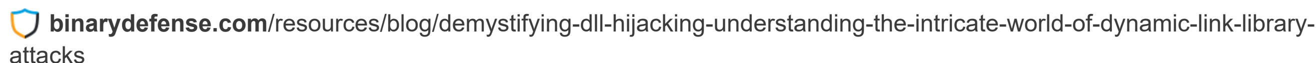
Demystifying DLL Hijacking Understanding the Intricate World of Dynamic Link Library Attacks


binarydefense.com/resources/blog/demystifying-dll-hijacking-understanding-the-intricate-world-of-dynamic-link-library-attacks

August 23, 2023

[
By: Jonny Johnson, Senior Researcher of Adversarial Techniques and Capabilities at Binary Defense

Introduction

DLL Hijack-based attacks have been popular within the offensive community for several years. This technique has been used to achieve initial access, persistence, or privilege escalation in several environments. Due to the volume of DLL loads that happen in an environment, these attacks have been historically perceived as difficult to identify and detect. I think there is a different lens we can look through when it comes to this attack that would help us understand how to identify this behavior. During this post I will talk about the basics of DLL hijacking, the different types of hijacks, and detection ideas.

Internals

A lot of vulnerabilities come by way of weak file/folder permissions within a computer. DLL hijacking isn't any different. DLL hijacks were created whenever it was identified that there were a lot of application folders out there that weren't limited access across users. DLL hijack-based attacks are simple, from the most basic perspective: An attacker with write access to a folder, drops a DLL in the place of a (missing) DLL so that the application that needs it will load it. However, there are some variational nuances that has come out over those years of people performing hijacking techniques. We are going to focus on three in this post:

1. Search Order Hijacking
2. Sideloading
3. DLL Proxying

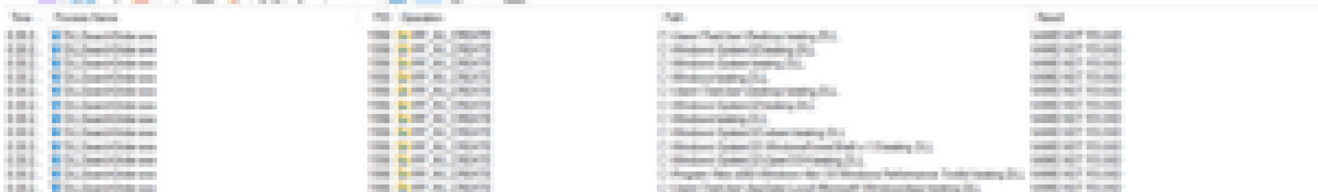
DLL Search Order Hijack:

It is very common for applications to call one of the LoadLibrary functions load a DLL they need within their internal functionality. Often these applications will make these calls without specifying an absolute path like so:

LoadLibrary(L"testing.DLL");

When this happens Windows will go through a pre-defined search order to attempt to find the DLL. On 64-bit systems that search looks like the following:

1. The folder the application is launched out of
2. C:\Windows\System32\
3. C:\Windows\system\
4. C:\Windows\



This gives a unique opportunity for an attacker to add their DLL in one of those file paths. However, in order to have that ability the attacker needs to have write access to that folder. Due to this this it is common to see DLLs dropped in the applications folder, like <Application>, Temp, or AppData. It's possible to see this within the Program Files directory as well if the application manually changes the security descriptor of the applications folder, giving the proper write access the attacker would need. This leads into the next DLL hijack technique that is probably the most common.

DLL Sideloading

Sideloading seems to be the most popular hijacking technique. However, in my opinion, DLL sideloading is the same as DLL search order hijacking because someone is either:

1. Dropping the DLL in the applications folder path and executing the application.
2. Dropping a signed application binary (.exe) with the malicious DLL and executing the application.

Which if you remember above, is possible because the applications folder path is the first directory checked when the application searches for the DLL via LoadLibrary without an absolute path. Sideloading does seem to be quite popular and a go-to technique in the wild/red teams to secure code execution. If you can find an application that you know loads a DLL you can substitute, then it is easy to drop those files and secure the desired execution.

