Apress™
Books for Professionals by Professionals

Chapter Sixteen: “Security”

A Programmer’s Introduction to PHP 4.0

by William Jason Gilmore
ISBN # 1-893115-85-2

info@apress.com

CHAPTER 16

Security

"Non sum qualis eram." ("I am not as I used to be.")
—Horace

When I happened across this quotation from Horace some time ago, I thought it so fittingly described the true essence of network security that I tucked it into the depths of my harddrive in hopes of being able to later use it. Of course, many of you are scratching your heads wondering what Horace, the ancient Roman poet, could possibly have to say that could be related to network security. In fact, network security is one of those subjects that spews forth a never-ending amount of information and is always changing to the tune of emerging technology. Thus, it is never what it used to be. You can never rely solely on what you already know about the subject, as it became most likely outdated the moment it hit the mainstream information market or is soon doomed to become so. The only way to feel the sense of being *relatively* secure in building reliable server-based applications is either to constantly stay abreast of the latest developments regarding the subject or to hire a reliable third party capable of effectively handling the problem for you.

Security considerations as applicable to PHP take many faces, some of which tie into the security of the server itself. After all, the degree of vulnerability built into the server is paramount in many ways to determining that of the data handled by the PHP scripts I strongly suggest that you read as much as you can about your Web server and be on the watch for upgrades and recommended fixes. Provided that many readers will likely be using the Apache server, I recommend checking out the Apache site (`http://www.apache.org`) and the great Apache resource Apache Week (`http://www.apacheweek.com`). Beyond your server, PHP can be also held accountable for providing some degree of security through its configuration options and cautious coding.

This final chapter, devoted to introducing many of these issues to you, is divided into five sections:

- Configuration Issues
- Coding Issues
- Data Encryption

- Ecommerce Solutions
- User Authentication

Although none of these sections will provide you with *all* of the answers regarding how to build an impregnable PHP application system, they will provide you with the basis from which you can begin your own investigation into this important topic.

Configuration Issues

There are several configuration options you should consider immediately after installing PHP to begin safeguarding your system. Of course, your configuration choices should depend on your particular situation. For example, if solely you or your development team are going to be programming PHP, then your security configuration may be vastly different from an ISP that has decided to allow all clients to develop PHP scripts for use on the server. Regardless of your situation, it is a good idea to evaluate all of the configuration options and implement only those that you deem necessary. These options are in the php.ini file.

safe_mode boolean

Enabling `safe_mode` places restrictions on several potentially dangerous PHP options. It can be enabled by setting `safe_mode` to the Boolean value of on, or disabled by setting it to off Its restriction scheme is based on the comparison of the UID (user ID) of the executing script and the UID of the file that that script is attempting to access. If the UIDs are the same, the function can execute; otherwise, the function fails.

It isn't possible to use `safe_mode` when PHP is compiled as an Apache module. This is because, when run as an Apache module, all PHP scripts run under the same user as Apache, making it impossible to differentiate between script owners. Please see the section "Safe_mode and the PHP Apache Module," later in this chapter, for more information.

Specifically, when `safe_mode` is enabled, several restrictions come into effect:

- Use of all input/output functions (`fopen()`, `file()`, and `include()`, for example) is restricted to usage only with files that have the same owner as the script that is calling these functions. For example, assuming that `safe_mode` is enabled, `fopen()` called from a script owned by Mary calling will fail if it attempts to open a file owned by John. However, if Mary owns the script calling `fopen()` and the file called by `fopen()`, the function will be successful.

- Attempts by a user to create a new file will be restricted to creating the file in a directory in which the user is the owner.

- Attempts to execute external scripts via functions like `popen()`, `system()`, or `exec()` are only possible when the external script resides in the directory specified by `safe_mode_exec_dir`. This directive is discussed later in this section.

- HTTP authentication is further strengthened because the UID of the owner of the authentication script is prepended to the authentication realm. User authentication is discussed in further detail in the later section "User Authentication."

- The username used to connect to a MySQL server must be the same as the username of the owner of the file calling `mysql_connect()`.

Table 16-1 provides a complete list of functions that are affected when `safe_mode` is enabled.

