PHP Top 5

From OWASP

PHP is a very popular language. Every PHP developer and hoster should understand the primary attack vectors being used by attackers against PHP applications.

This article is the underlying research behind the SANS Top 20 2005's PHP section (http://www.sans.org/top20/#c3). The methodology used in the preparation of this article is to review all Bugtraq postings containing the word "PHP" and categorize each unique flaw. The author analyzed the most popular flaws / attacks, and researched prevention techniques, resulting in this article.

If you need more information on how to write solid, secure PHP, please consult the references.

Special notes for this edition

About "safe_mode"

Safe Mode is a set of controls, which when turned on:

- forces PHP to test for UID permission before opening files. You can relax this to GID by enabling safe_mode_gid, but this usually implies access as if safe mode was disabled)
- prevents system() and other calls from working (unless safe_mode_exec_dir is set)
- restrictions on setting most environment variables (but not reading them)
- open_basedir allows hosters to force file access to stay within a virtual directory, but otherwise it is not set

However, there are many ways around safe_mode. Some legitimate programs contain these methods. Almost all PHP exploits use such methods to get around safe_mode restrictions.

Hosters, do not just switch safe_mode on and lock it down hard. Such controls rarely work as expected, and more to the point it does not prevent any of the five major attack vectors presented in this paper. Only code reviews and ensuring that code is tested for security flaws can the risk of attack be reduced.

This is not to say that safe_mode is useless. Well designed software can use thoughtfully chosen safe_mode restrictions to improve defense in depth. Such software should be given that opportunity by using .htaccess (or similar mechanisms) to selectively enable safe_mode restrictions as they need.

Safe Mode has been removed in PHP 6.0.

addslashes() / magic_quotes

An earlier version of this article recommended the use of addslashes(). In general, this is poor advice today, particularly when PHP is coupled with the most popular open source database, MySQL.

To prevent SQL injections, it is essential that:

- magic_quotes_gpc is disabled in all PHP installations
- addslashes() should be deprecated - it does not protect against SQL injections
- Software should move post-haste to support safe(r) database access mechanisms, such as PDO or mysql_real_escape_string()

PDO requires hosters to upgrade their PHP to the latest PHP 5.1. Unfortunately, there has been low take up of the safer versions of PHP due to perceived incompatibilities with much of the PHP software base. Hosters should move to the latest PHP as quickly as possible to encourage the adoption of safer software.

P1: Remote Code Execution

Description

The most widespread PHP security issue since July 2004 is remote code execution, mostly via file system calls.

The root causes of this issue are:

- Insufficient validation of user input prior to dynamic file system calls, such as require or include or fopen()
- allow_url_fopen and PHP wrappers allow this behavior by default, which is unnecessary for most applications
- Poor permissions and planning by many hosters allowing excessive default privileges and wide ranging access to what should be off limits areas.

From PHP 4.0.4 onwards, allow_url_fopen was enabled by default, making poorly written applications vulnerable through no changes of their own. As of PHP 4.3.4, the PHP project changed the access on this feature to PHP_INI_SYSTEM, which prevents code from easily disabling this feature via the use of ini_set(). This has the unfortunate effect that only hosters can change this setting, many of whom cannot make wholesale changes to php.ini without breaking their customers code, including well-written code which relies upon these features.

Operating Systems Affected

- PHP 4 (after PHP 4.0.4), 5.x

CVE/CAN Entries

There are more than 100 such vulnerabilities reported since July 30, 2004. These are a representative sample:

phpBB Remote Code Execution Vulnerability http://www.securityfocus.com/bid/14086

TikiWiki Remote Code Execution Vulnerabilities http://www.securityfocus.com/bid/12328

XML-RPC Remote Code Execution (many vendors) http://www.securityfocus.com/bid/14088

How to Determine if you are Vulnerable

Inspect your code for constructs like:

```
$report = $_POST[‘report_name’];
include $report;
```

or

```
$username = $_POST[‘username’];
eval(“echo $username”);
```

The above snippets are not exhaustive. Other code constructs to look for include:

- fopen(), fsockopen()
- Direct command execution - popen(), system(), ` (backtick operator). Allows remote attackers to execute code on the system without necessarily introducing remote code.
- Direct PHP code execution via eval()
- Limited evaluation if the attacker supplied PHP code is then used within double quotes in the application code – most useful as an information disclosure
- include, include_once, require, require_once with dynamic inputs
- file_get_contents()
- imagecreatefromXXX()
- mkdir(), unlink() and rmdir() and so on - PHP 5.0 and later has limited support for some URL wrappers for almost all file functions

How to Protect Against It

Developers should

- Review existing code for file operations, include/require, and eval() statements to ensure that user input is properly validated prior to first use

- When writing new code, try to limit the use of dynamic inputs from users to vulnerable functions either directly or via wrappers

Hosters should:

- Disable allow_url_fopen in php.ini by setting it to 0
- Enable safe_mode and set open_basedir restrictions (if you know what you're doing - it's not really that safe!)
- Lockdown the server environment to prevent the server from making new outbound requests

OWASP calls on the PHP Project to by default disable remote file support and associated wrappers, and allow applications that require these features to selectively enable them on a per application basis.

References

- OWASP, OWASP Guide (http://www.owasp.org/index.php/Category:OWASP_Guide_Project), © 2005 OWASP Project

Books

- Alshanetsky, I., php|architect's Guide to PHP Security, (C) 2005 Marco Tabini & Associates, Inc ISBN 0973862106
- Shiflett, C., Essential PHP Security, © October 2005, O'Reilly ISBN 059600656X

Web sites

- http://php.net/features.safe-mode
- http://php.net/features.remote-files.php

P2: Cross-site scripting

Description

Cross-site scripting (also known as HTML injection or user agent injection) with PHP is possible in all three modes: reflected, persistent and DOM injection.

- Reflected The attacker provides a link or other payload containing embedded malicious content, which the application immediately displays back to the victim. This is the primary form of phishing via e-mail (such as eBay scams, bank scams, etc)

- Persistent The attacker stores malicious content within a database, which is then exposed to victims at a later time. This is the most common form of cross-site scripting attack against forum and web mail software.

- DOM The attacker uses the victim site’s JavaScript code to perform reflected cross-site scripting. This technique is not widely used as yet, but it is just as devastating as any form of cross-site scripting.

PHP has no inbuilt mechanism to protect against cross-site scripting in an automated fashion, therefore unless developers remediate this issue, an application will be vulnerable by default.

Operating Systems Affected

All versions of PHP

CVE/CAN Entries

There are more than 100 XSS CVE / CAN entries since July 2004. The following are representative samples:

VBulletin Cross-site scripting http://www.securityfocus.com/bid/14874

Coppermine Display Image Cross-site scripting http://www.securityfocus.com/bid/14625

WordPress Edit Cross-site Scripting http://www.securityfocus.com/bid/13664

How to Determine if you are Vulnerable

Developers

- Does the application rely upon register_globals to work? If so, your application is at a slightly higher risk, particularly if you do not validate input correctly.
- Inspect user input handling code for unsafe inputs:

```
echo $_POST[‘input’];
```

At the very least, the code should do this:

```
echo htmlentities($_POST[‘input’], ENT_QUOTES, ‘UTF-8’);
```

The best code will inspect user input for type, length, and syntax. Chris Shiflett recommends validation code like this:

```
$html = array();

$html['username'] = htmlentities($clean['username'], ENT_QUOTES, 'UTF-8');

echo "<p>Welcome back, {$html['username']}.</p>";
```

For more details on Shiflett’s approach, please refer to the references below. As every application has different input fields, is up to every application to do this properly – there is no automatic way of doing this task.

If you rely upon stored data that may potentially be tainted, your output code should protect users regardless of if you currently filter new input for HTML entities:

```
echo htmlentities($output);
```

Code that performs basic black listing is particularly prone to attack:

```
$input = str_replace(“<”, “&lt;”, $input);
$input = str_replace(“>”, “&gt;”, $input);
$input = str_replace(“script”, “”, $input);
```

Such code should be refactored to use HTML entities and white listing approaches, for example, the simpler and safer:

```
$input = htmlentities($input, ENT_QUOTES, ‘UTF-8’);
```

If you use Javascript to redirect the user (via document.location or window.open any similar means), output to the user via document.write, or modifies the DOM in any way, you are likely to be at risk of DOM injection.

