Secure Web Programming Techniques

Build security into your applications

by Alan Seiden

I N "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)