmers are stressed because of tight deadlines. That's not something we can get out of, it has been like that for decades now. However, this also has the unfortunate effect that a lot of sloppy code gets written, which can lead to strange results at times. For example, too often I've experienced scripts that only checked if a variable existed after submitting a form, but did not look at the data that was coming in at all. Listing 3 shows an example. But who is that guy whose name consists of only a space? Nobody! That's why we could at least check if the value of the field we want to validate contains any characters besides a possible space. Listing 4 shows how to do just this for a field in a form which is being submitted using the *HTTP POST* method. In the if-statement we use *trim()* to get rid of the

whitespace at the beginning and the end of the string. Therefore, there can be nothing left but other characters than space. We see if there are in fact any characters left by invoking *strlen()* on the remaining string. If that value is bigger than zero, we know it is set. If it's not, there were only spaces in the field. Of course this is by no means a strict way of dealing with your data, but it sure is better than just testing if the variable is there and it can save some trouble. If you really want strict validation of incoming data you're better off with regular expressions in most cases.

Listing 1

```
// pass encoded value
header("Location: http://www.yourdomain.com/letsgohere.php
        ?u=".rawurlencode(base64_encode($username)));
// decode value at the other end
$username = rawurldecode (base64_decode($_GET['u']));
```

Listing 2

```
$unique_code = substr(md5(uniqid(rand(), 1)), 0, 6);
```

Listing 3

```
if(isset($_POST['your_name']))
{
    // do things
}
```

Listing 4

```
if(strlen(trim($_POST['your_name'])) > 0)
{
    // do things
}
```

Listing 5

```
<?php
$foo = array(
    'one' => 'two',
    'three' => 'four',
    'five' => 'six'
);
$csv_data = "";
foreach($foo as $key => $val)
{
    $csv_data .= "$key,$val\r\n";
}
header("Content-type: application/ms-excel");
header("Content-Disposition: attachment; filename=data.csv");
echo($csv_data);
?>
```

Listing 6

```
#!/usr/bin/php -q
<?php
mail("foo@bar.com", "This is PHP talking", "Hey the cron daemon was
                                         running me!");
?>
```

Listing 7

```
#!/bin/sh
/usr/bin/lynx -dump -auth=username:secretpass
http://www.yourdomain.com/path/to/script.php
```

Overriding `safe_mode` with the CGI binary

A lot of us probably have faced situations in which we don't have much say about the environment that's going to be used for the things that are being programmed. That can be extremely annoying at times, primarily because webhosting companies tend to limit what you can do with PHP on their web servers, thus limiting the set of functionality that you can use. There's a nice trick to bypass this kind of "security" in some situations. A lot of companies (although not recommended for performance reasons) still install PHP as a CGI binary. They also tend to be a bit meaner than that by not letting us use *.htaccess* files to influence the PHP configuration (which is caused by Apache's *AllowOverride* directive not allowing Options), and on top of that, they will run PHP in safe mode. That's not a very nice working environment, is it? Fortunately, there's a hack, or rather, a fact, that a lot of people don't know about. When running PHP as a CGI program, the PHP interpreter always tries to look for a *php.ini* in the directory in which the script resides. That allows us to just override the *safe_mode* directive by putting *safe_mode = Off* in a *php.ini*, stuff it in the relevant directory, and boom.

throwing data at MS Excel

I have often seen questions from people who want to output data in MS Excel format. Most of the time, the only reason people want to do that is so they can look at the data in nice, organized columns. In that case, you would not need the logic a spreadsheet program provides and thus, you do not actually need to use an MS Excel file format for that. MS Excel can read comma-delimited files as well, which are a lot easier to create and only hold data. Listing 5 shows a simple example to accomplish this. As you can see, it's pretty easy. I created an array, looped through the contents of it and added it to a string. Sending the correct HTTP headers is next on the list. Ideally, we would like the browser to come up with a dialog for downloading or direct opening. As the idea is to load the data into MS Excel, we can simply use the string *application/ms-excel* as the value for the Content-type header. That will create the awareness that we're dealing with a MS Excel file here. We set the Content-Disposition header to *attachment* as we do not want the content to appear inline (in the browser) and after this we come up with a name that will be used to save the file to the client's disk. I've chosen *data.csv*. Lastly we print the contents of the string to the client. The script will now cause the dialog to show up and (depending on your browser) will give you the option to download or open the file directly. If MS Excel is installed the contents will now be shown nicely in MS Excel. That's all there's to it.

Running a PHP script as a cron job

On some occasions you want to automate certain tasks, tasks that PHP is particularly good at. So you thought that's not possible with PHP and ported your idea to a bash script? Too bad. Doing that with PHP is pretty trivial. There are actually two common ways to achieve this. The first is to use the command line interpreter directly, so that means you'd just do the shell scripting in PHP. Starting from version 4.3.0, PHP is compiled with *—enable-cli* by default, which means that the command line interpreter will be available. Listing 6 shows an example of how to write a shell script in PHP. We just put a she-bang (path to the interpreter) on top of the file to point to the command line interpreter (the *-q* parameter is used to suppress HTTP headers). Give this file permissions so that it can execute, let the cron daemon know when to execute it (how to do this is in the docs of your OS), and there you have a nice cron job written in PHP. Just in case you do not have access to the CLI (because PHP was compiled with *—disable-cli* or on older PHP versions that don't have it enabled), there's an alternative way of doing it which is a bit more tricky, but still a fairly clean hack. You can just put the script which needs to be executed in a web directory. In Listing 7 you can see a regular shell script. In this script, we invoke the text-based Lynx browser to execute the PHP file. The *-dump* parameter makes sure Lynx will exit once the request is completed. Assuming we don't want the script to be executed by accident, it's probably best if you protect the directory the script resides in with a password. When using HTTP authentication, Lynx needs to know the authentication data so that it can access the script. This is accomplished by using the *-auth* parameter, which can be given a username and password, delimited by a colon. The PHP script you're calling can be a regular script, there's nothing special about this. As with the method mentioned earlier, you give the shell script execute permissions, tell cron when to execute it, and we're done. ■

Links & Literature

• Comments and Questions: forum.php-mag.net/



NEW YORK PHP

training • open source projects • amp technology • php consulting
<http://nyphp.org/>

Any Questions?



Ask a guru!

editors@php-mag.net

forum.php-mag.net/askaphpguru



Bug Off

A tutorial on how to resolve and prevent bugs from impeding your PHP scripts.

by Ilia Alshanetsky

The ability to write bug free code is perhaps the holiest of the programming grails that every programmer tries to achieve at least once in their career. This seemingly simple goal often proves to be nearly impossible to accomplish, resulting in countless delays that drive the release date beyond the horizon. Fortunately, there are a number of tools and techniques that can help you to avoid bugs and if any are found, resolve them in the most expedient manner.

The first step in writing bug free code starts before even a single line of code is written. It begins with a development of a precise plan of action that is to be followed to the letter. If you happen to be working in a team, make sure that each team member is aware of their part and has at least a conceptual idea about the final outcome. Stray as little as possible from the predetermined plan, since that often leads to bloat, bugs and undefined behavior. If possible, establish some intermediate steps where you will be able to test the written code extensively to ensure that it is working as expected. This will allow you to test your code a small piece at a time and if there is a bug, reduce the code base you need to search through.

To allow for intermediate code testing, you should try to make your code as modular as possible. This means breaking it down into individual functions and/or classes. Consequently your code will be easy to read and equally easy to debug because you will be able to test each part as soon as it is written. Remember to avoid making large functions as this is detrimental to the goal of keeping a small modular code base that is easy to test and debug. As a rule of thumb, functions should be no longer than forty to fifty lines. If a function is longer, consider breaking it down into two or more functions.

When writing your functions, take the time to document the purpose of each function as well as arguments it accepts and the data, if any, that it returns. It is very important that while you document your code you do it based on the code that you have written rather than on an idea of what code is supposed to do. This will force you to go through the written code and allow you

to compare the actual code to the concept in your head. Such an approach will result in accurate comments and should a mismatch between the two occur you will be made aware of the problem immediately. Accuracy of comments is very important, since you will most likely be relying on the comments when trying to determine the expected behavior during debugging. Accurately commented code is also less likely to be broken during upgrading, since the maintainer will be able to understand what the code is supposed to do rather than have to make assumptions, which in many cases may not be entirely correct. A word of caution: don't get carried away when writing comments, keep them short and as simple as possible to avoid ambiguity. Confusing comments can be worse than no comments at all, since they could mislead the reader as to the nature and purpose of the code. In most cases the person reading your comments will be someone other than yourself, therefore you should try to make your intent as clear as possible. This is especially important when working with other people who may be basing portions of their own code on yours. Lack of clear understanding may result in conflicting APIs and lead to bugs that are extremely difficult to resolve. While the individual parts may work as expected, the sum of the parts will work outside of the norm.

While it may be slightly annoying having to spend time writing comments mostly for the benefit of others, you do gain from the process. By creating an environment in which your code can be easily understood by others you will be able to gain meaningful code reviews from your peers and co-workers. While they are unlikely to spot deep implementation problems,

```
<?php
function apply_magic_quotes(&$arr)
{
    $c = count($arr);
    for ($i = 0; $i < $c; $i++) {
        if (is_array($arr[$i])) {
            apply_magic_quotes($arr[$i]);
        } else {
            $arr[$i] = addslashes($arr[$i]);
        }
    }
}

if (!ini_get('magic_quotes_gpc')) {
    apply_magic_quotes($_GET);
    apply_magic_quotes($_POST);
    apply_magic_quotes($_COOKIE);
}
?>
```

Fig. 1: Replacement for setting magic-quotes-gpc in the php.ini

```
<?php
if (version_compare(PHP_VERSION(), "4.1.2") < 0) {
    exit("PHP version 4.1.2 or higher is required to run this program.");
}
?>
```

Fig. 2: Ensuring a minimum version of PHP

they will often spot typos that you simply were not be able to see due to your involvement with the code. Properly commented code should make the process comparatively simple and not terribly time consuming, thereby making the job of finding volunteers for the task relatively simple. Whenever possible select people whose programming skill you consider to be superior or at least equivalent to your own. You'll end up with a more meaningful review rather than just an ego stroking pat on a back by a person who will be amazed by your unending brilliance. Peer reviews should allow you to eliminate the majority of the small, but terribly annoying bugs that stem from uninitialized variables and type mismatches.

Even with peer reviews, certain typos may still be found and given the strict-less nature of PHP make it into the final revision of the code, resulting in an undefined behavior. Unlike other programming and scripting languages such as C and Perl, PHP does not have a strict mode of operation. Thus a typo could very well make it into the final code, while in other languages it would have resulted in an immediately noticeable fatal error had the strict mode been used. I recommend using PHP's error reporting to enforce a certain level of strictness. By setting the error reporting to the highest level, *E_ALL* you will be able to see notice messages. These generally occur when your code tries to do things that while are possible, should not be done. The most frequent source of notice messages stem from usage of uninitialized variables. This is rather handy, as it allows for the prevention of bugs that can occur due to users exploiting such ambiguities and passing random data that could alter the script's behavior. By having the error reporting level set to *E_ALL*, you will immediately see an error message indicating the vulnerable code. This of course may not be an option when dealing with production code. At this

point you probably want to specify an *error_log* directive inside your *php.ini*, pointing to a file where the errors are to be logged to. At the same time set the *display_errors* ini directive to 0, which will prevent the error messages from appearing on screen. Thus, you end up with a log of any and all errors that may have occurred and avoid startling the user with error messages they should not see in the first place.

The user is one of the most unpredictable things your application will encounter. Since the majority of the PHP scripts are designed for a web environment, it is virtually guaranteed that a certain amount of interaction will occur between the user and the application. This process will likely involve the user passing a certain amount of data to the script, which in turn will present an output based on the given parameters. It is imperative that you never assume that the input contains the information you expect – quite the opposite. You should assume that the data passed is completely different from what is expected and perform extensive input validation. Failure to do so may result in numerous bugs and security faults. Consider the following situation, you've designed a simple guest book script where a user may specify a number of guest book entries they wish to see per page. Internally, the passed data, which is expected to be an integer, is used inside the *LIMIT* portion of the SQL query. Given no usual circumstances the data will be an integer and the code will work as expected. However, suppose for a moment that somehow a string was passed instead of the number. The result is that a normally working SQL query now fails and possibly gives the user the ability to inject hostile SQL code via your script and compromise the server. Had the script contained proper input validation this would not have been possible.

To ease the life of developers, PHP comes equipped with numerous functions that can validate contents of variables. For example if you expect an input to be an integer, use the *is_numeric()* function to check if the variable is indeed an integer or the *intval()* function, which will convert any input to an integer. On the other hand, if the expected input is a string, escape it using the *addslashes()* function, to prevent special characters such as from breaking your code. Normally, PHP tries to help you validate the user input when it comes to strings, by having the *magic_quotes_gpc* ini directive enabled by default. This will automatically perform *addslashes()* on all the data passed via GET, POST and Cookies. You should however be very careful and not blindly rely that this or any other *ini* options are set to their default values. Always use the *ini_get()* function to confirm that the *ini* settings are set to their expected values and if they are not, either change their values or perform the necessary steps to ensure that your code can account for the difference. In the case of *magic_quotes_gpc*, you will not be able to change the value of this directive within the script context. Hence, you

```

Input:
<?php
function a($string, $array)
{
    b($array, $string);
}

function b($array, $string)
{
    nl2br("a");
    debug_print_backtrace();
}

a("aa", array('a','b','c'));
?>

Output:
#0 b(Array ([0] => a,[1] => b,[2] => c), aa) called at [test.php:4]
#1 a(aa, Array ([0] => a,[1] => b,[2] => c)) called at [test.php:13]

```

Fig. 3: Stack trace generated using debug-print-backtrace

should consider writing a function capable of emulating the functionality normally offered by the lacking feature. Figure 1, is an example of one such function, this particular example works as a replacement for *magic_quotes_gpc*.

Magic quotes is not the only option that can significantly alter the nature of user input, it's cousin, *register_globals*, can be far more influential. When PHP was first designed, to simplify the development process, all input passed via GET, POST, COOKIE and FILES was registered as a variable. Meaning that if your script got passed 'abc=123' via GET you would have variable *\$abc*, who's value will be 123. The problem with this approach is that it could allow the user to pass data expected from GET via a cookie and so on. This one of the most common causes of security vulnerabilities in PHP scripts. To address this issue, as of PHP 4.2.0 this option is no longer enabled by default and the *\$_GET*, *\$_POST*, *\$_COOKIE* and *\$_FILES* super-globals should be used to access user input. The benefit to this approach is that it eliminates the possibility of user input being taken from the wrong place and removes much of the ambiguity present in the old approach. Compare the code snippet *\$_COOKIE['my_var']* to *\$my_var*. Clearly the first refers to the *my_var* variable that is expected to be found in a cookie. While the second, *\$my_var*, may have been a variable created by you in some part of code or it may have been a parameter passed by GET. Here the numerous possibilities may present an unsolvable quandary to a maintenance programmer. Therefore, you should make every attempt to avoid relying on the functionality of *register_globals*, even if you are certain that you could avoid the possible security issues involved.

INI settings are only one part of the PHP environment that may change. Certain extensions that you have on your development server may not be available on the production server. It is very important that you identify all of the extensions that your code relies upon and via the *extension_loaded()* function confirm that the needed extensions are available. If they are not, either raise an error indicating that the user needs to enable this

feature or implement a work around. You should also ensure that all of the PHP functions you rely upon are available, since certain functions may have become available only in later versions. The PHP manual is a superb resource for this information, since it indicates the PHP version when each function was first introduced. An alternative approach would involve using *function_exists()* to check for the actual availability of the function regardless of the version. To avoid having to check for every single function, you should use *version_compare()* to enforce the minimum PHP version required to run your code. For example if you want to ensure that your program always runs on PHP 4.1.2 or later you could use the code snippet in Figure 2.