Table 16-1. Functions restricted by safe_mode

chgrp	include	require
chmod	link	rmdir
chown	passthru	symlink
exec	popen	system
fopen	readfile	unlink
file	rename	

TIP *The PHP documentation for safe_mode has unfortunately not been updated since PHP2.0, although its functionality remains largely unchanged. This documentation is at http://www.php.net/manual/phpfi2.html.*

safe_mode_exec_dir string

This directive specifies the residing directory in which any system programs reside that can be executed by functions such as `system()`, `exec()`, or `passthru()`. `Safe_mode` must be enabled for this to work.

disable_functions string

You can set this directive equal to a comma-delimited list of function names that you want to disable. Note that this directive is not in any way related to `safe_mode` For example, if you wanted to just disable `fopen()`, `popen()`, and `file()`, just set `disable_functions` as follows:

```
disable_functions = fopen,popen,file
```

doc_root string

This directive can be set to a path that specifies the root directory from which PHP files will be served. If `doc_root` is set to nothing (empty), it will be ignored, and the PHP scripts are executed exactly as the URL specifies. If `safe_mode` is enabled and `doc_root` is not empty, no PHP scripts lying outside of this directory will be executed.

max_execution_time integer

This directive specifies how many seconds a script can execute before being terminated. This can be useful to prevent users' scripts from eating up CPU time. By default, this is set to 30 seconds. If you set it to zero, no time limit will be set.

memory_limit integer

This directive specifies, in bytes, how much memory a script can use. By default, this is set to 8 megabytes (8,388,608 bytes).

sql.safe_mode integer

When enabled, `sql.safe_mode` ignores all information passed to `mysql_connect()` and `mysql_pconnect()`, allowing connection only under the user the Web server is running as.

user_dir string

This directive specifies the name of the directory in a user's home directory where PHP scripts must be placed in order to be executed. For example, if user_dir is set to scripts and user Alessia wants to execute somescript.php, then that user must create a directory named scripts in her home directory and place somescript.php

in it. This script can then be accessed via the URL http://www.yoursite.com/~alessia/somescript.php. Notice that the URL does not include the directory scripts. This directive is typically used in conjunction with Apache's UserDir configuration directive.

safe_mode and the PHP Module

Keep in mind that `safe_mode` is not useful when using PHP as a server module. This is because the PHP module runs as a part of the Apache server, and therefore all PHP scripts are executed under the same UID as the Apache server itself. Since `safe_mode` operates under the premise of comparing UIDs to restrict use of certain functions, it can only really be useful when the CGI version of PHP is used in conjunction with SuExec (http://www.apache.org/docs/suexec.html). This is because the CGI version of PHP runs as a separate process, and therefore the UID can be changed dynamically through the suExec functionality. If you are particularly interested in making use of PHP's `safe_mode` features, running PHP as a CGI along with suExec is probably your best bet, although it will be at a cost of speed and overall performance.

Another important configuration strategy is the prevention of certain files from being viewed in the browser. Certainly you wouldn't want those secret passwords or other configuration information to be viewed by an outside user, would you? That is the topic of this next section.

Hiding Data Files and Configuration Files

This is an extremely important security-oriented procedure to keep in mind, regardless of the programming language. I will use the Apache server configuration to illustrate just how easily your security can be compromised if sufficient steps aren't taken to "hide" files not meant to be viewed by the user.

In Apache's httpd.conf file is a configuration directive named DocumentRoot. This is set to the path from which you would like the server to consider to be the public HTML directory. Any file in this path is considered fair game in terms of being served to a user's browser, even if the file does not have a recognized extension. It is not possible for a user to view a file that resides outside of this path. Therefore, it is a very good idea to *always* place your configuration files outside of the DocumentRoot path!

As an exercise, create a file and inside this file type "my secret stuff." Save this file into your public HTML directory under the name of secrets with some really strange extension like .zkgjg. Obviously, the server isn't going to recognize this extension, but it's going to attempt to serve up the data anyway. Now, go to your browser and request that file, using the URL pointing to that file. Scary, isn't it? Fortunately, there are two simple ways to correct this problem.

