Pro PHP Security

Chris Snyder and Michael Southwell

Apress®

Pro PHP Security

ISBN (pbk): 1-59059-508-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore
Technical Reviewer: Timothy Boronczyk
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositors: Susan Glinert and Pat Christenson
Proofreader: April Eddy
Indexer: Michael Brinkman
Artist: Wordstop Technologies Pvt. Ltd., Chennai
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 12

Preventing SQL Injection

We began Part 3 with a discussion in Chapter 11 of keeping your PHP scripts secure by careful validation of user input. We continue that discussion here, focusing on user input that participates in your scripts' interaction with your databases. Your data is, after all, probably your most treasured resource. Your primary goal in writing scripts to access that data should be to protect your users' data at all costs. In the rest of this chapter, we'll show you ways to use PHP to do that.

What SQL Injection Is

There is no point to putting data into a database if you intend never to use it; databases are designed to promote the convenient access and manipulation of their data. But the simple act of doing so carries with it the potential for disaster. This is true not so much because you yourself might accidentally delete everything rather than selecting it. Instead, it is that your attempt to accomplish something innocuous could actually be hijacked by someone who substitutes his own destructive commands in place of yours. This act of substitution is called *injection*.

Every time you solicit user input to construct a database query, you are permitting that user to participate in the construction of a command to the database server. A benign user may be happy enough to specify that he wants to view a collection of men's long-sleeved burgundy-colored polo shirts in size large; a malicious user will try to find a way to contort the command that selects those items into a command that deletes them, or does something even worse.

Your task as a programmer is to find a way to make such injections impossible.

How SQL Injection Works

Constructing a database query is a perfectly straightforward process. It typically proceeds something like this (for demonstration purposes, we'll assume that you have a database of wines, where one of the fields is the grape variety):

1. You provide a form that allows the user to submit something to search for. Let's assume that the user chooses to search for wines made from the grape variety "lagrein."
2. You retrieve the user's search term, and save it by assigning it to a variable, something like this:

```
$variety = $_POST['variety'];
```

So that the value of the variable `$variety` is now this:

```
lagrein
```

3. You construct a database query, using that variable in the `WHERE` clause, something like this:

```
$query = "SELECT * FROM wines WHERE variety='$variety'";
```

so that the value of the variable `$query` is now this:

```
SELECT * FROM wines WHERE variety='lagrein'
```

4. You submit the query to the MySQL server.
5. MySQL returns all records in the wines table where the field variety has the value "lagrein."

So far, this is very likely a familiar and comfortable process.

Unfortunately, sometimes familiar and comfortable processes lull us into complacency. So let's look back at the actual construction of that query.

1. You created the invariable part of the query, ending it with a single quotation mark, which you will need to delineate the beginning of the value of the variable:

```
$query = "SELECT * FROM wines WHERE variety = '";
```

2. You concatenated that invariable part with the value of the variable containing the user's submitted value:

```
$query .= $variety;
```

3. You then concatenated the result with another single quotation mark, to delineate the end of the value of the variable:

```
$query .= "'";
```

The value of `$query` was therefore (with the user input in bold type) this:

```
SELECT * FROM wines WHERE variety = 'lagrein'
```

The success of this construction depended on the user's input. In this case, you were expecting a single word (or possibly a group of words) designating a grape variety, and you got it. So the query was constructed without any problem, and the results were likely to be just what you expected, a list of the wines for which the grape variety is "lagrein."

Let's imagine now that your user, instead of entering a simple grape variety like "lagrein" (or even "pinot noir"), enters the following value (notice the two included punctuation marks):

```
lagrein' or 1=1;
```

You now proceed to construct your query with, first, the invariable portion (we show here only the resultant value of the `$query` variable):

```
SELECT * FROM wines WHERE variety = '
```

You then concatenate that with the value of the variable containing what the user entered (here shown in bold type):

```
SELECT * FROM wines WHERE variety = 'lagrein' or 1=1;
```

And finally you add the closing quotation mark:

```
SELECT * FROM wines WHERE variety = 'lagrein' or 1=1;'
```

The resulting query is very different from what you had expected. In fact, your query now consists of not one but rather two instructions, since the semicolon at the end of the user's entry closes the first instruction (to select records) and begins another one. In this case, the second instruction, nothing more than a single quotation mark, is meaningless.

