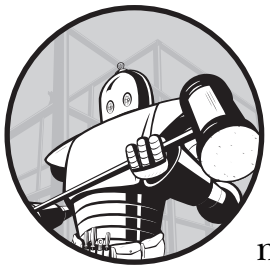
4

WORKING WITH FORMS



Forms are how your users talk to your scripts. To get the most out of PHP, you must master forms. The first thing you need to understand is that although PHP makes it easy to access form data, you must be careful of how you work with the data.

Security Measures: Forms Are Not Trustworthy

A common mistake that novices make is to trust the data provided by an HTML form. If you have a drop-down menu that only allows the user to enter one of three values, you must still check those values. As mentioned in Chapter 3, you also cannot rely on JavaScript to stop people from sending whatever they like to your server.

Your site's users can write their own form in HTML to use against your server; users can also bypass the browser entirely and use automatic tools to interact with web scripts. You should assume that people will mess around

with parameters when you put a script on the Web, because they might be trying to discover an easier way to use your site (though they could be attempting something altogether less beneficial).

To ensure that your server is safe, you must verify all data that your scripts receive.

Verification Strategies

There are two approaches to checking form data: blacklisting and whitelisting.

Blacklisting is the process of trying to filter out all bad data by assuming that form submissions are valid and then explicitly seeking out bad data. In general, this technique is ineffective and inefficient. For example, let's say that you're trying to eliminate all "bad" characters from a string, such as quotes. You might search for and replace quotation marks, but the problem is that there will *always* be bad characters you didn't think of. In general, blacklisting assumes that most of the data you receive is friendly.

A better assumption to make about form data you're receiving is that it's inherently malicious; thus, you should filter your data in order to *accept* only valid data submissions. This technique is called *whitelisting*. For example, if a string should consist of only alphanumeric characters, then you can check it against a regular expression that matches only an entire string of A-Za-z0-9. Whitelisting may also include forcing data to a known range of values or changing the type of a value. Here is an overview of a few specific tactics:

- If the value should be a number, use the `is_numeric()` function to verify the value. You can force a value to an integer using the `intval()` function. If the value should be an array, use `is_array()`.
- If the value should be a string, use `is_string()`. To force it, use `strval()`.
- If the value should be null, use `is_null()`.
- If the value should be defined, use `isset()`.

WHITELISTING INTEGERS

Here's a typical example of how you might whitelist for a numeric value. If the data is not numeric, then you use a default value of zero (of course, this assumes that zero is an acceptable value):

```
if (! is_numeric($data)) {
    // Use a default of 0.
    $data = 0;
}
```

In the case of integers, there is an alternative if you know that all integer values are safe. Using `$data = intval($data);` forces `$data` to its integral value. This technique is called *typecasting*.

Using $_POST, $_GET, $_REQUEST, and $_FILES to Access Form Data

In Chapter 2, we showed you how to turn off the `register_globals` setting that automatically sets global variables based on form data.

To shut down this dangerous setting, refer to "#14: Turning Off Registered Global Variables" on page 25. How do you use `$_POST`, `$_FILES`, and `$_GET` to retrieve form data? Read on.

#25: Fetching Form Variables Consistently and Safely

You should pull form data from *predefined server variables*. All data passed on to your web page via a posted form is automatically stored in a large array called `$_POST`, and all GET data is stored in a large array called `$_GET`. File upload information is stored in a special array called `$_FILES` (see "#54: Uploading Images to a Directory" on page 97 for more information on files). In addition, there is a combined variable called `$_REQUEST`.

To access the username field from a POST method form, use `$_POST['username']`. Use `$_GET['username']` if the username is in the URL. If you don't care where the value came from, use `$_REQUEST['username']`.

```
<?php

$post_value = $_POST['post_value'];
$get_value = $_GET['get_value'];
$some_variable = $_REQUEST['some_value'];

?>
```

`$_REQUEST` is a union of the `$_GET`, `$_POST`, and `$_COOKIE` arrays. If you have two or more values of the same parameter name, be careful of which one PHP uses. The default order is cookie, POST, then GET.

There has been some debate on how safe `$_REQUEST` is, but there shouldn't be. Because all of its sources come from the outside world (the user's browser), you need to verify everything in this array that you plan to use, just as you would with the other predefined arrays. The only problems you might have are confusing bugs that might pop up as a result of cookies being included.