Maintain the Document Outside of the Document Root

The first solution is to simply place any files that you do not want the user to view outside of document root. Then use `include()` to include those files into any PHP files. For example, assume that you set your document root to:

```
DocumentRoot C:/Program Files/Apache Group/Apache/htdocs    # Windows
DocumentRoot /www/apache/home                               # non-Windows
```

Suppose you have a file containing access information (hostname, username, password) for your MySQL database. You certainly wouldn't want anyone to view that file, so it would be a good idea to place it outside of the document root. Therefore, in Windows, you could save that file to:

```
C:/Program Files/mysecretdata/
```

or

```
/usr/local/mysecretdata/
```

for UNIX.

When you need to use this access information, just include these files using the full pathname where needed. For example:

```
INCLUDE("C:/Program Files/mysecretdata/mysqlaccess.inc");
```

for Windows, or

```
INCLUDE("/usr/local/mysecretdata/mysqlaccess.inc");
```

for UNIX.

Of course, if you have `safe_mode` disabled (see the previous section, "Configuration Issues"), this may not prevent other users with the capability to execute PHP scripts on the machine from attempting to include that file into their own scripts. Therefore, in a multiuser environment it would be a good idea to couple this safeguard with the enabling of `safe_mode`.

Configure httpd.conf File to Deny Certain File Extension Access

A second way to prevent users from viewing certain files is to deny access to certain extensions by configuring the httpd.conf file FILES directive. Assume that you don't want anyone to access files having the extension .inc. Simply place the following in your httpd.conf file:

```
<Files *.inc>
    Order allow,deny
    Deny from all
</Files>
```

After making this addition, restart the Apache server, and you will find that access is denied to any user making a request to view a file having the extension .inc via the browser. However, you can still include these files in your scripts. Incidentally, if you search through the httpd.conf file, you will see that this is the same premise used to protect access to .htaccess files. These files are used to password-protect certain directories and are discussed at the conclusion of this chapter.

Coding Issues

Even if you have a solid server configuration, you still must be constantly wary of introducing security holes into your PHP code. It's not that PHP is not a secure language. It is possible to introduce potentially dangerous holes in practically any programming language. However, given PHP's propensity to be used in a large-scale distributed environment (that is, the Web), the opportunity for users to attempt to "break" your code increases substantially. It's up to you to make sure that this does not happen.

Accepting User Input

While the ability to accept user input is an important part of practically any useful application, you must constantly be wary of the introduction of malicious data, both accidental and intentional. The danger involved in regard to a Web application is even more pronounced, since it is possible for a user to execute system commands through the use of functions such as `system()` or `exec()`.

One of the easiest ways to combat malicious user input is by using the predefined function `escapeshellcmd()`.

escapeshellcmd()

The function `escapeshellcmd()` will escape any questionable characters entered by the user that could result in the execution of a potentially damaging system command. Its syntax is:

```
string escapeshellcmd(str command)
```

To illustrate just how ugly things could get if you were not to control user input, suppose that you offered users the ability to execute system commands such as `'ls -l'`. However, what if the user entered something like `` `rm -rf * ` `` you were to then either echo this input or insert it into `exec()` or `system()`, it could potentially recursively delete files and directories from your server! You can eliminate these problems by first cleaning up the command with `escapeshellcmd()`. Reconsidering the input `` `rm -rf * ` ``, if you were to first pass it through `escapeshellcmd()`, the string would be converted to `` \`rm -rf \*\` ``.

> **NOTE** *Backticks are an execution operator, telling PHP to attempt to execute the contents found between backticks. The output can be echoed directly to the screen, or it can be assigned to a return variable.*

Another problem that arises from user input is the introduction of HTML content. This can be particularly problematic when the information is displayed back to the browser, as is the case with a message board. The introduction of HTML tags into a message board could alter the display of the page, causing it to be displayed incorrectly or not at all. This problem can be eliminated by passing the user input through `strip_tags()`.

strip_tags()

The function `strip_tags()` will remove all HTML tags from a string. Its syntax is:

```
string strip_tags(str string [, str allowed_tags])
```

The input parameter `string` is the string that will be examined for tags, while the optional input parameter `allowed_tags` specifies any tags that you would like to be allowed in the string. For example, italic tags (`<i></i>`) might be allowable, but table tags such as `<td></td>` could potentially wreak havoc on a page. An example of usage of the function follows:

```
$input = "I <i>really</i> love PHP!";
$input = strip_tags($input);
// $input now equals "I really love PHP!"
```

This sums up the brief synopsis of the two more widely used functions for sanitizing user input. Next I introduce data encryption, highlighting several of PHP's predefined functions capable of encrypting data.

