

SQL Injection

Are Your Web Applications Vulnerable?

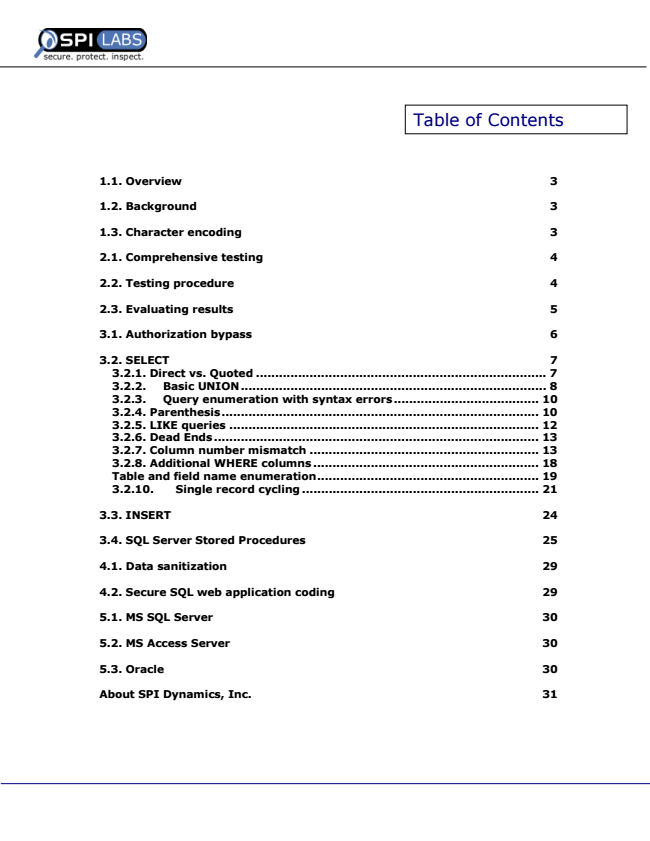


Table of Contents

**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**

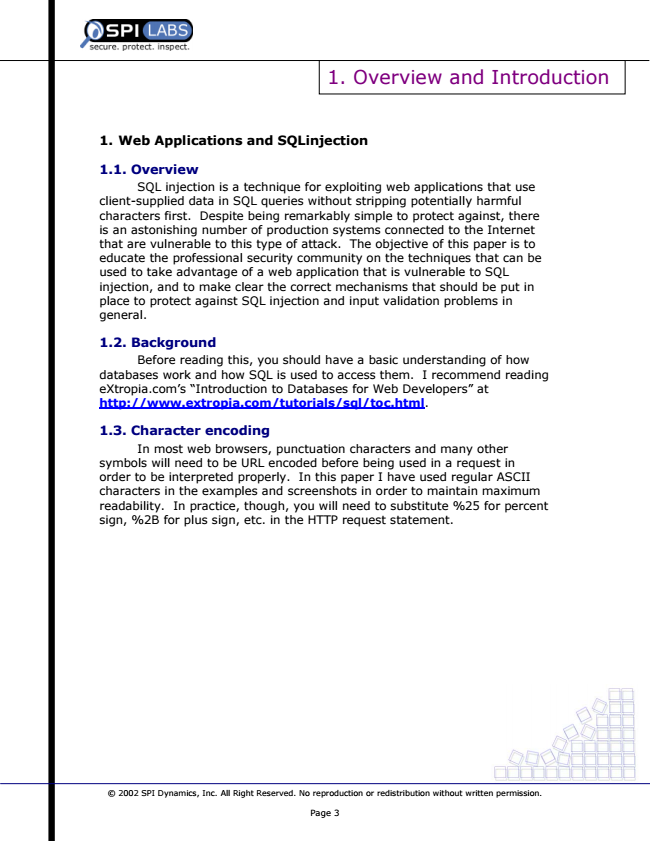
**4.2. Secure SQL web application coding 29**

