

Automating DLL Hijack Discovery



Justin Bui · Follow

Published in Posts By SpecterOps Team Members · 11 min read · Jun 30, 2020



--



1



Introduction

This blogpost will describe the concept of dynamic-link library (DLL) search order hijacking and how it can be used for userland persistence on Windows systems. This technique is mapped to MITRE ATT&CK under [DLL Search Order Hijacking.\(T1038\)](#).

DLL hijacking is useful to an attacker for a myriad of reasons, but this post will focus on its use for persistence when combined with autostart applications. For example, since Slack and Microsoft Teams start on boot (by default), a DLL hijack in one of these applications would allow an attacker persistent access to their target whenever the user logs in.

After introducing the concept of DLLs, DLL search order, and DLL hijacking, I explore the process of automating DLL hijack discovery (<https://github.com/slyd0g/DLLHijackTest>). This post will cover DLL hijack discovery in Slack, Microsoft Teams, and Visual Studio Code.

Lastly, I noticed numerous DLL hijacks that were shared between the different applications, investigated the root cause, and discovered that applications using certain Windows API calls are subject to a DLL hijack when not running out of `C:\Windows\System32\`.

I want to give a big shoutout to my coworker, Josiah Massari (@[Airzero24](#)), for initially finding some of these DLL hijacks, explaining his methodology, and inspiring me to automate the discovery.

What is a DLL?

A DLL is a library that contains code and data that can be used by more than one program at the same time. ([Source](#))

Functionality within a DLL can be leveraged by a Windows application using one of the `LoadLibrary*` functions. Applications can reference DLLs custom-created for their application or Windows DLLs already on disk in System32.

Developers can load DLLs from System32 to use functionality already implemented in Windows within their application without having to write the functionality from scratch.

For example, a developer that needs to make HTTP requests can leverage the WinHTTP library (`winhttp.dll`) instead of implementing HTTP requests using raw sockets.

DLL Search Order and Hijacking

Since DLLs exist as files on disc, you may ask yourself how does an application know where to load DLLs from? Microsoft has documented the DLL search order thoroughly [here](#).

Since Windows XP SP2, safe DLL search mode has been enabled by default (`HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode`). With safe DLL search mode enabled, the search order is as follows:

1. The directory from which the application loaded.
2. The system directory. Use the [GetSystemDirectory](#) function to get the path of this directory.
3. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched.
4. The Windows directory. Use the [GetWindowsDirectory](#) function to get the path of this directory.
5. The current directory.
6. The directories that are listed in the PATH environment variable. Note that this does not include the per-application path specified by the **App Paths** registry key. The **App Paths** key is not used when computing the DLL search path.

A system can contain multiple versions of the same dynamic-link library (DLL). Applications can control the location from which a DLL is loaded by specifying a full path or using another mechanism such as a manifest. ([Source](#))

If an application does not specify where to load a DLL from, Windows will default to the DLL search order shown above. The first location in the DLL search order, the directory from which the application is loaded, is of interest to attackers.