But the first instruction is not what you intended, either. When the user put a single quotation mark into the middle of his entry, he ended the value of the desired variable, and introduced another condition. So instead of retrieving just those records where the variety is "lagrein," in this case you are retrieving those records that meet either of two criteria, the first one yours and the second one his: the variety has to be "lagrein" or 1 has to be 1. Since 1 is always 1, you are therefore retrieving all of the records!

You may object that you are going to be using double rather than single quotation marks to delineate the user's submitted variables. This slows the abuser down for only as long as it takes for it to fail and for him to retry his exploit, using this time the double quotation mark that permits it to succeed. (We remind you here that, as we discussed in Chapter 11, all error notification to the user should be disabled. If an error message were generated here, it would have just helped the attacker by providing a specific explanation for why his attack failed.)

As a practical matter, for your user to see all of the records rather than just a selection of them may not at first glance seem like such a big deal, but in actual fact it is; viewing all of the records could very easily provide him with insight into the structure of the table, an insight that could easily be turned to more nefarious purposes later. This is especially true if your database contains not something apparently innocuous like wines, but rather, for example, a list of employees with their annual salaries.

And as a theoretical matter, this exploit is a very bad thing indeed. By injecting something unexpected into your query, this user has succeeded in turning your intended database access around to serve his own purposes. Your database is therefore now just as open to him as it is to you.

PHP and MySQL Injection

As we have mentioned previously, PHP, by design, does not do anything except what you tell it to do. It is precisely that hands-off attitude that permits exploits such as the one we described previously.

We will assume that you will not knowingly or even accidentally construct a database query that has destructive effects; the problem is with input from your users. Let's therefore look now in more detail at the various ways in which users might provide information to your scripts.

Kinds of User Input

The ways in which users can influence the behavior of your scripts are more, and more complex, than they may appear at first glance.

The most obvious source of user input is of course a text input field in a form. With such a field, you are deliberately soliciting a user's input. Furthermore, you are providing the user with a wide open field; there is no way that you can limit ahead of time what a user can type (although you can limit its length, if you choose to). This is the reason why the overwhelming source for injection exploits is the unguarded form field.

But there are other sources as well, and a moment's reflection on the technology behind forms (the transmission of information via the POST method) should bring to mind another common source for the transmission of information: the GET method. An observant user can easily see when information is being passed to a script simply by looking at the URI displayed in the browser's navigation toolbar. Although such URIs are typically generated programmatically, there is nothing whatsoever stopping a malicious user from simply typing a URI with an improper variable value into a browser, and thus potentially opening a database up for abuse.

One common strategy to limit users' input is to provide an option box rather than an input box in a form. This control forces the user to choose from among a set of predetermined values, and would seem to prevent the user from entering anything unexpected. But just as an attacker might spoof a URI (that is, create an illegitimate URI that masquerades as an authentic one), so might she create her own version of your form, with illegitimate rather than predetermined safe choices in the option box. It's extremely simple to do this; all she needs to do is view the source and then cut-and-paste the form's source code, which is right out in the open for her. After modifying the choices, she can submit the form, and her illegal instruction will be carried in as if it were original.

So users have many different ways of attempting to inject malicious code into a script.

Kinds of Injection Attacks

There may not be quite as many different kinds of attacks as there are motives for attacks, but once again, there is more variety than might appear at first glance. This is especially true if the malicious user has found a way to carry out multiple query execution, a subject to which we will return in a moment.

If your script is executing a `SELECT` instruction, the attacker can force the display of every row in a table by injecting a condition like `1=1` into the `WHERE` clause, with something like this (the injection is in bold type):

```
SELECT * FROM wines WHERE variety = 'lagrein' OR 1=1;'
```

As we said earlier in this chapter, that can by itself be very useful information, for it reveals the general structure of the table (in a way that a single record cannot), as well as potentially displaying records that contain confidential information.

An `UPDATE` instruction has the potential for more direct damage. By inserting additional properties into the `SET` clause, an attacker can modify any of the fields in the record being updated, with something like this (the injection is in bold type):

```
UPDATE wines SET type='red','vintage'='9999' WHERE variety = 'lagrein'
```

And by adding an always-true condition like `1=1` into the `WHERE` clause of an `UPDATE` instruction, that modification can be extended to every record, with something like this (the injection is in bold type):

```
UPDATE wines SET type='red','vintage'='9999 WHERE variety = 'lagrein' OR 1=1;'
```

The most dangerous instruction may be `DELETE`, although it's not hard to imagine that a buried and therefore overlooked change might in the long run be more destructive than a wholesale deletion, which is likely to be immediately obvious. The injection technique is the same as what we have already seen, extending the range of affected records by modifying the `WHERE` clause, with something like this (the injection is in bold type):

```
DELETE FROM wines WHERE variety = 'lagrein' OR 1=1;'
```

Multiple-query Injection

Multiple-query injection multiplies the potential damage an attacker can cause, by allowing more than one destructive instruction to be included in a query.