Careful validation of the PHP environment will ensure that your code will always work as expected and the required functionality will always be available to your scripts.

In addition to the tools that allow you to confirm the availability of functionality, PHP also has a number of functions that can help you debug your code. Perhaps, the most useful of those functions is *var_dump()*. This function allows you to dump the contents and the type (i.e, integer, string, array, etc...) of any number of variables to screen, allowing you to quickly determine their contents. This function also supports arrays of any complexity and will print to screen the entire contents of an array recursively. The *var_dump()* function is particularly useful when you have a bug that you suspect is a result of invalid parameters being passed. *var_dump()* will allow you to quickly see the contents of all the parameters without needing to print each and every one of them manually. However, *var_dump()* has one limitation, the function will always dump it's data to screen. If displaying the debug information on screen is not an option, you will need to place the function within a PHP wrapper that would allow you to store the returned data in a variable that can be handled internally. Below is sample wrapper around *var_dump()* that demonstrates this concept.

```

<?php
function fvar_dump()
{
    ob_start();
    var_dump(func_get_args());
    $trace = ob_get_contents();
    ob_end_clean();

    return $trace;
}

$trace = fvar_dump($_ENV, $_SERVER);
?>

```

If you are just looking for the contents of an array and do not wish to see the type information shown by *var_dump()*, you can use it's sister function, *print_r()*. This function will recursively print the contents of an array, without any type information. Just

Function trace:

```
0.0000 37800 -> {main} () test.php:0
0.0004 38056 -> a("aa", array (0 => 'a', 1 => 'b', 2 => 'c')) test.php:13
0.0125 38240 -> b(array (0 => 'a', 1 => 'b', 2 => 'c'), 'aa') test.php:4
0.0126 38376 -> nl2br('a') test.php:9
```

Fig. 4: Stack trace generated with Xdebug

Input:

```
<?php
function a($string, $array)
{
    b($array, $string);
}

function b($array, $string)
{
    nl2br("a",2);
}

a("aa", array('a','b','c'));
?>
```

Output:

Warning: Wrong parameter count for nl2br() in test.php on line 9

Call Stack:

```
0.0000 1. {main} () test.php:0
0.0004 2. a('aa', array (0 => 'a', 1 => 'b', 2 => 'c')) test.php:12
0.0012 3. b(array (0 => 'a', 1 => 'b', 2 => 'c'), 'aa') test.php:4
0.0013 4. nl2br('a',2) test.php:9
```

Fig. 5: Stack trace using Xdebug with a buggy script

as with *var_dump()*, the function will always print the data to screen, so if you want to use its output it will need to be buffered.

In PHP 4.3.0 another two functions join the ranks of PHP's built-in debugging tools, *debug_backtrace()* and *debug_print_backtrace()*. These functions generate a stack trace of the code up until the function is called. In a way it is similar to placing *var_dump()* within each function and printing the passed arguments. It does have an advantage over *var_dump()*, since virtually the same result is accomplished with a single function call and each entry contains a file and a line number where the function can be found. Another advantage of these functions over *var_dump()* is that they do not require manual buffering. If you just want to display the stack trace you can use the *debug_print_backtrace()* function. On the other hand, if you want to get the trace data use *debug_backtrace()*, which will return an associated array containing the stack trace. Figure 3 is an example of a stack trace generated using the *debug_print_backtrace()* function.

The built-in debug tools will only get you so far, their usability is mostly geared to identifying a problem once you are aware of its general location. To truly debug a program, especially a complex one, you will need a debugger. The advantage of a debugger over the mostly manual debugging tools covered above is that it allows real time analysis of the script while it is running, thus enabling you to quickly locate the offending code segment and resolve the problem within that segment. At the moment the only multi-platform tools capable of accomplishing such a task are Xdebug and Zend Studio. Zend Studio is

GUI development suit written in Java that among other features integrates an extremely functional debugger. The full and trial versions can be found at: www.zend.com/. Xdebug, on the other hand, is an Open Source project released under the PHP license and can be downloaded from xdebug.derickrethans.nl/. Unlike Zend Studio, Xdebug works via a command line interface, which is slightly less user friendly than the GUI interface offered by Zend Studio.

The Xdebug extension consists of several components, the first of which is a stack trace generator, somewhat similar to the one found in PHP. The stack traces generated by Xdebug, comparatively speaking, are far more detailed and offer greater degree of control to the user. The ability to generate them is enabled by default when Xdebug extension is loaded. It allows tracing of code from the very moment the execution began up until the point that the stack trace is requested using one of the Xdebug functions. You can also manually specify the point from which to begin tracing of the code by using the *xdebug_start_trace()* function. Another option at your disposal is the ability to log the stack traces to a file, allowing tracing of a script without the need to modify even a single line of code. Simply enable Xdebug's tracing function via *php.ini* and specify a filename where the stack traces are to be written to. This is highly useful when debugging production code as it frees you from having to make extensive modifications for the sake of debugging.

The stack traces themselves are far more detailed than the ones offered natively by PHP providing a great deal of additional information that may help to identify possible problems. Consider the stack trace in Figure 4, which was generated using the PHP code in the previous example.

The first two columns represent the time taken to execute each function (in seconds) and the script's total memory usage (in bytes) after each function was executed. This information is very useful when hunting for possible bottlenecks that are caused due to code inefficiencies or a sharp increase in memory usage. Unlike PHP's native *debug_backtrace()*, Xdebug also includes PHP's native functions in its traces, making sure that no stone is left unturned in your search for the elusive bug. The trace itself is also displayed in a manner that makes it simple to identify the context of each function, through the use of tabs preceding each function. This formatting style allows even a person not familiar with the code to clearly see the flow of the script. As with native traces, this information can be either displayed on screen via *xdebug_dump_function_trace()* or be fetched in a form of an associated array for later use via *xdebug_get_function_trace()*. Of course, as previously mentioned, you can choose to have this information written to a file for later analysis. The advantage of this approach is that the original code remains the same and at the same time you have a

```
(init) run
Starting program: /home/php/debug.php
Breakpoint, a('aa', array (0 => 'a', 1 => 'b', 2 => 'c'))
at /home/php/debug.php:13
4      $ret = b($array, $string);
```

Fig. 6: Breakpoint output using Xdebug

Time Taken	Number of Calls	Function Name	Location
-> 0.0015069246	1	*{main}	debug.php:0
-> 0.0006510142	1	*a	debug.php:15
-> 0.0001620578	1	*b	debug.php:6
-> 0.0000220400	1	implode	debug.php:12
-> 0.0000089572	1	addslashes	debug.php:11
Opcache Compiling:			0.0004729033
Function Execution:			0.0006510142
Ambient Code Execution:			0.0008559105
Total Execution:			0.0015069246
Total Processing:			0.0019798279

Fig. 7: Profiling output using Xdebug

record of the script's state, which may be of use when comparing the script with future revisions.

The code tracing done by Xdebug is not limited to the generation of stack traces, it is also integrated into Xdebug's error handler which replaces the one offered by PHP. Consequently, when an error occurs, the error message includes a stack trace indicating the instructions executed by the script that have caused the error. This is especially important when you need to discover the exact set of conditions that have lead to the error, saving the time normally spent trying to replicate the bug. Figure 5 is an example of a Call Stack generated when an error occurred within a script.

As you can see, by simply having the Xdebug module enabled you get a far more informative error message than the one you would normally see.

Xdebug's abilities go further than just error reporting, it can actually save you from a rather nasty bug that has to do with infinite recursion. PHP uses the system's native stack without implementing any sort of protection and since the stack is limited it means that it is possible to overflow the stack and crash PHP. The most common cause of stack overflows in PHP are recursive functions that call themselves several thousand times (the mileage may vary depending on the complexity of the function and the size of the stack). When this happens, PHP overflows the stack and promptly crashes. Xdebug allows you to prevent this from happening by setting a maximum allowed recursion limit via the `xdebug/max_nesting_level` directive. If the limit is reached the script will simply stop, rather than crashing. When using this feature be careful not to set its value too low as it may cause termination of scripts with deeply nested functions. Setting the value to 200-300 should normally keep you safe in such cases. If you are enabling Xdebug predominantly for the sake of stack protection, I recommend setting the `xde-`

`bug.max_nesting_level` to 0 (this will disable parameter tracking) to ensure minimal performance penalties for utilizing this safety check. Keep in mind that unless you code uses recursive functions you probably should not load Xdebug just for the sake of stack protection.

While stack protection and code tracing are rather useful features offered by Xdebug, the core of the extension is its interactive debugger, which allows real-time debugging of PHP code. It works in two parts: server and client. The client is a small program separate from the Xdebug extension, which allows you to place breakpoints and analyze the state of the code. The server is the Xdebug extension itself, which connects to the client once a script is executed. This allows the debugger to work with any SAPI, and if need be, even permit remote debugging. By default Xdebug does not come with a compiled debug client and unless you are using Windows, you will need to compile one yourself. This is a fairly simple process, which involves a total of 5 commands (you may need to install *libedit* library, unless you already have it installed):

```
cd debugclient
./buildconf
./configure
make
make install
```

Once these commands have been run, you will have 'debug-client' inside `/usr/local/bin`, which is the client side part of the debugger. Now that all the necessary utilities are in place, the debugging process may begin.

It begins with the start of the *debugclient* application, which starts to listen on port 17869 for incoming debug data, followed by the execution of a script. This will result in the Xdebug extension connecting to the open socket and prompting you to specify the debugging parameters such as break points. A break point is an indicator that tells the debug server to pause execution allowing you to examine the state of the script at that point. To add break points use the *break* command with which you may already be familiar if you have used GNU Debugger (GDB). The command's syntax is quite simple, "*breakfile_name:line_number*". Once the breakpoints have been set, you can start the execution of the program by passing the *debug-client* the *run* command. This will begin the execution and continue executing the script until the first break point is reached. At that point, the execution will pause and display on the debug client's terminal the line on where the execution has been paused as demonstrated in Figure 6.

This by itself may not be enough information for your purposes and you will need to see this code within a certain context to fully understand the purpose of the current segment.

To do so, you can use the list command that accepts a line number as its single argument. Once executed, it will then display the ten lines which follow the specified line, inclusively.

```
(cmd) list 2
2    function a($string, $array)
3    {
4        $ret = b($array, $string);
5    }
6
7    function b($array, $string)
8    {
9        $string = addslashes($string);
10       return implode($string, $array);
11    }
```

Once the context of the code has been realized, it would be prudent to check the values of the available variables. This is accomplished via two commands, *show*, which will display all of the currently available variables, and *print*, which will print the value of a variable. When passing variable names to *print*, be sure not to include the \$ part of the variable name.

```
(cmd) show
```

```
$ret
$string
$array
```

```
(cmd) print array
```

```
$array = array (0 => 'a', 1 => 'b', 2 => 'c')
```

Assuming that no bugs were found at the current breakpoint, you'll probably want to advance to the next breakpoint, which is done by executing the *continue* command. Once executed, the debugger will resume the execution of the script until the next break point is reached. As an alternative to the *continue* command, you can use the *next* command. It will allow you to move through the code one instruction at a time. Which is rather handy when going through a suspicious code segment trying to determine the exact piece of code causing the problem. You can also use the *finish* command to advance execution forward. This command will continue executing the script until a breakpoint or the end of the current function is reached. The debug client has a number of other commands whose purpose may be uncovered by running the *help* command.

Even with the help of a debugger certain bugs, such as scalability issues, may not be easily solved. To assist in resolving such bugs, Xdebug comes with highly flexible profiler capable of performing detailed performance analysis of the code. The profiler supports a variety of operational modes that control the nature of the output and are tailored to identify various performance issues. The profiler can be activated in a manner very similar to that of the backtrace generator, via two simple functions, *xdebug_start_profiling()* and *xdebug_stop_profiling()*. To generate the actual profile you use the *xdebug_dump_function_profile()* function, which will output the profiling data to screen, or *xde-*

bug_get_function_profile(), which will return an associated array containing the profiling data. Both of these functions accept a single argument that defines what profiling mode to use. Figure 7 is an example of the profiling output generated using the *XDEBUG_PROFILER_SD_CPU_D* profiling mode.

Note that certain functions have an asterisk prefixed to their name. This is used to indicate that this function has been created in user space rather than being a native PHP function. The indicator is there to allow for a quick visual identification of your function rather than the functions which are part of PHP. In the majority of cases optimizing your code will prove to be far easier than optimizing the underlying C code behind PHP itself. The profiling data also includes summary information about how long the compilation (conversion of the script into Zend Engine opcodes) and interpretation steps took as well as the total time taken to execute the script.

It should be noted that while you have relatively little choice when it comes to debuggers, there is a fair bit of variety as far as profilers are concerned. The PECL repository contains a profiler APD – written by George Schlossnagles. If you are using PHP 4.3.0 or later you can install this profiler by simply running pear install APD. Another profiler, written by Steven Brown, can be found at www-cse.ucsd.edu/~sbrown/profiler/php-profiler.html. Given the growing demand for profilers, I suspect that even more will emerge in the near future.

Even with all of the available tools to help you resolve bugs you should be vigilant when writing your scripts in the first place. The time it takes to debug will almost always be greater than the time it would have taken to carefully write bug free code. This said you should not despair when you encounter bugs in your carefully crafted code and throw in the towel. Given the tools you are now familiar with, exterminating a few pesky bugs should be a walk in the park. So take your time, be careful and hopefully your quest for the holy grail will prove fruitful. ■

Ilia Alshanetsky has been developing PHP based applications since 1998. During this time he has made a number of contributions to the PHP project. Last year he joined the PHP's Quality Assurance team. He is also involved in the development of FUDforum, which is a PHP based bulletin board application and the profiling portion of the Xdebug extension.

Links & Literature

- Xdebug's website: xdebug.derickrethans.nl/
- Zend Studio website: www.zend.com/
- PHP Manual: www.php.net/manual/
- APD Manual: pear.php.net/manual/en/pecl.apd.php
- Profiling patch by Steven Brown: www-cse.ucsd.edu/~sbrown/profiler/php-profiler.html
- Comments and Questions: forum.php-mag.net/3/2/debug

Writing PHP Extensions

One of the most powerful web platforms in use today

by Zeev Suraski

One of the key factors of PHP's tremendous success was the very easy to use extensibility API. The simplicity of adding new functionality to the PHP engine, such as support for a new database or a new protocol, enabled a wide audience of developers to join in the project. The purpose of this article is to explain the process of creating a new PHP extension, and to explain how to implement some of the features commonly used in extensions.

A bit of history

Prior to PHP 3.0 with its extensibility API, there was PHP/FI 2.0B. In order to add functionality to the language, for instance, support for a new database, developers had to actually modify the language itself – including the lexical scanner and syntactic parser. There was very little infrastructure to work with; For example, managing resources (such as open files or SQL result sets) were left entirely to be implemented by each developer for each functionality block. Another major gotcha included the responsibility of each function (and in turn, of each developer) to take care of each and every argument that was passed to it. Failing to do so resulted in a messed up stack, and eventually brought the entire program down.

PHP 3.0 was a whole new ball game. This complete rewrite was mainly designed to address the deficiencies of PHP/FI – the limited, unreliable scripting engine, and the cumbersome extensibility options. It introduced a (almost completely) well-defined syntax, increased reliability and superior performance, but most important for our purpose – a complete and easy-to-use extensibility API.

