Extracted from:

SQL Antipatterns

Avoiding the Pitfalls of Database Programming

This PDF file contains pages extracted from *SQL Antipatterns*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

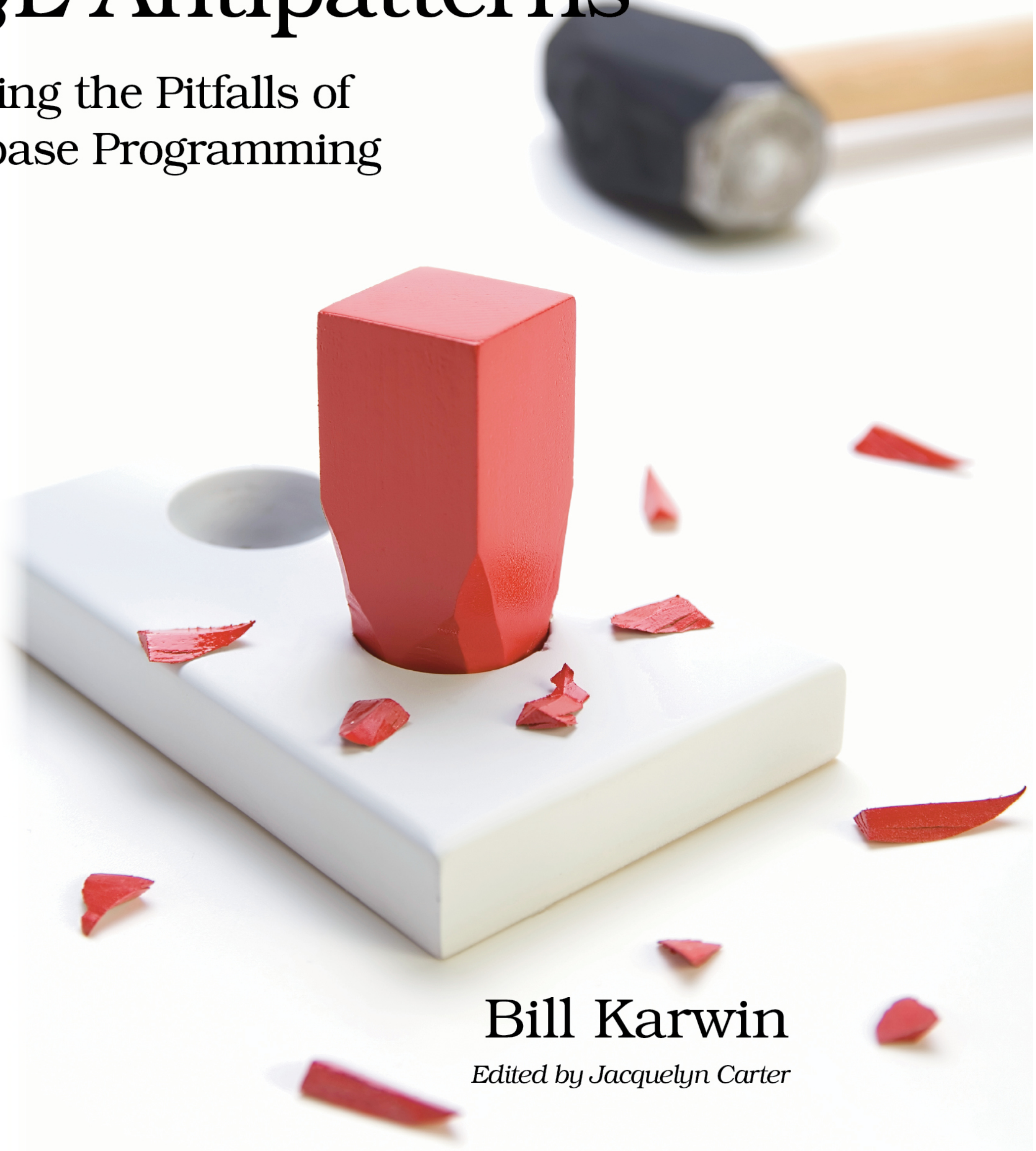
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

SQL Antipatterns

Avoiding the Pitfalls of Database Programming



Bill Karwin

Edited by Jacquelyn Carter

SQL Antipatterns

Avoiding the Pitfalls of Database Programming

Bill Karwin

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-934356-55-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P3.0—March 2012

In March 2010, serial computer hacker Albert Gonzalez was convicted for his role in the largest identity theft in history. He acquired an estimated 130 million credit and debit card numbers by hacking into ATM machines and payment systems of several major retail store chains and the credit-card processing companies that serve them.

