

SECURITY TECHNIQUES

With more and more personal information being stored on the Web—credit card data, social security numbers, maiden names, favorite pets—today’s PHP developer cannot afford to be ignorant when it comes to security. Sadly, most beginning programmers fail to understand the truth about security: there is no such thing as “secure” or “insecure.” The wise programmer knows that the real question is *how secure* a site is. Once any piece of data is stored in a database, in a text file, or on a Post-it note in your office, its security is compromised. The focus in this chapter is therefore how to make your applications more secure.

This chapter will begin by rehashing the fundamentals of secure PHP programming. These are the basic things that I hope/assume you’re already doing. After that a quick example shows ways to validate different kinds of data that might come from an HTML form. The third topic is the new-to-PHP 5 PECL library called Filter. Its usage isn’t very programmer-friendly, but the way it wraps all of the customary data filtering and sanitizing methods into one interface makes it worth knowing. After that, two different uses of the PEAR Auth package show an alternative way to implement authorization in your Web applications. The chapter will conclude with coverage of the MCrypt library, demonstrating how to encrypt and decrypt data.

Remembering the Basics

Before getting into demonstrations of more particular security techniques, I want to take a moment to go over the basics: those fundamental rules that every PHP programmer should abide by all of the time.

To ensure a basic level of security:

1. Do not rely upon `register_globals`.

The advent of `register_globals` once made PHP so easy to use, while also making it less secure (convenience often weakens security). The recommendation is to program as if `register_globals` is off. This is particularly important because `register_globals` will likely disappear in future versions of PHP.

2. Initialize variables prior to using them.

If `register_globals` is still enabled—even if you aren't using them—a malicious user could use holes created by noninitialized variables to hack your system. For example:

```
if (condition) {
    $auth = TRUE;
}
```

If `$auth` is not preset to `FALSE` prior to this code, then a user could easily make themselves authorized by passing `$_GET['auth']`, `$_POST['auth']`, or `$_COOKIE['auth']` to this script.

3. Verify and purify all incoming data.

How you verify and purify the data depends greatly upon the type of data. You'll see many different techniques in this chapter and the book.

Avoiding Mail Abuses

A security concern exists in any Web application that uses the `mail()` function with form data. For starters, if someone enters their “to” email address as *someone@example.com,someone.else@example.com*, you'll now be sending two emails. If a malicious user enters 500 addresses (perhaps by creating their own form that submits to your same page), you're now sending out spam! You can avoid this by using regular expressions to guarantee that the submitted value contains just one address. Or you could search for a comma in the submitted email address, which wouldn't be allowed. But that won't solve the problem entirely.

Although the `mail()` function takes separate arguments for the “to” address, “from” address (or other additional headers), subject, and body, all four values are put together to create the actual message. By submitting specifically formatted text through any of these inputs, bad people can still use your form to send their spam. To guard against this, you should watch for newline (`\n`) and carriage returns (`\r`) within the submitted data. Either don't send emails with these values or replace them with spaces to invalidate the intended message format. You should probably also make sure that you (or someone involved with the site) receives a copy of every email sent so that close tabs can be kept on this area of the server.

4. Be careful if you use variables for included files.

If your code does something like

```
require($page);
```

then you should either make sure that `$page` does not come from an outside source (like `$_GET`) or, if it does, that you've made certain that it has an appropriate value. See the technique in Chapter 2, "Developing Web Applications."

5. Be extra, extra careful when using any function that runs commands on the server.
This includes `eval()`, `exec()`, `system()`, `passthru()`, `popen()`, and the backticks (```). Because each of these runs commands on the server itself, they should never be used casually. And if you must use a variable as part of the command to execute, perform any and all security checks on that variable first. Also use the `escapeshellarg()` and `escapeshellcmd()` functions as an extra precaution.
6. Consider changing the default session directory or using a database to store session data.
An example as to how you would do this is discussed in Chapter 3, "Advanced Database Concepts."
7. Do not use browser-supplied filenames for storing uploaded files on the server.
When you move a file onto your server, rename it to something safe, preferably something not guessable.

8. Watch for HTML (and more important, JavaScript) in submitted data if it will be redisplayed in a Web page.

Use the `strip_tags()` or similar functions to clear HTML and potential JavaScript from submitted text.

9. Do not reveal PHP errors on live sites.
One of the most common ways to hack a site is to try to "break" it—do something unexpected to cause errors—in the hopes that the errors reveal important behind-the-scenes information.

10. Nullify the possibility of SQL injection attacks.

Use a language-specific database escaping function, like `mysqli_real_escape_data()`, to ensure that submitted values will not break your queries.

11. Program with error reporting on its highest level.

While not strictly a security issue, programming with error reporting on its highest level can often show potential holes in your code.

12. Never keep `phpinfo()` scripts on the server.

Although vital for developing and debugging PHP applications, `phpinfo()` scripts reveal too much information and are too easily found if left on a live site.

Validating Form Data

Handling form data is still far and away the most common use of PHP (in this author's humble opinion, anyway). The security concern lies in the fact that the PHP page handling the form will do something with the information the user enters: store it in a database, pass it along to another page, or use it in an email. If the information the user enters is tainted, you could have a major problem on your hands. As a rule, do not trust the user! Mistakes can happen, either on purpose or by accident, that could reveal flaws in your code, cause the loss of data, or bring your entire system to a crashing halt.

Some good validation techniques are:

- ◆ Use the `checkdate()` function to confirm that a given date is valid.
- ◆ Typecast numbers.
- ◆ Use regular expressions to check email addresses, URLs, and other items *with definable patterns* (see the sidebar).

When to Use Regular Expressions

I often see what I would call an overuse of regular expressions. You should understand that regular expressions require extra processing, so they shouldn't be used flippantly. Many types of data—comments and addresses being just two examples—really don't have a definable pattern. A regular expression that allows for any valid comment or address would allow for just about anything. So skip the server-intensive regular expressions in such cases.

As a guide, regular expressions *may be* the most exacting security measure, but they're almost definitely the least efficient and possibly the most problematic. I'm not suggesting you shouldn't use them—just make sure they're really the best option for the data being validated.

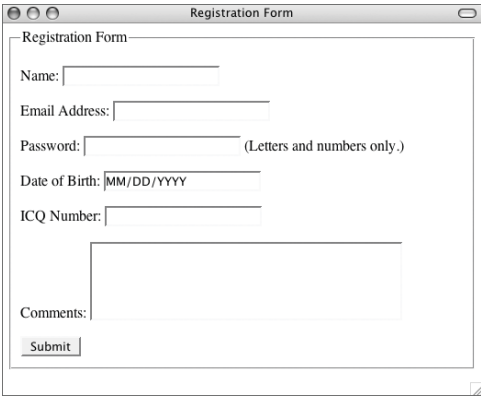


Figure 4.1 When users first come to the registration page, this is the form they will see.

As with the basic security techniques already reviewed, the hope is that as a somewhat-experienced PHP programmer, you already know most of these things. To be certain, this next example will present a sample registration form (**Figure 4.1**), taking various types of information, which will then be precisely validated. In doing so, I'll make use of a couple of Character Type functions, added to PHP in version 4.3. Listed in **Table 4.1**, these functions test a given value against certain constraints for the current locale (established by the `setlocale()` function).

Table 4.1 The Character Type functions provide validation specific to the given environment (i.e., the locale setting).

Character Type Functions	
FUNCTION	CHECKS IF VALUE CONTAINS
<code>ctype_alnum()</code>	Letters and numbers
<code>ctype_alpha()</code>	Letters only
<code>ctype_cntrl()</code>	Control characters
<code>ctype_digit()</code>	Numbers
<code>ctype_graph()</code>	Printable characters, except spaces
<code>ctype_lower()</code>	Lowercase letters
<code>ctype_print()</code>	Printable characters
<code>ctype_punct()</code>	Punctuation
<code>ctype_space()</code>	White space characters
<code>ctype_upper()</code>	Uppercase characters
<code>ctype_xdigit()</code>	Hexadecimal numbers

To validate a form:

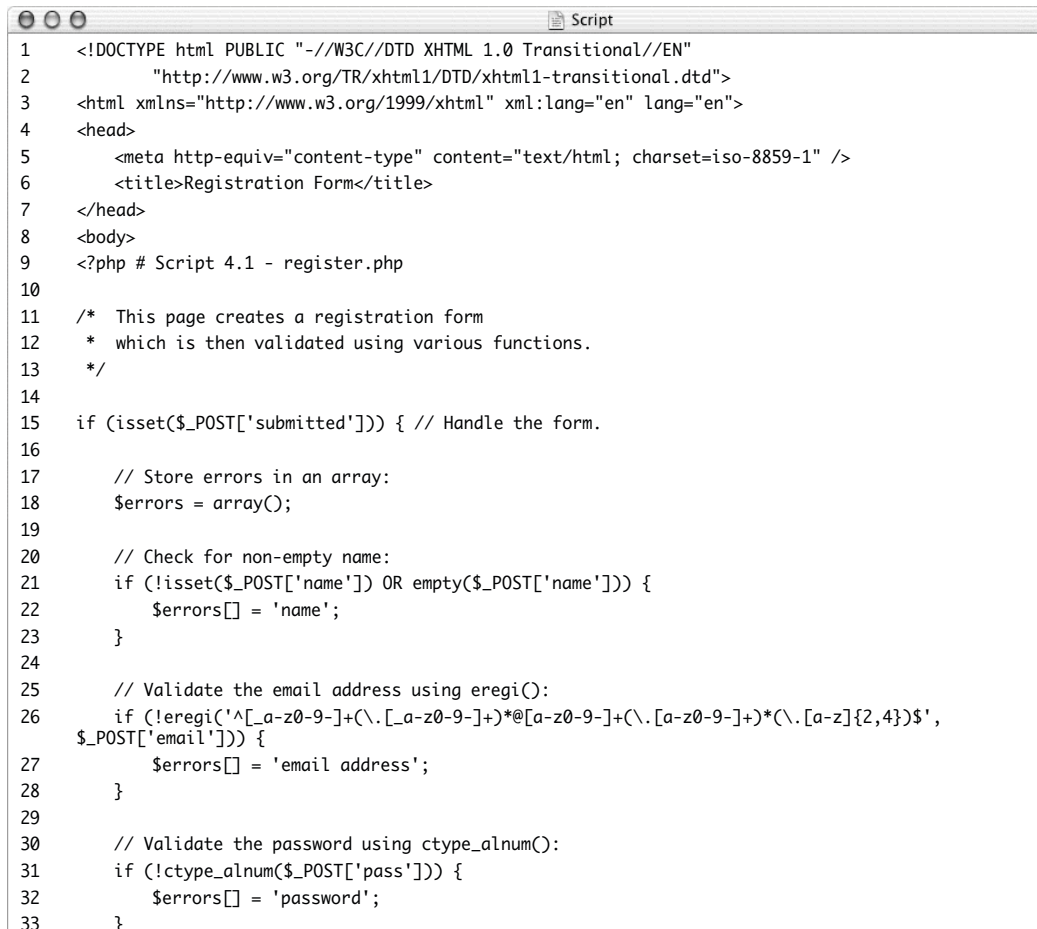
1. Begin a new PHP script in your text editor or IDE, starting with the HTML (**Script 4.1**).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD
→ XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/
→ xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/
→ xhtml" xml:lang="en" lang="en">
```

```
<head>
    <meta http-equiv="content-type"
→ content="text/html; charset=
→ iso-8859-1" />
    <title>Registration Form</title>
</head>
<body>
<?php # Script 4.1 - register.php
```

continues on page 131

Script 4.1 This page both displays a registration form and processes it. The script validates the submitted data using various functions, and then reports any errors.