How to Protect Against It

Developers should:

- Turn off register_globals and ensure all variables are properly initialized
- Obtain user input directly from the correct location ($_POST, $_GET, etc) rather than relying on register_globals or the request object ($_REQUEST)
- Validate input properly for type, length, and syntax
- Free text input can only be safely re-displayed to the user after using HTML entities
- Variables sent back to the user via URLs must be URL encoded using urlencode(), although the use of GET requests is deprecated for anything besides navigation purposes:

```
$html = array();
$url = array();

$url['username'] = urlencode($clean['username']);

$link = "http://example.org/index.php?username={$url['username']}";

$html['link'] = htmlentities($link, ENT_QUOTES, 'UTF-8');

echo "<a href=\"{$html['link']}\">Link</a>";
```

- Validate JavaScript code against Klein's DOM Injection paper (see references) to ensure that they are immune from DOM injection attacks

At this time, OWASP recommends all applications move to directly accessing only those variables they require from the correct user input array ($_POST, $_GET, $_COOKIE, etc) rather than rely upon the get, post, cookie (GPC) behavior of register globals or $_REQUEST. It is strongly recommended you do not use $_REQUEST.

Hosters are unable to configure PHP to be cross-site scripting safe, so should consider removing applications that have a history of being vulnerable to XSS attacks. This will force developers to improve their XSS protections.

References

- OWASP, OWASP Guide (http://www.owasp.org/index.php/Category:OWASP_Guide_Project), © 2005 OWASP Project

Books

- Alshanetsky, I., php|architect's Guide to PHP Security, (C) 2005 Marco Tabini & Associates, Inc ISBN 0973862106
- Shiflett, C., Essential PHP Security, © October 2005, O'Reilly ISBN 059600656X

Web sites

- PHP htmlentities():http://php.net/htmlentities
- PHP urlencode(): http://php.net/urlencode
- Klein, Amit, Cross-site Scripting Explained, http://crypto.stanford.edu/cs155/CSS.pdf
- Klein, Amit, Cross-site Scripting of the Third Kind, http://www.webappsec.org/projects/articles/071105.shtml
- Shiflett, C., Ideology, http://shiflett.org/articles/security-corner-nov2004

P3: SQL Injection

Description

SQL injection is one of the oldest attacks against web applications. However, as it is well known, there are many techniques to defend against it:

- Validating data prior to using it within dynamic SQL queries
- Always prefer positive validation rather than black listing
- Using PDO (available via PECL for PHP 5.0, and is included in PHP 5.1 and later)
- Using MySQLi's or PEAR::DB's parameterized statements
- At the very least, use functions like mysql_real_escape_string(). All PHP database interfaces have basic SQL escaping functions

Unfortunately, with PHP 4.x, it is up to the PHP coder to write robust code that is safe against SQL injections. At this time, PHP programs should be migrating to PHP 5.1 and using PDO, which has a safe SQL interface which avoids all of these issues.

Using addslashes() is simply insufficient. As Chris Shiflett has demonstrated (http://shiflett.org/archive/184), addslashes() is simply insufficient to protect against SQL injection. It is the OWASP Project's view that addslashes() must be deprecated in favor of safe interfaces.

There is some dissent in the PHP non-security community about the default behavior of PHP's magic quotes, which "magically" adds slashes (not quotes) to input data on the basis that it might be destined for a database. As this behavior is not always available, code must be written with the assumption it will not be configured. OWASP's issue with magic quotes is that it gives a false sense of security – it is not a silver bullet and it is insufficient to protect against advanced SQL injection techniques.