The attacker sets this up by introducing an unexpected termination of the query. This is easily done with MySQL, where first an injected quotation mark (either single or double; a moment's experimentation will quickly reveal which) marks the end of the expected variable; and then a semicolon terminates that instruction. Now an additional attacking instruction may be added onto the end of the now-terminated original instruction. The resulting destructive query might look something like this (again, the injection, running over two lines, is in bold type):

```
SELECT * FROM wines WHERE variety = 'lagrein';
GRANT ALL ON *.* TO 'BadGuy@%' IDENTIFIED BY 'gotcha';'
```

This exploit piggybacks the creation of a new user, BadGuy, with network privileges, all privileges on all tables, and a facetious but sinister password, onto what had been a simple `SELECT` statement. If you took our advice in Chapter 10 to restrict severely the privileges of process users, this should not work, because the webserver daemon no longer has the `GRANT` privilege that you revoked. But theoretically, such an exploit could give BadGuy free rein to do anything he wants to with your database.

There is considerable variability in whether such a multiple query will even be processed by the MySQL server. Some of this variability may be due to different versions of MySQL, but most is due to the way in which the multiple query is presented. MySQL's monitor program allows such a query without any problem. The common MySQL GUI, `phpMyAdmin`, simply dumps everything before the final query, and processes that only.

But most if not all multiple queries in an injection context are managed by PHP's `mysql` extension. This, we are happy to report, by default does not permit more than one instruction to be executed in a query; an attempt to execute two instructions (like the injection exploit just shown) simply fails, with no error being set and no output being generated. It appears that this behavior is impossible to circumvent. In this case, then, PHP, despite its default hands-off behavior, does indeed protect you from the most obvious kinds of attempted injection.

PHP 5's new `mysqli` extension (see `http://php.net/mysqli`), like `mysql`, does not inherently permit multiple queries, but possesses a `mysqli_multi_query()` function that will let you do it if you really want to. If you decide that you do really want to, however, we urge you to remember that by doing so you are making an injector's job a lot easier.

The situation is more dire, however, with SQLite, the embeddable SQL database engine that is bundled with PHP 5 (see http://sqlite.org/ and http://php.net/sqlite), and that has attracted much attention recently for its ease of use. SQLite defaults to allowing such multiple-instruction queries in some cases, because the database can optimize batches of queries, particularly batches of INSERT statements, very effectively. The sqlite_query() function will not, however, allow multiple queries to be executed if the result of the queries is to be used by your script, as in the case of a SELECT to retrieve records (see the warning at http://php.net/sqlite_query for more information).

INVISION POWER BOARD SQL INJECTION VULNERABILITY

Invision Power Board is a widely known forum system (see http://www.invisionboard.com for information). On 6 May 2005 a SQL injection vulnerability was found in the login code, by James Bercegay of GulfTech Security Research (see http://www.gulftech.org/?node=research&article_id=00073-05052005 for more information).

The login query is as follows:

```
$DB->query("SELECT * FROM ibf_members WHERE id=$mid AND password='$pid'");
```

The member ID variable $mid and the password ID variable $pid are retrieved from the my_cookie() function with these two lines:

```
$mid = intval($std->my_getcookie('member_id'));
$pid = $std->my_getcookie('pass_hash');
```

The my_cookie() function retrieves the requested variable from the cookie with this line:

```
return urldecode($_COOKIE[$ibforums->vars['cookie_id'].$name]);
```

The value returned from the cookie is not sanitized at all. While $mid is cast to an integer before being used in the query, $pid is left untouched. It is therefore subject to the kinds of injection we have discussed earlier.

This vulnerability was addressed by modifying the my_cookie() function as follows (in relevant part; see http://forums.invisionpower.com/index.php?showtopic=168016 for more information):

```
if ( ! in_array( $name, array('topicsread', 'forum_read', 'collapseprefs') ) )
{
  return $this->➥
    clean_value(urldecode($_COOKIE[$ibforums->vars['cookie_id'].$name]));
}
else
{
  return urldecode($_COOKIE[$ibforums->vars['cookie_id'].$name]);
}
```

With this correction, the critical variables are returned after having been passed through the global clean_value() function, while other variables are left (not inappropriately) unsanitized.