```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
4 <head>
5     <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
6     <title>Registration Form</title>
7 </head>
8 <body>
9 <?php # Script 4.1 - register.php
10
11 /* This page creates a registration form
12  * which is then validated using various functions.
13  */
14
15 if (isset($_POST['submitted'])) { // Handle the form.
16
17     // Store errors in an array:
18     $errors = array();
19
20     // Check for non-empty name:
21     if (!isset($_POST['name']) OR empty($_POST['name'])) {
22         $errors[] = 'name';
23     }
24
25     // Validate the email address using eregi():
26     if (!eregi('^[_a-z0-9-]+(\\.[_a-z0-9-]+)*@[a-z0-9-]+(\\.[a-z0-9-]+)*$',
27         $_POST['email'])) {
28         $errors[] = 'email address';
29     }
30
31     // Validate the password using ctype_alnum():
32     if (!ctype_alnum($_POST['pass'])) {
33         $errors[] = 'password';
34     }
35 }
```

(script continues on next page)

Script 4.1 continued

```

34
35 // Validate the date of birth using check_date():
36 if (isset($_POST['dob']) AND
37     (strlen($_POST['dob']) >= 8) AND
38     (strlen($_POST['dob']) <= 10) ) {
39
40     // Break up the string:
41     $dob = explode('/', $_POST['dob']);
42
43     // Were three parts returned?
44     if (count($dob) == 3) {
45
46         // Is it a valid date?
47         if (!checkdate((int) $dob[0], (int) $dob[1], (int) $dob[2])) {
48             $errors[] = 'date of birth';
49         }
50
51     } else { // Invalid format.
52         $errors[] = 'date of birth';
53     }
54
55 } else { // Empty or not the right length.
56     $errors[] = 'date of birth';
57 }
58
59 // Validate the ICQ number using ctype_digit():
60 if (!ctype_digit($_POST['icq'])) {
61     $errors[] = 'ICQ number';
62 }
63
64 // Check for non-empty comments:
65 if (!isset($_POST['comments']) OR empty($_POST['comments'])) {
66     $errors[] = 'comments';
67 }
68
69 if (empty($errors)) { // Success!
70
71     // Print a message and quit the script:
72     echo '<p>You have successfully registered (but not really).</p></body></html>';
73     exit();
74
75 } else { // Report the errors.
76
77     echo '<p>Problems exist with the following field(s):<ul>';
78
79     foreach ($errors as $error) {
80         echo "<li>$error</li>\n";
81     }
82

```

(script continues on next page)

Script 4.1 *continued*

```
83         echo '</ul></p>';
84
85     }
86
87 } // End of $_POST['submitted'] IF.
88
89 // Show the form.
90 ?>
91 <form method="post">
92 <fieldset>
93 <legend>Registration Form</legend>
94 <p>Name: <input type="text" name="name" /></p>
95 <p>Email Address: <input type="text" name="email" /></p>
96 <p>Password: <input type="password" name="pass" /> (Letters and numbers only.)</p>
97 <p>Date of Birth: <input type="text" name="dob" value="MM/DD/YYYY" /></p>
98 <p>ICQ Number: <input type="text" name="icq" /></p>
99 <p>Comments: <textarea name="comments" rows="5" cols="40"></textarea></p>
100
101 <input type="hidden" name="submitted" value="true" />
102 <input type="submit" name="submit" value="Submit" />
103 </fieldset>
104 </form>
105
106 </body>
107 </html>
```

2. Create the section of the script that handles the submitted form.

```
if (isset($_POST['submitted'])) {
    $errors = array();
```

Your script should always handle the form before it could possibly redisplay it (on errors found). I like to use a hidden form input to check if a form was submitted. The hidden form input will always be passed to the page upon submission, unlike any other input (on Internet Explorer, if a user submits a button by pressing Enter, then the submit button won't be set).

One way I like to validate forms is to use an array that stores the errors as they occur. By checking if this array is empty, the script can tell if all validation tests have been passed. If the array isn't empty, its values can be used to print the error messages.

3. Check for a name.

```
if (!isset($_POST['name']) OR
    → empty($_POST['name'])) {
    $errors[] = 'name';
}
```

A person's name is one of those things that you can use regular expressions on, but it may not be worthwhile. A valid name can contain letters, spaces, periods, hyphens, and apostrophes. Under most circumstances, just checking for a nonempty name is sufficient.

4. Validate the submitted email address.

```
if (!eregi('^[a-z0-9-]+(\.[a-z0-9-
→ 9-]+)*@[a-z0-9-]+(\.[a-z0-9-
→ +)*(\.[a-z]{2,4})$')
    → $_POST['email'])) {
    $errors[] = 'email address';
}
```

There are any number of patterns you can use to validate an email address, depending on how strict or liberal you want to be. This one is commonly seen. Certainly some invalid email addresses could slip through this expression, but it does add a sufficient level of security. Feel free to use a different pattern if you have one to your liking. Keep in mind that a user could enter a valid e-mail address that does not actually exist. Only some sort of activation process (sending the user an email containing a link back to the Web site) can confirm a real address.

5. Validate the submitted password.

```
if (!ctype_alnum($_POST['pass'])) {
    $errors[] = 'password';
}
```

The form indicates that the password must contain only letters and numbers. To validate such values, the function `ctype_alnum()` works perfectly.

In a real registration form, I would also recommend confirming the password with a second password input, then making sure both values match. I'm skipping that step here for brevity's sake.

continues on next page

6. Begin checking to see if the user entered a valid date of birth.

```
if (isset($_POST['dob']) AND
(strlen($_POST['dob']) >= 8) AND
(strlen($_POST['dob']) <= 10) ) {
    $dob = explode('/',
    → $_POST['dob']);
    if (count($dob) == 3) {
```

There is really no way of knowing if the information users enter is in fact their birthday, but PHP's built-in `checkdate()` function can confirm whether or not that date existed. Since the form takes the date of birth as a simple string in the format *MM/DD/YYYY*, the script must first confirm that something was entered. I also check if the string's length is at least eight characters long (e.g., 1/1/1900) but no more than ten characters long (e.g., 12/31/2000).

This string is then exploded on the slashes to theoretically retrieve the month, day, and year values. Next, a conditional checks that exactly three parts were created by the explosion.

7. Check if the date of birth is a valid date.

```
if (!checkdate((int) $dob[0], (int)
→ $dob[1], (int) $dob[2])) {
    $errors[] = 'date of birth';
}
```

The `checkdate()` function confirms that a date is valid. You might want to also check that a user didn't enter a date of birth that's in the future or the too-recent past. Each value is typecast as an integer as an extra precaution.

8. Complete the date of birth conditionals.

```
} else {
    $errors[] = 'date of
    → birth';
}
} else {
    $errors[] = 'date of birth';
}
```

The first `else` applies if the submitted value cannot be exploded into three parts. The second `else` applies if the value isn't of the right length.

9. Validate the ICQ number.

```
if (!ctype_digit($_POST['icq'])) {
    $errors[] = 'ICQ number';
}
```

The ICQ number can only contain digits, so it makes sense to use the `ctype_digit()` function.

10. Check for some comments.

```
if (!isset($_POST['comments']) OR
→ empty($_POST['comments'])) {
    $errors[] = 'comments';
}
```

Comments really cannot be run through a regular expression pattern because any valid pattern would allow just about anything. Instead, a check for some value is made.

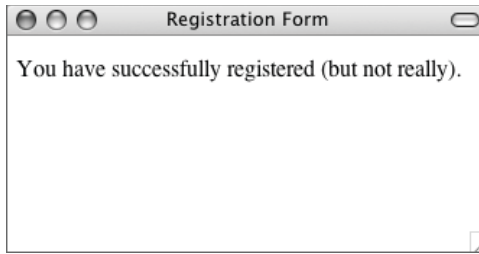


Figure 4.2 If all of the data passed through the various checks, this message is displayed.

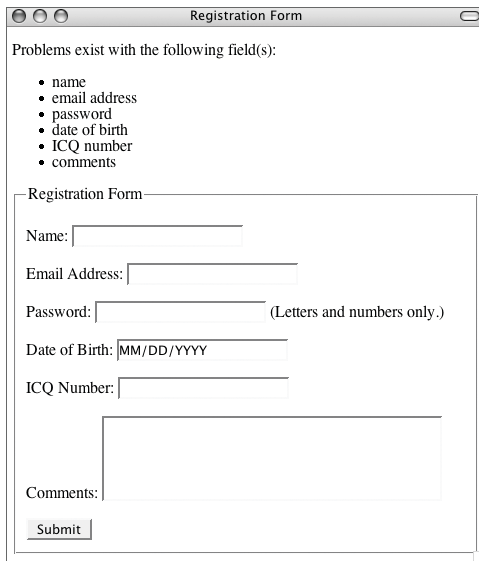


Figure 4.3 A lot of mistakes were made in this registration attempt, each reported back to the user.

- 11.** If there were no errors, report upon the success.