Operating Systems Affected

All versions of PHP

CVE/CAN Entries

As there are more than 100 CVE / CAN entries from multiple vendors (in fact, Bugtraq usually offers up two to three different PHP applications with SQL injection vulnerabilities per day), this is a representative sample only:

PHP Nuke SQL Injection http://www.securityfocus.com/bid/9544

OS Commerce SQL Injection http://www.securityfocus.com/bid/15023

VBulletin SQL Injection http://www.securityfocus.com/bid/14872

How to Determine if you are Vulnerable

Find code which calls mysql_query() or similar database interfaces Inspect if any calls create dynamic queries using user input:

```
if($doublee == "off" && strstr($email, "@")) {
$email = trim($email);
$email1 = ", email";
$email2 = "OR email='$email'";
}

$username = trim($username);
$query = $db->query("SELECT username$email1 FROM $table_members WHERE username='$username' $email2");
```

(Code from XMB 1.8, where $db->query() is essentially a wrapped mysql_query($conn, $sql))

Such code is vulnerable to attack due to the lack of escaping.

Stefan Esser points out (http://blog.php-security.org/archives/35-Teaching-the-Teachers.html) that using user-supplied data for the table name (in the above example, $table_members, is for all intents and purposes unsecurable, and should not be used, as mysql_escape_realstring() or $mysqli->escape_string() and other escaping mechanisms do not expect to be dealing with data prior to the WHERE clause, only WHERE, ORDER BY, etc data enclosed by quotes. Therefore, we do not recommend the use of dynamic table names if it can be avoided.

How to Protect Against It

Developers should:

- Migrate code to PHP 5.1 and use PDO, or if this is not possible, at least migrate code to safer constructs, such as PEAR::DB's parameterized statements or the MySQLi interfaces:

Using mysql_real_escape_string()

UltimaBB, XMB's descendant, wraps mysql_real_escape_string() in $db->escape() as it has a data factory which will eventually support other databases than MySQL. formVar() forces the data to come from the $_POST array, which reduces the attack surface area.

```
$regusername = $db->escape(formVar('regusername'));
$regemail = $db->escape(formVar('regemail'));

if (empty($regusername))
{
    cp_error($lang['regempty'], false, '', '</td></tr></table>');
}

if ($SETTINGS['doublee'] == 'off' && false !== strpos($regemail, "@"))
{
    $email1 = ", email";
    $email2 = "OR email='$regemail'";
}
else
{
    $email1 = $email2 = '';
}

if (preg_match("/[\]\['".'",!@#~$%\^&*()+=\/\\\\:;?|.<>{}]/', $regusername))
{
    cp_error($lang['badusername'], false, '', '</td></tr></table>');
}

$query = $db->query("SELECT username$email1 FROM ".X_PREFIX."members WHERE username = '$regusername' $email2");
$usercheck = $db->num_rows($query);
$db->free_result($query);
```

(From current CVS UltimaBB 1.0, cp_reguser.php)

If this code base was converted from its own data layer to PDO, this is what it might look like:

```
$regusername = $db->escape(formVar('regusername'));
$regemail = $db->escape(formVar('regemail'));

if (empty($regusername))
{
    cp_error($lang['regempty'], false, '', '</td></tr></table>');
}

if (preg_match("/[\]\['".'",!@#~$%\^&*()+=\/\\\\:;?|.<>{}]/', $regusername))
{
    cp_error($lang['badusername'], false, '', '</td></tr></table>');
}

$sql = '';
if ($SETTINGS['doublee'] == 'off' && false !== strpos($regemail, "@"))
{
```

```
        $sql = 'SELECT username, email FROM '.X_PREFIX.'members WHERE username = ":regusername" OR email=":regemail"';
        $stmt = $db->prepare($sql);
        $stmt-> bindParam(':regemail', $regemail, PDO::PARAM_STR, 32);
    }
    else
    {
        $sql = 'SELECT username FROM '.X_PREFIX.'members WHERE username = "':regusername'"';
        $stmt = $db->prepare($sql);
    }
    $stmt->bindParam(':regusername', $regusername, PDO::PARAM_STR, 32);
    $stmt->execute();
    $usercheck = $stmt-> rowCount();
```

- Validate data for correct type, length, and syntax.
- (From Stefan Esser) Do not use dynamic table names - escape functions are not designed for this use and are not safe for this use.
- Always prefer white listing (positive validation) data over black listing, which is akin to virus patterns – always out of date, and always insufficient against advanced attacks
- As a last resort, code should be using mysql_real_escape_string() (but not addslashes() which is insufficient). This provides limited protection to simple SQL injections, but is the absolute minimum required for all applications trying to use the native database interfaces.
- Provide a .htaccess file to ensure that register_globals and magic_quotes are forced off, and that all variables are properly initialized and validated

Developers should carefully test their applications against SQL injection prior to release, particularly those SQL statements that consume user data. A good tool for this is WebScarab that tests for basic SQL injections in a relatively automated fashion.