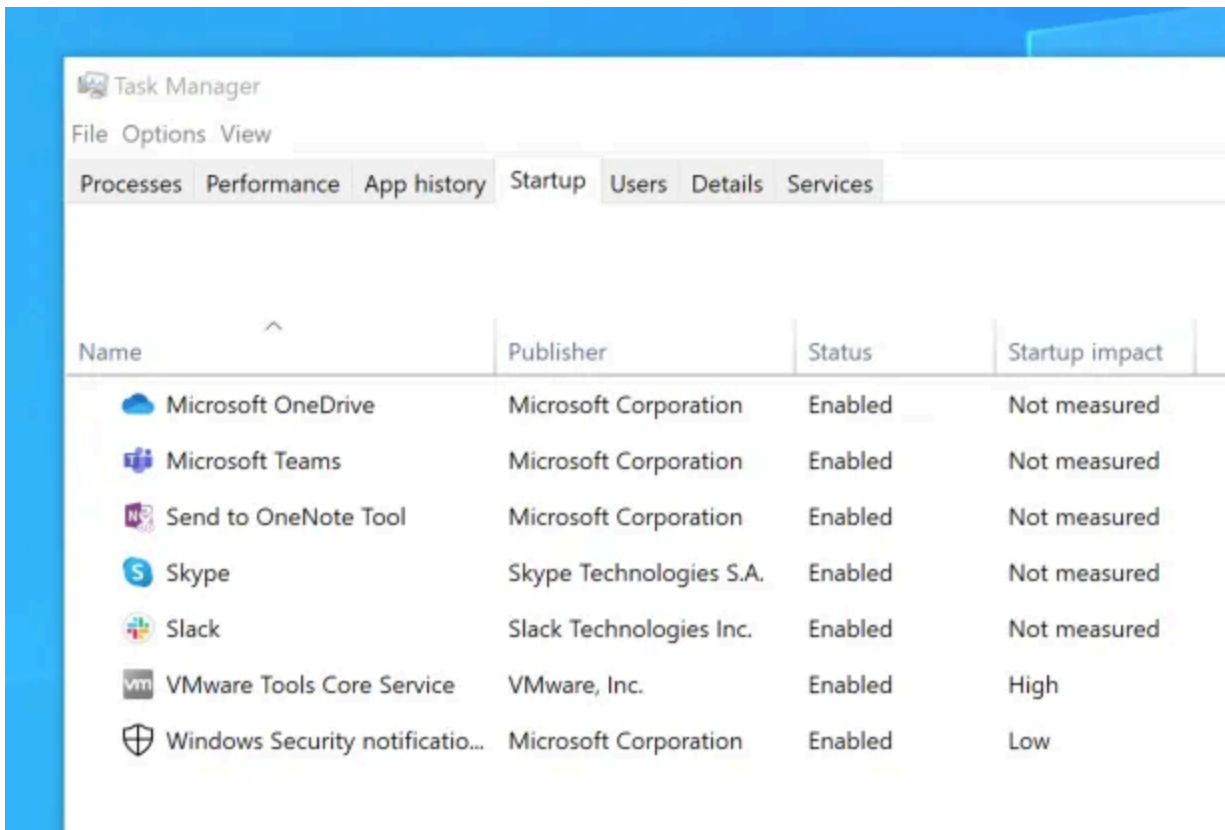
If the application developer wants to load DLLs from `C:\Windows\System32` , but did not explicitly write the application to do so, a malicious DLL planted in the application directory would be loaded before the legitimate DLL in

System32. This malicious DLL load is referred to as a DLL hijack and is used by attackers to load malicious code into trusted/signed applications.

Using DLL Hijacking for Persistence

DLL hijacking can be used for persistence when a vulnerable application/service is started and a malicious DLL has been planted in the vulnerable location. My coworker, [@Airzero24](#), discovered a DLL hijack in Microsoft OneDrive, Microsoft Teams, and Slack in the form of `userenv.dll`.

These particular programs were targeted because, by default, they are configured to start when Windows boots up. This can be seen below in the Task Manager:



Windows Applications Configured to Automatically Start

To verify the DLL hijack, I created a DLL shellcode loader that would launch a Cobalt Strike Beacon. I renamed the malicious DLL to `userenv.dll` and copied it to the directory of the vulnerable application. I started the application and saw my new Beacon callback.

Cobalt Strike Beacon through DLL Hijack

Using Process Explorer, I'm able to verify that my malicious DLL was indeed loaded by the vulnerable application.

Process Explorer Showing Malicious DLL Loaded

Automating DLL Hijack Discovery

After confirming the previously known DLL hijack, I wanted to see if I could find other DLL hijacks that could be used operationally.

The code used in my testing can be found here:

<https://github.com/slyd0g/DLLHijackTest>

Case Study: Slack

To begin this process, I started Process Monitor (ProcMon) with the following filters:

- **Process Name** is *slack.exe*
- **Result** contains *NOT FOUND*
- **Path** ends with *.dll*

Filters Missing DLLs with ProcMon

Next, I started Slack and observed ProcMon for any DLLs that Slack was searching for but could not find.

Potential DLL Hijacks Found with ProcMon

I exported this data from ProcMon as a CSV file for easy parsing within PowerShell.

With my current shellcode loader DLL, I wouldn't be able to easily determine the names of the DLLs that were successfully loaded by Slack. I created a new DLL that used `GetModuleHandleEx` and `GetModuleFileName` to determine the name of the loaded DLL and write it to a text file.

My next goal was to parse the CSV file for a list of DLL paths, loop through this list of paths, copy my test DLL to the specified path, start the target process, stop the target process, and remove the test DLL. If the test DLL was successfully loaded, it would write its file name to a results file.

When this process completes, I would (hopefully) have a list of valid DLL hijacks written to a text file.

The PowerShell script in my DLLHijackTest project does all the magic. It accepts a path to the CSV file generated by ProcMon, a path to your

malicious DLL, a path to the process you want to start, and any arguments you want to pass to the process.