**5.1. MS SQL Server 30**

**5.2. MS Access Server 30**

**5.3. Oracle 30**

**About SPI Dynamics, Inc. 31**

****

1. Overview and Introduction

**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**

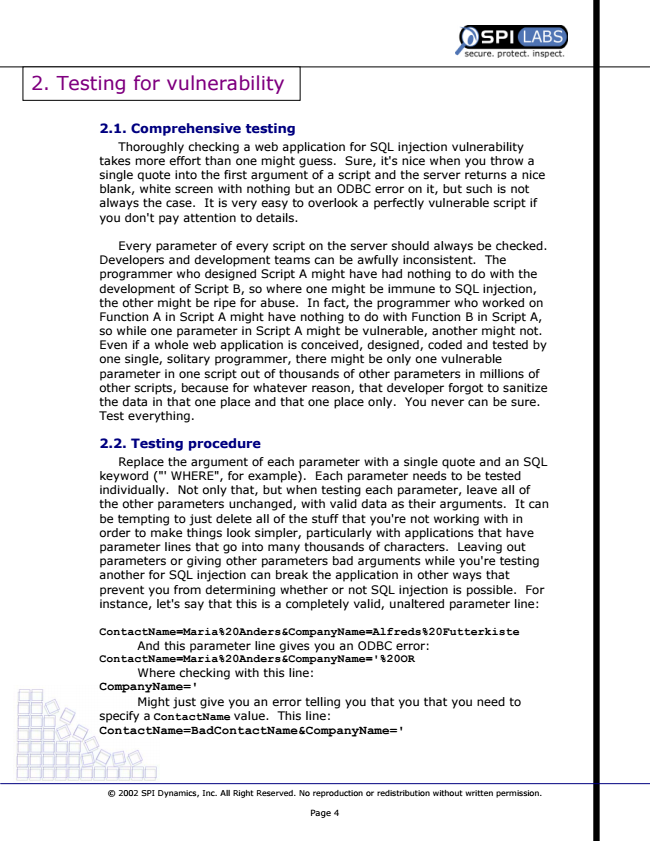
Before reading this, you should have a basic understanding of how databases work and how SQL is used to access them. I recommend reading eXtropia.com’s “Introduction to Databases for Web Developers” at http://www.extropia.com/tutorials/sql/toc.html.

**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.

Page 3



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**

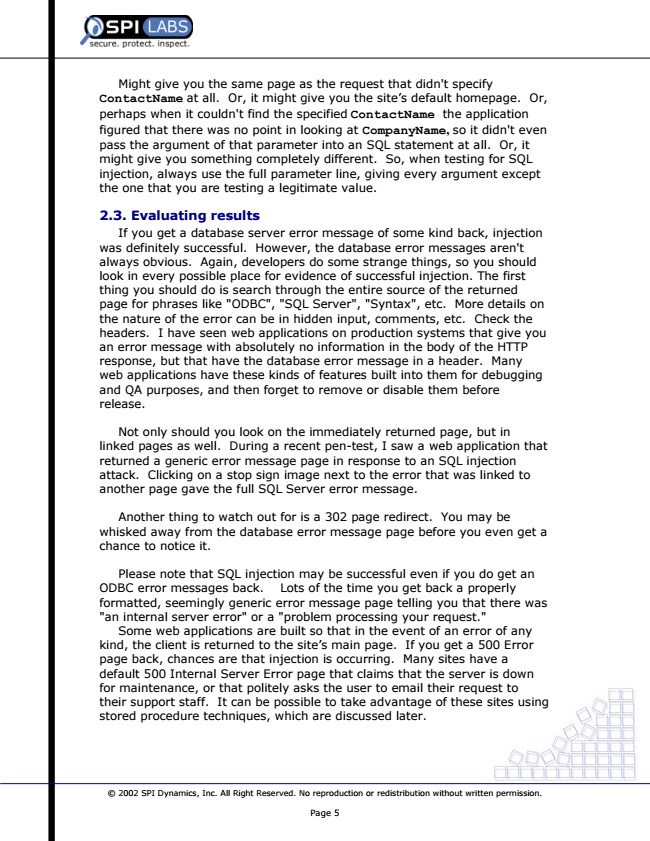
**And this parameter line gives you an ODBC error: ContactName=Maria%20Anders&CompanyName='%20OR**

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.

Page 4



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.