#26: Trimming Excess Whitespace

Excess whitespace is a constant problem when working with form data. The `trim()` function is usually the first tool a programmer turns to, because it removes any excess spaces from the beginning or end of a string. For example, " Wicked Cool PHP " becomes "Wicked Cool PHP." In fact, it's so handy that you may find yourself using it on almost every available piece of user-inputted, non-array data:

```
$user_input = trim($user_input);
```

But sometimes you have excessive whitespace *inside* a string—when someone may be cutting and copying information from an email, for instance. In that case, you can replace multiple spaces and other whitespace with a single space by using the `preg_replace()` function. The *reg* stands for *regular expression*, a powerful form of pattern matching that you will see several times in this chapter.

```
<?php
function remove_whitespace($string) {
    $string = preg_replace('/\s+/', ' ', $string);
    $string = trim($string);
    return $string;
}
?>
```

You'll find many uses for this script outside of form verification. It's great for cleaning up data that comes from other external sources.

#27: Importing Form Variables into an Array

One of the handiest tricks you can use in PHP is not actually a PHP trick but an HTML trick. When a user fills out a form, you'll frequently check the values of several checkboxes. For example, let's say you're taking a survey to see what sorts of movies your site's visitors like, and you'd like to automatically insert those values into a database called `customer_preferences`. The hard way to do that is to give each checkbox a separate name on the HTML form, as shown here:

```
<p>What movies do you like?</p>
<input type="checkbox" name="action" value="yes"> Action
<input type="checkbox" name="drama" value="yes"> Drama
<input type="checkbox" name="comedy" value="yes"> Comedy
<input type="checkbox" name="romance" value="yes"> Romance
```

Unfortunately, when you process the form on the next page, you'll need a series of `if/then` loops to check the data—one loop to check the value of `$action`, one to check the value of `$drama`, and so forth. Adding a new checkbox to the HTML form results in yet another `if/then` loop to the processing page.

A great way to simplify this procedure is to store all of the checkbox values in a single array by adding `[]` after the name, like this (see Figure 4-1):

```
<form action="process.php" method="post">
<p>What is your name?</p>
<p><input type="text" name="customer_name"></p>

<p>What movies do you like?</p>
<p><input type="checkbox" name="movie_type[]" value="action"> Action
<input type="checkbox" name="movie_type[]" value="drama"> Drama
```

```
<input type="checkbox" name="movie_type[]" value="comedy"> Comedy
<input type="checkbox" name="movie_type[]" value="romance"> Romance</p>
<input type="submit">
</form>
```

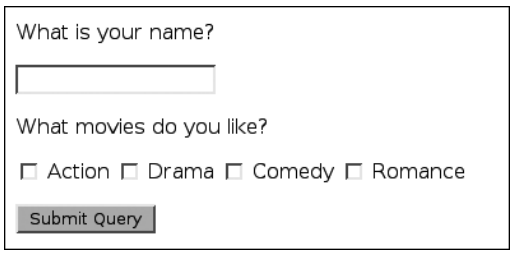


Figure 4-1: A form with an array of checkboxes

When PHP gets the data from a form like this, it stores the checked values in a single array. You can loop through the array this way:

```
<?php
$movie_type = $_POST["movie_type"];
$customer_name = strval($_POST["customer_name"]);

if (is_array($movie_type)) {
    foreach ($movie_type as $key => $value) {
        print "$customer_name likes $value movies.<br>";
    }
}

?>
```

Not only does this technique work for checkboxes, but it's extremely handy for processing arbitrary numbers of rows. For example, let's say we have a shopping menu where we want to show all the items in a given category. Although we may not know how many items will be in a category, the customer should be able to enter a quantity into a text box for all items he wants to buy and add all of the items with a single click.