Gonzales broke the previous record, which he also held, for stealing 45.6 million credit and debit card numbers in 2006. He performed that earlier crime by exploiting vulnerable wireless networks.

How did Gonzalez nearly triple his own record? We imagine a daring plot from a James Bond movie, with black-clad agents rappelling down elevators shafts, using supercomputers to crack state-of-the-art encrypted passwords, or sabotaging electrical power to an entire city.

The indictment describes a more mundane reality. Gonzalez exploited a vulnerability that is one of the most common security weaknesses on the Internet. He was able to use an attack technique called *SQL Injection* to gain privileged access to upload files to the corporate victims' servers. After Gonzalez and his coconspirators gained this access, the indictment states:[1]

> **Executing the Attacks: The Malware**
> ...they would install "sniffer" programs that would capture credit and debit card numbers, corresponding Card Data, and other information on a real-time basis as the information moved through the Corporate Victims' credit and debit card processing networks, and then periodically transmit that information to the coconspirators.

The retailers whose websites Gonzalez attacked have said that they've made changes to correct these security holes. However, they've plugged only one hole, while new web applications are created every day that contain other holes. SQL Injection attacks remain an easy target for hackers, because software developers don't understand the nature of the vulnerability or how to write code to prevent it.

21.1 Objective: Write Dynamic SQL Queries

SQL is intended to be used in concert with application code. When you build SQL queries as strings and combine application variables into the string, this is commonly called *dynamic SQL.*[2]

[1] http://voices.washingtonpost.com/securityfix/heartlandIndictment.pdf

[2] Technically, any query parsed at runtime is dynamic SQL, but in common usage, it describes SQL that includes variable data.

SQL-Injection/obj/dynamic-sql.php

```
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id = $bug_id";
$stmt = $pdo->query($sql);
```

This simple example shows interpolating a PHP variable into a string. We intend that $bug_id is an integer so that by the time the database receives the query, the value of $bug_id is part of the query.

Dynamic SQL queries are a natural way to get the most out of a database. When you use application data to specify how you want to query a database, you're using SQL as a two-way language. Your application is having a kind of dialogue with the database.

However, it's not too hard to make your software do tasks that you want it to do—the harder challenge is making your software secure so it doesn't allow actions that you don't want it to do. Software defects resulting from SQL Injection are failures to satisfy the latter.

21.2 Antipattern: Execute Unverified Input As Code

SQL injection happens when you interpolate some content into an SQL query string and the content modifies the syntax of your query in ways you didn't intend. In the classic example of SQL Injection, the value you interpolate into your string finishes the SQL statement and executes a second complete statement. For instance, if the value of the $bug_id variable is *1234; DELETE FROM Bugs*, the resulting SQL shown earlier would look like this:

SQL-Injection/anti/delete.sql

```
SELECT * FROM Bugs WHERE bug_id = 1234; DELETE FROM Bugs
```

This type of SQL Injection can be spectacular, as shown in Figure 18, *Exploits of a mom*, on page 7.[3] Usually these flaws are more subtle, but still dangerous.

Accidents May Happen

Suppose you are writing a web interface to view the bugs database and one page allows you to view a project based on its name:

SQL-Injection/anti/ohare.php

```
<?php
$project_name = $_REQUEST["name"];
$sql = "SELECT * FROM Projects WHERE project_name = '$project_name'";
```

[3] Cartoon by Randall Munroe, used with permission (http://xkcd.com/327/).