```
if (empty($errors)) {
    echo '<p>You have successfully
    → registered (but not
    → really).</p></body></html>';
    exit();
}
```

If no errors occurred, then `$errors` would still be empty. The script could then register the user (probably in a database). Here it just prints a message and terminates the script (so that the form isn't redisplayed) instead (**Figure 4.2**).

- 12.** Report the errors.

```
} else {
    echo '<p>Problems exist with
    → the following field(s):<ul>';
    foreach ($errors as $error) {
        echo
    → "<li>$error</li>\n";
    }
    echo '</ul></p>';
}
```

If `$errors` isn't empty, it contains all of the fields that failed a validation test. These can be printed in a list (**Figure 4.3**).

- 13.** Complete the main conditional and the PHP code.

```
} // End of $_POST['submitted'] IF.
?>
```

continues on next page

14. Create the HTML form.

```

<form method="post">
<fieldset>
<legend>Registration Form</legend>
<p>Name: <input type="text"
→ name="name" /></p>
<p>Email Address: <input type="text"
→ name="email" /></p>
<p>Password: <input type="password"
→ name="pass" /> (Letters and
→ numbers only.)</p>
<p>Date of Birth: <input type="text"
→ name="dob" value="MM/DD/YYYY" />
→ </p>
<p>ICQ Number: <input type="text"
→ name="icq" /></p>
<p>Comments: <textarea
→ name="comments" rows="5"
→ cols="40"></textarea></p>
<input type="hidden"
→ name="submitted" value="true" />
<input type="submit" name="submit"
→ value="Submit" />
</fieldset>
</form>

```

There's not much to say about the form except to point out that it does indicate the proper format for the password and date of birth fields. If you are validating data to a specification, it's important that the end user be made aware of the requirements as well, prior to submitting the form.

15. Complete the page.

```

</body>
</html>

```

16. Save the file as `register.php`, place it in your Web directory, and test in your Web browser.

Using Captcha

Popular in many of today's forms is *captcha*, short for “completely automated public Turing test to tell computers and humans apart” (now *that's* an acronym!). A captcha test displays an image with a word or some letters written in it, normally in a nonlinear fashion. In order to successfully complete the form, the text from the image has to be typed into a box. This is something a human user could do but a bot could not.

If you do want to add this feature to your own sites, using the PEAR Text_CAPTCHA package would be the easiest route. Otherwise, you could generate the images yourself using the GD library. The word on the image should be stored in a session so that it can be compared against what the user typed.

The main caveat with captcha tests is that they do restrict the visually impaired from completing that form. You should be aware of this, and provide alternatives. Personally, I think that bots can be effectively stopped by just adding another input to your form, with an easy-to-answer question (like “What is 2 + 2?”). Humans can submit the answer, whereas bots could not.

✓ Tips

- If possible, use the POST method in your forms. POST has a limitation in that the resulting page cannot be bookmarked, but it is far more secure and does not have the limit on transmittable data size that GET does. If a user is entering passwords, you really must use the POST method lest the password be visible.
- Placing hidden values in HTML forms can be a great way to pass information from page to page without using cookies or sessions. But be careful what you hide in your HTML code, because those hidden values can be seen by viewing a page's source. This technique is a convenience, not a security measure.
- Similarly, you should not be too obvious or reliant upon information PHP passes via the URL. For example, if a `homepage.php` page requires receipt of a user ID—and that is the only mandatory information for access to the account—someone else could easily break in (e.g., `www.example.com/userhome.php?user=2` could quickly be turned into `www.example.com/userhome.php?user=3`, granting access to someone else's information).

Using PECL Filter

New in PHP 5 and quite promising is the Filter library of PECL code. Being developed by PHP's creator and other major contributors, the future of Filter looks bright, even though it's still in beta form (at the time of this writing). The Filter package provides two types of security:

- ◆ Data validation by type
- ◆ Data sanitization

What Filter offers is a unified interface for performing common types of validation and sanitization. For example, I might commonly use code like this:

```
if (isset($_GET['id'])) {  
    if (is_numeric($_GET['id'])) {  
        $id = (int) $_GET['id'];  
        if ($id > 0) {  
            // Do whatever.  
        }  
    }  
}
```

I could instead do this:

```
$id = filter_input(INPUT_GET, 'id',  
    → FILTER_VALIDATE_INT, array('options'  
    → =>array('min_range'=>1)));  
if ($id) { ...
```

That might look like jabberwocky, but once you get the hang of Filter, the amount of work you can do in just a line of code will be worth the learning curve.

Figure 4.4 This new registration form lacks the password and date of birth inputs.

To filter individual variables, there are two functions you’ll use: `filter_input()` and `filter_var()`. The first one is for working with variables coming from an outside source, like forms, cookies, sessions, and the server. The second is for variables within your own code. I’ll focus on `filter_input()` here. Its syntax is:

```
$var = filter_input($variable_source,  
→ $variable_name, $filter, $options);
```

The sources, which the PHP manual calls “types,” are: `INPUT_GET`, `INPUT_POST`, `INPUT_COOKIE`, `INPUT_SERVER`, `INPUT_ENV`, `INPUT_SESSION`, and `INPUT_REQUEST`. As you can probably guess, each of these corresponds to a global variable (`$_GET`, `$_POST`, etc.). For example, if a page receives data in the URL, you’d use `INPUT_GET` (not `$_GET`).

The second argument—the variable name—is the specific variable within the source that should be addressed. The `$filter` argument indicates the filter to apply, using the constants in **Table 4.2**. This argument is optional, as a default filter will be used if none is specified. Some filters also take options, like the `FILTER_VALIDATE_INT` in the preceding example (which can take a range).

The `filter_input()` function will return the filtered variable if the filtration or validation was successful, the Boolean `FALSE` if the filter didn’t apply to the data, or the value `NULL` if the named variable didn’t exist in the given input. Thus you have multiple levels of validation in just one step.

There’s really a lot of information packed into just a few functions here, but I want to present a sample of how you would use the Filter library. To do so, I’ll create a modified version of the registration form (**Figure 4.4**). Note that as of PHP 5.2, Filter is built into PHP. If you’re using an earlier version, you may need to install it using the pecl installer (see the PHP manual for more).

Table 4.2 These constants represent some of the filters that can be applied to data. For a complete list, see the PHP manual or invoke the `filter_list()` function.

Filters by Name	
CONSTANT NAME	ACTION
<code>FILTER_VALIDATE_INT</code>	Confirms an integer, optionally in a range
<code>FILTER_VALIDATE_FLOAT</code>	Confirms a float
<code>FILTER_VALIDATE_REGEXP</code>	Matches a PCRE pattern
<code>FILTER_VALIDATE_URL</code>	Matches a URL
<code>FILTER_VALIDATE_EMAIL</code>	Matches an email address
<code>FILTER_SANITIZE_STRING</code>	Strips tags
<code>FILTER_SANITIZE_ENCODED</code>	URL-encodes a string

To use PECL Filter:

1. Begin a new PHP script in your text editor or IDE, starting with the HTML (**Script 4.2**).

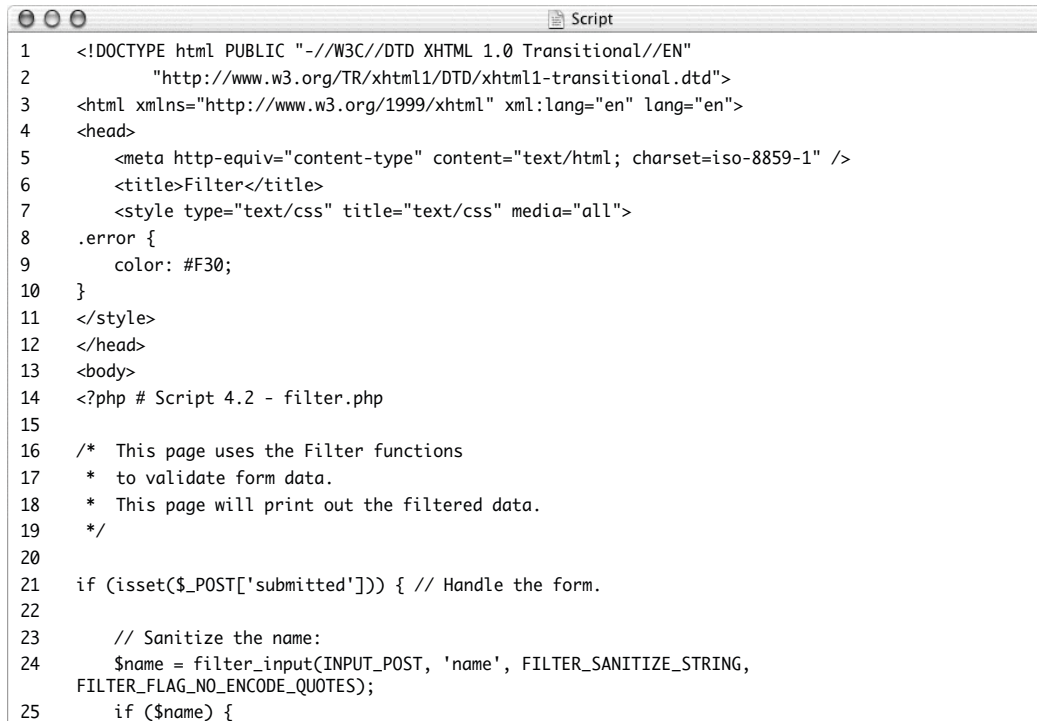
```
<!DOCTYPE html PUBLIC "-//W3C//DTD
→ XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/
→ xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/
→ xhtml" xml:lang="en" lang="en">
<head>
    <meta http-equiv="content-type"
→ content="text/html; charset=iso-
→ 8859-1" />
```

```
<title>Filter</title>
<style type="text/css"
→ title="text/css" media="all">
.error {
    color: #F30;
}
</style>
</head>
<body>
<?php # Script 4.2 - filter.php
```

The script has one CSS class for printing errors in a different color.

continues on page 140

Script 4.2 With this minimalist registration form, the Filter library is used to perform data validation and sanitization.