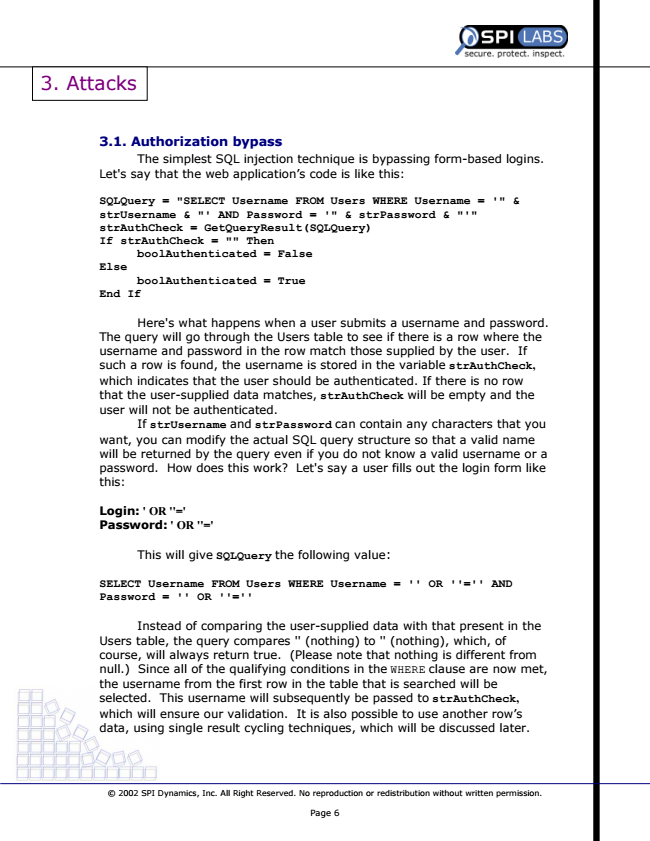
Another thing to watch out for is a 302 page redirect. You may be whisked away from the database error message page before you even get a chance to notice it.

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.

Page 5



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 ''='**

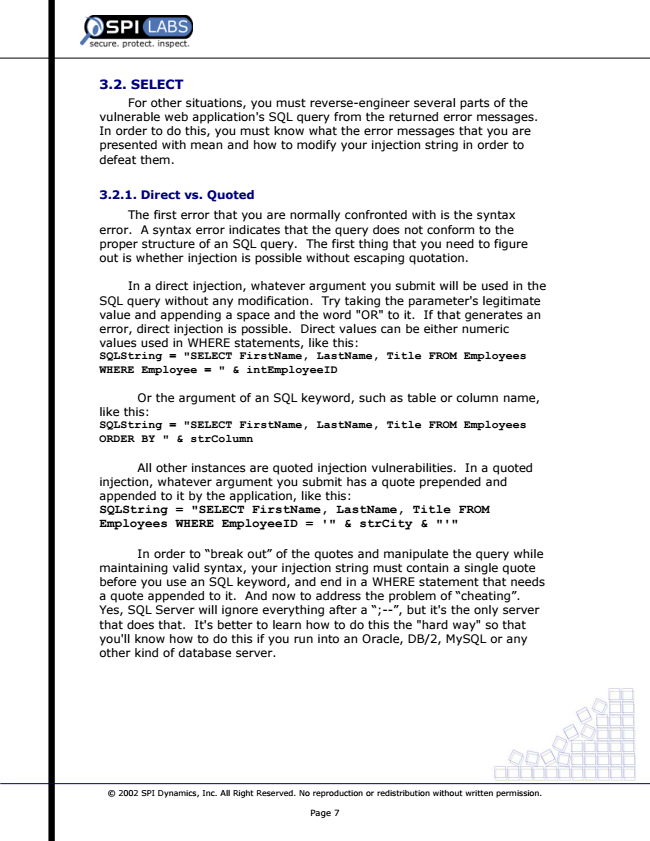
This will give SQLQuery the following value:

**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.

Page 6



**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

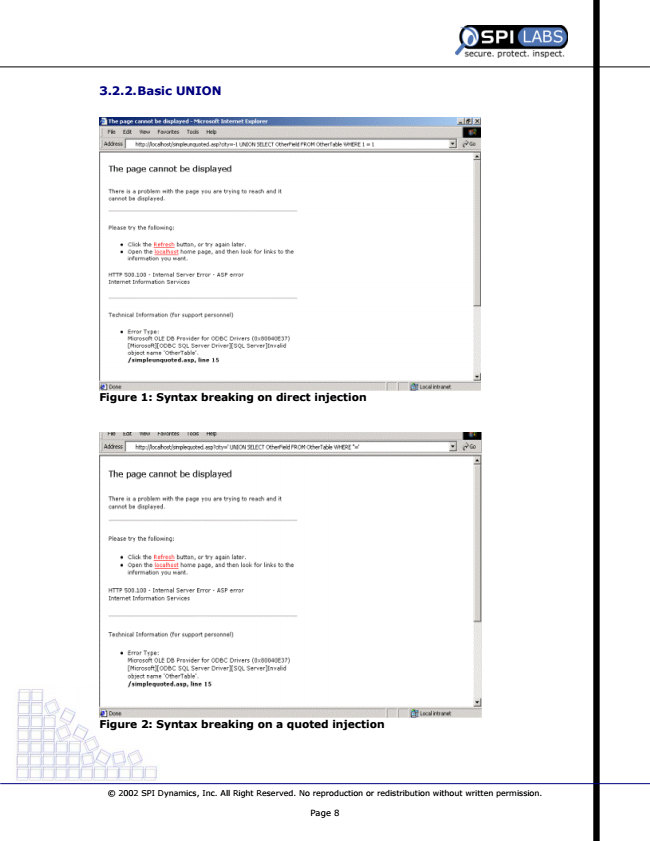
**Or the argument of an SQL keyword, such as table or column name, like this: SQLString = "SELECT FirstName, LastName, Title FROM Employees ORDER BY " & strColumn**

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.