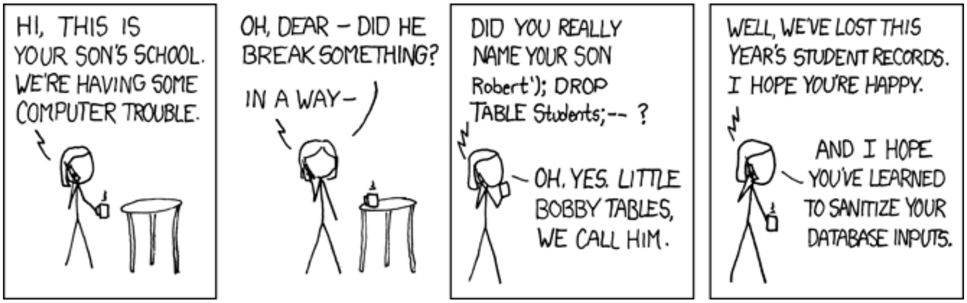

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Figure 18—Exploits of a mom

The trouble begins when your team is hired to develop software for O'Hare International Airport in Chicago. You naturally give the project a name like "O'Hare." How do you submit a request to view the project in your web application?

http://bugs.example.com/project/view.php?name=O'Hare

Your PHP code takes the value of that request parameter and interpolates it into the SQL query, but it produces a query that neither you nor the user intended:

SQL-Injection/anti/ohare.sql

```
SELECT * FROM Projects WHERE project_name = 'O'Hare'
```

Because a string is terminated by the first quote character it finds, the resulting expression contains a short string, 'O', followed by some extra characters, Hare', that make no sense in this context. The database can only report this as a syntax error. This is an honest accident. The risk of anything bad happening is low, because a statement with a syntax error can't execute. The greater risk is that the statement executes without error but does something you didn't intend.

The Top Web Security Threat

SQL Injection becomes a greater threat when an attacker can use this to manipulate your SQL statements. For example, your application may allow a user to change his or her password:

SQL-Injection/anti/set-password.php

```
<?php
$password = $_REQUEST["password"];
$userid = $_REQUEST["userid"];
$sql = "UPDATE Accounts SET password_hash = SHA2('$password')
    WHERE account_id = $userid";
```

A clever attacker who can guess how the request parameters are used in your SQL statement can send a carefully chosen string to exploit it:

```
http://bugs.example.com/setpass?password=xyzzy&userid=123 OR TRUE
```

After interpolating the string from the userid parameter into your SQL expression, the string has changed the syntax of the statement. Now it changes the password for *every* account in the database, not for one specific account:

SQL-Injection/anti/set-password.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = 123 OR TRUE;
```

This is key to understanding SQL Injection and also how to combat it: SQL Injection works by changing the syntax of the SQL statement before the statement is parsed. As long as you insert dynamic portions to the statement before it's parsed, you have a risk of SQL Injection.

There are countless ways a maliciously chosen string can alter the behavior of your SQL statements. It's limited only by the imagination of the attacker and your ability to protect your SQL statements.

The Quest for a Cure

Now that we know the threat of SQL Injection, the next natural question is, what do we need to do to protect code from being exploited? You may have read a blog or an article that described some single technique and claimed it's the universal remedy against SQL Injection. In reality, none of these techniques is proof against all forms of SQL Injection, so you need to use all of them in different cases.

Escaping Values

The oldest way to protect SQL queries from accidental unmatched quote characters is to *escape* any quote characters to prevent them from becoming the end of the quoted string. In standard SQL, you can use two quote characters to make one literal quote character:

SQL-Injection/anti/ohare-escape.sql

```
SELECT * FROM Projects WHERE project_name = 'O''Hare'
```

Most brands of database also support the backslash to escape the following quote character, just like most other programming languages do:

SQL-Injection/anti/ohare-escape.sql

```
SELECT * FROM Projects WHERE project_name = 'O\'Hare'
```

The idea is that you transform application data before you interpolate it into SQL strings. Most SQL programming interfaces provide a convenience function. For example, in PHP's PDO extension, use the quote() function to both delimit a string with quote characters and escape any literal quote characters within the string.

SQL-Injection/anti/ohare-escape.php