Data Encryption

Encryption can be defined as the translation of data into a format that is, in theory, unreadable by anyone except the intended party. The intended party can then decode, or decrypt, the encrypted data through the use of a secret key or password. PHP offers support for several encryption algorithms. Several of the more prominent ones are described here.

General Encryption Functions

It is important to realize that encryption over the Web is largely useless unless the scripts running the encryption schemes are operating via a secured server. Why? Since PHP is a server-side scripting language, information must first be sent to the server in plain text format *before* it can be encrypted. There are many ways that an unwanted third party can watch this information as it is transmitted from the user to the server if the user is not operating via a secured connection. For more information about setting up a secure Apache server, check out http://www.apache-ssl.org. For those readers implementing a different Web server, refer to your documentation. Chances are that there exists at least one, if not several different, security solutions for your particular server.

md5()

Md5 is a third-party hash algorithm used for creating digital signatures (among other things), which can be used to uniquely identify the sending party. PHP provides support to it:

```
string md5(string string)
```

It is considered to be a "one-way" hashing algorithm, which means there is no way to dehash data that has been hashed using `md5()`.

The Md5 algorithm can also be used as a password verification system. Since it is in theory extremely difficult to retrieve the original string that has been

hashed using the Md5 algorithm, you could hash a given password using Md5 and then compare that encrypted password against those that a user enters in order to gain access to restricted information.

For example, assume that our secret password toystore has an Md5 hash of 745e2abd7c52ee1dd7c14ae0d71b9d76. You store this hashed value on the server and compare it to the Md5 hash equivalent of the password the user attempts to enter. Even if an intruder were to get hold of the encrypted password, it wouldn't make much difference, since that intruder couldn't (in theory) decrypt it. An example of hashing a string follows:

```
$val = "secret";
$hash_val = md5 ($val);
// $hash_val = "c1ab6fb9182f16eed935ba19aa830788";
```

Now I'll introduce another way to secure a data string, that is, through another one of PHP's predefined functions: `crypt()`.

crypt()

`Crypt()` offers a convenient way to one-way encrypt a piece of data. By one-way encrypt, I mean that the data can only be encrypted; there is no known algorithm to decrypt the data once it is encrypted using `crypt()`. Its syntax is:

```
string crypt(string string [, salt])
```

The input parameter `string` is the string that will be encrypted by the `crypt()` algorithm. The optional input parameter, `salt`, determines the type of encryption that will be used to encrypt `string`. Specifically, the encryption type is determined by the length of the salt. The various encryption types and their determinant salt lengths are shown in Table 16-2.

Table 16-2. Encryption Types and Corresponding Salt Lengths

ENCRYPTION TYPE	LENGTH
CRYPT_STD_DES	2
CRYPT_EXT_DES	9
CRYPT_MD5	12 (starting with first character of unencrypted password)
CRYPT_BLOWFISH	12 (starting with first two characters of unencrypted password)

Not all encryption formats are available on each system, but you can easily determine which of the formats listed in Table 16-2 are available by printing the encryption type to the browser. A 1 will be displayed if it is available, 0 otherwise.

Listing 16-1 illustrates the use of `crypt()` to create and compare encrypted passwords.