New functionality was now encapsulated in self-contained modules, or *extensions*. These extensions no longer required any changes to the language's code – the scanner and parser were completely generic, and no hacks were necessary in order to extend PHP's functionality. PHP 3.0's core exposed powerful and efficient APIs for doing common things, such as memory

management, resource management and debugging. Last but not least, the API's moved many responsibilities away from module developers and down to the infrastructure, such as cleaning up the stack, freeing function arguments, etc.

The new APIs, along with the revamped language, marked the start of a new era and were some of the decisive factors for PHP's unprecedented growth.

Where are we now?

Even though PHP 3.0 came out as early as 1998, as far as the extensibility APIs are concerned, no far-reaching changes have been made. Most modules written for PHP 3.0 in 1998 can still be patched to work with PHP 4.0, and even the upcoming PHP 5.0 with minimal efforts. The design of PHP 3.0, which decoupled the scripting engine and services from the extensions' implementation code proved itself, and allowed us to completely renovate the engine, while still supporting the legacy code. Of course, the APIs did go through fine-tuning, and PHP 4.0 did introduce an optional higher-performance API, but in general, the contract between extension modules and PHP remained intact throughout the years.

Getting down to business

Getting carried away with historical overviews is a personal hobby of mine, but unfortunately, we still have to go through explaining how to write a module, and now is a good time to start. Before we get into actually implementing our functions,

there are three pieces of infrastructure that we have to get familiar with, as we would be using them in our code: memory management, resource management and file management.

Memory Management

The Zend Engine implements its own efficient memory manager. Memory managed by the memory manager takes advantage of the following features:

- **Caching**
Deallocated blocks are not always freed, but stored for fast reuse.
- **Leak prevention**
All memory allocated by the memory manager is implicitly freed at the end of the request, even if you forget to free it yourself (do not rely on that, a well-written extension should always clean after itself and free any memory it allocated on its own). In debug mode, the memory manager will also report any leaks it may detect.
- **Overflow/overrun detection (debug mode only)**
Some of the nastiest bugs have to do with writing too much information into a memory buffer (overflow), or having your data overwritten by mistake (overrun). The memory manager detects overflows and underflows, and reports the details to the user. Note that detecting these errors does not mean a crash may not still occur, as the memory manager cannot actually fix the problem, only find out about it.

It is therefore highly recommended to use the engine's memory manager in place of the standard *libc* memory manager. In order to do so, you should simply add an *e* prefix to the standard *libc* memory management calls, as illustrated in table 1.

Once you do that, you would be automatically taking advantage of the features mentioned before. Note that you may *NOT* mix between calls to the *libc* memory manager and the Zend memory manager for the same pointer. Memory allocated by *libc* must be freed/reallocated by *libc*'s functions only, and vice versa.

Also, note that in many cases, you *must* use the engine's memory manager in order for things to work properly. For instance, when you allocate a return value, or add elements to PHP's symbol table – you must allocate them using the memory manager. The engine, as well as other parts in PHP, expect this memory to be coming from the memory manager, and will try to deallocate or reallocate it using the memory manager. In such cases, using memory from *libc* will result in a crash. The rule of the thumb is to always use the memory manager when you are implementing functions, unless you have a very, very good reason to do otherwise (you do not). You should also use the mem-

ory manager when you are implementing the per-request startup/shutdown hooks, but *NOT* when implementing the server-wide startup/shutdown hooks.

Resource Management

Many PHP modules interact with external resources. For instance, a typical SQL module is capable of opening a link to a remote server, issue queries and manipulate result sets that come back from that server. As you probably know, each such external resource (SQL link, SQL result set, etc.) is referred to by PHP as a resource handle. For instance, if you try running the following code:

```
<?php print mysql_connect(); ?>
```

You would see:

Resource id #1

(provided you have a locally available MySQL server that does not require authentication to connect).

In order to simplify the implementation of modules that deal with resources, the engine features an extensive resource management API. As long as you use this API throughout your extension module, the engine will take care of keeping track of your resource and deallocating it as soon as it is no longer necessary.

The simplest way to explain resource management is through examples, and since our example extensions make use of resources to denote open files, we will simply explain each API function as we first use it.

File Management

File management in PHP is also handled by a special piece of infrastructure, instead of through the standard *libc* functions. The reason we need to take care of it ourselves is that in the operating system level, CWDs are process-wide, so different threads share the same CWD. Since we do not want a *chdir()* call in one thread to affect the code in other threads, PHP uses a *Virtual*

Description	libc name	Zend Engine equivalent
Allocate memory	<i>malloc()</i>	<i>emalloc()</i>
Free memory	<i>free()</i>	<i>Efree()</i>
Reallocate memory	<i>realloc()</i>	<i>erealloc()</i>
Allocate&initialize memory	<i>calloc()</i>	<i>ecalloc()</i>
Duplicate string	<i>strdup()</i>	<i>estrdup()</i>
Duplicate string (binary safe)	N/A	<i>estrndup()</i>

Table 1: Standard libc memory manager

CWD system, that provides a separate *virtual* current working directory for each thread. Naturally, this only affects the thread-safe version of PHP, but you are strongly encouraged to use this piece of infrastructure, to help to make your extension play nicely in multithreaded servers – you never know in which environments users would end up using your extension. Using the virtual CWD subsystem is extremely easy - simply use the *libc* functions, except make them uppercase, and prefix them with *VCWD_*. For instance, instead of calling

```
fopen("/etc/passwd", "r");
```

use

```
VCWD_FOPEN("/etc/passwd", "r");
```

The rest is done for you automatically.

When are we going to write some code?

Well, not quite yet, but we are almost there. In order to demonstrate the simplicity of adding new extensions to PHP, we are going to develop a simple extension, called *myfile*, that shockingly enough, deals with files. We are going to implement functions to open and close files, read from and write to files, and we will also not forget the ever-useful end-of-file checker. Unlike the old days, where you boldly had to go where no one has gone before, today PHP does a fair amount of the dirty work for you automatically. In order to get started, all you need to do

is write a file with the prototypes of the functions that you intend to implement in your extension, in the format:

return_type function_name(type1 arg1, type2 arg2, ...) description

In our case, the prototypes file would look similar to this:

```
resource myfile_open(string filename, string mode) opens a file
bool myfile_close(resource file_handle) closes a file
string myfile_read(resource file_handle, int size) reads from a file
int myfile_write(resource file_handle, string data) writes to a file
bool myfile_eof(resource file_handle) checks for end of file
```

Note that all function names are of the form *myfile_XYZ()*, which follows PHP's function naming convention rules:

All functions in module *foo* should be prefixed with *foo_* i.e., always prefer *foo_connect()* to a plain *connect()*.

Use underscores to separate between words i.e., use *foo_list_databases()* and not *FooListDatabases()*. Use lower-case letters.

Listing 1

```
static int le_myfile_handle;

/* destroy a resource of type 'myfile' */
static void myfile_close_file(zend_rsrc_list_entry *rsrc)
{
    FILE *fp = (FILE *) rsrc->ptr;

    fclose(fp);
    MYFILE_G(open_files)--;
}

PHP_MINIT_FUNCTION(myfile)
{
    ...

    /* Register a resource type for our file handle */
    le_myfile_handle
    = zend_register_list_destructors_ex(myfile_close_file,
    NULL,
    "myfile handle",
    module_number);
    return SUCCESS;
}
```

Listing 2

```
PHP_FUNCTION(myfile_open)
{
    char *filename = NULL;
    char *mode = NULL;
    int argc = ZEND_NUM_ARGS();
    int filename_len;
    int mode_len;
    FILE *fp;

    if (zend_parse_parameters(argc TSRMLS_CC, "s|s", &filename, &filename_len,
    &mode, &mode_len) == FAILURE) {
        RETURN_NULL();
    }

    if (!mode) {
        /* Assume mode is read-only if it's missing */
        mode = "r";
    }

    fp = VCWD_FOPEN(filename, mode);

    if (!fp) {
        RETURN_NULL();
    }

    /* Associate the file pointer with a resource handle of
    * our le_myfile_handle registered type
    */
    ZEND_REGISTER_RESOURCE(return_value, fp, le_myfile_handle);
    MYFILE_G(open_files)++;
}
```

Optional arguments can also be described by surrounding them in square brackets. For instance, let us say that in *myfile_open()*, we want to assume that if the user neglects to supply the mode argument, he meant to open the file for reading. In that case, *myfile_open()*'s prototype would look like this:

```
resource myfile_open(string filename [, string mode]) opens a file
```

This will create a function that requires the filename argument, it will not complain if it also gets the mode argument, but it will not complain if it does not get it either.

Now, that we have the prototypes file, we are all settled to start generating some code. Change to the *ext* directory under the PHP source tree, and issue the command:

```
./ext_skel --extname=myfile --proto=/path/to/myfile_prototypes.txt
```

ext/myfile will be created, along with all of the necessary files to get your extension up and running. You will also receive a list of instructions about how to use your extension once you are done implementing it; It would be a good idea to save them for later use.

Listing 3

```
PHP_FUNCTION(myfile_close)
{
    int argc = ZEND_NUM_ARGS();
    zval *file_handle = NULL;
    FILE *fp;

    if (zend_parse_parameters(argc TSRMLS_CC, "r", &file_handle) == FAILURE) {
        return;
    }

    if (!file_handle) {return;}
}

/* Obtain the file pointer from the resource handle that
 * was passed
 */
ZEND_FETCH_RESOURCE(fp, FILE *, &file_handle, -1,
    "myfile_handle", le_myfile_handle);

/* If we got to this point, we have a valid file_handle.
 * Removing it from the resource table will erase
 * it automatically.
 */
zend_list_delete(Z_RESVAL_P(file_handle));

RETURN_TRUE;
```

New Module Overview

Your newly created module, *ext/myfile/myfile.c*, contains a skeleton for a full-fledged PHP Extension. It contains hooks for server-wide startup and shutdown, hooks for per-request startup and shutdown, skeleton implementations for all of the functions specified in the prototype file, and a bit of infrastructure for your module. It also contains the *zend_module_entry* record for your module, that encapsulates all of the module's hooks in a single structure. Let us explore each of these.

Startup/Shutdown hooks

For the purpose of our module, we will not have to make significant use of the startup/shutdown hooks, but for more complex, they are exceptionally important. It is important to understand the difference between the two different kinds of startup and shutdown modules.

The **server-wide** startup hook, is typically called just once, for the entire duration of the server's uptime. In it, you should allocate resources and perform calculations that you would use throughout the server's lifetime. This is the standard place to initialize the configuration directives your module is aware of (INI entries).

The **server-wide** shutdown hook is called when the server shuts down. In it, you should deallocate and destroy each and every resource that you allocated in the server-wide startup.

Note: In practice, for historical reasons, the Apache Web server actually starts up its modules two times (calls startup, shutdown, and then startup again). Therefore, it is extremely important to implement a full shutdown hook that deallocates the resources, otherwise, your module will leak.

The **per-request** startup and shutdown hooks are called at the beginning and end of each and every request PHP serves, respectively. In the startup hook, you would typically initialize counters and values that are manipulated and used by your functions. For instance, in our module, we will keep a counter of how many open files we have, and this is the perfect place to initialize it to zero.

There is actually a third set of initialization/destruction functions, that is used by the thread-safe version of PHP – the per-thread constructor/destructor. These hooks are typically used to initialize information which is local to each thread, and to a certain extent replace the server-wide startup/shutdown hooks, in the threaded version of PHP. Because of its relative complexity, we will not get into it in depth in this article.

What do these callbacks look like?

```
PHP_MINIT_FUNCTION(myfile)      /* server-wide start-up function */
PHP_MSHUTDOWN_FUNCTION(myfile) /* server-wide shut-down function */
PHP_RINIT_FUNCTION(myfile)     /* per-request start-up function */
PHP_RSHUTDOWN_FUNCTION(myfile) /* per-request shut-down function */
```

To modify any of these hooks, simply locate them in *myfile.c*, and add your implementation code there.

At this point, we will initialize the counter for opened files in the per-request initialization hook:

```
PHP_RINIT_FUNCTION(myfile)
{
    MYFILE_G(open_files) = 0;
    return SUCCESS;
}
```

The *MYFILE_G* macro is used to access any global variables that your module may use. If you use it consistently, it will significantly reduce the pain involved in making your module thread-safe. Of course, every property that you use has to be declared as well. Open *php_myfile.h*, locate the section that looks like this:

```
ZEND_BEGIN_MODULE_GLOBALS(myfile)
...
ZEND_END_MODULE_GLOBALS(myfile)
```

And add there:

```
ZEND_BEGIN_MODULE_GLOBALS(myfile)
...
    int open_files;
ZEND_END_MODULE_GLOBALS(myfile)
```

Registering our resource type

As we are going to be using resources in our extension, we have to tell the engine about this at the server-wide startup stage (listing 1).

As you can see, we declare a global integer, *le_myfile_handle*, which is assigned the return value of *zend_register_list_destructors_ex()*. We will be using this integer for subsequent calls to the resource management subsystem, whenever we want to register a resource of this type.

The first argument passed to *zend_register_list_destructors_ex()* is the destructor function, which should perform all the operations necessary to deallocate our resource. The second one is the destructor function for persistent resources – which I will not cover in this article. The third one is a human-readable name for the resource (displayed in error messages, etc.) and the fourth is always the variable *module_number*.

Function Implementation Hooks

This is where the fun stuff really begins, where you actually get to implement your code. Each function implementation is of the following format:

```
PHP_FUNCTION(function_name)
```

So, in our file you should be able to locate five such declarations:

```
PHP_FUNCTION(myfile_open)
PHP_FUNCTION(myfile_close)
PHP_FUNCTION(myfile_read)
PHP_FUNCTION(myfile_write)
PHP_FUNCTION(myfile_eof)
```

As a matter of fact, your file would contain a 6th implementation, that was added automatically for you:

```
PHP_FUNCTION(confirm_myfile_compiled)
```

This function, as its name implies, would help you find out whether your extension was successfully built into PHP. Once you verified that, it is safe to remove it.

Note that each and every function is also registered in a centralized place in the beginning of the file:

```
function_entry myfile_functions[] = {
    PHP_FE(confirm_myfile_compiled, NULL)
    PHP_FE(myfile_open, NULL)
    PHP_FE(myfile_close, NULL)
    PHP_FE(myfile_read, NULL)
    PHP_FE(myfile_write, NULL)
    PHP_FE(myfile_eof, NULL)
    {NULL, NULL, NULL}
};
```

This structure gives the engine information about which functions this module implements. If you add/remove functions from your module, you must also remember to add/remove them from this structure, otherwise, your function will not be available to you (if you forget to add it) or you would experience build problems (if you forget to remove it).

Let us take a closer look at our functions, and start adding the implementation code. Here is the template for *myfile_open*, generated based on the prototype we supplied:

```
PHP_FUNCTION(myfile_open)
{
    char *filename = NULL;
    char *mode = NULL;
    int argc = ZEND_NUM_ARGS();
    int filename_len;
    int mode_len;

    if (zend_parse_parameters(argc TSRMLS_CC, "s|s", &filename, &filename_len, &mode,
                                &mode_len) == FAILURE) {return;}

    php_error(E_WARNING, "myfile_open: not yet implemented");
}
```

Right now, all this function does is accept the arguments. It uses `zend_parse_parameters()`, which is a relatively new and easy to use way to check what arguments were supplied to your function, and convert them automatically to the right types. The first argument you supply to it is the number of arguments you wish to check for, then `TSRMLS_CC` (note, no comma!) which is used for passing thread-safety information, and then a string that describes the types of arguments you wish to receive. In this case, for instance, we expect to always receive a string as the first argument (the name of the file to read), and optionally, a second string as the second argument (if the user wishes to specify the mode in which to open the file). The description string is therefore `sls` (`s` denotes a string, `l` denotes that the following arguments are optional; Full documentation including the list of supported types is available at <http://www.php.net/manual/en/printwn/zend.arguments.retrieval.php>).