Source: http://www.securityfocus.com/archive/1/397672

Preventing SQL Injection

Now that we have surveyed just what SQL injection is, how it can be carried out, and to what extent you are vulnerable to it, let's turn to considering ways to prevent it. Fortunately, PHP has rich resources to offer, and we feel confident in predicting that a careful and thorough application of the techniques we are recommending will essentially eliminate any possibility of SQL injection in your scripts, by sanitizing your users' data before it can do any damage.

Demarcate Every Value in Your Queries

We recommend that you make sure to demarcate every single value in your queries. String values must of course be delineated, and for these you should normally expect to use single (rather than double) quotation marks. For one thing, doing so may make typing the query easier, if you are using double quotation marks to permit PHP's variable substitution within the string; for another, it (admittedly, microscopically) diminishes the parsing work that PHP has to do to process it.

We illustrate this with our original, noninjected query:

```
SELECT * FROM wines WHERE variety = 'lagrein'
```

Or in PHP:

```
$query = "SELECT * FROM wines WHERE variety = '$variety'";
```

Quotation marks are technically not needed for numeric values. But if you were to decide not to bother to put quotation marks around a value for a field like `vintage`, and if your user entered an empty value into your form, you would end up with a query like this:

```
SELECT * FROM wines WHERE vintage =
```

This query is, of course, syntactically invalid, in a way that this one is not:

```
SELECT * FROM wines WHERE vintage = ''
```

The second query will (presumably) return no results, but at least it will not return an error message, as an unquoted empty value will (even though you have turned off all error reporting to users—haven't you? If not, look back at Chapter 11).

Check the Types of Users' Submitted Values

We noted previously that by far the primary source of SQL injection attempts is an unexpected form entry. When you are offering a user the chance to submit some sort of value via a form, however, you have the considerable advantage of knowing ahead of time what kind of input you should be getting. This ought to make it relatively easy to carry out a simple check on the validity of the user's entry. We discussed such validation at length in Chapter 11, to which we now refer you. Here we will simply summarize what we said there.

If you are expecting a number (to continue our previous example, the year of a wine vintage, for instance), then you can use one of these techniques to make sure what you get is indeed numeric:

- Use the `is_int()` function (or `is_integer()` or `is_long()`, its aliases).
- Use the `gettype()` function.
- Use the `intval()` function.
- Use the `settype()` function.

To check the length of user input, you can use the `strlen()` function.

To check whether an expected time or date is valid, you can use the `strtotime()` function.

It will almost certainly be useful to make sure that a user's entry does not contain the semicolon character (unless that punctuation mark could legitimately be included). You can do this easily with the `strpos()` function, like this:

```
if ( strpos( $variety, ';' ) ) exit ( "$variety is an invalid value for variety!" );
```

As we suggested in Chapter 11, a careful analysis of your expectations for user input should make it easy to check many of them.

Escape Every Questionable Character in Your Queries

Again, we discussed at length in Chapter 11 the escaping of dangerous characters. We simply reiterate here our recommendations, and refer you back there for details:

- Do not use the `magic_quotes_gpc` directive or its behind-the-scenes partner, the `addslashes()` function, which is limited in its application, and requires the additional step of the `stripslashes()` function.
- The `mysql_real_escape_string()` function is more general, but has its own drawbacks.

Abstract to Improve Security

We do not suggest that you try to apply the techniques listed earlier manually to each instance of user input. Instead, you should create an abstraction layer. A simple abstraction would incorporate your validation solutions into a function, and would call that function for each item of user input. A more complex one could step back even further, and embody the entire process of creating a secure query in a class. Many such classes exist already; we discuss some of them later in this chapter.

Such abstraction has at least three benefits, each of which contributes to an improved level of security:

1. It localizes code, which diminishes the possibility of missing routines that circumstances (a new resource or class becomes available, or you move to a new database with different syntax) require you to modify.
2. It makes constructing queries both faster and more reliable, by moving part of the work to the abstracted code.
3. When built with security in mind, and used properly, it will prevent the kinds of injection we have been discussing.