Listing 16-1: Using `crypt()` (STD_DES) to store and compare passwords

```
$user_pass = "123456";
// extract the first two characters of $user_pass for use as salt.
$salt = substr ($user_pass, 0, 2);
// encrypt and store password somewhere
$crypt1 = crypt($user_pass, $salt);
// $crypt1 = "12tir.zIbWQ3c";

// . . . user enters password
$entered_pass = "123456";

// get the first two characters of the stored password
$salt1 = substr ($crypt, 0, 2);
// encrypt $entered_pass using $salt1 as the salt.
$crypt2 = crypt ($entered_pass, $salt1);
// $crypt2 = "12tir.zIbWQ3c";
// Therefore, $crypt1 = $crypt2
```

As you can see in Listing 16-1, `$crypt1` equals `$crypt2`, but *only* because I correctly used the first two characters of `$crypt1` as the salt for the encryption of `$entered_pass`. I suggest that you experiment with this example, inserting different salt values so that you can see firsthand that `$crypt1` and `$crypt2` will only end up equivalent using this procedure.

> **TIP** *When choosing between* `crypt()` *and* `md5()` *to carry out your site encryption procedures, go with* `md5()`. *It's more secure.*

mhash()

The function `mhash()` offers support for a number of hashing algorithms, allowing developers to implement checksums, message digests, and various other digital signatures into their PHP application. Hashes are also used for storing passwords. Integrating the `mhash()` module into your PHP distribution is rather simple:

1. Go to http://mhash.sourceforge.net and download the source.

2. Extract the contents of the compressed distribution and follow the instructions as specified in the INSTALL document.

3. Compile PHP with the `-with-mhash` option.

Easy enough. There is, however, one quirk that tends to cause trouble when compiling mhash into a PHP/Apache system. Apparently, many find that they have to configure `mhash` as follows: `"./configure -disable-pthreads"`. (You'll understand what I'm talking about when you read the mhash INSTALL document.) Keep this in mind when compiling your distribution.

On completion of the installation process, you have the functionality offered by `mhash` at your disposal. `Mhash` currently supports the hashing algorithms listed in Table 16-3.

Table 16-3. Hashing Algorithms Currently Supported by mhash()

SHA1	RIPEMD160	MD5
GOST	TIGER	SNEFRU
HAVAL		CRC32
RIPEMD128		
CRC32B		

mcrypt()

`Mcrypt` is a popular data-encryption package available for use with PHP, providing support for two-way encryption (that is, encryption and decryption). The `mcrypt` module offers support for the four types of encryption modes discussed here:

> **TIP** *For more information about encryption modes, I recommend the textbook* Applied Cryptography Second Addition, *by Bruce Schneier (John Wiley & Sons, 1996).This is a fantastic resource for learning more about cryptographic protocols, techniques, and algorithms.*

CBC: Cipher Block Chaining

CBC mode is typically the encryption mode that is the most frequently used of the four. Unlike ECB (described below), using CBC results in different encryption patterns of identical plain text blocks, making it more difficult for an attacker to discern patterns. If you don't know which of the four modes you should be using, use this one. However, I would suggest learning more about each mode before making a final decision.

CFB: Cipher Feedback

CFB combines certain characteristics of the stream cipher, resulting in the elimination of the need to amass blocks of data before enciphering takes place. Typically, you won't need to use this mode.

ECB: Electronic Code Book

ECB mode encrypts each plain text block independently with the block cipher, making it susceptible to attack when used to encrypt relatively small block sizes of language text. This is because ECB will encrypt two plain text blocks with identical encipherments, providing an attacker with a means to base a decipherment strategy. Therefore, unless you have a valid reason for using ECB, you'll probably want to use CBC mode instead.

OFB: Output Feedback

OFB mode has many of the same characteristics as the CFB mode. Like CFB, you typically won't need to use this mode.

NOTE *To use the functionality offered by* `mcrypt`*, you must first download the mcrypt package from ftp://argeas.cs-net.gr/pub/unix/mcrypt/.*

A Final Note About Data Encryption

The methods in this section are only those that are in some way incorporated into the PHP extension set. However, you are not limited to these encryption/hashing solutions. Keep in mind that you can use functions like `popen()` or `exec()` to work with any of your favorite third-party encryption technologies, PGP (http://www.pgpi.org) or GPG (http://www.gnupg.org), for example.