The rest of the arguments to `zend_parse_parameters()` are placeholders where the arguments should be stored.

Note: When storing strings, each string has two placeholders – a `char*` pointer that points to the string, and an integer that holds the length of the received string. Using this length instead of using `strlen()` is exceptionally important because it is much more efficient, and is also an important step towards making your extension binary-safe.

In listing 2 you can see what this function would look like after we add our implementation code.

The added code is mostly straightforward. We check whether the user supplied a mode, and default to read-only if she has not. We then attempt to open the file, using the *Virtual CWD* subsystem. If we fail – we return `null`. If we succeed – we register our file pointer with the engine's resource management subsystem, and increase the number of successfully opened files.

Let us concentrate for a second at the `ZEND_REGISTER_RESOURCE()` call. The first argument to it is always the variable `return_value`; It is a special variable, passed by the engine to each and every implementation function. As its name implies, it is used to store the return value of the function. Since our return value from this function is going to be the resource handle, we pass it on to `ZEND_REGISTER_RESOURCE()`, which will update it with the value of the newly acquired resource handle. The second argument, `fp`, is the resource itself. Resources would typically be pointers to some larger piece of information, e.g., file pointers (as in our case), SQL result sets, etc. The third and final argument, `le_myfile_handle`, is the resource type that we wish to register. As you may recall, we registered this variable in the server-wide startup callback.

Congratulations – you have just implemented your first PHP function! (with a bit of help, but still...)

Let us take a look at another function (listing 3), this time a function that *uses* a resource.

This function, especially with the comments, is pretty self-explanatory. One thing that you may find interesting is the data type of the placeholder where we store the *resource* – `zval`. `zval`, which stands for Zend Value, is the multipurpose value holder used by the engine and throughout PHP. This structure may contain scalar types including integers (long), floating point numbers (double), strings (`char *` pointer and an integer for the length) and booleans, as well as compound structures like arrays and objects (in PHP 5.0, `zval`'s will no longer contain object, but only **object handles**, but that is a different story). Finally, they may also hold resource handles, which is what we use it for in this particular case. As you get into more advanced module development, you would have to become more familiar with the internals of the `zval` structure – the API functions will only take you so far. For the purpose of our entry-level module, however, we will not get any deeper than using the API functions and macros.

Listing 4

```
PHP_FUNCTION(myfile_read)
{
    int argc = ZEND_NUM_ARGS();
    long size;
    zval *file_handle = NULL;
    char *buf;
    FILE *fp;
    int read_bytes;

    if (zend_parse_parameters(argc TSRMLS_CC, "rl", &file_handle, &size)
        == FAILURE) {return;}

    if (!file_handle) {
        return;
    }

    ZEND_FETCH_RESOURCE(fp, FILE *, &file_handle, -1,
        "myfile_handle", le_myfile_handle);

    /* emalloc() can never fail */
    buf = (char *) emalloc(size+1);

    read_bytes = fread(buf, 1, Z_LVAL_PP(size), fp);
    /* zval strings always have to be NULL terminated */
    buf[read_bytes] = 0;

    /* Initialize the return value as a string */
    Z_TYPE_P(return_value) = IS_STRING;
    Z_STRVAL_P(return_value) = buf;
    Z_STRLEN_P(return_value) = read_bytes;
}
```

Now that we know how to both register resources and also reuse them in later function calls, we are almost ready for writing real world modules. But not quite yet – so far, our functions did not return anything interesting, just simplistic boolean success/failure values, or, at most, a resource handle. What if we want to return something more interesting, such as an integer, or a string?

Consider the next function implementation in listing 4.

Let us take a look at a few of the elements of this function. Right after retrieving the passed arguments, we allocate room for storing the data that we will read from the file. Note the comment *emalloc() can never fail*. Of course, it does not mean that if you try to allocate a terabyte of memory, *emalloc()* would magically find it for you; It means that if *emalloc()* fails to find enough memory to satisfy your request, it will bail the whole of PHP out, cleanly. In practice, this means you do not have to check for error return values – if control arrived to the line after the *emalloc()* call, it means, for certain, that the allocation was successful.

After allocating the buffer, we read the amount of requested bytes from the file into it. We then ensure that the string is NULL terminated by putting a NULL at its end. Note the comment that reads *zval strings always have to be NULL terminated*; This is extremely important. If you forget to terminate your strings with NULLs, you would experience all sorts of unexpected bugs, which may eventually result in crashes.

Once we have the buffer with the information inside it, we still need to tell the engine that this is our return value from this function. As before, setting the return value is done by manipulating the special *return_value* variable (which is also a *zval*). In order to do it neatly, we use the API macros to tell PHP it is a string, then update the string pointer to point to our allocated buffer, and finally, update the length property with the proper value. Note that this last step, setting the length of the string, is exceptionally important. Failing to set the accurate length will result in odd behavior and will almost surely result in a crash. Also note that when we return strings from functions, they *must* be allocated, and specifically, they *must* be allocated using the engine's memory manager (e.g. *emalloc()*). If you assign a pointer to a static buffer, or to a buffer which was allocated by *malloc()* – your module would crash before you can say supercalifragilisticexpialdocius, which is not that impressive, but it would also crash before you can say Jack.

Returning Other Data Types

The engine exposes a complete API for returning values from all of PHP's supported data types. If you want your return value to be an integer with the value of 7, for instance, you can use *RETURN_LONG (7)*, and if you want it to be 7.3,

you should use *RETURN_DOUBLE(7.3)*. A full list of the available macros is available at <http://www.php.net/manual/en/printwn/zend.returning.php>.

Also, for every *RETURN_** macro, there is a twin macro that is prefixed with *RETVAL_* instead of *RETURN_*. The difference between them is that *RETURN_** macros actually explicitly return control from your function, whereas the *RETVAL_** macros only set the return value, without returning control. Effectively, this means that using the *RETVAL_** macros, you can continue to perform operations even after you set the return value, and possibly even change the return value. In order to actually return control to the engine, you would have to explicitly *return*; (or reach the end of the function body).

Building Your Extension

There is one final step that needs to be taken before you can call functions from your new extensions, and that is – building it. If you have been a good boy and kept the build instructions from *ext_skel*, it should be a fairly straightforward step. If you have not, though, no worries – these steps should get you through it:

- Under the root of PHP's source directory, run:

```
./buildconf
```

- If this script complains about any missing or outdated versions of required software – please download and install them (<ftp://ftp.gnu.org/pub/gnu/> is one place you can get them from).
- Reconfigure PHP, using the same configure line you usually use, only this time – add *--with-extname* (in our case, *--with-myfile*).
- Rebuild PHP, and install it as necessary.

Once you run the newly built PHP, you should be able to call all of the functions declared in the module. In our example, a good start would be:

```
<?php print confirm_myfile_compiled(); ?>
```

Conclusion

At this point, you should have enough information to begin writing your own extensions, and probably got a feel for how simple yet powerful it is. There is still a lot to be learned, namely – how to use INI entries, advanced resource management, how to use compound types (such as arrays or objects), and more. We shall leave that, however, to a future article. Good luck! ■

We are looking for you!



We receive requests from all over the world, from avid PHP enthusiasts, keen to know if they can purchase the International PHP Magazine from their local newstands. While we are already committed to managing distribution in several countries, and have several others on the anvil, we are always on the lookout for partners who can help us reach many more customers worldwide. The Software & Support Global Alliance Program is intended to make PHP enthusiasts aware of where they can buy our magazine locally, from either a store or online. We are also interested in hearing from you, about publishing our magazine in additional languages as a local version. Join the Software & Support Global Alliance Program, and we will build our business by partnering with you to build yours.

go to: www.php-mag.net/gap

Global Alliance Program



Come and visit us!

**NYPHP and Software & Support Media
at the LinuxWorld Expo**

Booth #5



January 21-23, 2004 – New York

www.software-support.biz | www.nyphp.org

The Truth about Sessions

Session Management Exposed

by Chris Shiflett

Nearly every PHP application uses sessions. This article takes a detailed look at implementing a secure session management mechanism with PHP. Following a fundamental introduction to the Web's underlying architecture, the challenge of maintaining state, and the basic operation and intent of cookies, I will step you through some simple and effective methods that can be used to increase the security and reliability of your stateful PHP applications.

It is a common misconception that PHP provides a certain level of security with its native session management features. On the contrary, PHP simply provides a convenient mechanism. It is up to the developer to provide the complete solution, and as you will see, there is no one solution that is best for everyone.

Statelessness

Hypertext Transfer Protocol (HTTP), the protocol that powers the Web, is a stateless protocol. This is because there is nothing within the protocol that requires the browser to identify itself during each request, and there is also no established connection between the browser and the Web server that persists from one page to the next. When a user visits a Web site, the user's browser sends an HTTP request to a Web server, which in turn sends an HTTP response in reply. This is the extent of the communication, and it represents a complete HTTP transaction.

Because the Web relies on HTTP for communication, maintaining state in a Web application can be particularly challenging for developers. Cookies are an extension of HTTP that were introduced to help provide stateful HTTP transactions, but privacy concerns have prompted many users to disable support for cookies. State information can be passed in the URL, but accidental disclosure of this information poses serious security risks. In fact, the very nature of maintaining state requires that the client identify itself, yet the security-conscious

among us know that we should never trust information sent by the client.

Despite all of this, there are elegant solutions to the problem of maintaining state. There is no perfect solution, of course, nor is there any one solution that can satisfy everyone's needs. This article introduces some techniques that can reliably provide statefulness as well as defend against session-based attacks such as impersonation (session hijacking). Along the way, you will learn how cookies really work, what PHP sessions do, and what is required to hijack a session.

HTTP Overview

In order to appreciate the challenge of maintaining state as well as choose the best solution for your needs, it is important to understand a little bit about the underlying architecture of the Web – the Hypertext Transfer Protocol (HTTP).

A visit to <http://www.example.org/> requires the Web browser to send an HTTP request to www.example.org on port 80. The syntax of the request is similar to the following:

```
GET / HTTP/1.1
Host: www.example.org
```

The first line is called the request line, and the second parameter (a slash in this example) is the path to the resource being requested. The slash represents the document root; the Web server translates the document root to a specific path in the filesystem. Apache users might be familiar with setting this path with the *DocumentRoot* directive. If *http://www.example.org/path/to/script.php* is requested, the path to the resource given in the request is */path/to/script.php*. If the document root is defined to be */usr/local/apache/htdocs*, the complete path to the resource that the Web server uses is */usr/local/apache/htdocs/path/to/script.php*.

The second line illustrates the syntax of an HTTP header. The header in this case is *Host*, and it identifies the domain name of the host from which the browser intends to be requesting a resource. This header is required by HTTP/1.1 and helps to provide a mechanism to support virtual hosting, multiple domains being served by a single IP address (often a single server). There are many other optional headers that can be included in the request, and you may be familiar with referencing these in your PHP code; examples include `$_SERVER['HTTP_REFERER']` and `$_SERVER['HTTP_USER_AGENT']`.

Of particular note, in this example request, is that there is nothing within it that can be used to uniquely identify the client. Some developers resort to information gathered from TCP/IP (such as the IP address) for unique identification, but this approach has many problems. Most notably, a single user

can potentially use a different IP address for each request (as is the case with AOL users), and multiple users can potentially use the same IP address (as is the case in many computer labs using an HTTP proxy). These situations can cause a single user to appear to be many, or many users to appear to be one. For any reliable and secure method of providing state, only information obtained from HTTP can be used.

The first step in *maintaining state* is to somehow uniquely identify each client. Because the only reliable information that can be used for such identification must come from the HTTP request, there needs to be something within the request that can be used for unique identification. There are a few ways to do this, but the solution designed to solve this particular problem is the *cookie*.

Cookies

The realization that there must be a method of uniquely identifying clients has resulted in cookies, a fairly creative solution. Cookies are easiest to understand if you consider them to be an extension of the HTTP protocol, which is precisely what they are. Cookies are defined by RFC 2965, although the original specification written by Netscape (wp.netscape.com/newsref/std/cookie_spec.html) more closely resembles industry support.

There are two HTTP headers that are necessary to implement cookies, *Set-Cookie* and *Cookie*. A Web server includes a *Set-Cookie* header in a response to request that the browser include this cookie in future requests. A compliant browser that has cookies enabled includes the *Cookie* header in all subsequent requests (that satisfy the conditions defined in the *Set-*

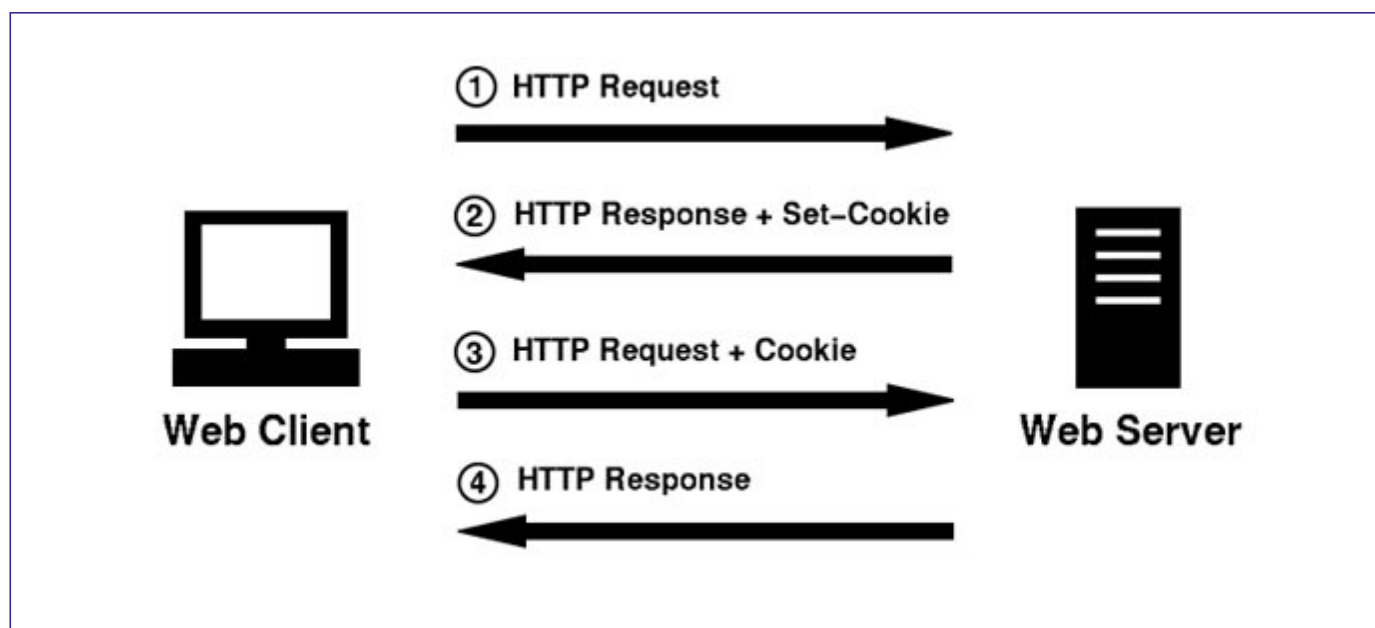


Figure 1: A typical Cookie exchange

Cookie header) until the cookie is expired. A typical scenario consists of two transactions (four HTTP messages):

- Client sends an HTTP request
- Server sends an HTTP response that includes the *Set-Cookie* header
- Client sends an HTTP request that includes the *Cookie* header
- Server sends an HTTP response

This exchange is illustrated in Figure 1.