Get-PotentialDLLHijack Parameters

Get-PotentialDLLHijack.ps1

A few minutes later, I check the text file I specified in my “malicious” DLL for valid DLL hijacks. I found the following hijacks for Slack:

```
PS C:\Users\John\Desktop> Get-PotentialDLLHijack -CSVPath
.\Logfile.CSV -MaliciousDLLPath .\DLLHijackTest.dll -ProcessPath
"C:\Users\John\AppData\Local\slack\slack.exe"

C:\Users\John\AppData\Local\slack\app-4.6.0\WINSTA.dll
C:\Users\John\AppData\Local\slack\app-4.6.0\LINKINFO.dll
C:\Users\John\AppData\Local\slack\app-4.6.0\ntshrui.dll
C:\Users\John\AppData\Local\slack\app-4.6.0\svcli.dll
C:\Users\John\AppData\Local\slack\app-4.6.0\cscapi.dll
C:\Users\John\AppData\Local\slack\app-4.6.0\KBDUS.DLL
```

Case Study: Microsoft Teams

Running through the above process again:

- 1. Use ProcMon to identify potential DLL hijacks, export this data as a CSV file.
- 2. Identify the path of the process to start.
- 3. Identify any arguments you want to pass to the process.
- 4. Run `Get-PotentialDLLHijack.ps1` with the appropriate arguments.

I found the following hijacks for Microsoft Teams:

```
PS C:\Users\John\Desktop> Get-PotentialDLLHijack -CSVPath
.\Logfile.CSV -MaliciousDLLPath .\DLLHijackTest.dll -ProcessPath
"C:\Users\John\AppData\Local\Microsoft\Teams\Update.exe" -
ProcessArguments '--processStart "Teams.exe"

C:\Users\John\AppData\Local\Microsoft\Teams\current\WINSTA.dll
C:\Users\John\AppData\Local\Microsoft\Teams\current\LINKINFO.dll
C:\Users\John\AppData\Local\Microsoft\Teams\current\ntshrui.dll
C:\Users\John\AppData\Local\Microsoft\Teams\current\srvcli.dll
C:\Users\John\AppData\Local\Microsoft\Teams\current\cscapi.dll
C:\Users\John\AppData\Local\Microsoft\Teams\current\WindowsCodecs.
dll
C:\Users\John\AppData\Local\Microsoft\Teams\current\TextInputFrame
work.dll
```

Note: I had to make a small modification to the PowerShell script to kill `Teams.exe` since my script attempts to kill the process that it tried to start, which in this case was `Update.exe`.

Case Study: Visual Studio Code

Repeating the process outlined above, I found the follow hijacks for Visual Studio Code:

```
PS C:\Users\John\Desktop> Get-PotentialDLLHijack -CSVPath
.\Logfile.CSV -MaliciousDLLPath .\DLLHijackTest.dll -ProcessPath
"C:\Users\John\AppData\Local\Programs\Microsoft VS Code\Code.exe"

C:\Users\John\AppData\Local\Programs\Microsoft VS Code\WINSTA.dll
C:\Users\John\AppData\Local\Programs\Microsoft VS
Code\LINKINFO.dll
C:\Users\John\AppData\Local\Programs\Microsoft VS Code\ntshrui.dll
C:\Users\John\AppData\Local\Programs\Microsoft VS Code\srvcli.dll
C:\Users\John\AppData\Local\Programs\Microsoft VS Code\cscapi.dll
```

Shared DLL Hijacks

I noticed that Slack, Microsoft Teams, and Visual Studio Code shared the following DLL hijacks:

- `WINSTA.dll`
- `LINKINFO.dll`
- `ntshrui.dll`
- `srvcli.dll`
- `cscapi.dll`

I found this interesting and wanted to understand what was causing this behavior.

Methodology: Understanding Shared DLL Hijacks

I observed the stack trace when Slack attempted to load `WINSTA.dll`, `LINKINFO.dll`, `ntshrui.dll`, `srvcli.dll`, and `cscapi.dll`.

Delay-Loaded DLLs

I noticed similarities in the stack trace when `WINSTA.dll`, `LINKINFO.dll`, `ntshrui.dll`, and `srvcli.dll` were loaded.

Stack Trace when Code.exe Attempts to Load WINSTA.dll

Stack Trace when Teams.exe Attempts to Load LINKINFO.dll

Stack Trace when Slack Attempts to Load ntshrui.dll

The stack trace consistently contained a call to `_tailMerge_<dllname>.dll`, `delayLoadHelper2`, followed by `LdrResolveDelayLoadedAPI`. This behavior was consistent between all three applications.

I determined this behavior was related to delay-loaded DLLs. From the stack trace when `WINSTA.dll` was being loaded, I could see the module responsible for this delayed-load was `wsapi32.dll`.