```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
4 <head>
5     <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
6     <title>Filter</title>
7     <style type="text/css" title="text/css" media="all">
8     .error {
9         color: #F30;
10    }
11 </style>
12 </head>
13 <body>
14 <?php # Script 4.2 - filter.php
15
16 /* This page uses the Filter functions
17  * to validate form data.
18  * This page will print out the filtered data.
19  */
20
21 if (isset($_POST['submitted'])) { // Handle the form.
22
23     // Sanitize the name:
24     $name = filter_input(INPUT_POST, 'name', FILTER_SANITIZE_STRING,
25         FILTER_FLAG_NO_ENCODE_QUOTES);
26     if ($name) {
```

(script continues on next page)

Script 4.2 *continued*

```

26     echo "<p>Name: $name<br />\$_POST['name']: {\$_POST['name']}</p>\n";
27 } else {
28     echo '<p class="error">Please enter your name.</p>';
29 }
30
31 // Validate the email address using FILTER_VALIDATE_EMAIL:
32 $email = filter_input(INPUT_POST, 'email', FILTER_VALIDATE_EMAIL);
33 if ($email) {
34     echo "<p>Email Address: $email</p>\n";
35 } else {
36     echo '<p class="error">Please enter your email address.</p>';
37 }
38
39 // Validate the ICQ number using FILTER_VALIDATE_INT:
40 $icq = filter_input(INPUT_POST, 'icq', FILTER_VALIDATE_INT);
41 if ($icq) {
42     echo "<p>ICQ Number: $icq</p>\n";
43 } else {
44     echo '<p class="error">Please enter your ICQ number.</p>';
45 }
46
47 // Strip tags but don't encode quotes:
48 $comments = filter_input(INPUT_POST, 'comments', FILTER_SANITIZE_STRING);
49 if ($comments) {
50     echo "<p>Comments: $comments<br />\$_POST['comments']: {\$_POST['comments']}</p>\n";
51 } else {
52     echo '<p class="error">Please enter your comments.</p>';
53 }
54
55 } // End of $_POST['submitted'] IF.
56
57 // Show the form.
58 ?>
59 <form method="post" action="filter.php">
60 <fieldset>
61 <legend>Registration Form</legend>
62 <p>Name: <input type="text" name="name" /></p>
63 <p>Email Address: <input type="text" name="email" /></p>
64 <p>ICQ Number: <input type="text" name="icq" /></p>
65 <p>Comments: <textarea name="comments" rows="5" cols="40"></textarea></p>
66
67 <input type="hidden" name="submitted" value="true" />
68 <input type="submit" name="submit" value="Submit" />
69 </fieldset>
70 </form>
71
72 </body>
73 </html>

```

2. Check for the form submission.

```
if (isset($_POST['submitted'])) {
```

3. Filter the name data.

```
$name = filter_input(INPUT_POST,
    → 'name', FILTER_SANITIZE_STRING,
    → FILTER_FLAG_NO_ENCODE_QUOTES);
```

For the name field, there's no type to validate against, but it can be filtered to remove any HTML tags. The `FILTER_SANITIZE_STRING` filter will accomplish that. The last argument, `FILTER_FLAG_NO_ENCODE_QUOTES`, says that any quotation marks in the name (e.g., O'Toole) shouldn't be turned into an HTML entity equivalent.

4. Print the name value or an error.

```
if ($name) {
    echo "<p>Name: $name<br
    → />\$_POST['name']:"
    → {\$_POST['name']}</p>\n";
} else {
    echo '<p class="error">Please
    → enter your name.</p>';
}
```

The conditional `if ($name)` will be true if the `$_POST['name']` variable was set and passed the filter. In that case, I'll print the filtered version and the original version, just for comparison.

5. Validate the email address.

```
$email = filter_input(INPUT_POST,
    → 'email', FILTER_VALIDATE_EMAIL);
if ($email) {
    echo "<p>Email Address:
    → $email</p>\n";
} else {
    echo '<p class="error">Please
    → enter your email address.</p>';
}
```

The `FILTER_VALIDATE_EMAIL` filter is perfect here. If the submitted email address has a valid format, it will be returned. Otherwise, `$email` will equal either `FALSE` or `NULL`.

6. Validate the ICQ number.

```
$icq = filter_input(INPUT_POST,
    → 'icq', FILTER_VALIDATE_INT);
if ($icq) {
    echo "<p>ICQ Number:
    → $icq</p>\n";
} else {
    echo '<p class="error">Please
    → enter your ICQ number.</p>';
}
```

This is validated as an integer.

7. Filter the comments field.

```
$comments = filter_input(INPUT_POST,
    → 'comments', FILTER_SANITIZE_
    → STRING);
if ($comments) {
    echo "<p>Comments: $comments<br
    → />\$_POST['comments']:"
    → {\$_POST['comments']}</p>\n";
} else {
    echo '<p class="error">Please
    → enter your comments.</p>';
}
```

For the comments, any tags will be stripped (as with the name), but the quotation marks will also be encoded.

8. Complete the main conditional and the PHP code.

```
} // End of $_POST['submitted'] IF.
?>
```

Figure 4.5 These values will be submitted, then filtered, resulting in Figure 4.6.

Figure 4.6 At the top of the form the filtered values are displayed.

9. Create the HTML form.

```
<form method="post"
→ action="filter.php">
<fieldset>
<legend>Registration Form</legend>
<p>Name: <input type="text"
→ name="name" /></p>
<p>Email Address: <input type="text"
→ name="email" /></p>
<p>ICQ Number: <input type="text"
→ name="icq" /></p>
<p>Comments: <textarea
→ name="comments" rows="5"
→ cols="40"></textarea></p>
<input type="hidden"
→ name="submitted" value="true" />
<input type="submit" name="submit"
→ value="Submit" />
</fieldset>
</form>
```

10. Complete the page.

```
</body>
</html>
```

11. Save the file as `filter.php`, place it in your Web directory, and test in your Web browser (**Figures 4.5** and **4.6**).

12. View the HTML source of the page to see how the name and comments fields were treated (**Figure 4.7**).

continues on next page

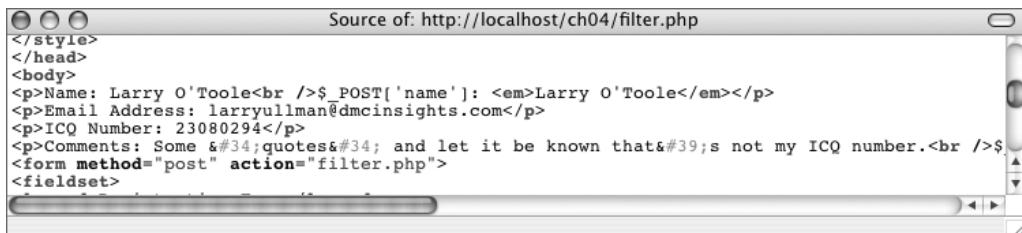


Figure 4.7 The HTML source code shows how all tags are stripped from the name and comments fields, plus how quotation marks in the comments are encoded.

✓ Tips

- The `filter_has_var()` function checks to see if a variable with a given name exists within a greater array of variables. In this script, you could use this code to see if the form has been submitted:

```
if (filter_has_var(INPUT_POST,  
→ 'submitted')) {
```

- To filter an array of variables, use `filter_input_array()`. In `filter.php`, you could just do this:

```
$filters = array(  
    'name' => FILTER_SANITIZE_STRING,  
    'email' => FILTER_VALIDATE_EMAIL,  
    'icq' => FILTER_VALIDATE_INT,  
    'comments' => array('filter' =>  
→ FILTER_SANITIZE_STRING, 'flags' =>  
→ FILTER_FLAG_NO_ENCODE_QUOTES)  
);
```

```
$data = filter_input_array  
→ (INPUT_POST, $filters);
```

From that point, you could just refer to `$data['name']`, etc.

- The `filter_var_array()` applies a filter, or an array of filters, to an array of data.

Authentication with PEAR Auth

One of the more common elements in today's Web sites is an authentication system: users register with a site, they log in to gain access to some parts, and restricted pages allow or deny access accordingly. Such systems aren't hard to implement—I've done so in some of my other books—but here I'd like to look at what PEAR has to offer.

The PEAR Auth package provides a really easy, yet customizable authentication system. To show it off, I'll start with one very simple example. This will mostly demonstrate its basic usage. Then I'll show how to customize the authentication system to fit it into a larger application. For both examples, you'll need to install the PEAR Auth package. Because the authentication information is stored in a database, the PEAR DB package must also be installed. If you're not familiar with PEAR and its installation, see Chapter 12, "Using PEAR," or <http://pear.php.net>.

✓ Tip

- For these examples I will put both the authentication code and the restricted page data in the same file. In a larger Web site, you'll likely want to separate the authentication code into its own file, which is then included by any file that requires authentication.

Simple authentication

This first, simple authentication example shows how easily you can implement authentication in a page. I'll run through the syntax and concepts first, and then create a script that executes it all.

To begin, require the Auth class:

```
require_once ('Auth.php');
```

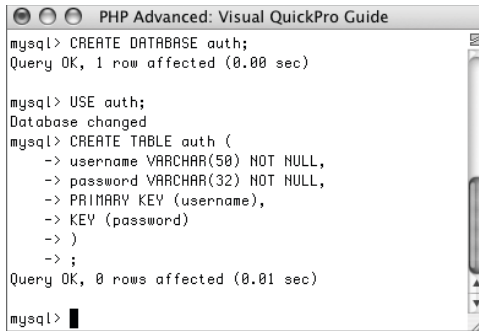
Next, you'll need to define a function that creates a login form. This function will be called when an unauthorized user is trying to access a page. The form should use the POST method and have inputs called *username* and *password*.

Then, for database-driven authentication, which is the norm, you'll need to create a "DSN" within an options array. DSN stands for *data source name*. It's just a string of information that indicates the type of database application being used, the username, password, and hostname to connect as, and the database to select. That code might be:

```
$options = array('dsn' =>
→ 'mysql://username:password@localhost/
→ databasename');
```

Now that those two things have been defined—the function that makes the login form and the DSN—you can create an object of Auth type. Provide this object three arguments: the type of authentication back end to use (e.g., database or file), the options (that correspond to the authentication type), and the name of the login function:

```
$auth = new Auth('DB', $options,
→ 'login_form_function_name');
```



```

mysql> CREATE DATABASE auth;
Query OK, 1 row affected (0.00 sec)

mysql> USE auth;
Database changed
mysql> CREATE TABLE auth (
  -> username VARCHAR(50) NOT NULL,
  -> password VARCHAR(32) NOT NULL,
  -> PRIMARY KEY (username),
  -> KEY (password)
  -> )
  -> ;
Query OK, 0 rows affected (0.01 sec)

mysql>

```

Figure 4.8 Creating the database and table required by the simple authentication example.

The *DB* option tells Auth to use the PEAR DB package. If you wanted to use a file system instead, you would use *File* as the first argument and the name of the file as the second.

Now, start the authentication process:

```
$auth->start();
```

From there, you can check if a user is authenticated by calling the `checkAuth()` method:

```
if ($auth->checkAuth()) {
    // Do whatever.

```

And that's simple authentication in a nutshell! This next example will implement all this. It will also invoke the `addUser()` method to add a new authenticated user, which can then be used for logging in. One last note: this example will make use of a database called `auth`, which must be created prior to writing this script. It should have a table called `auth`, defined like so:

```

CREATE TABLE auth (
  username VARCHAR(50) NOT NULL,
  password VARCHAR(32) NOT NULL,
  PRIMARY KEY (username),
  KEY (password)
)

```

Be certain that you've created this database and table (**Figure 4.8**), and that you have created a MySQL user that has access to them, prior to going any further.

To perform simple authentication:

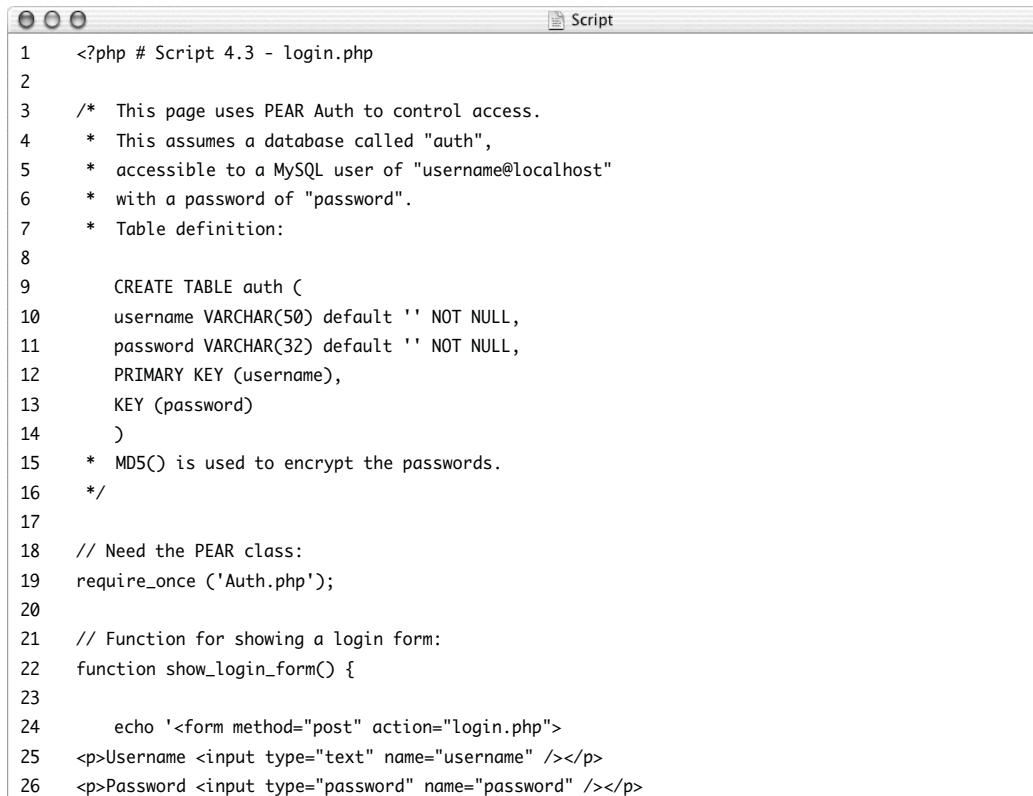
1. Begin a new PHP script in your text editor or IDE (**Script 4.3**).

```
<?php # Script 4.3 - login.php
```

Because Auth relies on sessions (it'll start the sessions for you), it's best to do as much as you can before sending any HTML to the Web browser. So I'll write most of the authentication code, and only then begin the HTML page.

continues on page 148

Script 4.3 Using PEAR Auth and a MySQL table, this script enforces authentication.



```

1  <?php # Script 4.3 - login.php
2
3  /* This page uses PEAR Auth to control access.
4   * This assumes a database called "auth",
5   * accessible to a MySQL user of "username@localhost"
6   * with a password of "password".
7   * Table definition:
8
9   CREATE TABLE auth (
10    username VARCHAR(50) default '' NOT NULL,
11    password VARCHAR(32) default '' NOT NULL,
12    PRIMARY KEY (username),
13    KEY (password)
14  )
15  * MD5() is used to encrypt the passwords.
16  */
17
18  // Need the PEAR class:
19  require_once ('Auth.php');
20
21  // Function for showing a login form:
22  function show_login_form() {
23
24      echo '<form method="post" action="login.php">
25      <p>Username <input type="text" name="username" /></p>
26      <p>Password <input type="password" name="password" /></p>
  
```

(script continues on next page)

Script 4.3 *continued*

```

27 <input type="submit" value="Login" />
28 </form><br />
29 ';
30
31 } // End of show_login_form() function.
32
33 // Connect to the database:
34 $options = array('dsn' => 'mysql://username:password@localhost/auth');
35
36 // Create the Auth object:
37 $auth = new Auth('DB', $options, 'show_login_form');
38
39 // Add a new user:
40 $auth->addUser('me', 'mypass');
41
42 ?><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
43     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
44 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
45 <head>
46     <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
47     <title>Restricted Page</title>
48 </head>
49 <body>
50 <?php
51
52 // Start the authorization:
53 $auth->start();
54
55 // Confirm authorization:
56 if ($auth->checkAuth()) {
57
58     echo '<p>You are logged in and can read this. How cool is that?</p>';
59
60 } else { // Unauthorized.
61
62     echo '<p>You must be logged in to access this page.</p>';
63
64 }
65
66 ?>
67 </body>
68 </html>

```

2. Include the Auth class.

```
require_once ('Auth.php');
```

If you haven't installed PEAR Auth yet, do so now. See the PEAR manual for instructions.

3. Define the function that creates the login form.

```
function show_login_form() {
    echo '<form method="post"
    → action="login.php">
    <p>Username <input type="text"
    → name="username" /></p>
    <p>Password <input type="password"
    → name="password" /></p>
    <input type="submit" value="Login" />
    </form><br />
    ';
}
```

The only requirements are that this form has one input called *username* and another called *password*.

4. Create the options array.

```
$options = array('dsn' =>
    → 'mysql://username:password@
    → localhost/auth');
```

This code says that a connection should be made to a MySQL database called *auth*, using *username* as the username, *password* as the password, and *localhost* as the host.

5. Create the Auth object.

```
$auth = new Auth('DB', $options,
    → 'show_login_form');
```

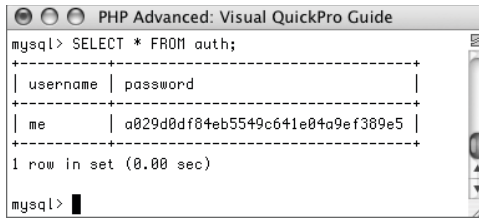


Figure 4.9 One user has been added to the table. The password is encrypted using the MD5() function.

6. Add a new user and complete the PHP section.

```
$auth->addUser('me', 'mypass');
?>
```

The `addUser()` function takes the username as its first argument and the password as the second. This record will be added to the database as soon as the script is first run (**Figure 4.9**). Because the `username` column in the table is defined as a primary key, MySQL will never allow a second user with the name of *me* to be added.

In a real application, you'd have a registration process that would just end up calling this function in the end.

7. Add the initial HTML code.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD
→ XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/
→ xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/
→ xhtml" xml:lang="en" lang="en">
<head>
    <meta http-equiv="content-type"
→ content="text/html; charset=iso-
→ 8859-1" />
    <title>Restricted Page</title>
</head>
<body>
```

8. Start the authentication.

```
<?php
$auth->start();
```

continues on next page

9. Display different messages based upon the authentication status.

```
if ($auth->checkAuth()) {
    echo '<p>You are logged in and
    → can read this. How cool is
    → that?</p>';
} else {
    echo '<p>You must be logged in
    → to access this page.</p>';
}
```

When a user first comes to this page, and `$auth->checkAuth()` is false, they'll see the login form plus this second message (**Figure 4.10**). After logging in with a valid username/password combination, they'll see this first message (**Figure 4.11**).

10. Complete the page.

```
?>
</body>
</html>
```

11. Save the file as `login.php`, place it in your Web directory, and test in your Web browser.

Use *me* as the username and *mypass* as the password.

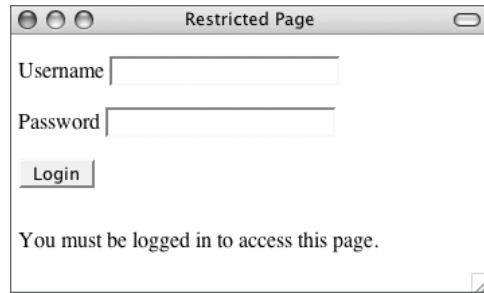


Figure 4.10 When first arriving at this page, or after an unsuccessful login attempt, a user sees this.

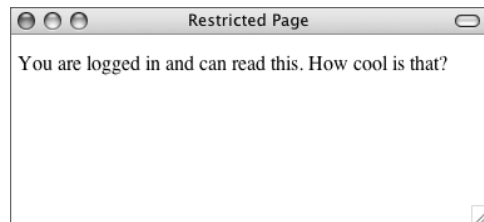


Figure 4.11 The result after successfully logging in.