The addition of the *Cookie* header in the client's second request (Step 3) provides information that the server can use to uniquely identify the client. It is also at this point that the server (or a server-side PHP script) can determine whether the user has cookies enabled. Although the user can choose to disable cookies, it is fairly safe to assume that the user's preference will not change while interacting with your application. This fact can prove to be very useful, as will soon be demonstrated.

GET and POST Data

There are two additional methods that a client can use to send data to a server, and these methods predate cookies. A client can include information in the URL being requested, whether in the query string or simply the path, although the latter case requires specific programming that is not covered in this article. As an example of utilizing the query string, consider the following example request:

```
GET /index.php?foo=bar HTTP/1.1
Host: www.example.org
```

The receiving script, *index.php*, can reference `$_GET['foo']` to get the value of *foo*. Because of this, most PHP developers refer to this data as GET data (others sometimes refer to it as query data or URL variables). One common point of confusion is that GET data can exist in a POST request, because it is simply part of the URL being requested and not reliant on the actual request method.

The other method that a client can use to send information is by utilizing the content portion of an HTTP request. This technique requires that the request method be POST, and an example of such a request is as follows:

```
POST /index.php HTTP/1.1
Host: www.example.org

Content-Type: application/
          x-www-form-urlencoded
Content-Length: 7

foo=bar
```

In this case, the receiving script (*index.php*) can reference `$_POST['foo']` to get the value of *foo*. PHP developers typically refer to this data as POST data, and this is how a browser passes data submitted from a form where *method="post"*.

A request can potentially have both types of data, like this:

```
POST /index.php?getvar=foo HTTP/1.1
Host: www.example.org

Content-Type: application/
          x-www-form-urlencoded
Content-Length: 11

postvar=bar
```

These two additional methods of sending data in a request can provide substitutes for cookies. Unlike cookies, GET and POST data support is not optional, so these methods can also be more reliable. Consider a unique identifier called *PHPSESSID* included in the request URL as follows:

```
GET /index.php?PHPSESSID=12345 HTTP/1.1
Host: www.example.org
```

This achieves the same goal as the *Cookie* header, because the client identifies itself, but it is much less automatic for the developer. Once a cookie is set, it is the browser's responsibility to return it in subsequent requests. To propagate the unique identifier through the URL, the developer must ensure that all links, form submission buttons and the like, contain the appropriate query string (PHP can help with this, however, if you enable the PHP directive *session.use_trans_sid*). In addition, GET data is displayed in the URL and is much more exposed than a cookie. In fact, unsuspecting users might bookmark such a URL and send it to a friend or do any number of things that can accidentally reveal the unique identifier.

Although POST data is less likely to be exposed, propagating the unique identifier as a POST variable requires that all user requests are POST requests. This is typically not a convenient option, although your application design might make it more viable.

Session Management

Until now, I have been discussing state. This is a rather low-level detail that involves associating one HTTP transaction with another. The more useful feature that you are likely to be familiar with is *session management*. Session management not only relies on the ability to maintain state, but it also requires that you maintain client data for each user session. This data is more commonly called session data, because it is associated with a

Listing 1

```
<?php
session_start();
$_SESSION['foo'] = 'bar';
<a href="session_continue.php">session_continue.php</a>
?>
```

Listing 2

```
<?php
session_start();
echo $_SESSION['foo']; /* bar */
?>
```

specific user session. If you use PHP's built-in session management mechanism, session data is maintained for you (in */tmp* by default) and available in the `$_SESSION` superglobal. A simple example of using sessions involves the persistence of session data from one page to the next. Listing 1, which presents the *session_start.php* script, demonstrates how this can be done.

Assuming the user clicks the link in *session_start.php*, the receiving script (*session_continue.php*) will be able to access the same session variable, `$_SESSION['foo']`. This is detailed in Listing 2.

Serious security risks exist when you write code, similar to the above, without understanding what PHP is doing for

you. Without this knowledge, you will find it difficult to debug session errors or provide any reasonably safe level of security.

Impersonation

It is a common misconception that PHP's native session management mechanism provides safeguards against common session-based attacks. On the contrary, PHP simply provides a convenient mechanism. It is the developer's responsibility to provide the appropriate safeguards for security. As mentioned previously, there is no perfect solution, nor a best solution that is right for everyone.

To explain the risk of impersonation, consider the following series of events:

- Good Guy visits *http://www.example.org/* and logs in
- The *example.org* Web site sets a cookie, *PHPSESSID=12345*
- Bad Guy visits *http://www.example.org/* and presents a cookie, *PHPSESSID=12345*
- The *example.org* Web site mistakenly believes that Bad Guy is indeed Good Guy

These events are illustrated in Figure 2.

Of course, this scenario requires that Bad Guy somehow discovers or guesses the valid *PHPSESSID* that belongs to Good Guy. While this may seem unlikely, it is an example of se-

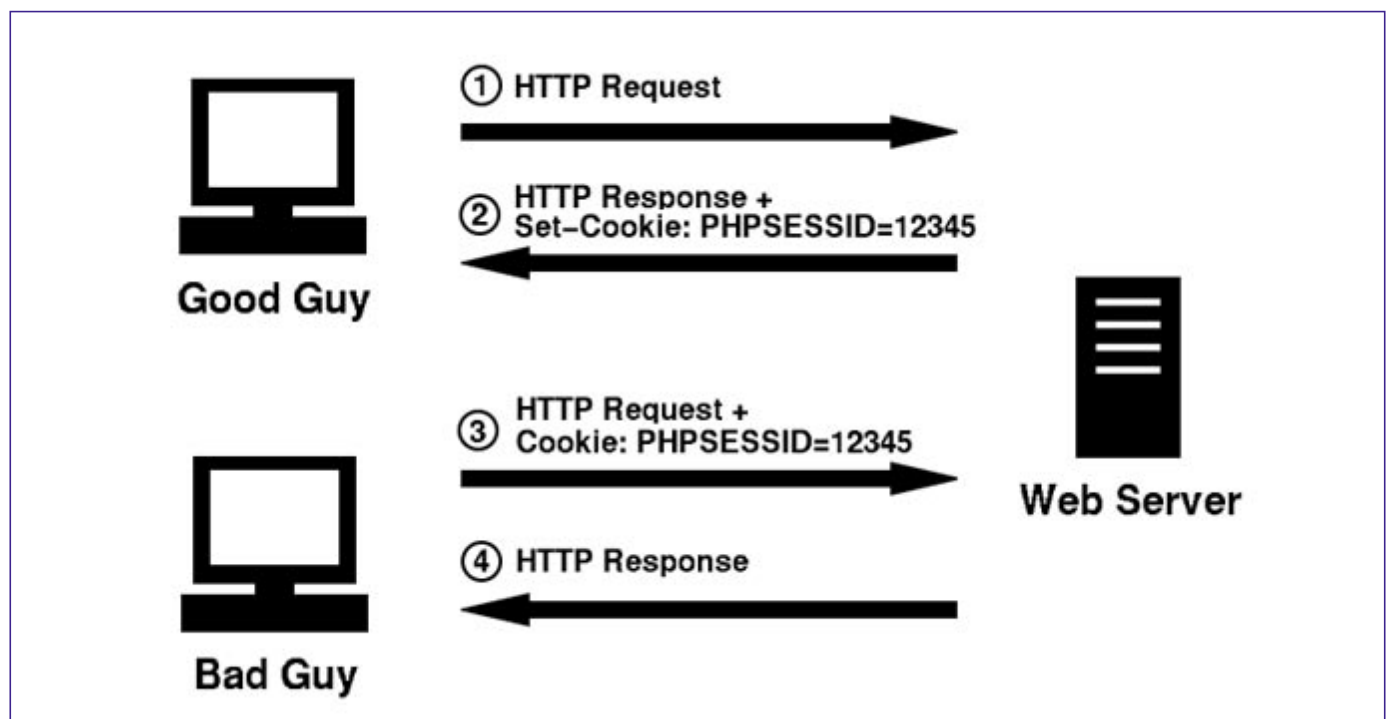


Figure 2: An Impersonation attack

curity through obscurity and is not something that should be relied upon. Obscurity isn't a bad thing, of course, and it can help, but there needs to be something more substantial in place that offers reliable protection against such an attack.

Preventing Impersonation

There are many techniques that can be used to complicate impersonation or other session-based attacks. The general approach is to make things as convenient as possible for your legitimate users and as complicated as possible for the attackers. This can be a very challenging balance to achieve, and the perfect balance largely depends on the application design. So you are ultimately the best judge.

The simplest valid HTTP/1.1 request, as mentioned earlier, consists of a request line and the *Host* header:

```
GET / HTTP/1.1
Host: www.example.org
```

If the client is passing the session identifier as *PHPSESSID*, this can be passed in a *Cookie* header as follows:

```
GET / HTTP/1.1
Host: www.example.org
Cookie: PHPSESSID=12345
```

Alternatively, the client can pass the session identifier in the request URL:

```
GET /?PHPSESSID=12345 HTTP/1.1
Host: www.example.org
```

The session identifier can also be included as POST data, but this typically involves a less friendly user experience and is the least popular approach.

Because information gathered from TCP/IP cannot be reliably used to help strengthen the security of the mechanism, it seems that there is little that a Web developer can do to complicate impersonation. After all, an attacker must only provide the same unique identifier that a legitimate user would in order to impersonate that user and hijack the session. Thus, it would appear that the only protection is to either keep the session identifier hidden or to make it difficult to guess (preferably both).

PHP generates a random session identifier that is practically impossible to guess, so this concern is already mitigated. Preventing the attacker from discovering a valid session identifier is much more difficult, because much of this responsibility lies outside of the developer's realm of control.

There are many situations that can result in the exposure of a user's session identifier. GET data can be mistakenly cached, observed by an onlooker, bookmarked, or e-mailed. Cookies provide a somewhat safer mechanism, but users can disable support for cookies, ruling out the possibility of using them, and past browser vulnerabilities have been known to accidentally leak cookie information to unauthorized sites (see www.peacefire.org/security/iecookies/ and www.solutions.fi/iebug/ for more information).

Thus, a developer can be fairly certain that a session identifier cannot be guessed, but the possibility that it can be revealed to an attacker is much more likely, regardless of the method used to propagate it. Something additional is needed to help prevent impersonation.

In practice, a typical HTTP request includes many optional headers in addition to *Host*. For example, consider the following request:

```
GET / HTTP/1.1
Host: www.example.org
Cookie: PHPSESSID=12345
User-Agent: Mozilla/5.0 Galeon/1.2.6 (X11;
Linux i686; U;) Gecko/20020916
Accept: text/html;q=0.9, */*;q=0.1
Accept-Charset: ISO-8859-1, utf-8;q=0.66,
*;q=0.66
Accept-Language: en
```

This example includes four optional headers, *User-Agent*, *Accept*, *Accept-Charset*, and *Accept-Language*. Because these headers are optional, it is not very wise to rely on their presence. However, if a user's browser does send these headers, is it safe to assume that they will be present in subsequent requests from the same browser? The answer is yes, with very few exceptions. Assuming that the previous example is a request sent from a current user with an active session, consider the following request sent shortly thereafter:

```
GET / HTTP/1.1
Host: www.example.org
Cookie: PHPSESSID=12345
User-Agent: Mozilla/5.0 (compatible; IE 6.0
Microsoft Windows XP)
```

Because the same unique identifier is being presented, the same PHP session will be accessed. If the browser is identifying itself differently than noted in previous interactions, should it be assumed that this is the same user?

It is hopefully clear to you that this is not desirable, yet this is exactly what happens if you do not write code that specifically checks for such situations. Even in cases where you cannot be sure that the request is an impersonation attack, simply

Listing 3

```
<?php
session_start();

if (md5($_SERVER['HTTP_USER_AGENT']) !=
    $_SESSION['HTTP_USER_AGENT']) {
    /* Prompt for password */
    exit;
}

/* Rest of code */
?>
```

Listing 4

```
<?php
session_start();
$_SESSION['HTTP_USER_AGENT'] =
    md5($_SERVER['HTTP_USER_AGENT']);
?>
```

Listing 5

```
<?php
session_start();

$fingerprint = 'SECRETSTUFF' .
    $_SERVER['HTTP_USER_AGENT'] .
    $_SERVER['HTTP_ACCEPT_CHARSET'];

$_SESSION['fingerprint'] = md5($fingerprint
    . session_id());
?>
```

prompting the user for a password can help prevent impersonation without adversely affecting your users too much.

You can add user agent checking to your security model with code similar to that in Listing 3.

Of course, you will need to first store the MD5 digest of the user agent whenever you first begin a session, as shown in Listing 4.

While it is not necessary that you use the MD5 digest instead of the entire user agent, it helps provide consistency and eliminates the necessity to validate `$_SERVER['HTTP_USER_AGENT']` before storing it in the session. Because this data originates from the client, it should not be blindly trusted, but the format of an MD5 digest is guaranteed, regardless of the input data.

Now that we have added user agent checking, an attacker must complete two steps in order to hijack a session:

- Obtain a valid unique identifier
- Present the same *User-Agent* header in the impersonation attack

While this is certainly possible, it is at least more complicated than if Step 2 was omitted. Thus, we have already strengthened the security of the session mechanism.

Other headers can be added in this way, and you can even use a combination of headers as a fingerprint. If you also include some secret padding of some sort, this fingerprint becomes practically impossible to guess. Consider the example in Listing 5.

The *Accept* header should not be used in the fingerprint, because Microsoft's Internet Explorer is known to vary the value of this header when the user refreshes as opposed to clicking on a link.

With a fingerprint that is difficult to guess, little is gained without leveraging this information in an additional way than demonstrated thus far. With the existing mechanism, there are still basically two steps required for impersonation, although the second step is more complicated now that the attacker has to reproduce multiple headers.

To add increased security, it is necessary to begin including data in addition to the unique identifier. Consider a session management mechanism where the unique identifier is propagated as GET data. If the fingerprint generated in the previous example is also propagated as GET data, an attacker must complete the following three steps to successfully hijack a session:

- Obtain a valid unique identifier
- Present the same HTTP headers being validated
- Present a valid fingerprint

If both the unique identifier and the fingerprint are propagated as GET data, it is possible that an attacker who can obtain one will also have access to the other. A safer approach is to utilize two different methods of propagation – GET data and cookies. Of course, this is reliant upon the user's preference, but an extra level of protection can be granted to those who enable cookies. Thus, if an attacker obtains the unique identifier by way of a browser vulnerability, the fingerprint is still likely to be unknown.

There are many more techniques that can be used to help strengthen the security of your session management mechanism. Hopefully you are well on your way to creating some techniques of your own. After all, you are the expert of your own applications, so armed with a good understanding of sessions, you are the best person to implement some added security.

Obscurity

I would like to dispel a common myth about obscurity. The myth is that there is no security through obscurity. As mentioned previously, obscurity is not something that offers ade-

quate protection, nor should it be relied upon. However, this does not mean that there is absolutely no security that can be provided through obscurity. On the contrary, backed by an already secure session management mechanism, obscurity can offer a small degree of additional security.