Hosters cannot prevent SQL injection via technical means, but they can reduce their exposure by using PHP 5.1, and providing PDO to their clients. They should carefully configure MySQL or their favorite database in the most secure fashion possible. Users should not be granted administrative privileges over their databases if at all possible, and the database should be running in a chrooted environment to minimize damage from any successful attacks.

References

- OWASP, OWASP Guide (http://www.owasp.org/index.php/Category:OWASP_Guide_Project), © 2005 OWASP Project
- Category:OWASP_WebScarab_Project

Books

- Alshanetsky, I., php|architect's Guide to PHP Security, (C) 2005 Marco Tabini & Associates, Inc ISBN 0973862106
- Shiflett, C., Essential PHP Security, © October 2005, O'Reilly ISBN 059600656X

Web sites

- PEAR::DB, http://pear.php.net/package/DB
- Shiflett, C., SQL Injection, http://shiflett.org/articles/security-corner-apr2004

P4: PHP Configuration

Description

PHP Configuration has a direct bearing on the severity of attacks. Although no particular CVE entries are found against configuration, poor configuration choices maximize the attacker’s advantage and damage they can cause poorly configured systems. What is worse, many “security” options in PHP are set incorrectly by default and give a false sense of security.

It is surprising that there is no agreed "secure" PHP configuration, and even more surprising that this is not how PHP is configured by default. There are arguments for and against the most common security options:

- register_globals (off by default in modern PHP, should be off)
- allow_url_fopen (enabled by default, should be off)
- magic_quotes_gpc (on by default in modern PHP, should be off)
- magic_quotes_runtime (off by default in modern PHP, should be off)
- safe_mode and open_basedir (disabled by default, should be enabled and correctly configured. Be aware that safe_mode really isn't safe and can be worse than useless)

Due to disagreement on the best settings, and the PHP Project's preference to enable features over security, all but a very few PHP installations are properly secured.

OWASP strongly recommends the PHP Project coordinate with acknowledged PHP security professionals and the web hosting community to come up with a secure default configuration, even at the expense of backward compatibility.

Operating Systems Affected

All versions of PHP

CVE/CAN Entries

As this relates to post deployment, there are few CVE / CAN references, however, the severity of attacks is far worse than a properly secured PHP solution.

For example, if remote code execution were blocked by the simple expedient of it being disabled, a vulnerable system would not be compromised. If "helpers" such as register_globals and magic_quotes_gpc were always disabled, developers would have to practice good hygiene to simply obtain their data.

How to Determine if you are Vulnerable

Unless the PHP installation has been hardened by a security professional, it is highly likely that all configurations are sub-optimal.

How to Protect Against It

Developers

- Upgrade applications to use PHP 5. PHP 5's slightly stricter conformance requirements help eliminate many latent PHP security issues
- Configure a .htaccess file – most PHP hosting is via Apache, and setting appropriate PHP variables to suit you reduces the guess work. In particular, disable register globals and magic_quotes_gpc
- During installation, test using ini_get() for common hosting mistakes, such as allowing register_globals and warn the user that the hoster has sub-standard security

Hosters

- Hosters should be upgrading to PHP 5 now and assisting their clients with the move

PHP Project

- PHP Project to coordinate with professional security organizations such as SANS or OWASP, acknowledged PHP security experts, and web hosting companies to produce a "safe" out of the box configuration

- PHP Project to always allow applications to disable risky functionality they do not need that may be available (either by mis-configuration or by design).