Let's access product name and ID data in the `product_info` MySQL table described in the appendix to build the form as follows:

```
<?php
/* Insert code for connecting to $db here. */

$category = "shoes";
/* Retrieve products from the database. */
$sql = "SELECT product_name, product_id
          FROM product_info
         WHERE category = '$category'";

$result = @mysql_query($sql, $db) or die;
```

```
/* Initialize variables. */
$order_form = ""; /* Will contain product form data */
$i = 1;

print '<form action="addtocart.php" method="post">';

while($row = mysql_fetch_array($result)) {
    // Loop through the results from the MySQL query.
    $product_name = stripslashes($row['product_name']);
    $product_id = $row['product_id'];

    // Add the row to the order form.
    print "<input type=\"hidden\" name=\"product_id[$i]\" value=\"$product_id\
">";
    print "<input type=\"text\" name=\"quantity[$i]\"
    size=\"2\" value=\"0\"> $product_name<br />";

    $i++;
}

print '<input type="submit" name="add" value="Add to Cart"></form>';


?>
```

To process the form, you need to examine the two arrays passed to the processing script—one array containing all of the product IDs (`$product_id`) and another containing the corresponding values from the quantity text boxes (`$quantity`). It doesn't matter how many items are displayed on the page; `$product_id[123]` contains the product ID for the 123rd item displayed and `$quantity[123]` holds the number the customer entered into the corresponding text box.

The processing script `addtocart.php` is as follows:

```
<?php

$product_id = $_POST["product_id"];
$quantity = $_POST["quantity"];

if (is_array($quantity)) {
    foreach ($quantity as $key => $item_qty) {
        $item_qty = intval($item_qty);
        if ($item_qty > 0) {
            $id = $product_id[$key];
            print "You added $item_qty of Product ID $id.<br>";
        }
    }
}

?>
```

As you can see, this script depends wholly on using the index from the `$quantity` array (`$key`) for the `$product_id` array.

#28: Making Sure a Response Is One of a Set of Given Values

As I told you earlier, you can *never* assume that the data passed on by a form is safe. Let's look at this simple form item:

```
<SELECT NAME="card_type">
<OPTION value="visa">Visa</OPTION>
<OPTION value="amex">American Express</OPTION>
<OPTION value="mastercard">MasterCard</OPTION>
</SELECT>
```

How do you ensure that the data you're looking at is really Visa, American Express, or MasterCard? Simple: You store the data in array keys and then look at the array to make sure that there's an exact match. Here's an example:

```
<?php

$credit_cards = array(
    "amex"         => true,
    "visa"         => true,
    "mastercard"   => true,
);
$card_type = $_POST["card_type"];
if ($credit_cards[$card_type]) {
    print "$card_type is a valid credit card.";
} else {
    print "$card_type is not a valid credit card.";
}
```

Hacking the Script

One advantage of this method of data storage is that you can temporarily disable an item by changing its value to `false`. You can also alter the script slightly to provide both verbose values and data values. For example, you may store American Express cards in your database as *amex*, but when the name of the card is displayed on the screen you want it to show up as *American Express*.

In that case, you can use a map to remember what's what by storing the database value as the key in the array and the display name as the value. The following example demonstrates that technique.

```
<?php
$credit_cards = array(
    "amex"         => "American Express",
    "visa"         => "Visa",
    "mastercard"   => "MasterCard",
);
$card_type = $_POST["card_type"];
if (count($credit_cards[$credit_card_type]) > 0) {
    print "Your payment type: $credit_cards[$card_type].";
} else {
    print "Invalid card type.";
}
```

NOTE *The previous example is extremely useful information to store in a central configuration file.*

#29: Using Multiple Submit Buttons

Occasionally you want a form that does two separate things depending on which button a user clicks—one button updates a post while the other button deletes it. You can put two forms on one page that will send the user to two separate pages, but then you have to worry about inserting redundant information into both forms, not to mention that this would be unbearable to the user.

In HTML, buttons also have values, and you can read those values. Construct your form as follows:

```
<form action="process.php" method="post">
<input name="postid" type="hidden" value="1234">
<input name="action" type="submit" value="Update">
<input name="action" type="submit" value="Delete">
</form>
```

Now, in process.php, access `$_POST['action']` to get the button the user clicked.

#30: Validating a Credit Card

Here's a brief overview of how online credit card transactions work. First, you need to find a *merchant solution* (an online provider, such as Authorize.net or Secpay.com) that provides you with a *merchant account*. This account is like a bank account, except that it allows you to process charges for credit card transactions. The merchant provider typically charges a per-transaction fee for each credit card action.

If you have a physical store that accepts credit cards, you almost certainly have a merchant solution. However, not all merchant solutions offer online transactions. The ones that do offer online transactions give you access to a *payment gateway*, a secure server for processing credit card charges. Usually, the transactions occur via an XML datastream. You can use cURL to exchange XML with the payment gateway (see Chapter 11 for more details).