Simply using misleading variable names for the unique identifier and fingerprint can help. You can also propagate decoy data to mislead a potential attacker. These techniques certainly should never be relied upon for protection, of course, but you will not waste your time by implementing a bit of obscurity in your own mechanism. For those who do not have a basic understanding of session security, it is probably best to support the myth about obscurity, else someone might be misled into believing that it provides a sufficient level of protection.

Summary

I hope that you have gained several things from this article. Notably, you should now have a basic understanding of how the Web works, how statefulness is achieved, what a cookie really

is, how PHP sessions work, and some techniques that you can use to improve the security of your sessions.

If you have any questions or comments, my contact information is available on my Web site at shiflett.org/; alternatively, you could also post your feedback on this article at the PHP Magazine forum at forum.php-mag.net/. I would love to hear about your own solutions for secure session management, and I hope that this article provides the background information that you need to support your own creativity. ■

Links & Literature

There are many more resources available on this topic. A few notable ones freely available on the Web are as follows:

- <http://www.php.net/session> – PHP Manual Page on Sessions
- <http://www.ietf.org/rfc/rfc2616.txt> – HTTP/1.1 Specification
- <http://shiflett.org/books/http-developers-handbook/chapters/11> – Chapter 11 from the HTTP Developer's Handbook



Any Questions?

www.php-mag.net/forum

Clean Up Your Code

Making things better

by Leendert Brouwer

So you've finished your project, and it all works. Just like your boss and your client expected it to work. Everybody happy, all good, let's forget all about it, right? Yes. In a perfect world. The week after you had the champagne party to celebrate the successful finish of the project, the client makes a phonecall to your office.

It's about the survey application you created. He likes how he can present graphs of the survey results to the marketing staff now. He likes that a lot. He liked it so much, that he has decided he does not only want to have horizontal bar graphs of the results, he wants vertical bar graphs, plot graphs and pie charts as well. In a flash of a second you

remember the code you wrote to generate the horizontal bar graphs. You remember it was kind of a topdown clutter of database queries and calls to the GD library, but hey, it did the job well so you were satisfied.

The first thing that jumps to mind might be to dive into the source again, and do the same for the vertical bar graphs, the plot graphs and the pie charts. It will be a nasty job, it may be quite some work, but it does the trick, and the client will be happy. We'll just hope the client does not have similar demands in the future, because that would make things real messy. After all, this is the only way, right? Unless you, what? Apply refactoring?

If a change of plan during or after a software development process was exceptional, it would make the lives of programmers and software architects much easier. Unfortunately, the scenario from the introduction, and countless variations on that scenario, occur too often in reality. People's demands change, applications have to change, and these applications are often not prepared for that particular kind of change. This is one of the main reasons why applications become messy, full of hacks, resulting in unreadable code, and therefore become hard to maintain. In a worst case scenario, applications might become so hard to maintain that making changes is hardly doable, and is very time consuming. Something our bosses and clients do not generally like. And this is where refactoring can help us. Let us start by defining refactoring.

The definition of refactoring

Refactoring is changing the design of existing software without modifying its external behaviour, by applying a number of refactorings.

I will explain what those "refactorings" are later. First let's look at the first part of the definition. This is the revolutionary part of the refactoring process. It does not say: adding to the ex-

isting code so we will have those additional features. It is much more drastic. It explicitly says to change the design of that code. Those changes are not even directly visible from the outside, only in the code itself. The application will behave just like it used to. What sense does that make? I will discuss this in the next section.

Are you ready for the future?

An old software engineering paradigm goes along the lines of the following: if it works, don't touch it. That was then. Software engineering has matured a lot since then, and that paradigm should be buried and forgotten, because nowadays this statement just doesn't apply anymore. Constantly adding and not changing the current design turns out to be inefficient in the long run. If you don't change the design when there might be situations in the future that require more nasty hacks, you're inviting obfuscation into your project. Don't get this the wrong way - refactoring is not about making your code as flexible as possible, so that any change or new feature can easily be implemented. This would cost time, and your application would be more flexible than necessary. On top of that, it tends to get way too complex. However, when refactoring you must look at your

code and wonder if you can easily change the design in the future, in case it is needed. If so, you stick with the current code. If not, you make your code more flexible so that it does allow these changes easily.

When to refactor?

The process of recognizing situations that need refactoring take experience and a bright mind. However, once you've done some successful refactorings it sort of becomes natural. Keep in mind that refactoring is not about specific code problems or syntax errors, it is merely about design problems. The code should already work before you apply refactoring. Here are a few examples of obvious situations in which refactoring is often necessary.

- **Code that needed to be repeated**

Code duplication might or might not happen the first time when you're coding. Especially when features have to be added later on that are slightly different from what was already written, the chances of writing almost the same code are high. The real problem of duplicated code is not only that your code gets messy, but when changes are required for these duplicated pieces of code, you'll have to make the changes in all these places, which can be quite cumbersome and time consuming.

- **The entity that could do it all**

In our passion for programming, we might sometimes find ourselves lost in the art of creation. Creating and seeing your creature working can easily become obsessive. Nothing wrong with that, except that we might implement just too many features. Implementing more features than we really need creates unnecessary overhead for that particular entity. Usually you just shouldn't implement any more functionality than the specification requires.

- **The objects that liked each other a lot**

In an object oriented or modular application you will want to abstract separate objects or modules as well as you can, however, in practice this is often not the case. For certain operations, one object might rely a lot on another. Thus, if you want to change things in one object, chances are high you'll have to modify the other object as well, since they are tightly coupled to each other. The worst case scenario being that once you modify the class you relied on too much it longer works as it is supposed to, so you'll have to redesign a much larger part of your application structure.

- **The fish that could change a tire if asked**

When you study your entities carefully, you might find things that really do not belong inside that entity. For example, a function that is merely responsible to take a segment from a character string should not also have the functionality to filter out certain unwanted characters. You would create a separate

Listing 1

```
function parseRecipients($recipients)
{
    include_once 'Mail/RFC822.php';

    // if we're passed an array, assume addresses are valid
    // and implode them before parsing.
    if (is_array($recipients)) {
        $recipients = implode(' ', $recipients);
    }

    // Parse recipients, leaving out all personal info. This is
    // for smtp recipients, etc. All relevant personal
    // information should already be in the headers.
    $addresses = Mail_RFC822::parseAddressList($recipients, 'localhost', false);
    $recipients = array();
    if (is_array($addresses)) {
        foreach ($addresses as $ob) {
            $recipients[] = $ob->mailbox . '@' . $ob->host;
        }
    }

    return $recipients;
}
```

Listing 2

```
function parseRecipients($recipients)
{
    include_once 'Mail/RFC822.php';

    $recipients = $this->prepareRecipients($recipients);

    $addresses = Mail_RFC822::parseAddressList($recipients, 'localhost', false);
    $recipients = array();
    if (is_array($addresses)) {
        foreach ($addresses as $ob) {
            $recipients[] = $ob->mailbox . '@' . $ob->host;
        }
    }

    return $recipients;
}

function prepareRecipients($recipients)
{
    if (is_array($recipients)) {
        $recipients = implode(' ', $recipients);
    }

    return $recipients;
}
```

Listing 3

```

class TemplateEmail
{
    var $contentType;
    var $recipient;
    var $body;
    var $from;
    var $replyTo;
    var $subject;
    var $templateContents = null;

    function Email($recipient, $subject)
    {
        // ...
    }

    function setTemplate($templateFile)
    {
        // ...
    }

    function parseTemplate()
    {
        // ...
    }

    function setContentType($contentType)
    {
        // ...
    }

    function setBody($body)
    {
        // ...
    }

    function setFrom($from)
    {
        // ...
    }

    function setReplyTo($replyTo)
    {
        // ...
    }

    function send()
    {
        // ...
    }
}

```

Listing 4

```

class Email
{
    var $contentType;
    var $recipient;
    var $body;
    var $from;
    var $replyTo;
    var $subject;

    function Email($recipient, $subject)
    {
        // ...
    }

    function setContentType($contentType)
    {
        // ...
    }

    function setBody($body)
    {
        // ...
    }

    function setFrom($from)
    {
        // ...
    }

    function setReplyTo($replyTo)
    {
        // ...
    }

    function send()
    {
        // ...
    }
}

class TemplateEmail extends Email
{
    var $templateContents;

    function setTemplate($templateFile)
    {
        // ...
    }

    function parseTemplate()
    {
        // ...
    }
}

```

Listing 5

```
class PlotBand
{
    var $prect = null;
    var $depth;
    var $dir, $min, $max;

    // ..
}
```

function for that. A class that sends an HTML email should not have functionality to make XML dumps of your database. A fish doesn't have to be able to change a tire - the world is strange enough as it is.

When Martin Fowler wrote his book about refactoring, he referred to this kind of problems as "bad smells" in code. You learn how to recognize these problems as you gain experience. You start to "smell" them. Despite the fact that most publications about refactoring are quite heavily based on the object oriented side of things, this does not mean that refactoring is only applicable to object oriented applications. Refactoring is very applicable to modular or procedural code, so it doesn't really matter what your preferred design style is, you can still take advantage of the concept.

Moments to refactor

So now that you are fairly familiar with the situations in which you might want to refactor, when is the best moment to refactor? It really depends. I will try to give you a brief overview of moments when you refactor.

- **While coding**

Sometimes when writing code, the situation might occur where, for example, you find yourself writing almost the same code twice. This is when the word "refactoring" should blink in red in front of your eyes. Writing the same code twice should hardly ever be necessary, and is pretty much always avoidable, given that your design permits this kind of flexibility.

- **While reviewing code**

Reviewing is important, whether it's your own code, code from someone on your team or even third party open source software you're using might be worth giving a closer look. Not only does it make you think about your code in a more intensive way, it's also a learning process in which you will quickly learn how to recognize situations that could not stand the future in the long run, and how to improve the design after carefully studying the current code.

Listing 6

```
class PlotBand
{
    var $plotRectangular = null;
    var $depth;
    var $direction, $min, $max;

    // ..
}
```

- **While repairing your code**

When you're looking to repair a bug, you sometimes look over the code and realize the design could have been better. You could just fix your bug and forget all about the design. You could also take that little extra time and change the design, then fix the bug, given the timeline you got from your superior permits you to take a little more time than you would need to fix only the bug.

- **When adding features later**

We're probably all familiar with change. Seemingly the client always wants the most impossible things to be changed, often in a later phases of the development process. Changing the design at this point is often more effective than simply adding code to the existing design. Because the design has changed you will not face the same problem the next time a change is to be made to that piece of your application.

Fear vs. Refactoring

What stops a lot of programmers actually changing their application design after it has been designed is fear. Their code works, and even though it is messy, it still serves a purpose. This way of thinking is understandable, but as Don Wells said in his explanation of Extreme Programming, you should realize that your application design might become obsolete over time. You must be open to the fact that once a design works, that does not mean the design will be working forever. You can be facing a situation in the future where your design just isn't the most elegant solution to the problem anymore. And it is that moment where you should not deny the fact that your design is obsolete, nor be afraid of drastically changing your design. It is not wrong – you are even encouraged to be aggressive about changing your application structure if this leads to a great improvement in the flexibility and maintainability of your application in regard to problems you could run into in the future, or problems you're already running into. Of course these kind of drastic changes are not without risk, since you're actually changing working code. But in the end, refactoring turns out to be an efficient way to prepare yourself for coming changes, and that's often worth a risk.

What are these “refactorings”?

Now that you understand the general idea behind refactoring, let’s move to the practical side of things. What are the “refactorings” that I mentioned earlier? I shall try to give a definition of what a refactoring is.

A refactoring is a structured way of improving an existing design.

There is no such thing as the “ultimate refactoring catalogue”. The list of refactorings grows every once in a while. This list can be found at one of the resources at the end of this article. It is probably best to show you how it works by giving a number of examples. I will show you a number of refactorings that are fairly well-known, by using code that comes from a variety of existing PHP applications to make the examples more realistic. Of course I’m not implying that the code I’m using to describe the initial situations is in any way bad. The code examples are merely used to illustrate how things work when refactoring.

Listing 7

```
class Table
{
    // ..

    function SetCellBGColor($row, $col, $bgcolor)
    {
        $this->table[$row][$col]["bgcolor"] = $bgcolor;
    }

    // ..
}
```

Listing 8

```
class Table
{
    var $rows;

    // ..
}

class Row
{
    var $cols;

    // ..
}

class Column
{
    var $color;

    // ..

    function setBackgroundColor($bgcolor)
    {
        return $this->bgcolor = $bgcolor;
    }
}
```

Extract Method

In Listing 1 there is some code that comes from the *Mail* class in the PEAR Mail package (cvs.php.net/cvs.php/pear/Mail/Mail.php). There is a function called *parseRecipients()* in this code, which takes a variable *\$recipients*, and parses the recipients out of the *\$recipients* variable to put them in an array. So far so good. Let us take a closer look to this function. You will probably see that there are comments inside the function that each have a block of code under them, so it is likely that the comments explain what goes on in the blocks. Once we read the comments, we can conclude that the function actually does two things, which are:

- preparing the recipients to be parsed, and
- parsing the recipients.

This is, in a way, more than the name of the function implies it can do. The function name implies nothing about any

Listing 9

```
class TellAFriend
{
    var $email;
    var $bonusAmount = 10;
    var $ident;

    function TellAFriend($email)
    {
        // ..
    }

    function tell($userid)
    {
        // ..
    }

    function emailExists()
    {
        // ..
    }

    function generateIdent()
    {
        // ..
    }

    function increaseUserBonus($userid)
    {
        // ..
    }

    function wonGiftBox($userid)
    {
        // ..
    }
}
```

preparations being made to the passed data before the actual parsing of the data. We can also rephrase this observation by concluding that the function has more responsibilities than the function name says it has. Such a situation can decrease readability, thus making debugging harder. A way to solve this is to write a new function that does the preparation of the data, and then call that function from the function we are refactoring. Once this is done, you can remove the now redundant code from the function. This is shown in Listing 2. Your code is not only more readable now, it is also easier to add other things that might be required to prepare the data in the future, in case some logic changes. A refactoring that is close to Extract Method is called Extract Class. Like Extract Method, Extract Class can be applied in object oriented environments.

Listing 10

```
class TellAFriend
{
    var $email;
    var $userid;
    var $bonusAmount = 10;
    var $ident;

    function TellAFriend($email, $userid)
    {
        // ..
    }

    function tell()
    {
        // ..
    }

    function emailExists()
    {
        // ..
    }

    function generateIdent()
    {
        // ..
    }

    function increaseUserBonus()
    {
        // ..
    }

    function wonGiftBox()
    {
        // ..
    }
}
```

With this refactoring you look for functionality (usually a combination of methods and variables) inside a class that does not belong to that class. You could also say, it removes functionality that does not directly relate to the class name, and puts it in one or more separate classes so that they are properly abstracted. If something inside your entity does not entirely match with the name of that entity, be it a class, a module, a namespace (PHP 5), think it over - better abstraction might be appropriate.