Retrofitting an Existing Application

A simple abstraction layer is most appropriate if you have an existing application that you wish to harden. The code for a function that simply sanitizes whatever user input you collect might look something like this:

```
function safe( $string ) {
  return "'" . mysql_real_escape_string( $string ) . "'"
}
```

Notice that we have built in the required single quotation marks for the value (since they are otherwise hard to see and thus easy to overlook), as well as the `mysql_real_escape_string()` function. This function would then be used to construct a `$query` variable, like this:

```
$variety = safe( $_POST['variety'] );
$query = "SELECT * FROM wines WHERE variety=" . $variety;
```

Now your user attempts an injection exploit by entering this as the value of `$variety`:

```
lagrein' or 1=1;
```

To recapitulate, without the sanitizing, the resulting query would be this (with the injection in bold type), which will have quite unintended and undesirable results:

```
SELECT * FROM wines WHERE variety = 'lagrein' or 1=1;'
```

Now that the user's input has been sanitized, however, the resulting query is this harmless one:

```
SELECT * FROM wines WHERE variety = 'lagrein\' or 1=1\;'
```

Since there is no variety field in the database with the specified value (which is exactly what the malicious user entered: `lagrein' or 1=1;`), this query will return no results, and the attempted injection will have failed.

Securing a New Application

If you are creating a new application, you can start from scratch with a more profound layer of abstraction. In this case, PHP 5's improved MySQL support, embodied in the brand new `mysqli` extension, provides powerful capabilities (both procedural and object-oriented) that you should definitely take advantage of. Information about `mysqli` (including a list of configuration options) is available at `http://php.net/mysqli`. Notice that `mysqli` support is available only if you have compiled PHP with the `--with-mysqli=path/to/mysql_config` option.

A procedural version of the code to secure a query with `mysqli` follows, and can be found also as `mysqliPrepare.php` in the Chapter 12 folder of the downloadable archive of code for *Pro PHP Security* at `http://www.apress.com`.

```
<?php

// retrieve the user's input
$animalName = $_POST['animalName'];
```

```
// connect to the database
$connect = mysqli_connect( 'localhost', 'username', 'password', 'database' );
if ( !$connect ) exit( 'connection failed:  ' . mysqli_connect_error() );

// create a query statement resource
$stmt = mysqli_prepare( $connect,
 "SELECT intelligence FROM animals WHERE name = ?" );

if ( $stmt ) {
  // bind the substitution to the statement
  mysqli_stmt_bind_param( $stmt, "s", $animalName );

  // execute the statement
  mysqli_stmt_execute( $stmt );

  // retrieve the result...
  mysqli_stmt_bind_result( $stmt, $intelligence );

  // ...and display it
  if ( mysqli_stmt_fetch( $stmt ) ) {
    print "A $animalName has $intelligence intelligence.\n";
  } else {
    print 'Sorry, no records found.';
  }

  // clean up statement resource
  mysqli_stmt_close( $stmt );
}

mysqli_close( $connect );

?>
```

The `mysqli` extension provides a whole series of functions that do the work of constructing and executing the query. Furthermore, it provides exactly the kind of protective escaping that we have previously had to create with our own `safe()` function. (Oddly, the only place this capacity is mentioned in the documentation is in the user comments at `http://us2.php.net/mysqli_stmt_bind_param`.)

First you collect the user's submitted input, and make the database connection. Then you set up the construction of the query resource, named `$stmt` here to reflect the names of the functions that will be using it, with the `mysqli_prepare()` function. This function takes two parameters: the connection resource, and a string into which the `?` marker is inserted every time you want the extension to manage the insertion of a value. In this case, you have only one such value, the name of the animal.

In a `SELECT` statement, the only place where the `?` marker is legal is right here in the comparison value. That is why you do not need to specify which variable to use anywhere except in the `mysqli_stmt_bind_param()` function, which carries out both the escaping and the substitution; here you need also to specify its type, in this case `"s"` for "string" (so as part of its provided

protection, this extension casts the variable to the type you specify, thus saving you the effort and coding of doing that casting yourself). Other possible types are "i" for integer, "d" for double (or float), and "b" for binary string.

Appropriately named functions, mysqli_stmt_execute(), mysqli_stmt_bind_result(), and mysqli_stmt_fetch(), carry out the execution of the query and retrieve the results. If there are results, you display them; if there are no results (as there will not be with a sanitized attempted injection), you display an innocuous message. Finally, you close the $stmt resource and the database connection, freeing them from memory.

Given a legitimate user input of "lemming," this routine will (assuming appropriate data in the database) print the message "A lemming has very low intelligence." Given an attempted injection like "lemming' or 1=1;" this routine will print the (innocuous) message "Sorry, no records found."