References

- OWASP, OWASP Guide (http://www.owasp.org/index.php/Category:OWASP_Guide_Project), © 2005 OWASP Project

Books

- Alshanetsky, I., php|architect's Guide to PHP Security, (C) 2005 Marco Tabini & Associates, Inc ISBN 0973862106
- Shiflett, C., Essential PHP Security, © October 2005, O'Reilly ISBN 059600656X

Web sites

- Hardened Security Project, http://www.hardened-php.net/
- Making PHP code portable, http://www.mt-dev.com/2002/09/make-your-php-code-portable/

P5: File system attacks

Description

PHP developers have many ways to obviate security on shared hosts with local file system attacks, particularly in shared environments:

- Local file inclusion (such as /etc/passwd, configuration files, or logs)
- Local session tampering (which is usually in /tmp)
- Local file upload injection (usually part of image attachment handling)

As most hosters run PHP as “nobody” under Apache, local file system vulnerabilities affect all users within a single host.

Operating Systems Affected

PHP 3, 4, 5

CVE/CAN Entries

As there have been many examples over the last year, the following are representative examples only:

MyPHPAdmin Local File Inclusion http://www.securityfocus.com/bid/15053

MyPHPFAQ Logs Unauthorized Access Vulnerability http://www.securityfocus.com/bid/14930

MAXdev MD-Pro Arbitrary Remote File Upload Vulnerability http://www.securityfocus.com/bid/14750

How to Determine if you are Vulnerable

Inspect all file related functionality and determine

- If user input is being used as part of the filenames
- If variables involved in the file operations are not initialized prior to first use and register_globals may be available

If so, your application is at risk.

How to Protect Against It

Developers

- Ensure that all variables are properly initialized prior to first use
- Ensure that the users can only affect file operations to the degree you had in mind
- Try to move secrets and logs out of the web root if at all possible – see the references on "Shared Hosting" by Chris Shiflett
- Ensure that your scripts are compatible with safe mode restrictions and will work under suPHP or other user execution wrapper
- Session tampering can be hindered by setting your scripts to use a non-default session dir. Use session_save_path() (http://no2.php.net/session_save_path) or set session_save_path (http://no2.php.net/manual/en/ref.session.php#ini.session.save-path) in a .htaccess file

Hosters

- Ensure your environment is hardened (such as using SELinux or similar hardened environments)
- Move PHP's session variables from the default location and secure the new location to prevent cross-user session disclosure and tampering (this is not easy)
- Enable safe_mode as appropriate (there are many options :-) but be aware that safe_mode really isn't safe and can be worse than useless)
- Use open_basedir (http://no2.php.net/manual/en/features.safe-mode.php#ini.open-basedir) restrictions
- Ensure that users have a place outside the web root (public_html) to store logs and secrets
- Run PHP under a least privilege model, preferably as the user, via the use of PHPsuExec, php_suexec or suPHP

References

- OWASP, OWASP Guide (http://www.owasp.org/index.php/Category:OWASP_Guide_Project), © 2005 OWASP Project

Books

- Alshanetsky, I., php|architect's Guide to PHP Security, (C) 2005 Marco Tabini & Associates, Inc ISBN 0973862106
- Shiflett, C., Essential PHP Security, © October 2005, O'Reilly ISBN 059600656X

Web sites

- suPHP: http://www.suphp.org
- PHP Security Consortium, http://phpsec.org/projects/guide/5.html#5.1
- Shiflett, C., Shared Hosting, http://shiflett.org/articles/security-corner-mar2004

About the reviewers

The author, Andrew van der Stock, had the article peer reviewed by:

- Chris Shiflett, 14th October, peer review received 15th October. Most changes incorporated. Errors remaining are mine.
- OWASP Top 10 mail list, 13th October, no peer review received
- Amit Klein (only the XSS section), 13th of October 2005. Updated XSS section to include DOM injection.

Stefan Esser provided feedback via his blog (http://blog.php-security.org/archives/35-Teaching-the-Teachers.html), which has been incorporated 12 July 2006.

OWASP welcomes peer review and constructive criticism for all its materials. If you wish to provide feedback, please e-mail the author Andrew van der Stock.

Vanderaj 00:15, 25 June 2006 (EDT)

Retrieved from "https://www.owasp.org/index.php?title=PHP_Top_5&oldid=206970"

Category: PHP