Implementing Optional Authentication

If some of your Web site's pages do not require authentication but could still acknowledge logged-in users, that's an option with Auth, too. To make authentication optional, add a fourth parameter when creating the Auth object:

```
$auth = new Auth('DB', $options,
    → 'show_login_form', true);
```

To limit aspects of a page to authenticated users, invoke the `getAuth()` method:

```
if ($auth->getAuth()) {
    // Restricted access content.
}
```


Custom authentication

The preceding example does a fine job of showing how easy it is to use PEAR Auth, but it doesn't demonstrate how you would actually use it in a more full-fledged application. By this I mean a site that has a table with more than two columns and needs to store, and retrieve, other information as well.

The first change you'll need to make is to the options array used when creating the Auth object. Different storage types ("containers" in Auth parlance) have different options. **Table 4.3** lists some of the other options you can use with DB.

For example, the DB container will use a combination of the *usernamecol* and *passwordcol* (encrypted using *cryptType*) to authenticate the user against the submitted values. The preceding example used the defaults, but you can change this information easily. Just as important, you can specify what other database columns should be retrieved. These will then be available in the session data and can be retrieved in your script through the `getAuthData()` function:

```
echo $auth->getAuthData('column_name');
```

Table 4.3 These are some of the parameters you can set when creating a new Auth object that uses DB.

DB Container Options	
OPTION	INDICATES
dsn	The Data Source Name
table	The database table to use
usernamecol	The name of the username column
passwordcol	The name of the password column
db_fields	What other table fields should be selected
cryptType	The function used to encrypt the password

Three other functions you can use to customize the authentication are `setExpire()`, `setIdle()`, and `setSessionName()`. The first takes a value, in seconds, when the session should be set to expire. The second takes a value, in seconds, when a user should be considered idle (because it's been too long since their last activity). The third function changes the name of the session (which is *PHPSESSID*, by default).

For this next example, a new table will be used, still in the auth database. To create it, use this SQL command (**Figure 4.12**):

```
CREATE TABLE users (
  user_id INT UNSIGNED NOT NULL
  → AUTO_INCREMENT,
  email VARCHAR(60) NOT NULL,
  pass CHAR(40) NOT NULL,
  first_name VARCHAR (20) NOT NULL,
  last_name VARCHAR(40) NOT NULL,
  PRIMARY KEY (user_id),
  UNIQUE (email),
  KEY (email, pass)
)
```

This table represents how you might already have some sort of user table, with its own columns, that you'd want to use with Auth.

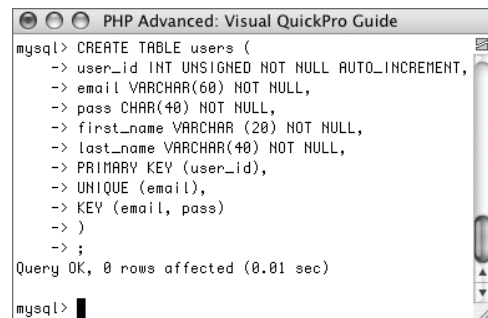


Figure 4.12 Creating the table used by the custom authentication system.

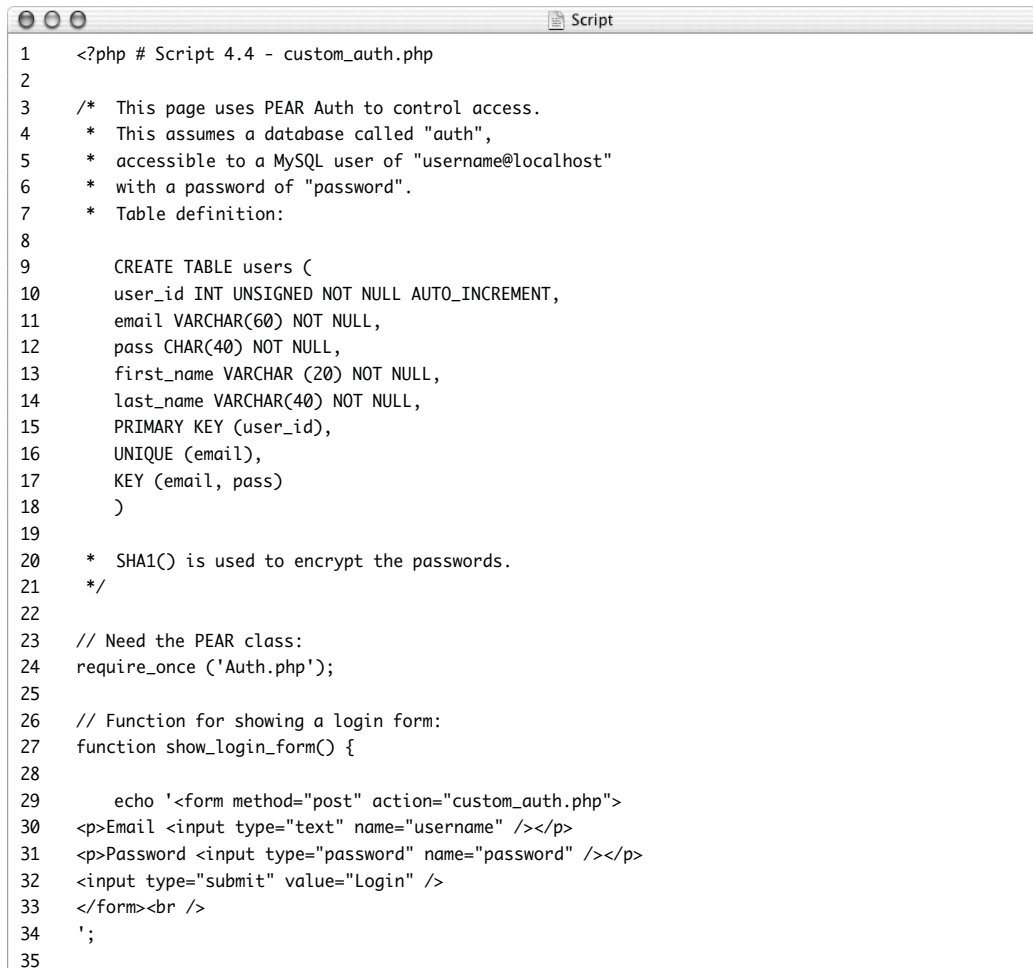
To use custom authentication:

1. Begin a new PHP script in your text editor or IDE, starting with the HTML (**Script 4.4**).

```
<?php # Script 4.4 - custom_auth.php
require_once ('Auth.php');
```

continues on page 154

Script 4.4 In this script, Auth uses a different table, different column names, and a different encryption function for the password. It selects every column from the table, making all the previously stored data available to the page.



```
1  <?php # Script 4.4 - custom_auth.php
2
3  /* This page uses PEAR Auth to control access.
4   * This assumes a database called "auth",
5   * accessible to a MySQL user of "username@localhost"
6   * with a password of "password".
7   * Table definition:
8
9   CREATE TABLE users (
10     user_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
11     email VARCHAR(60) NOT NULL,
12     pass CHAR(40) NOT NULL,
13     first_name VARCHAR (20) NOT NULL,
14     last_name VARCHAR(40) NOT NULL,
15     PRIMARY KEY (user_id),
16     UNIQUE (email),
17     KEY (email, pass)
18   )
19
20   * SHA1() is used to encrypt the passwords.
21   */
22
23 // Need the PEAR class:
24 require_once ('Auth.php');
25
26 // Function for showing a login form:
27 function show_login_form() {
28
29     echo '<form method="post" action="custom_auth.php">
30     <p>Email <input type="text" name="username" /></p>
31     <p>Password <input type="password" name="password" /></p>
32     <input type="submit" value="Login" />
33     </form><br />
34     ';
35 }
```

(script continues on next page)

Script 4.4 *continued*

```

36 } // End of show_login_form() function.
37
38 // All options:
39 // Use specific username and password columns.
40 // Use SHA1() to encrypt the passwords.
41 // Retrieve all fields.
42 $options = array(
43     'dsn' => 'mysql://username:password@localhost/auth',
44     'table' => 'users',
45     'usernamecol' => 'email',
46     'passwordcol' => 'pass',
47     'cryptType' => 'sha1',
48     'db_fields' => '*'
49 );
50
51 // Create the Auth object:
52 $auth = new Auth('DB', $options, 'show_login_form');
53
54 // Add a new user:
55 $auth->addUser('me@example.com', 'mypass', array('first_name' => 'Larry', 'last_name' =>
    'Ullman'));
56
57 ?><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
58     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
59 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
60 <head>
61     <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
62     <title>Restricted Page</title>
63 </head>
64 <body>
65 <?php
66
67 // Start the authorization:
68 $auth->start();
69
70 // Confirm authorization:
71 if ($auth->checkAuth()) {
72
73     // Print the user's name:
74     echo "<p>You, {"$auth->getAuthData('first_name')} {"$auth->getAuthData('last_name')}, are
    logged in and can read this. How cool is that?</p>";
75
76 } else { // Unauthorized.
77
78     echo '<p>You must be logged in to access this page.</p>';
79
80 }
81
82 ?>
83 </body>
84 </html>

```

2. Define the `show_login_form()` function.

```
function show_login_form() {
    echo '<form method="post"
    → action="custom_auth.php">
    <p>Email <input type="text"
    → name="username" /></p>
    <p>Password <input type="password"
    → name="password" /></p>
    <input type="submit" value="Login" />
    </form><br />
    ';
}
```

The function is mostly the same as it was before, except this time the action points to this script, `custom_auth.php`. The form also labels the one input as *Email* (**Figure 4.13**), even though it's named *username* (as required).

3. Establish the authorization options and create the object.

```
$options = array(
    'dsn' => 'mysql://username:password@
    → localhost/auth',
    'table' => 'users',
    'usernamecol' => 'email',
    'passwordcol' => 'pass',
    'cryptType' => 'sha1',
    'db_fields' => '*'
);
$auth = new Auth('DB', $options,
    → 'show_login_form');
```

The DSN is the same as it was before. Next, the *table*, *usernamecol*, and *passwordcol* values are all specified. These match the table already created (Figure 4.12). The *cryptType* value says that the passwords should be encoded using *SHA1()*, instead of the default *MD5()*. The final element in the `$options` array says that every column from the table should be retrieved. In this particular script, this will allow the page to refer to the logged-in user by name.

Figure 4.13 The customized login form.