Page 7



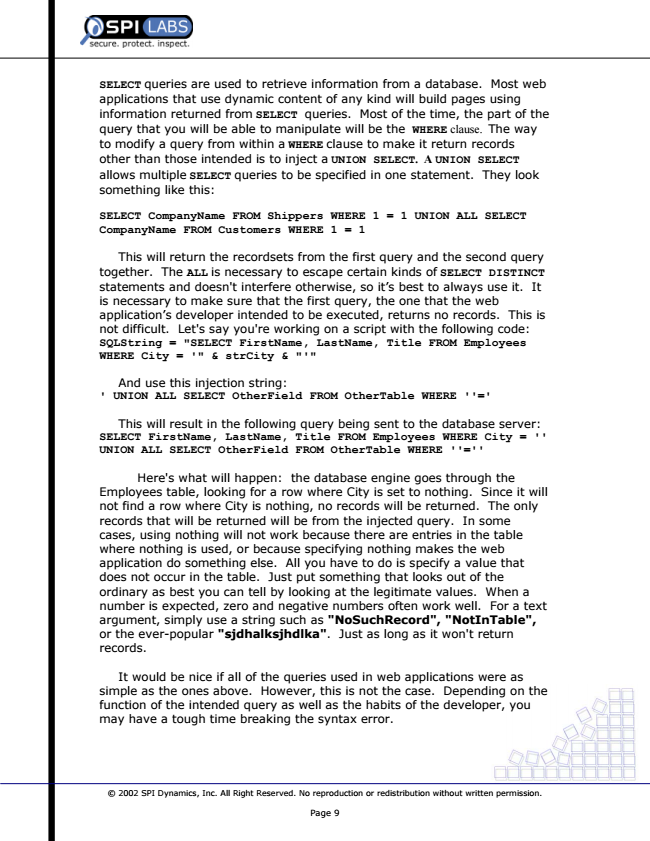
**3.2.2. Basic UNION**

**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.

Page 8



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 ''='**

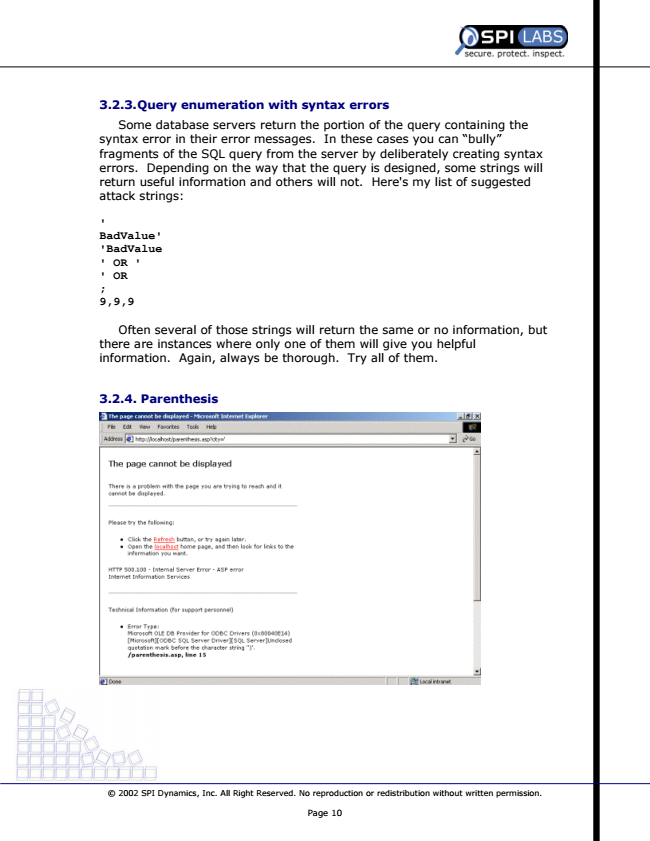
**This will result in the following query being sent to the database server: SELECT FirstName, LastName, Title FROM Employees WHERE City = '' UNION ALL SELECT OtherField FROM OtherTable WHERE ''=''**

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.

Page 9



**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.

Page 10