However, you can do some preliminary form validation work before talking to the payment gateway to save on transactions and transaction fees and possibly speed things for the user if they typed their credit card number incorrectly. It turns out that you can weed out completely incorrect credit card numbers with an easy algorithm. Furthermore, you can even determine a credit card type from a valid number. Keep in mind, though, that passing these tests is no guarantee that a card isn't stolen or canceled or that it belongs to a different person.

```
<?php
function validate_cc_number($cc_number) {
    /* Validate; return value is card type if valid. */
    $false = false;
    $card_type = "";
    $card_regexes = array(
        "/^4\d{12}(\d\d\d){0,1}$/"     => "visa",
        "/^5[12345]\d{14}$/"           => "mastercard",
        "/^3[47]\d{13}$/"              => "amex",
        "/^6011\d{12}$/"               => "discover",
        "/^30[012345]\d{11}$/"         => "diners",
        "/^3[68]\d{12}$/"              => "diners",
    );

    foreach ($card_regexes as $regex => $type) {
        if (preg_match($regex, $cc_number)) {
            $card_type = $type;
            break;
        }
    }

    if (!$card_type) {
        return $false;
    }

    /* mod 10 checksum algorithm */
    $revcode = strrev($cc_number);
    $checksum = 0;

    for ($i = 0; $i < strlen($revcode); $i++) {
        $current_num = intval($revcode[$i]);
        if ($i & 1) {              /* Odd position */
            $current_num *= 2;
        }
        /* Split digits and add. */
        $checksum += $current_num % 10;
        if ($current_num > 9) {
            $checksum += 1;
        }
    }

    if ($checksum % 10 == 0) {
        return $card_type;
    } else {
        return $false;
    }
}
?>
```

This function has two main stages. The first determines card type, and the second determines whether the card checksum is correct. If the card passes both tests, the return value is the card type as a string. If a card is invalid, you get `false` (you can change this return value to whatever you like with the `$false` variable).

The first stage is where the big trick comes in, where we determine the card type and confirm the prefix in one quick step. Credit card numbers follow a certain format. For example, all Visas start with 4 and have 13 or 16 digits, all MasterCards start with 51 through 55 and have 16 digits, and all American Express cards start with 34 or 37 and have 15 digits. These rules are easily expressed in a few regular expressions, and because they are unique rules, we can map the regular expressions to card types in an array called `$card_regexes`. To check for a valid format, we just cycle through the regular expressions until one matches. When we get a match, we set `$card_type` and move to the next stage. If no expressions match, we return `failure`.

The checksum test for the credit card number uses a mod 10 algorithm, a reasonably simple-to-implement check that does the following:

- It starts with a checksum value of 0.
- It runs through the credit card number digit-by-digit from right to left.
- If the current digit has an odd index (that is, every other digit, starting at index 0), the digit is doubled. If the value of the doubled digit is over 9, the two numbers are added together and added to the checksum (so an 8 becomes 16, which becomes 1 + 6, which becomes 7). Otherwise the current (doubled if on an odd index) digit is added to the checksum.
- After running through all the digits, the final checksum must be divisible by 10. If not, the number fails the test.

There are several ways to code this algorithm; the implementation here is on the compact side, but easy enough to follow.

Using the Script

Just feed a string with a number to `validate_cc_number()` and check the return value. The only thing you should be careful about is nondigits in the string; you should take care of this with `preg_replace()` before running the function. Here is a snippet that runs the function on several test numbers:

```
$nums = array(
    "3721 0000 0000 000",
    "340000000000009",
    "5500 0000 0000 0004",
    "4111 1111 1111 1111",
    "4222 2222 22222",
    "4007000000027",
    "30000000000004",
    "6011000000000004",
);
```

```
foreach ($nums as $num) {
    /* Remove all non-digits in card number. */
    $num = ereg_replace("[^0-9]", "", $num);

    $t = validate_cc_number($num);
    if ($t) {
        print "$num valid (type: $t).\n";
    } else {
        print "$num invalid.\n";
    }
}
```

Hacking the Script

You can add other major credit cards if you know their format. An excellent resource for other cards is http://www.sitepoint.com/print/card-validation-class-php.