I opened `wsapi32.dll` in Ghidra and used `Search -> For Strings -> Filter: WINSTA.dll`. Double-clicking the found string leads you to its location in memory.

“WINSTA.dll” String in wsapi32.dll

Right-clicking the location in memory, we are able to find any references to this address.

Following References to WINSTA.dll

Following the references, we see the `WINSTA.dll` string is being passed to a structure called `ImgDelayDescr`. Looking at [documentation](#) on this structure, we can confirm it is related to delay-loaded DLLs.

```
typedef struct ImgDelayDescr {
    DWORD      grAttrs;           // attributes
    RVA         rvaDLLName;       // RVA to dll name
    RVA         rvaHmod;          // RVA of module handle
    RVA         rvaIAT;           // RVA of the IAT
    RVA         rvaINT;           // RVA of the INT
    RVA         rvaBoundIAT;      // RVA of the optional bound IAT
    RVA         rvaUnloadIAT;     // RVA of optional copy of
original IAT
    DWORD      dwTimeStamp;       // 0 if not bound,
                                   // O.W. date/time stamp of DLL
bound to (Old BIND)
} ImgDelayDescr, * PImgDelayDescr;
```

This structure can be passed to `__delayLoadHelper2`, which will use `LoadLibrary`/`GetProcAddress` to load the specified DLL and patch the address of the imported function in the delay load import address table (IAT).

```
FARPROC WINAPI __delayLoadHelper2(
    PCImgDelayDescr pidd, //Const pointer to a ImgDelayDescr
    struct
        FARPROC * ppfnIATEntry //A pointer to the slot in delay load
IAT
);
```

Finding other references to our `ImgDelayDescr` structure, we can find the call to `__delayLoadHelper2`, which then calls `ResolveDelayLoadedAPI`. I've renamed the function name, types, and variables to make it easier to understand.

__delayLoadHelper2 and ResolveDelayLoadedAPI in Ghidra

Great! This matches what we saw in our ProcMon stack trace when Slack attempted to load `WINSTA.dll`.

__delayLoadHelper2 and ResolveDelayLoadedAPI in ProcMon

This behavior was consistent between `WINSTA.dll`, `LINKINFO.dll`, `ntshrui.dll`, and `srvcli.dll`. The primary difference between each delay-loaded DLL was the “parent” DLL. In all three applications:

- `wtsapi32.dll` delay-loaded `WINSTA.dll`
- `shell32.dll` delay-loaded `LINKINFO.dll`
- `LINKINFO.dll` delay-loaded `ntshrui.dll`
- `ntshrui.dll` delay-loaded `srvcli.dll`

Observe anything interesting? It seems `shell32.dll` loads `LINKINFO.dll`, which loads `ntshrui.dll`, which finally loads `srvcli.dll`. This leads us to our final shared DLL hijack, `cscapi.dll`.

DLL Hijacks in NetShareGetInfo and NetShareEnum

I observed the stack trace when Slack attempted to load `cscapi.dll` and saw a `LoadLibraryExW` call which appeared to originate from `srvcli.dll`.

Stack Trace when Loading cscapi.dll

I opened `srvcli.dll` in Ghidra and used `Search -> For Strings -> Filter: cscapi.dll`. Double-clicking the found string and following the references leads to the expected `LoadLibrary` call.

srvcli.dll calls LoadLibrary on cscapi.dll

Renaming the function containing the `LoadLibrary` call and following the references yielded two function locations:

- [NetShareEnum](#)
- [NetShareGetInfo](#)

NetShareEnum Loads cscapi.dll

NetShareGetInfo Loads cscapi.dll

I verified this with PoC programs that call `NetShareEnum` and `NetShareGetInfo`:

NetShareEnum.exe Loads cscapi.dll

NetShareGetInfo.exe Loads cscapi.dll

Results

The following DLL hijacks exist in Slack:

```
C:\Users\John\AppData\Local\slack\app-4.6.0\WINSTA.dll
C:\Users\John\AppData\Local\slack\app-4.6.0\LINKINFO.dll
C:\Users\John\AppData\Local\slack\app-4.6.0\ntshrui.dll
C:\Users\John\AppData\Local\slack\app-4.6.0\srvcli.dll
C:\Users\John\AppData\Local\slack\app-4.6.0\cscapi.dll
C:\Users\John\AppData\Local\slack\app-4.6.0\KBDUS.DLL
```