Creating a Logout Feature

To add a logout to your authentication system, place this code on a logout page:

```
$auth = new Auth('DB', $options,
    → 'show_login_form');
$auth->start();
if ($auth->checkAuth()) {
    $auth->logout();
    $auth->start();
}
```

Just as when using sessions, you need to start the authentication in order to destroy it. You should then confirm that the user is authenticated, using `checkAuth()`, prior to logging out. Then call the `logout()` method to de-authenticate the user. Calling the `start()` method again will redisplay the login form.

4. Add a new user and complete the initial PHP section (**Figure 4.14**).

```
$auth->addUser('me@example.com',
→ 'mypass', array('first_name' =>
→ 'Larry', 'last_name' => 'Ullman'));
?>
```

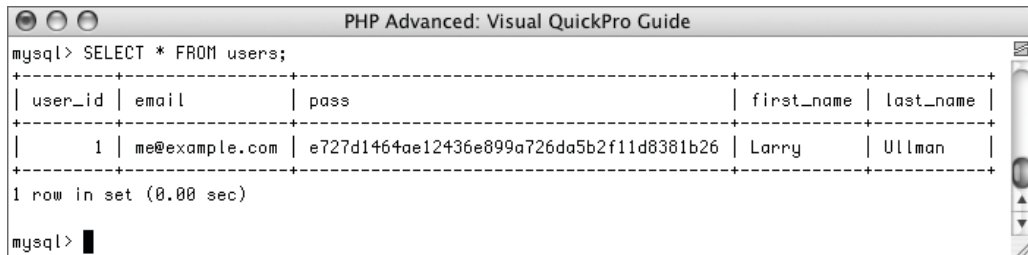
Because the table has more than just the two columns, the extra columns and values have to be provided, as an array, as the third argument to the `addUser()` method. This call of the function is the equivalent of running this query:

```
INSERT INTO users (email, pass,
→ first_name, last_name) VALUES
→ ('me@example.com', SHA1('mypass'),
→ 'Larry', 'Ullman')
```

5. Create the initial HTML code.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD
→ XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/
→ xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/
→ xhtml" xml:lang="en" lang="en">
<head>
    <meta http-equiv="content-type"
→ content="text/html; charset=iso-
→ 8859-1" />
    <title>Restricted Page</title>
</head>
<body>
```

continues on next page



```
mysql> SELECT * FROM users;
```

user_id	email	pass	first_name	last_name
1	me@example.com	e727d1464ae12436e899a726da5b2f11d8381b26	Larry	Ullman

```
1 row in set (0.00 sec)

mysql>
```

Figure 4.14 A sample user has been added to the users table.

6. Start the authorization.

```
<?php
$auth->start();
```

7. Print the authorization status.

```
if ($auth->checkAuth()) {
    echo "<p>You, {$auth->get
    → AuthData('first_name')}
    → {$auth->getAuthData('last_name')},
    → are logged in and can read this.
    → How cool is that?</p>";
} else {
    echo '<p>You must be logged in
    → to access this page.</p>';
}
```

The result if the user isn't logged in looks like Figure 4.13. When the user does log in, they are greeted by name (Figure 4.15). The `getAuthData()` function can access the values selected from the table and stored in the authentication session.

8. Complete the page.

```
?>
</body>
</html>
```

9. Save the file as `custom_auth.php`, place it in your Web directory, and test in your Web browser.

✓ Tips

- You can add, on the fly, other data to the authentication session using `setAuthData()`:
`setAuthData($name, $value);`
- You can also improve authentication security via the `setAdvancedSecurity()` method. It uses both cookies and JavaScript to lessen the possibility of someone hacking an authenticated session.

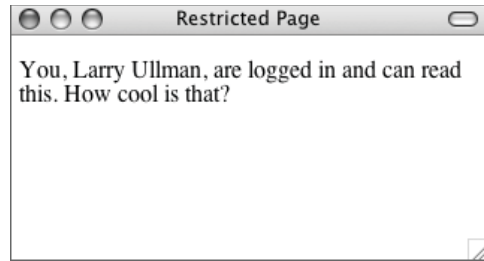


Figure 4.15 After successfully logging in, the user is greeted by name. The name was pulled from the table and stored in the session.

Using Auth_HTTP

One of the potential problems with Auth is that it relies upon sessions, which can introduce some security concerns. A more secure option is to use HTTP authentication via `Auth_HTTP`. HTTP authentication uses a pop-up window, separate from the HTML page, that takes a username and password.

The benefits of HTTP authentication are these:

- ◆ The entered username and password are remembered without needing to send cookies or establish sessions.
- ◆ The clean interface will not interfere with your page design.

The downsides are:

- ◆ Inability to create a logout feature
- ◆ Inability to establish user groups or specify access levels
- ◆ Inability to set an expiration time

mcrypt

mcrypt support	enabled	
Version	2.5.7	
Api No	20021217	
Supported ciphers	cast-128 gost rijndael-128 twofish arcfour cast-256 loki97 rijndael-192 saferplus wake blowfish-compat des rijndael-256 serpent xtea blowfish enigma rc2 tripledes	
Supported modes	cbc cfb ctr cfb nofb nofb ofb stream	
Directive	Local Value	Master Value
mcrypt.algorithms_dir	no value	no value
mcrypt.modes_dir	no value	no value

Figure 4.16 Run a `phpinfo()` script to confirm your server's support for MCrypt.

Using MCrypt

Frequently Web applications will encrypt and decrypt data stored in a database, using the database-supplied functions. This is appropriate, as you want the database to do the bulk of the work whenever possible. But what if you want to encrypt and decrypt data that's not being stored in a database? In that situation, MCrypt is the best solution. To use MCrypt with PHP, you'll need to install the MCrypt library (`libmcrypt`, available from <http://mcrypt.sourceforge.net>) and configure PHP to support it (**Figure 4.16**).

For this example, I'll show you how to encrypt data stored in a cookie, making it that much more secure. Because the encryption process creates binary data, the `base64_encode()` function will be applied to the encrypted data prior to storing it in a cookie. Therefore, the `base64_decode()` function needs to be used prior to decoding the data. Other than that little tidbit, the focus in the next two scripts is entirely on MCrypt.

Do keep in mind that in the next several pages I'll be introducing and teaching concepts to which people have dedicated entire careers. The information covered here will be secure, useful, and valid, but it's just the tip of the proverbial iceberg.

Encrypting data

With MCrypt libraries 2.4.x and higher, you start by identifying which algorithm and mode to use by invoking the `mcrypt_module_open()` function:

```
$m = mcrypt_module_open (algorithm,
→ algorithm_directory, mode,
→ mode_directory);
```

MCrypt comes with dozens of different algorithms, or *ciphers*, each of which encrypts data differently. If you are interested in how each works, see the MCrypt home page or search the Web. In my examples, I'll be using the Rijndael algorithm, also known as the Advanced Encryption Standard (AES). It's a very popular and secure encryption algorithm, even up to United States government standards. I'll be using it with 256-bit keys, for extra security.

As for the mode, there are four main modes: ECB (electronic codebook), CBC (cipher block chaining), CFB (cipher feedback), and OFB (output feedback). CBC will suit most of your needs, especially when encrypting blocks of text as in this example. So to indicate that you want to use Rijndael 256 in CBC mode, you would code:

```
$m = mcrypt_module_open('rijndael-
→ 256', '', 'cbc', '');
```

The second and fourth arguments fed to the `mcrypt_module_open()` function are for explicitly stating where PHP can find the algorithm and mode files. These are not required unless PHP is unable to find a cipher and you know for certain it is installed.

Once the module is open, you create an IV (initialization vector). This may be required, optional, or unnecessary depending upon the mode being used. I'll use it with CBC, to increase the security. Here's how the PHP manual recommends an IV be created:

```
$iv = mcrypt_create_iv
→ (mcrypt_enc_get_iv_size ($m),
→ MCRYPT_DEV_RANDOM);
```

By using the `mcrypt_enc_get_iv_size()` function, a properly sized IV will be created for the cipher being used. Note that on Windows, you should use `MCRYPT_RAND` instead of `MCRYPT_DEV_RANDOM`, and call the `srand()` function before this line to ensure the random generation.

The final step before you are ready to encrypt data is to create the buffers that MCrypt needs to perform encryption: `mcrypt_generic_init ($m, $key, $iv);`

The second argument is a key, which should be a hard-to-guess string. The key must be of a particular length, corresponding to the cipher you use. The Rijndael cipher I'm using takes a 256-bit key. Divide 256 by 8 (because there are 8 bits in a byte and each character in the key string takes one byte) and you'll see that the key needs to be exactly 32 characters long. To accomplish that, and to randomize the key even more, I'll run it through `MD5()`, which always returns a 32-character string:

```
$key = MD5('some string');
```

Once you have gone through these steps, you are ready to encrypt data:

```
$encrypted_data = mcrypt_generic ($m,
→ $data);
```


Finally, after you have finished encrypting everything, you should close all the buffers and modules:

```
mcrypt_generic_deinit ($m);  
mcrypt_module_close($m);
```

For this example, I'm going to create a cookie whose value is encrypted. The cookie data will be decrypted in the next example. The key and data to be encrypted will be hard-coded into this script, but I'll mention alternatives in the following steps. Also, because the same key and IV are needed to decrypt the data, the IV will also be sent in a cookie. Surprisingly, doing so doesn't hurt the security of the application.

To encrypt data:

1. Begin a new PHP script in your text editor or IDE (**Script 4.5**).

```
<?php # Script 4.5 -  
→ set_mcrypt_cookie.php
```

Because the script will send two cookies, most of the PHP code will come before any HTML.

2. Define the key and the data.

```
$key = md5('77 public drop-shadow  
→ Java');  
$data = 'rosebud';
```

For the key, some random words and numbers are run through the MD5() function, creating a 32-character-long string. Ideally, the key should be stored in a safe place, such as a configuration file located outside of the Web document root. Or it could be retrieved from a database.

The data being encrypted is the word *rosebud*, although in real applications this data might come from the database or another source (and be something more worth protecting).

3. Open the cipher.

```
$m = mcrypt_module_open('rijndael-  
→ 256', '', 'cbc', '');
```

This is the same code outlined in the text before these steps.

continues on page 162

Script 4.5 This script uses MCrypt to encrypt some data to be stored in a cookie.