```
<?php
$project_name = $pdo->quote($_REQUEST["name"]);
$sql = "SELECT * FROM Projects WHERE project_name = $project_name";
```

This technique can reduce the risk of SQL Injection resulting from unmatched quote characters within the dynamic content. But it doesn't work as well for nonstring content.

SQL-Injection/anti/set-password-escape.php

```
<?php
$password = $pdo->quote($_REQUEST["password"]);
$userid = $pdo->quote($_REQUEST["userid"]);
$sql = "UPDATE Accounts SET password_hash = SHA2($password)
    WHERE account_id = $userid";
```

SQL-Injection/anti/set-password-escape.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = '123 OR TRUE'
```

You can't compare a numeric column directly to a string containing digits in all brands of database. Some databases may implicitly cast the string to a sensible numeric equivalent, but in standard SQL you have to use the CAST() function deliberately to convert a string to a numeric data type.

There are also obscure corner cases where strings in non-ASCII character sets can pass through a function intended to escape the quote characters but leave unescaped quote characters intact.[4]

Query Parameters

The solution most frequently cited as a panacea to SQL Injection is to use *query parameters*. Instead of interpolating dynamic values into your SQL string, leave *parameter placeholders* in the string as you prepare the query. Then provide a parameter value as you execute the prepared query.

SQL-Injection/anti/parameter.php

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Projects WHERE project_name = ?");
$params = array($_REQUEST["name"]);
$stmt->execute($params);
```

[4] See http://bugs.mysql.com/8378 for an example.

Many programmers recommend this solution because you don’t have to escape dynamic content or worry about flawed escaping functions. In fact, query parameters are a very strong defense against SQL Injection. But parameters aren’t a universal solution because the value of a query parameter is always interpreted as a single literal value.

- No lists of values can be a single parameter:

  SQL-Injection/anti/parameter.php

  ```
  <?php
  $stmt = $pdo->prepare("SELECT * FROM Bugs WHERE bug_id IN ( ? )");
  $stmt->execute(array("1234,3456,5678"));
  ```

  This works as though you provided a single string value composed of digits and commas, which doesn’t work the same as a series of integers:

  SQL-Injection/anti/parameter.sql

  ```
  SELECT * FROM Bugs WHERE bug_id IN ( '1234,3456,5678' )
  ```

- No table identifier can be a parameter:

  SQL-Injection/anti/parameter.php

  ```
  <?php
  $stmt = $pdo->prepare("SELECT * FROM ? WHERE bug_id = 1234");
  $stmt->execute(array("Bugs"));
  ```

  This works as though you had entered a string literal in place of the table name, which is simply a syntax error:

  SQL-Injection/anti/parameter.sql

  ```
  SELECT * FROM 'Bugs' WHERE bug_id = 1234
  ```

- No column identifier can be a parameter:

  SQL-Injection/anti/parameter.php

  ```
  <?php
  $stmt = $pdo->prepare("SELECT * FROM Bugs ORDER BY ?");
  $stmt->execute(array("date_reported"));
  ```

  In this example, the sort is a no-op, because the expression is a constant string, the same on every row:

  SQL-Injection/anti/parameter.sql

  ```
  SELECT * FROM Bugs ORDER BY 'date_reported';
  ```

- No SQL keyword can be a parameter:

  SQL-Injection/anti/parameter.php

  ```
  <?php
  $stmt = $pdo->prepare("SELECT * FROM Bugs ORDER BY date_reported ?");
  $stmt->execute(array("DESC"));
  ```

The parameter is interpreted as a literal string, not an SQL keyword. In this example, the result is a syntax error.

SQL-Injection/anti/parameter.sql

