

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.