You might find the following links particularly useful for learning more about cryptography and information privacy:

- http://jya.com/crypto-free.htm
- http://www.io.com/~ritter/LEARNING.HTM
- http://www.rsasecurity.com/rsalabs/faq/
- http://www.cs.auckland.ac.nz/~pgut001/links.html
- http://www.thawte.com/support/crypto/contents.html

To close out this section, I would like to throw caution into the wind by saying that before you begin implementing mission-critical applications involving encryption, take some time to really learn about the mechanics of data encryption. Remember that in the world of data security, ignorance is certainly not bliss. For those new to the subject, take a moment to check out the links that I've included. They are widely regarded as great introductions to the many facets of encryption and data security.

E-Commerce Functions

One can hardly deny the frenzy that the advent of ecommerce has instilled into the populations of the world, not to mention the advantages and conveniences that have resulted from it. Thankfully, those of you who are interested in developing your own ecommerce sites have a number of trusted third-party applications that you can easily integrate into PHP scripts. I make brief note of some of the more popular ones in this section.

Verisign

Verisign, Inc. (http://www.verisign.com) offers a wide array of ecommerce-related products and services. PHP provides support for interfacing with Verisign's Payflow Pro service.

> **NOTE** *To use the Verisign functionality, PHP must be compiled with the —with-pfpro [=DIR] directive. Also, there are several Payflow Pro configuration directives available in the php.ini file.*

PHP's Payflow Pro functionality is extremely easy to use and requires a minimum of time and knowledge to begin performing transactions. However, just because you compile Verisign support into your PHP installation does not mean that you are capable of using the Verisign services! To do so, you must first register at the Verisign site and download Verisign's SDK package. At the time of this writing, setup of Payflow Pro involved a one-time fee of $249, in addition to a monthly fee of $59.95 for a maximum of 5,000 monthly transactions, or a monthly fee of $995 for unlimited transactions.

One further note to keep in mind: Before you purchase a Verisign account, you can test your script interface with Verisign's test account, offered free of charge. Performing test transactions with this test account will eliminate unnecessary expenditures when debugging your code. Check out the Verisign site for more information.

You can find more information regarding Verisign at:

- http://www.verisign.com
- http://www.php.net/manual/ref.pfpro.php

Cybercash

Cybercash, Inc. (http://www.cybercash.com) offers a variety of credit card authorization and transaction services and software to those wishing to incorporate these services into their Web application.

> **NOTE** *To make use of the Cybercash functionality, PHP must be compiled with the —with-cybercash=[DIR] directive.*

Cybercash provides C and Perl scripts capable of interfacing with the Cybercash transaction service. With this in mind, PHP users generally choose one or a combination of the following methods for incorporating Cybercash into their site:

- Make use of the cyberlib.php API, included in the PHP distribution. This provides you with the functionality necessary to perform the transactions. (Recommended.)
- Use the existing Perl and C scripts to interface with the Cybercash service, calling them from your own PHP scripts. (Recommended.)
- Rewrite the existing Perl and C scripts in PHP. (Not recommended.)

As with Verisign, keep in mind that just because you compile Cybercash into your PHP installation does not mean that you can use the service! Cybercash integration services are not free and can be rather costly. (The setup for the Cybercash ecommerce CashRegister service currently runs $495, in addition to a $20/month fee plus $0.20 per transaction.) However, despite these costs, many PHP developers feel that Cybercash is one of the best solutions available.

One further note: Before you purchase a Cybercash account, you can test your script interface with the Cybercash test account, offered free of charge. Performing test transactions with this test account will eliminate unnecessary expenditures when debugging your code. Check out the Cybercash site for more information.

Further information regarding Cybercash is at:

- http://www.cybercash.com
- http://www.php.net/manual/ref.cybercash.php

CCVS

CCVS, or the Credit Card Verification System, is a technology developed by RedHat (http://www.redhat.com) that allows you to independently process credit card transactions, directly accessing the credit card agencies rather than going through a third party (such as Cybercash). CCVS is compatible with many of the major Linux/UNIX platforms and can be easily modified since RedHat provides you with the source code to make changes as you wish.