Extract Superclass

When I was working on a project once, one of the project requirements was that the application should be able to send emails that were based on templates. You can see the API for the class I created to do this task in Listing 3. It could not only mail a regular email, it could send an email that was based on an HTML template as well. In my *send()* method I would just look if the *\$templateContents* variable was still null, and if so, it would send a regular email. If it was anything but null, I would call *parseTemplate()* first, which would put the appropriate values inside the template that I was using, and it would send a templated email, after doing everything that was needed to send an HTML email instead of sending a regular email. So far so good. Then I started thinking. How future-ready would my class be like this? What if in the future there would be a need for emails that are based on XML feeds, or where the content needs to be pulled from a database? This could very well happen, since my project is backed by a rather large CMS and it would be a logical thing to do at some point. I could of course simply add it to the current class, but it'd be a big mess. This is where I decided I should be able to add features like this without too much effort. I then started redesigning this part of the project, and came up with the code that you can see in Listing 4. I determined which features of the class would be common among all variations that could be made of this class, and put those in a separate class that I called *Email*. I then extended this *Email* superclass in a class that I called *TemplateEmail*. This class now encapsulates the functionality that is specifically needed to send an email based on a template, and thus I could extend other classes with purposes along these lines just like I did with this one. This example shows one way to apply Extract Superclass. I applied it at the same time as developing, since I was thinking ahead as I looked over my code after I wrote it. Another moment to apply Extract Superclass is when two classes show obvious similarities. You move the similarities to the superclass, and just extend from that class so you can program by difference, which is much more efficient when you need to add features later.

Rename Method

As I was browsing through the source code out of jpGraph (www.aditus.nu/jpgraph/), I found a nice example to illustrate the Rename Method refactoring. While this is probably one of the easiest refactorings to apply, it is certainly a very important one. A great deal of lack of readability in code is caused simply by picking wrong names for your entities. Listing 5 shows an example of unclear names for class variables. The advantage of using, probably longer, but clearer names for your entities far outweighs the disadvantage. You should be able to read the code when you look over it. A few more characters won't hurt performance very much. Have a look at Listing 6 to see what I mean. Since there is no need to go too far, I left the *\$min* and *\$max* variables intact, because they are very common. This might all sound very small in a way, but it cannot be emphasized enough that it all matters a great deal. Obviously this refactoring cannot only be applied to variable names, it can also be applied on every name you define. If you spot unclear naming in your code, carefully change it. Keep in mind that there might be references to this entity somewhere else in the code, so thorough testing is required.

Replace magic number with symbolic constant

Time for another small, but important refactoring:

```
$TotalPriceFormatted = number_format($TotalPrice, 2);
```

What's wrong with that line? It might seem that there's nothing wrong with it at first sight. You can tell that it probably formats a price to have two decimals behind the decimal point. Fine, I can read that too when it's in a context that is just one line, or maybe ten. But now, imagine that number in a context of, say, hundred lines. When you glance over the code real quick then, would you see what happens? Maybe in this case yes, but there are a lot of cases where you would see a number, and not understand its meaning. Therefore, it is generally better to do so:

```
$TotalPriceFormatted = number_format($TotalPrice, DECIMALS_TOTAL);
```

We have defined that number with a clear name, *DECIMALS_TOTAL* (I called it *DECIMALS_TOTAL* on purpose, since there might be more types of decimals what it is would remain clear because every constant that indicates a decimal constant would be prefixed with *DECIMALS_*), and used that instead of the magic number. Now it is probably clearer to you that whenever you see the name of that constant, you have an idea what that code is doing, instead of being puzzled about that number floating around just like that. Another way to make things more readable is replacing the "magic" number with a function that would return the appropriate value, so as:

```
function getTotalPriceDecimals($price)
{
    return number_format($price, 2);
}
```

Replace Array with Object, Move Method and Rename Method

This subheading might confuse you a bit. All of a sudden I'm going to explain three refactorings at once? Well, yes. Also, I'm going to make a point. When you're refactoring, it is not a rule that you can only apply one refactoring on a piece of code. It is very well possible to combine two, three or even more refactorings when there is an obvious need for it. For taking things a little more hardcore, I've used an example that is shown in Listing 7. The class in Listing 7 is capable of programmatically showing customized HTML tables, which can be generated at runtime by PHP. The class is called *Table*, and in the implementation I found a method that will be interesting to illustrate applying a series of refactorings sequentially. The class method is called *SetCellBGColor()*, and it can, as the method's name correctly implies, set the background colour of a cell in the table. Good, we don't have to apply Rename Method here since it's clear enough already. Let's have a look at the method body. I think it is safe to assume that most of us are fairly familiar with HTML tables. We know that tables consist of rows, and that rows consist of columns. From the method body, we can conclude that these rows and cells are not actually being seen as separate entities. They are stored in a multidimensional array instead. The code would most likely be more readable and flexible if rows and cells are represented through their own objects. This is where we apply Replace Array with Object. Twice, in this case. In Listing 8 you can see how that works out. Next to the *Table* class, we will have both a class to represent a table row called *Row*, and a class to represent a table cell called *Cell*. The *Table* class will be composed of a stack of *Row* objects, and these *Row* objects will be composed of a stack of *Cell* objects. Now we have a nice composition of all these separate entities. We have actually meanwhile applied another refactoring, Extract Class, without even knowing it. Some refactorings show similarities, and sometimes without even being aware of it, we're applying one. That's not a bad thing, it's more of a natural thing, and we don't have to do anything about that. Now that we've done this, there's another thing to do. We would have turned the *SetCellBGColor()* method into a sort of a stranger in the *Table* class. And that's where we apply Move Method refactoring. We take the method out of the *Table* class, and implement it in the *Cell* class. Meanwhile, we have yet again applied yet another refactoring, namely Rename Method, which we discussed earlier in this article. Since we're inside *Cell* already, it

does not make much sense to refer to the entity that the method acts on in the method name, so we can just leave that out. While I was at that, I figured writing the full word “Background” might be clearer than its acronym, “BG”. We have now successfully applied a series of refactorings, and drastically changed our design. While doing this kind of larger refactorings, make sure you test your code well. Go in small steps, so the chances of breaking the whole thing are smaller. It might break at some points, but in the end better code makes it all worth the effort.

Move Parameter to Field

It is not a rule to purely stick to existing refactorings. In fact, you are encouraged to think about how you could improve the designs of your applications. Although I should say that you should be careful when you invent your own refactorings. I invented Move Parameter to Field myself, and after I did, I surfed the web and found out someone invented the exact same thing. That’s fine. In fact, that made me more sure about my conclusion. It happened when I was doing a tell-a-friend application for a members area of some website we were doing. It was nothing special, as you can see from Listing 9. Members of the website were able to tell a ‘friend’ about the website, and if they referred more than 20 friends to the website and they actually visited, the member would get a present. The part where members can get presents was added later. As I was writing the *increaseUserBonus()* and *wonGiftBox()* methods, which both took a variable *\$userid* as a parameter, I started thinking. I was writing the same parameter twice. Also, when I scrolled up, I saw that it was also a parameter of the *tell()* method. I concluded that over time the responsibility of the class grew a little bit. It had to manipulate a few things that were related to a member which was referred to by the user’s ID in the database. This meant that the application would rely much more on that ID. I then decided that the user’s ID should be available upon class instantiation, and made it a class member (also called a field), which was going to be set through the constructor, and removed the *\$userid* parameter from the methods that had them as a parameter. This shows that the responsibility of a variable might change when making changes to a class. It is then a good idea to represent that responsibility by giving it more scope and availability. Actually, the reverse might happen as well. A variable might lose some of its responsibility, and you could choose to move the variable from being a class member to a method parameter. A lot of refactorings have an opposite.

A small word of warning

While improving things is a good thing, watch out for getting obsessed with changing designs. Instead of refactoring, you

would then actually obfuscate your project by making it all too flexible, beyond requirements. Sooner or later the point of your application could become unclear. Also, perfectionism has a disadvantage, and that’s time. You should only refactor when it is needed, or likely to be needed, not otherwise. It would be a shame not to meet your deadlines because you’re still in the middle of a rather drastic refactoring process that was not really needed. However, even though a lot of managers probably would not agree, time to do code reviews and refactoring needs to be scheduled into the project deadline. It is important to think about your code and be prepared for the future. Eventually the time taken to carry out these activities will pay itself back when changes are to be made to a project in the future; they would have required much more effort if the whole project turned out to be one big messy hack. The importance of readable code should not be underestimated. If a new programmer has to dig into your code it can be a real pain for him to figure out what you’re doing when the code is unreadable.

Final words

I hope this article served as a nice introduction into the world of improving code designs. Some of the things explained earlier may sound very obvious or even exaggerated. It’s not wrong to try them out though, you’ll see that most of them actually work. Once you get used to refactoring, you may or may not notice that parts of the refactoring catalogue actually start to extend your mind. After that, refactoring may start to feel like the natural thing to do. Of course one article does not cover all the aspects and related concepts of refactoring. There are plenty of other refactorings that are worth looking at and very applicable in a PHP environment. You can even apply refactoring on databases (a resource is listed at the end of this article). We did not discuss concepts like Unit Testing and the Extreme Programming methodology, which have strong links to the refactoring concept. However, this article, and the its resources should start you off. Have fun making things better! ■

Links & Literature

- Refactoring, Martin Fowler
www.refactoring.com
- www.extremeprogramming.org
- phpunit.sourceforge.net
- www.agiledata.org/essays/databaseRefactoring.html
- Comments and Questions:
forum.php-mag.net/3/3/refactoring



**The place to be for
PHP professionals!**

Amsterdam
May 3 to 5, 2004

International PHP Conference 2004

Spring Edition

www.phpconference.com

PHP at IntelleFLEET, LLC

A case study on how PHP is used by intelLeFLEET, LLC.

by Frank M. Kromann <frank@intellefleet.com>



PHP is a well-known and commonly used server scripting language for the creation of dynamic web sites. Still many new users ask why PHP should be preferred over other technologies/languages and many also ask for references to companies who have used PHP with success. This is the story about how PHP was helpful in making a success of a small startup company located in Southern California with customers all over USA.

Introduction

Having a gauge that indicates how much gasoline there is left in the tank of the car is a great help that informs the driver when to stop to refuel. When a battery replaces the gasoline tank it becomes more difficult to tell when the “tank” is empty. One indication of an empty tank is when the vehicle stops, but then it's usually too late. Batteries used in Industrial Vehicles (lift trucks, ground support equipment etc.), as well as other rechargeable batteries, are different from an ordinary gasoline tank. After a number of recharges, the battery is unable to perform as it did when it was new. The amount of usable energy in each cycle also depends on how you use it. Most of these batteries are designed to operate for 1500-2000 cycles at 6 hours per cycle, or approx. 5 years. Discharging the battery faster than it is rated for will reduce the length of each cycle as well as the expected life of the battery. High temperature, shorted cells and low water levels are other factors that influence the life expectancy of a battery.

It is possible but often time consuming and very expensive to measure the state of health for a battery, and doing so without knowing the exact history of usage makes it difficult to estimate the remaining life. Furthermore these tests often require that the battery is taken out of production and sent to a lab for analysis. The key to optimizing performance and getting the most out of batteries is to measure how the batteries are used on an individual cycle basis. The collected data can then be analysed from the management reports that are generated.

So what does this have to do with PHP? Well, some years ago a small group of people set out to create a system that would make it possible to monitor industrial batteries and with this information create a tool that would allow the fleet manager to optimize the operation by knowing which batteries to replace next, how to schedule maintenance and how to tackle problems indicated by the acquired data. Late in the process, PHP was selected as one of the key tools for the system, and is

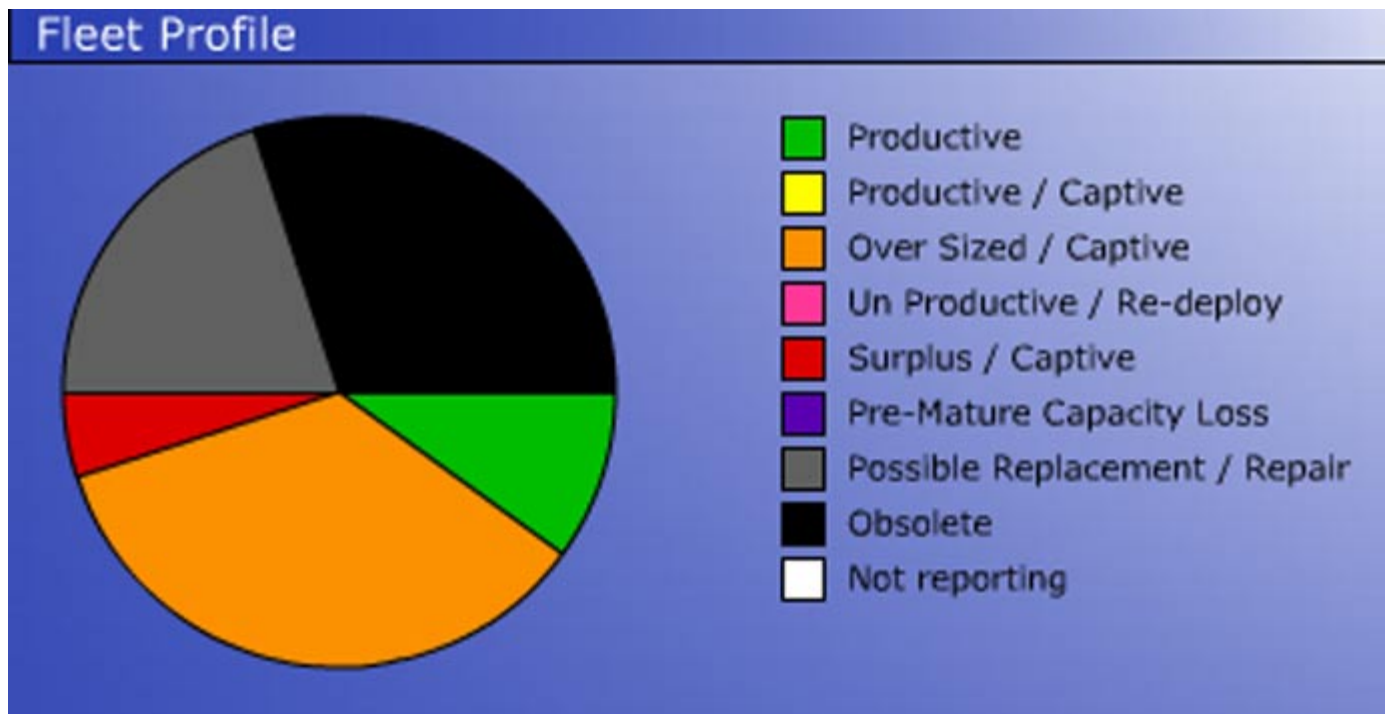


Fig. 1: Profile of the status of batteries for a fleet

now used in different areas from the web server to offline data manipulation etc.

Background

The first system was developed as a “manual” system, where a small sensor was mounted on each battery. An infrared reader was used to collect the data and transfer it to a computer for analysis. After some field-testing it was evident that the manual process of collecting data from each battery (several hundred in some cases) was too time consuming and it was difficult to keep track of the batteries.

A new version of the system was developed. It was based on radio frequency (RF) technology with no humans involved in the data acquisition process. After a few modifications to the hardware, the system was ready for beta testing in a real production environment. At this point there were no tools for data manipulation or reporting, but it became clear that these tools needed to be Internet based. This would minimize the requirements for maintaining and installing software on the client's networks, and it would make the development and support processes much easier to manage.

Many of the potential customers/users of a system like this have multiple locations from which they operate fleets of industrial vehicles and many of them have a need for a centralized monitoring/reporting system. Having a web based solution makes it possible for the end users to access the data at any time

(up to date information) and eliminates the process of preparing and mailing printed reports, as done in other fleet management systems in this industry. Not knowing exactly how the tools were going to work, how many users (hits per day) and what type of hardware/OS the system should operate on, PHP was an easy choice for the scripting language behind the web server. PHP is known to run on many different platforms, integrate with a large number of database servers as well as other tools and it is fast and scalable.