```

1  <?php # Script 4.5 - set_mcrypt_cookie.php
2
3  /* This page uses the MCrypt library
4   * to encrypt some data.
5   * The data will then be stored in a cookie,
6   * as will the encryption IV.
7   */
8
9  // Create the key:
10 $key = md5('77 public drop-shadow Java');
11
12 // Data to be encrypted:
13 $data = 'rosebud';
14
15 // Open the cipher:
16 // Using Rijndael 256 in CBC mode.
17 $m = mcrypt_module_open('rijndael-256', '', 'cbc', '');
18
19 // Create the IV:
20 // Use MCRYPT_RAND on Windows instead of MCRYPT_DEV_RANDOM.
21 $iv = mcrypt_create_iv(mcrypt_enc_get_iv_size($m), MCRYPT_DEV_RANDOM);
22
23 // Initialize the encryption:
24 mcrypt_generic_init($m, $key, $iv);
25
26 // Encrypt the data:
27 $data = mcrypt_generic($m, $data);
28
29 // Close the encryption handler:
30 mcrypt_generic_deinit($m);
31
32 // Close the cipher:
33 mcrypt_module_close($m);
34
35 // Set the cookies:
36 setcookie('thing1', base64_encode($data));
37 setcookie('thing2', base64_encode($iv));
38 ?><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
39     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
40 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
41 <head>
42     <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
43     <title>A More Secure Cookie</title>
44 </head>
45 <body>
46 <p>The cookie has been sent. Its value is '<?php echo base64_encode($data); ?>'.</p>
47 </body>
48 </html>

```

4. Create the IV.

```
$iv = mcrypt_create_iv
→ (mcrypt_enc_get_iv_size($m),
→ MCRYPT_DEV_RANDOM);
```

Again, this is the same code outlined earlier. Remember that if you are running this script on Windows, you'll need to change this line to:

```
srand();
$iv = mcrypt_create_iv
→ (mcrypt_enc_get_iv_size($m),
→ MCRYPT_DEV_RANDOM);
```

5. Initialize the encryption.

```
mcrypt_generic_init($m, $key, $iv);
```

6. Encrypt the data.

```
$data = mcrypt_generic($m, $data);
```

If you were to print the value of `$data` now, you'd see something like

Figure 4.17.

7. Perform the necessary cleanup.

```
mcrypt_generic_deinit($m);
mcrypt_module_close($m);
```

8. Send the two cookies.

```
setcookie('thing1',
→ base64_encode($data));
setcookie('thing2',
→ base64_encode($iv));
```

For the cookie names, I'm using meaningless values. You certainly wouldn't want to use, say, *IV*, as a cookie name! For the cookie data itself, you have to run it through `base64_encode()` to make it safe to store in a cookie. This applies to both the encrypted data and the IV (which is also in binary format).

If the data were going to be stored in a binary file or in a database (in a `BLOB` column), you wouldn't need to use `base64_encode()`.



Figure 4.17 This gibberish is the encrypted data in binary form.



Figure 4.18 The result of running the page.

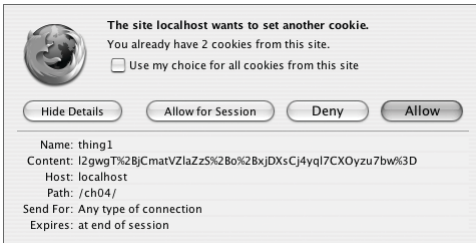


Figure 4.19 The first cookie stores the actual data.

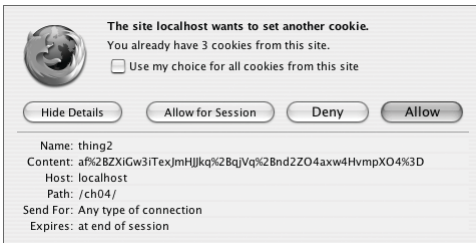


Figure 4.20 The second cookie stores the base64_encode() version of the IV.

9. Add the HTML head.

```
?><!DOCTYPE html PUBLIC "-//W3C//DTD
→ XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/
→ xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/
→ xhtml" xml:lang="en" lang="en">
<head>
    <meta http-equiv="content-type"
→ content="text/html; charset=iso-
→ 8859-1" />
    <title>A More Secure
→ Cookie</title>
</head>
<body>
```

10. Print a message, including the encoded, encrypted version of the data.

```
<p>The cookie has been sent. Its
→ value is '<?php echo
→ base64_encode($data); ?>'.</p>
```

I'm doing this mostly so that the page shows something (**Figure 4.18**), but also so that you can see the value stored in the cookie.

11. Complete the page.

```
</body>
</html>
```

12. Save the file as set_mcrypt_cookie.php, place it in your Web directory, and test in your Web browser.

If you set your browser to show cookies being sent, you'll see the values when you run the page (**Figures 4.19** and **4.20**).

✓ Tips

- If you want to determine the length of the key on the fly, use the `mcrypt_end_get_key_size()` function:
`$ks = mcrypt_end_get_key_size($m);`

- There's an argument to be made that you *shouldn't* apply the MD5() function to the key because it actually decreases the security of the key. I've used it here regardless, but it's the kind of issue that legitimate cryptographers think about.

Decrypting data

When it's time to decrypt encrypted data, most of the process is the same as it is for encryption. To start:

```
$m = mcrypt_module_open('rijndael-
→ 256', '', 'cbc', '');
mcrypt_generic_init($m, $key, $iv);
```

At this point, instead of using `mcrypt_generic()`, you'll use `mdecrypt_generic()`:

```
$data = mdecrypt_generic($m,
→ $encrypted_data);
```

Note, and this is very important, that to successfully decrypt the data, you'll need the *exact same key and IV* used to encrypt it.

Once decryption has taken place, you can close up your resources:

```
mcrypt_generic_deinit($m);
mcrypt_module_close($m);
```

Finally, you'll likely want to apply the `rtrim()` function to the decrypted data, as the encryption process may add white space as padding to the end of the data.

To decrypt data:

1. Begin a new PHP script in your text editor or IDE, starting with the HTML (**Script 4.6**).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD
→ XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/
→ xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/
→ xhtml" xml:lang="en" lang="en">
<head>
    <meta http-equiv="content-type"
→ content="text/html; charset=iso-
→ 8859-1" />
    <title>A More Secure
→ Cookie</title>
</head>
<body>
<?php # Script 4.6 -
→ read_mcrypt_cookie.php
```

continues on page 166

Script 4.6 This script reads in a cookie with encrypted data (plus a second cookie that stores an important piece for decryption); then it decrypts and prints the data.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
4  <head>
5      <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
6      <title>A More Secure Cookie</title>
7  </head>
8  <body>
9  <?php # Script 4.6 - read_mcrypt_cookie.php
10
11  /* This page uses the MCrypt library
12   * to decrypt data stored in a cookie.
13   */
14
15  // Make sure the cookies exist:
16  if (isset($_COOKIE['thing1']) && isset($_COOKIE['thing2'])) {
17
18      // Create the key:
19      $key = md5('77 public drop-shadow Java');
20
21      // Open the cipher:
22      // Using Rijndael 256 in CBC mode.
23      $m = mcrypt_module_open('rijndael-256', '', 'cbc', '');
24
25      // Decode the IV:
26      $iv = base64_decode($_COOKIE['thing2']);
27
28      // Initialize the encryption:
29      mcrypt_generic_init($m, $key, $iv);
30
31      // Decrypt the data:
32      $data = mcrypt_generic($m, base64_decode($_COOKIE['thing1']));
33
34      // Close the encryption handler:
35      mcrypt_generic_deinit($m);
36
37      // Close the cipher:
38      mcrypt_module_close($m);
39
40      // Print the data.
41      echo '<p>The cookie has been received. Its value is "' . trim($data) . '".</p>';
42
43  } else { // No cookies!
44      echo '<p>There\'s nothing to see here.</p>';
45  }
46  ?>
47  </body>
48  </html>

```

2. Check that the cookies exist.

```
if (isset($_COOKIE['thing1']) &&  
→ isset($_COOKIE['thing2'])) {
```

There's no point in trying to decrypt the data if the page can't read the two cookies.

3. Create the key.

```
$key = md5('77 public drop-shadow  
→ Java');
```

Not to belabor the point, but again, this must be the exact same key used to encrypt the data. This is another reason why you might want to store the key outside of these scripts.

4. Open the cipher.

```
$m = mcrypt_module_open('rijndael-  
→ 256', '', 'cbc', '');
```

This should also match the encryption code (you have to use the same cipher and mode for both encryption and decryption).

5. Decode the IV.

```
$iv = base64_decode  
→ ($_COOKIE['thing2']);
```

The IV isn't being generated here; it's being retrieved from the cookie (because it has to be the same IV as was used to encrypt the data). The `base64_decode()` function will return the IV to its binary form.

6. Initialize the decryption.

```
mcrypt_generic_init($m, $key, $iv);
```

7. Decrypt the data.

```
$data = mdecrypt_generic($m,  
→ base64_decode($_COOKIE['thing1']));
```

The `mdecrypt_generic()` function will decrypt the data. The data is coming from the cookie and must be decoded first.

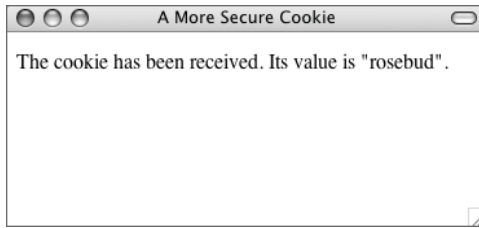


Figure 4.21 The cookie data has been successfully decrypted.

8. Wrap up the MCrypt code.


```
mcrypt_generic_deinit($m);
mcrypt_module_close($m);
```
9. Print the data.


```
echo '<p>The cookie has been
→ received. Its value is "' .
→ trim($data) . '"</p>';
```
10. Complete the page.


```
} else {
    echo '<p>There\'s nothing to
→ see here.</p>';
}
?>
</body>
</html>
```

The else clause applies if the two cookies were not accessible to the script.
11. Save the file as `read_mcrypt_cookie.php`, place it in your Web directory, and test in your Web browser (**Figure 4.21**).

✓ Tip

- If you rerun the first script, you'll see that the encrypted version of the data is actually different each time, even though the data itself is always the same. This is because the IV will be different each time. Still, the decryption will always work, as the IV is stored in a cookie.