NOTE *Note: To make use of the CCVS functions, PHP must be compiled with the —with-ccvs=[DIR] directive.*

You can find more information regarding CCVS at:

- http://www.php.net/manual/ref.ccvs.php
- http://www.redhat.com/products/ccvs/support/CCVS3.3docs/ProgPHP.html
- http://www.redhat.com/products/ccvs/

User Authentication

Just like knowing the "secret handshake" will get a person into the treehouse, knowing the correct username and password can grant a user the right to enter otherwise unauthorized server directories. These authentication systems are typically known as "challenge and response." The challenge is the prompt for the username and password, and the response is the input of a username and password combination. If the combination is correct, the user is permitted to enter the restricted directory; otherwise, the user is denied, and an appropriate message is displayed.

A pop-up authentication prompt is often used to query the user for a username and password. This prompt can be activated via calling an authentication header, shown in Listing 16-2.

Listing 16-2: Basic authentication prompt

```
<?
header('WWW-Authenticate: Basic realm="Secret Family Recipes"');
header('HTTP/1.0 401 Unauthorized');
exit;
?>
```

Executing the code in Listing 16-2 will only produce the pop-up window. The two calls to the `header()` function prompt the browser to display this window. This window will look similar to the one in Figure 16-1.

Figure 16-1. User authentication window

Now that you can set up the necessary interface, it is time to turn your attention to processing the username and password. In PHP, the login and password are stored in two global variables, namely, `$PHP_AUTH_USER` (username) and `$PHP_AUTH_PW` (password). Listing 16-3 shows how these variables can be checked for values. If they are not set, the authentication window is again displayed.

TIP *As you experiment with the scripts in this section, you may find that the authentication window does not always pop up as expected after you refresh the page. This does not necessarily imply a problem with the code; rather it is a function of the browser's implementation of the authentication window. You will need to close and relaunch the browser in order to receive the prompt.*

Listing 16-3 Checking PHP's global authentication variables

```
if ( (! isset ($PHP_AUTH_USER)) || (! isset ($PHP_AUTH_PW)) ):
     header('WWW-Authenticate: Basic realm="Secret Family Recipes"');
     header('HTTP/1.0 401 Unauthorized');
     print "You are attempting to enter a restricted area. Authorization is
required.";
     exit;
endif;
```

An easy albeit rather restrictive way to set up a restricted page is to simply hardcode the username and password into the authentication script. Consider Listing 16-4, which builds on the previous example.

Listing 16-4 Hardcoding the username and password into a script

```
if ( (! isset ($PHP_AUTH_USER)) || (! isset ($PHP_AUTH_PW)) ||
   ($PHP_AUTH_USER != 'secret') || ($PHP_AUTH_PW != 'recipes') ) :

     header('WWW-Authenticate: Basic realm="Secret Family Recipes"');
     header('HTTP/1.0 401 Unauthorized');
     print "You are attempting to enter a restricted area. Authorization is
     required.";
     exit;
endif;
```

Multiple User Authentication

Although the code in Listing 16-4 may be your solution when dealing with a small, static group of people, chances are you will be interested in a more robust and flexible solution to granting access to restricted areas of your Web site. Most likely, this involves granting a separate username and password for each user that you expect to visit the restricted area. There are several methods used to accomplish this, perhaps the most common being checking authentication information against a text file or database.

Storing Information in a Text File

A very simple yet effective solution for storing user authentication information is in a text file. Each line of this text file would contain a username/password pair that can be read in and tested one by one. A text file used for these purposes might look like the one shown in Listing 16-5.

Listing 16-5: A typical authentication text file (authenticate.txt)

```
brian:snaidni00
alessia:aiggaips
gary:9avaj9
chris:poghsawcd
matt:tsoptaes
```

As you can see, each line consists of a username, followed by a password, with a colon (`:`) separating the two. This means that there are five potential username/password combinations that can be used to enter the restricted area for which this text file is intended. Each time a user enters a username and password via the authentication window, the text file is opened and searched methodically for a matching pair. If a match is found, the user is permitted to enter the restricted area; otherwise, the user is denied access. This authentication procedure is displayed in Listing 16-6.