```
SELECT * FROM Bugs ORDER BY date_reported 'DESC'
```

Stored Procedures

Use of stored procedures is another method that many software developers claim is proof against SQL Injection vulnerabilities. Typically, stored procedures contain fixed SQL statements, parsed when you define the procedure.

However, it's possible to use dynamic SQL in stored procedures unsafely. In the following example, the input_userid argument is interpolated into the SQL query verbatim, which is unsafe.

SQL-Injection/anti/procedure.sql

```
CREATE PROCEDURE UpdatePassword(input_password VARCHAR(20),
  input_userid VARCHAR(20))
BEGIN
  SET @sql = CONCAT('UPDATE Accounts
    SET password_hash = SHA2(', QUOTE(input_password), ')
    WHERE account_id = ', input_userid);
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
END
```

Using dynamic SQL in a stored procedure is no more and no less safe than using dynamic SQL in application code. The input_userid argument can contain harmful content and produce an unsafe SQL statement:

SQL-Injection/anti/set-password.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = 123 OR TRUE;
```

Data Access Frameworks

You might see advocates of data access frameworks claim that their library protects your code from SQL Injection risks. This is a false claim for any framework that allows you to write SQL statements as strings.

Practice Good Hygiene

After I gave a presentation on a PHP data access framework that I had developed, a member of the audience approached me and asked, "Does your framework prevent SQL Injection?" I answered that it provides functions for quoting strings and using query parameters.

The young man looked puzzled. "But can it prevent SQL Injection?" he repeated. He was looking for an automatic way to ensure that he doesn't make a mistake that he doesn't know how to recognize himself.

What Was My Complete Query?

Many people think that using SQL query parameters is a way to quote values into an SQL statement automatically. This isn't accurate, and thinking about query parameters this way leads to misunderstanding about how they work.

The RDBMS server parses your SQL as you *prepare* the query. After this, nothing can change the syntax of that SQL query.

You provide values as you *execute* a prepared query. Each value you provide is used for each placeholder, one for one.

You can execute a prepared query again, substituting new parameter values for the old values. So, the RDBMS must keep track of the query and the parameter values separately. This is good for security.

This means that if you retrieve the prepared SQL query string, it doesn't contain any parameter values. It would be handy to see the SQL statement including parameter values if you're debugging or logging queries, but these values are never combined with the query in its human-readable SQL form.

The best way to debug your dynamic SQL statements is to log both the statement with parameter placeholders at prepare time and the parameter values at execute time.

> I told him the framework prevents SQL Injection like a toothbrush prevents cavities. You have to use it consistently to get the benefit.

No framework can force you to write safe SQL code. A framework may provide convenience functions to help you, but it's easy to bypass these functions and use common string manipulation to build an SQL statement unsafely.

21.3 How to Recognize the Antipattern

Practically every database application builds SQL statements dynamically. If you build any portion of an SQL statement by concatenating strings together or interpolating variables into strings, then the statement potentially exposes your application to SQL Injection attacks. SQL Injection vulnerabilities are so common that you should assume that you have some in any application that uses SQL, unless you've just completed a code review specifically to find and correct these issues.

21.4 Legitimate Uses of the Antipattern

This antipattern is different from most of the others in this book, in that there aren't any legitimate reasons for allowing your application to have a security vulnerability because of SQL Injection. It's your responsibility as a software developer to write code defensively and to help your peers to do so as well. Software is only as secure as its weakest link—make sure you're not responsible for that weakest link!

Rule #31: Check the Back Seat

If you like to watch monster movies, you know that creatures like to hide behind the driver seat of your car and grab you after you get in. The lesson is that you shouldn't assume there's no danger inside a familiar space like your car.

SQL Injection can take indirect forms. Even if you insert user-supplied data safely using query parameters, you might use that data later as you form dynamic SQL queries:

```
<?php
$sql1 = "SELECT last_name FROM Accounts WHERE account_id = 123";
$row = $pdo->query($sql1)->fetch();
$sql2 = "SELECT * FROM Bugs WHERE MATCH(description) AGAINST ('"
  . $row["last_name"] .  "')";
```

What would happen in the previous query if the user had spelled their name as *O'Hara*—or if they had deliberately entered their name to contain SQL syntax?