#31: Double-Checking a Credit Card's Expiration Date

When you accept a credit card, you'll need to know whether it has expired. In your HTML, it's best to create a drop-down menu that allows customers to choose their card's expiration date in order to avoid ambiguity in date formats:

```
<select name="cc_month">
<option value="01" >01 : January</option>
          <option value="02" >02 : February</option>
          <option value="03" >03 : March</option>
          <option value="04" >04 : April</option>
          <option value="05" >05 : May</option>
          <option value="06" >06 : June</option>
          <option value="07" >07 : July</option>
          <option value="08" >08 : August</option>
          <option value="09" >09 : September</option>
          <option value="10" >10 : October</option>
          <option value="11" >11 : November</option>
          <option value="12" >12 : December</option>
        </select>
<select name="cc_year">
<?php
    /* Create options for all years up to six years from now. */
    $y = intval(date("Y"));
    for ($i = $y; $i <= $y + 10; $i++) {
        print "<option value=\"$i\">$i</option>\n";
    }
?>

</select>
```

Now that you have a form for entering an expiration date, you need to validate the data sent by it.

```
<?php
function check_exp_date($month, $year) {
    /* Get timestamp of midnight on day after expiration month. */
    $exp_ts = mktime(0, 0, 0, $month + 1, 1, $year);

    $cur_ts = time();
    /* Don't validate for dates more than 10 years in future. */
    $max_ts = $cur_ts + (10 * 365 * 24 * 60 * 60);

    if ($exp_ts > $cur_ts && $exp_ts < $max_ts) {
        return true;
    } else {
        return false;
}
?>
```

To check a credit card expiration date, all you have to do is make sure that the date falls between the current date and some date in the future (this function uses 10 years). The best tools for that task are described in Chapter 6, so consider this a sneak preview.

The only trick here is that a credit card becomes invalid after the last day of the month in its expiration date. That is, if a card's expiration date was 06/2005, it actually stopped working on July 1, 2005. Thus, we have to add a month to the given date. This can be a pain because it can also set the actual target date ahead a year, but as you will learn in Chapter 6, the `mktime()` function that we're using here to compute the expiration timestamp automatically compensates for month numbers that are out of range. After computing the expiration timestamp, all you need are the current and maximum timestamps, and validating the expiration time boils down to a pair of simple comparisons.

Using the Script

```
if (check_exp_date($cc_month, $cc_year)) {
    // Approve the card.
} else {
    // The card has expired.
}
```

#32: Checking Valid Email Addresses

Customers enter all sorts of weird data into email form fields. The script in this section verifies that an email address mostly follows the rules outlined in RFC 2822. This won't prevent someone from entering a false (but RFC-compliant) email address such as leavemealone@nonexistentdomain.com, but it will catch some typos.

NOTE *If having a valid email address is critical, you need to require user accounts that are activated by links sent only via email. You'll see how to do this in "#65: Using Email to Verify User Accounts" on page 124. This is a fairly extreme measure; if you want more people to share their addresses with you, simply tell users that you won't spam them (and make good on that promise).*

```
<?php

function is_email($email) {
// Checks for proper email format

    if (! preg_match( '/^[A-Za-z0-9!#$%&\'*+-/=?^_`{|}~]+@[A-Za-z0-9-]+(\.[A-
Za-z0-9-]+)+[A-Za-z]$/', $email)) {
            return false;
    } else {
        return true;
    }
}
?>
```

This script utilizes a regular expression to check whether the given email uses proper email characters (alphabetical, dots, dashes, slashes, and so on), an @ sign in the middle, and at least one dot-something on the end. You can read more on regular expressions in "#39: Regular Expressions" on page 69.

#33: Checking American Phone Numbers

As with email addresses, there's no way to make sure a telephone number is valid outside of making a real telephone call. However, you can validate the number of digits and put it into standard format. The following function returns a pure 10-digit phone number if the number given is 10 digits or 11 digits starting with 1. If the number does not conform, the return value is `false`.

```
<?php
function is_phone($phone) {
    $phone = preg_replace('/[^\d]+/', '', $phone);
    $num_digits = strlen($phone);
    if ($num_digits == 11 && $phone[0] == "1") {
        return substr($phone, 1);
    } else if ($num_digits == 10) {
        return $phone;
    } else {
        return false;
    }
}
?>
```

This script shows the power of regular expressions combined with standard string functions. The key is to first throw out any character that's not a digit—a perfect task for the `preg_replace()` function. Once you know that you have nothing but digits in a string, you can simply examine the string length to determine the number of digits, and the rest practically writes itself.