The database server, FrontBase, was selected after evaluating other products. MySQL was a cheap option, but at the time when the development started it was not able to perform complex transactions and it did not support views or sub selects. Oracle's licensing fees made its solution too expensive. Microsoft SQL Server was only available for the Windows NT platform. One other key factor in the selection of FrontBase was the time needed to get the systems back online after a power outage. As soon as the power is back up the database server can be started, and with most database systems any unfinished transactions would be lost.

The System

Each battery is equipped with a Data Collection Module (DCM) and a Temperature Sending Unit (TSU). Both modules use RF technology to communicate with a Base Station (BS), a small computer with an embedded Linux system and a trans-

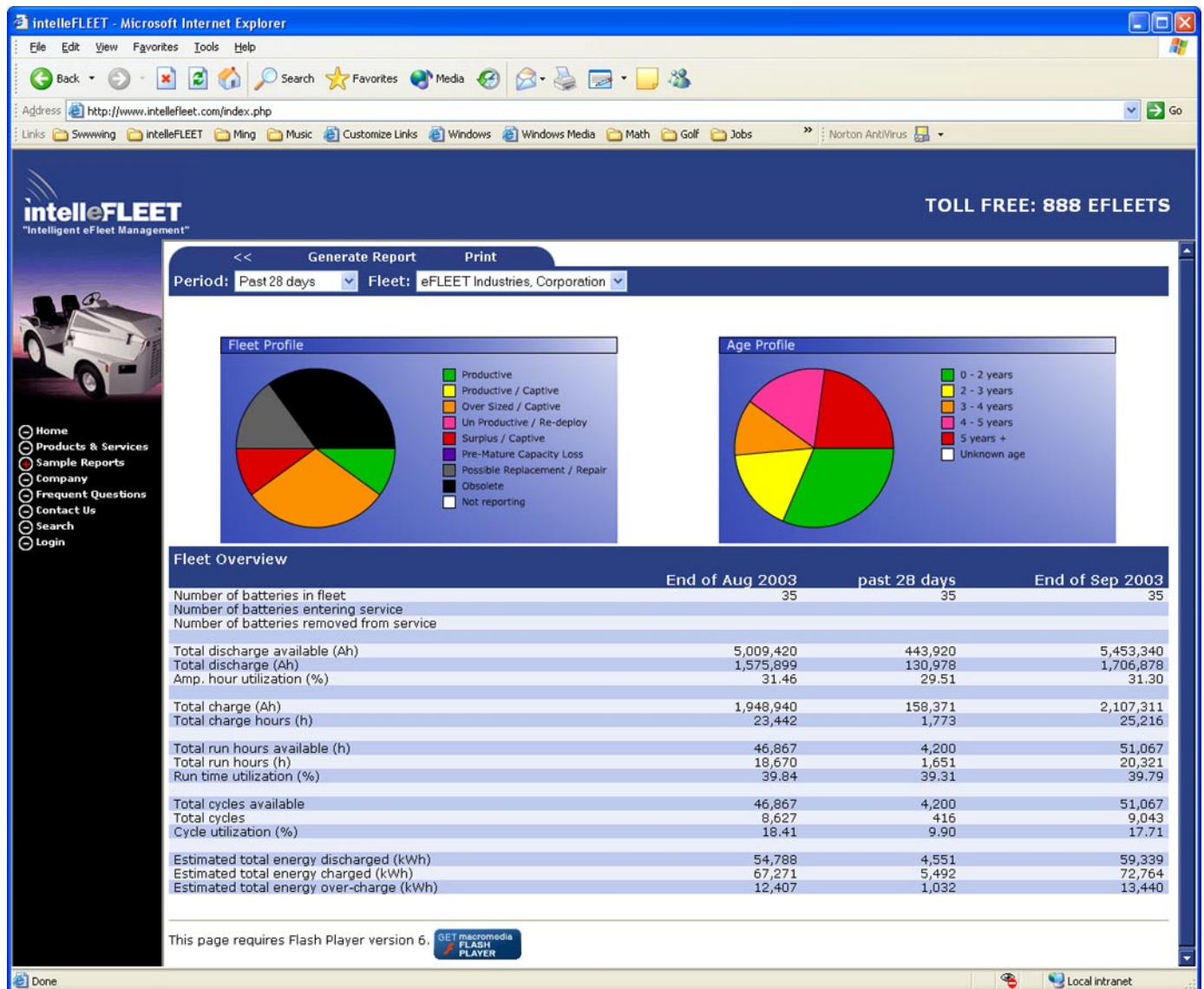


Fig. 2: Overview of fleet performance

ceiver connected to the serial port. The DCM and TSU are mounted on the battery in a non-invasive way such that it allows for reuse of the units on other batteries and it does not interfere with the operation of the battery or the equipment. Other monitoring systems require replacement of inner cell connectors (this will void any warranty) and use external wires to power up the equipment that would make the operation of the battery more difficult.

The DCM is mounted on the battery cables and collects data relating to Voltage, Amperage, Cable Temperature, Charge Time and Discharge time. These values are accumulated in the DCM and when the DCM is close enough to the Base Station it will upload the data. The Base Station is usually mounted in the charging area, where the batteries will

spend 6-10 hours on charge at the end of each cycle, close enough to upload data.

The TSU is mounted on the centre cell of the battery where it measures the temperature and transmits it to the DCM for inclusion in the data packet.

The Base Station will upload data from all the DCMs in the fleet (typical 50-500 batteries/DCMs per location), to a central computer, one or more times each day. Each package contains information about each DCM in the fleet, but no battery or customer information is included in this communication.

The data from each Base Station is loaded into a database, where individual battery data is extracted and performance data is calculated. This is the first step where PHP plays a role in

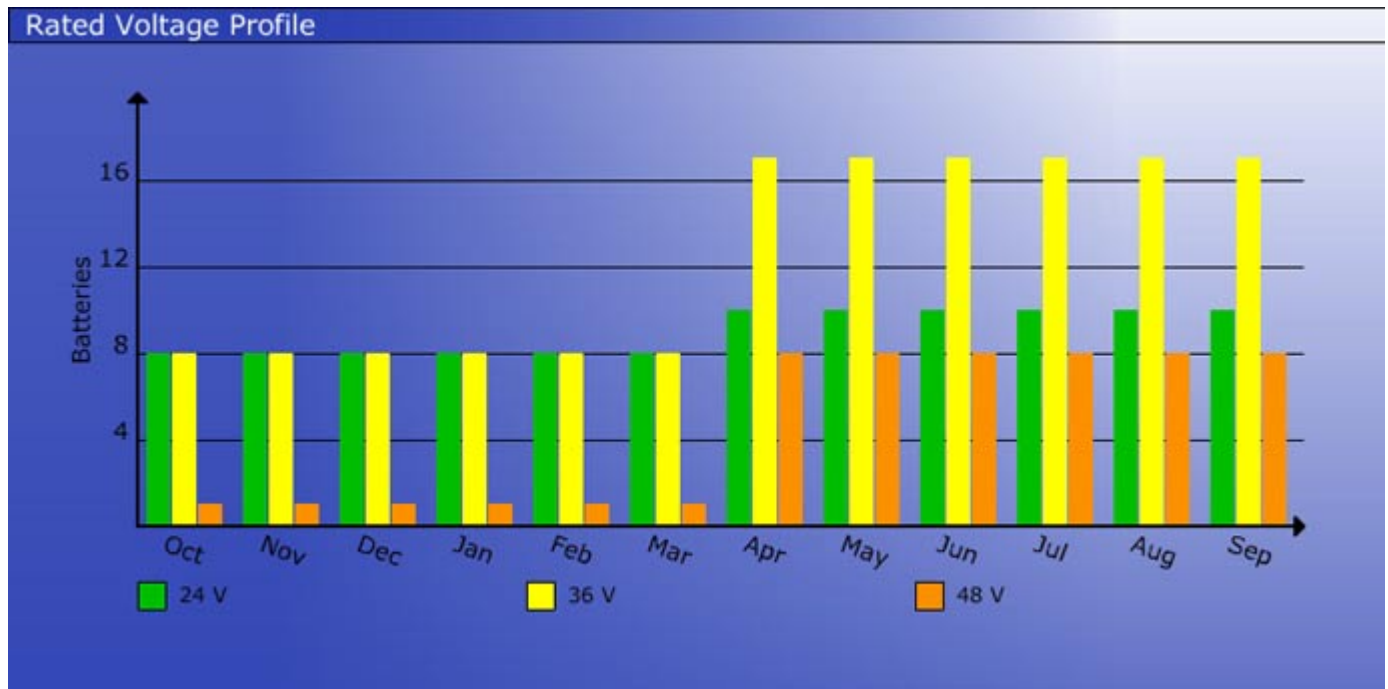


Fig 3: Overview of a fleet of batteries

the data acquisition process. A PHP script is activated by the cron demon and the data files are parsed. The collected data for each battery are paired with enrolment information in the database. If no enrolment information can be found the process will create the missing battery records with default values. In order to reduce the query time on reports requested by the users, several extra data fields are calculated and stored in the database.

During the data processing the system looks for “out of range” values for cable temperature, voltage and other parameters. If a high cable temperature is detected the system will force an email to the client informing them about the problem. A high temperature on the cable can be caused by a bad or broken connector, and with a battery capable of delivering several hundreds Amperes this can cause spikes that might lead to explosions and fires.

The web site provides a single GUI that allows the user to perform different tasks, depending on the access rights granted to the user:

- An anonymous user can browse all the public pages.
- An end-user can generate reports and view battery information for a single location or for selected or all locations within the corporation
- A client administrator can add, edit and delete user accounts and add enrolment information to new batteries. This is a simple way of moving some of the support tasks to the customer.

- A content manager can add, edit and delete content as articles or whole pages, without any knowledge of HTML or databases.
- A system administrator can manage data in lookup tables as well as the basic structure of companies, regions and locations etc.
- A developer can manage the underlying data model, without the need to access other tools.

The entire web site and the database model were designed with a few simple rules in mind. The content managers should be able to add and edit content without any knowledge of databases or HTML. Individual battery information (performance data etc.) should never be more than 3 mouse clicks away, after the user logs on. All reports are generated on request. This includes the charts (created in Flash with the Ming extension for PHP).

The data model supports access to battery information through a hierarchic structure. This allows the customers to assign user access to a single location, a region or the whole fleet.

Potential customers, and anonymous users in general, can follow the performance of a virtual fleet of batteries. This is made possible by the selection of a sample of real batteries. These batteries are renamed so it is impossible to see where they are operating, but all fleet reports are available on the web site.

Offline Tools

When a system is based on web technology and with a constantly growing and changing database there is a need for a tool to create a snapshot of the web site, with all the reports. PHP is used to create static HTML and image files. With all the links created for drill down features this ends up being more than 1200 files. These files can be copied to a CD or a notebook computer and used with a standard browser. There is no need for a web server, database server or PHP on the notebook. This tool has been very useful for sales presentations, where online access to the Internet was impossible.

For testing and data evaluation we have developed a PHP-GTK application, which allows the user to connect to the base station through a network connection and fetch the report file on request. This application uses the PHP socket extension to communicate with the base station and it parses the data file from encoded text to human readable values. PHP-GTK is an easy way to create GUI applications, which can be used by users who are more used to graphical than command line environments.

Future Development

The single most requested feature, from customers and prospects, to the system has been a “which battery next” feature. In a fleet operation, without the intelFLEET system, with 50-100 trucks and an average of 2-3 batteries per truck the amount of batteries on charge, at any given time, will make it nearly impossible to keep track of start and end of charge times as well as cooling periods. This causes the selection of undercharged batteries that will fail after a few hours and a trip back to the charging area is needed to replace the battery. Operating a battery right after the end of charging, will reduce the expected life, and in many cases void warrant.

With an automated monitoring system it is possible to use the acquired data to rank the batteries so the operator will know which battery to take next. This process will reduce the amount of time it takes change the battery in the truck (no guessing needed) and it will make it possible to optimize the usage of each battery and thereby increase the life. Getting one year more out of each battery makes a huge difference to the bottom line.

Standard Internet based web technology would in many cases not be usable for a solution like this. It would require a permanent Internet connection for a computer mounted in the charging area. Most companies do not allow this (because it would give employees permanent internet access). In order for the selection system to work in an optimal way, the data need to be acquired more than once per day, most likely every 5-10 min. The optimal solution would be to install a computer with

a touch screen, and have it equipped with a database server, a battery selection application and a network connection to the base station where data is collected.

The use of a web server and a browser for the battery selection application makes it possible to employ PHP technology, and reuse some of the code (data parsing etc.) developed for the web site, but the entire GUI would need a redesign, as this computer would not include a keyboard or a mouse.

It is expected that this new application will be ready for testing within a few weeks and ready for deployment before the end of the year 2003.

Data Sizes

As indicated in the beginning, intelFLEET is a small startup company. Currently the system is used to monitor approx 700 Batteries from 23 locations. The system has collected more than 450,000 data records and the database is growing with 1000-1500 new records per day.

Web statistics

The web site currently has a load of about 15,000-20,000 page impressions per month, excluding all internal usage. The Apache log files are parsed every month (this might change to every week in the future), and all the IP addresses are analysed in the database. With the use of reverse IP lookup and tools like GeoIP (from MaxMind) it is possible to make a good guess of country, state, city and organization for most IP addresses. This information is very useful for sales and marketing.

Conclusion

PHP and other open source technologies, as well as traditional closed source technologies, has made it possible for a small startup company like intelFLEET to create an advanced set of online web applications to be used by a growing number of users to analyse data about the usage of industrial batteries. PHP has provided the flexibility and tool integration needed to provide customers and end users with a “state of the art” monitoring system. The system will enable the users to improve performance and reduce the cost of operating a fleet of industrial electrical vehicles. ■

Links & Literature

- intellefleet.com
- www.frontbase.com
- maxmind.com
- Comments & Questions: forum.php-mag.net

Advertising Index

Global Alliance Program www.php-mag.net/gap	page 38	International PHP Magazine/News www.php-mag.net	page 09
International PHP Conference 2004 Spring Edition www.php-conference.com	page 54	New York PHP www.nyphp.com	pages 24/38
International PHP Magazine/Forum forum.php-mag.net	pages 24/45		

Imprint

PHP Magazine is published by
Software & Support Verlag GmbH

Address:
PHP Magazine
Software & Support Verlag GmbH
Kennedyallee 87
D-60596 Frankfurt am Main
Phone: +49 (0) 69 63 00 89 0
Fax: +49 (0) 69 63 00 89 89
eMail: editors@php-mag.net
www.php-mag.net

Editor: Indu Britto
eMail: ibritto@php-mag.net

Layout: Tobias Friedberg

Authors of this issue:

Ilia Alshanetsky, Leendert Brouwer, Frank M. Kromann,
Björn Schotte, Damien Seguy, Davey Shafik, Chris Shiflett,
Zeev Suraski

Advertising:

Software & Support Verlag GmbH
Kennedyallee 87
D-60596 Frankfurt am Main
Phone: +49 (0) 69 63 00 89 0
Fax: +49 (0) 69 63 00 89 89
eMail: editors@php-mag.net
www.php-mag.net

Subscription Service:

www.php-mag.net

© Copyright 2003

Software & Support Verlag GmbH
All rights reserved. No part of this publication may be reproduced in any form without the prior consent of the copyright holder.

While all reasonable attempts are made to ensure accuracy, Software & Support Verlag disclaims any liability whatsoever for any use of code or other information herein.

All trademarks and brands are usually registered trademarks of companies and organisations.