The mysqli extension provides also an object-oriented version of the same routine, and we demonstrate here how to use that class. This code can be found also as mysqliPrepareOO.php in the Chapter 12 folder of the downloadable archive of code for *Pro PHP Security* at http://www.apress.com.

```
<?php

$animalName = $_POST['animalName'];

$mysqli = new mysqli( 'localhost', 'username', 'password', 'database');

if ( !$mysqli ) exit( 'connection failed:  ' . mysqli_connect_error() );

$stmt = $mysqli->prepare( "SELECT intelligence➥
  FROM animals WHERE name = ?" );

if ( $stmt ) {
  $stmt->bind_param( "s", $animalName );
  $stmt->execute();
  $stmt->bind_result( $intelligence );

  if ( $stmt->fetch() ) {
    print "A $animalName has $intelligence intelligence.\n";
  } else {
    print 'Sorry, no records found.';
  }

  $stmt->close();
}

$mysqli->close();

?>
```

This code duplicates the procedural code described previously, using an object-oriented syntax and organization rather than strictly procedural code.

Full Abstraction

If you use external libraries like PearDB (see http://pear.php.net/package/DB), you may be wondering why we are spending so much time discussing code for sanitizing user input, for those libraries tend to do all of the work for you. The PearDB library takes abstraction one step beyond what we have been discussing, not only sanitizing user input according to best practices, but also doing it for whatever database you may happen to be using. It is therefore an extremely attractive option if you are concerned about hardening your scripts against SQL injection. Libraries like PearDB offer highly reliable (because widely tested) routines in a highly portable and database-agnostic context.

On the other hand, using such libraries has a clear downside: it puts you at the mercy of someone else's idea of how to do things, adds tremendously to the quantity of code you must manage, and tends to open a Pandora's Box of mutual dependencies. You need therefore to make a careful and studied decision about whether to use them. If you decide to do so, at least you can be sure that they will indeed do the job of sanitizing your users' input.

Test Your Protection Against Injection

As we discussed in previous chapters, an important part of keeping your scripts secure is to test them for protection against possible vulnerabilities.

The best way to make certain that you have protected yourself against injection is to try it yourself, creating tests that attempt to inject SQL code. For help and guidance in this task, you will probably find it useful to consult the amusing and revealing detailed instructions on just how to carry out an injection exploit, which can be found at http://www.issadvisor.com/columns/SqlInjection3/sql-injection-3-exploit-tables_files/frame.htm and http://www.imperva.com/application_defense_center/white_papers/blind_sql_server_injection.html.

Here we present a sample of such a test, in this case testing for protection against injection into a SELECT statement. This code can be found also as protectionTest.php in the Chapter 12 folder of the downloadable archive of code for *Pro PHP Security* at http://www.apress.com.

```
<?php

// protection function to be tested
function safe( $string ) {
  return "'" . mysql_real_escape_string( $string ) . "'"
}

// connect to the database

//////////////////////
// attempt an injection
//////////////////////
$exploit = "lemming' AND 1=1;";

// sanitize it
$safe = safe( $exploit );
```

```
$query = "SELECT * FROM animals WHERE name = $safe";
$result = mysql_query( $query );

// test whether the protection has been sufficient
if ( $result && mysql_num_rows( $result ) == 1 ) {
  exitt "Protection succeeded:\n
    exploit $exploit was neutralized.";
}
else {
  exit( "Protection failed:\n
    exploit $exploit was able to retrieve all rows." );
}
?>
```

If you were to create a suite of such tests, trying different kinds of injection with different SQL commands, you would quickly detect any holes in your sanitizing strategies. Once those were fixed, you could be sure that you have real protection against the threat of injection.

Summary

We began here in Chapter 12 our examination of specific threats to your scripts caused by faulty sanitizing of user input, with a discussion of SQL injection.

After describing how SQL injection works, we outlined precisely how PHP can be subjected to injection. We then provided a real-life example of such injection. Next we proposed a series of steps that you can take to make attempted injection exploits harmless, by making sure that all submitted values are enclosed in quotation marks, by checking the types of user-submitted values, and by escaping potentially dangerous characters in your users' input. We recommended that you abstract your validation routines, and provided scripts for both retrofitting an existing application and securing a new one. Then we discussed the advantages and disadvantages of third-party abstraction solutions.

Finally, we provided a model for a test of your protection against attempted SQL applications resulting in injection.

We turn in Chapter 13 to the next stage of validating user input in order to keep your PHP scripts secure: preventing cross-site scripting.