Listing 16-6 Text file–based user authentication

```
$file = "authenticate.txt";
$fp = fopen($file, "r");
$auth_file = fread ($fp, filesize($fp));
fclose($fp);

// assign each line of file as array element
$elements = explode ("\n", $auth_file);

foreach ($elements as $element) {

     list ($user, $pw) = split (":", $element);

     if (($user == $PHP_AUTH_USER) && ($pw == $PHP_AUTH_PW)) :
          $authorized = 1;
          break;
     endif;

} // end foreach

if (! $authorized) :
          header('WWW-Authenticate: Basic realm="Secret Family Recipes"');
          header('HTTP/1.0 401 Unauthorized');
          print "You are attempting to enter a restricted area. Authorization is
          required.";
          exit;
```

```
else :
          print "Welcome to the family's secret recipe collection";
endif;
```

Storing Information in a Database

Storing user authentication information in a database is advantageous for many reasons, many of them discussed in detail in Chapter 11, "Databases." Easy updating, scalability, and flexibility are just a few reasons why using a database is the logical choice for storing large amounts of user authentication data. Table 16-4 illustrates a sample database table used to store this data. After authentication lookup has successfully taken place, the user ID can then be used to tie into other tables hosting various other forms of user data and preferences. The idea of effectively quarantining related data to separate, smaller tables, rather than just grouping it all into one massive table, is known as *database normalization* and was briefly discussed in Chapter 11.

> **NOTE** *MySQL syntax is used to illustrate the examples in this section. The code is simple enough to be easily converted to other database servers.*

Table 16-4. A sample user authentication table (user_authenticate)

USER ID	USERNAME	PASSWORD
ur1234	brian	2b877b4b825b48a9a0950dd5bd1f264d
ur1145	alessia	6f1ed002ab5595859014ebf0951522d9
ur15932	gary	122a2a1adf096fe4f93287f9da18f664
ur19042	chris	6332e88a4c7dba6f7743d3a7a0c6ea2c
ur18930	matt	9252fe5d140e19d308f2037404a0536a

Listing 16-7 will first check to see whether or not the `$PHP_AUTH_USER` variable has been set. If it has not, the authentication window will pop up, prompting the user to enter the necessary information. Otherwise, a connection to the MySQL server is established and the user_authenticate table is queried using the username and password entered by the user. If no match is found, the authentication window will be displayed again. Otherwise, `$userid` is assigned the matching user ID, essentially authenticating the user.

Listing 16-7: Authenticating a user via database lookup

```
if (!isset($PHP_AUTH_USER)):
     header('WWW-Authenticate: Basic realm="Secret Family Recipes"');
     header('HTTP/1.0 401 Unauthorized');
     exit;
```

```
else :
     // connect to the mysql database
     mysql_connect ("host", "user", "password") or die ("Can't connect to
     database!");
     mysql_select_db ("user_info") or die ("Can't select database!");

     // query the user_authenticate table for authentication match
     $query = "select userid from user_authenticate where
                                          username = '$PHP_AUTH_USER' and
                                          password = '$PHP_AUTH_PW'";
     $result = mysql_query ($query);

     // if no match found, display authentication window
     if (mysql_numrows($result) != 1) :
          header('WWW-Authenticate: Basic realm="Secret Family Recipes"');
          header('HTTP/1.0 401 Unauthorized');
          exit;
     // else, retrieve user-Id
     else :
          $userid = mysql_result (user_authenticate, 0, $result);
     endif;
endif;
```

Conclusion

This chapter introduced a wide array of topics relating to security. As you've learned throughout this chapter, properly securing your PHP applications revolves around a combination of properly configuring your server and PHP installation, and employing prudent coding to prevent user input from wreaking havoc. Other variables such as encryption, credit card verification, and user authentication play important roles when applicable. To recap, I briefly introduced the following topics:

- PHP's configuration issues
- Safe mode and the PHP module
- Coding issues
- Data encryption

- Ecommerce functions
- User authentication

In closing, I would like to state that properly planning the level of security that your PHP application will require is as important as, if not more so than, planning the other features of your application that will make it a success. Therefore, always take time to properly outline the security features that you must employ *before* you begin coding. In the long run, it will save you time and aid in the prevention of potential security holes in your application.