The following DLL hijacks exist in Microsoft Teams:

```
C:\Users\John\AppData\Local\Microsoft\Teams\current\WINSTA.dll
C:\Users\John\AppData\Local\Microsoft\Teams\current\LINKINFO.dll
C:\Users\John\AppData\Local\Microsoft\Teams\current\ntshrui.dll
```

```
C:\Users\John\AppData\Local\Microsoft\Teams\current\srccli.dll
C:\Users\John\AppData\Local\Microsoft\Teams\current\cscapi.dll
C:\Users\John\AppData\Local\Microsoft\Teams\current\WindowsCodecs.dll
C:\Users\John\AppData\Local\Microsoft\Teams\current\TextInputFramework.dll
```

The following DLL hijacks exist in Visual Studio Code:

```
C:\Users\John\AppData\Local\Programs\Microsoft VS Code\WINSTA.dll
C:\Users\John\AppData\Local\Programs\Microsoft VS Code\LINKINFO.dll
C:\Users\John\AppData\Local\Programs\Microsoft VS Code\ntshrui.dll
C:\Users\John\AppData\Local\Programs\Microsoft VS Code\srccli.dll
C:\Users\John\AppData\Local\Programs\Microsoft VS Code\cscapi.dll
```

Additionally, I found that programs using `NetShareEnum` and `NetShareGetInfo` introduce a DLL hijack in the form of `cscapi.dll` due to a hard-coded `LoadLibrary` call. I confirmed this behavior with Ghidra and a PoC.

Conclusion

To recap, DLL hijacking is a method for attackers to obtain code execution in signed/trusted applications. I created tooling to help automate the discovery of DLL hijacks. Using this tooling, I found DLL hijacks within Slack, Microsoft Teams, and Visual Studio Code.

I noticed the three applications had overlap with their DLL hijacks and investigated the root cause. I highlighted my methodology for digging into this subject. I learned about delay-loaded DLLs and identified two API calls that introduce DLL hijacks into any program that calls them:

- `NetShareEnum` loads `cscapi.dll`
- `NetShareGetInfo` loads `cscapi.dll`

Thanks for taking the time to read this post, I hope you learned a little about the Windows API, Ghidra, ProcMon, DLLs, and DLL hijacking!

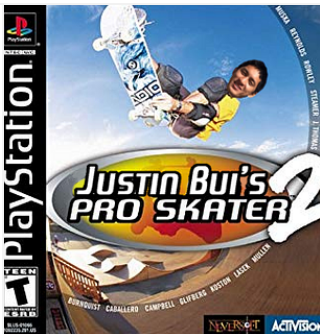
References

Big shoutout to my coworkers Daniel Heinsen (@hotnops), Lee Christensen (@tifkin), and Matt Hand (@matterpreter) for helping me stumble through Ghidra/ProcMon!

slyd0g/DLLHijackTest

DLL and PowerShell script to assist with finding DLL hijacks - slyd0g/DLLHijackTest

github.com



DLL Search Order Hijacking

Windows systems use a common method to look for required DLLs to load into a program. Adversaries may take advantage of...

attack.mitre.org



What is a DLL?

This article describes what a dynamic link library (DLL)...

support.microsoft.com

Dynamic-Link Library Search Order - Win32 apps

Applications can control the location from which a DLL is loaded by specifying a full path or using another mechanism...

docs.microsoft.com

Triaging a DLL planting vulnerability - Microsoft Security Response Center

DLL planting (aka binary planting/hijacking/preloading) resurface every now and then, it is not always clear on how...

msrc-blog.microsoft.com

How does the Import Library work? Details?

I know this may seem quite basic to geeks. But I want to make it crystal clear. When I want to use a Win32 DLL, usually...

stackoverflow.com



MicrosoftDocs/cpp-docs

The helper function for linker-supported delayed loading is what actually loads the DLL at run time. You can modify the...

github.com



Linker Support for Delay-Loaded DLLs

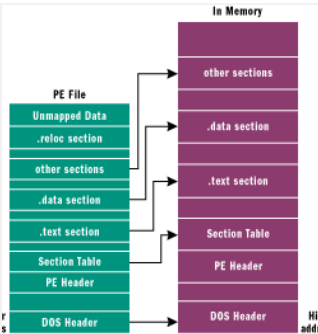
The MSVC linker now supports the delayed loading of DLLs. This relieves you of the need to use the Windows SDK...

docs.microsoft.com

Inside Windows: Win32 Portable Executable File Format in Detail



long time ago, in a galaxy far away, I wrote one of my first articles for Microsoft Systems Journal (now MSDN® ...

docs.microsoft.com



- Windows
- Red Team
- Dll
- Dll Hijacking
- Automation

 --  1



Written by Justin Bui

189 Followers · Writer for Posts By SpecterOps Team Members

Follow 

I break computers and skateboards and write about the former