DLL Proxy

DLL proxying, in my opinion, is one of the coolest DLL hijacking techniques because it involves the following:

1. Identify an application that leverages some export function from a DLL.

2. Create a DLL that forwards those exports to the legitimate DLL. Finding one that has fewer exports makes this easier.
3. Drop the DLL in the place of the legitimate DLL.
4. Start application to load the malicious DLL and the legitimate DLL.

Typically, on step 2-3 there are 2 options:

1. Renaming the legitimate DLL to a different name and have the malicious DLL point to that. This is useful if the application the attacker is targeting loads a DLL that exists within the same folder. Keep in mind the attacker needs write access to the folder path of the application they want to load and the appropriate access to change the name of the legitimate DLL.
2. The attacker has found an application that loads a DLL without a specifying an absolute path, but the legitimate DLL lives within a folder path lower in the pre-defined search order like C:\Windows\System32. So, they drop their malicious DLL in the applications path and point to the legitimate DLL in C:\Windows\System32.

This is a unique way of getting a malicious payload to load within an application, as an attacker isn't taking the place of a DLL per say, but more forwarding the requests.

Notable Others

There are other forms of DLL hijacking, like COM hijacking, but if I were to write a post about COM hijacking this post would be way too long. I plan to write one and some findings with COM hijacking in the near future. The only one of note that I think is valuable to add is path redirection, where the adversary changes where the application will search for DLLs instead of using the pre-defined search order. In this post we won't talk about either this method or COM hijacking. The 3 methods of DLL hijacking above are the main ones. There are nuance names like relative path hijacking, which is essentially writing a legitimate application and a malicious DLL to a folder the attacker has access to. However, this is the same as search order/sideloading.

Generalized Detection

DLL hijacking can be difficult to detect, as there are many DLLs loaded into different applications and it is difficult to determine if a DLL holds malicious code. However, there are a couple of tricks I have found to be useful in terms of a detection strategy for this. Note, these suggestions/queries are going to be generalized and if you want to use them you will probably have to do some internal tuning. There is also always nuance when it comes to detection suggestions, so these are not meant to be "catch-all" suggestions. Queries will be shown in Kusto for MDE, but the same could apply across different telemetry sensors like Sysmon.

Search Order/Sideloading:

- Unsigned DLL Creation from a medium/high integrity level.
  - Target the following directories:
    - C:\ProgramData\<Application Name>
    - Temp Directory
    - AppData Directory
    - Program Files directory

Sysmon Event 11 is a good choice in terms of telemetry collection for this. Unfortunately, Microsoft Defender for Endpoint (MDE) doesn't expose DLL signatures where you can query them. So, it would be hard to determine if the DLL was signed or not.

- The same DLL from above was loaded into an application where the process was launched at a higher integrity level.

```
//Query combines file creation events where the creator of the file is not SYSTEM and image load events where the file was loaded by a HIGH or SYSTEM level process
//Can be good for: SearchOrder Hijacking and SideLoading
//FPs can include legit software that create an application as a service and launches it, these can be filtered out accordingly.
DeviceFileEvents
| where ActionType == "FileCreated" and FileName endswith "dll" and InitiatingProcessIntegrityLevel != "System" and FolderPath has_any ("ProgramData", "Temp", "AppData")
| project Timestamp, DeviceName, FileName, PreviousFileName, FolderPath, PreviousFolderPath, RequestAccountName, InitiatingProcessAccountName
| join kind = inner (
DeviceImageLoadEvents
| where FileName endswith "dll" and  FolderPath has_any ("ProgramData", "Temp", "AppData") and (InitiatingProcessIntegrityLevel == "System" or InitiatingProcessIntegrityLevel == "High")
| project InitiatingProcessId, FileName, FolderPath, InitiatingProcessIntegrityLevel, InitiatingProcessCommandLine
) on $left.FileName == $right.FileName
```

DLL Proxying:

- File renames of a DLL:

```
DeviceFileEvents
| where ActionType == "FileRenamed" and FileName endswith "dll" and InitiatingProcessIntegrityLevel != "System"and FolderPath contains PreviousFolderPath
| project Timestamp, DeviceName, FileName, PreviousFileName, FolderPath, PreviousFolderPath, RequestAccountName, InitiatingProcessAccountName
```

- A DLL being renamed, but still loaded into an application.

```
DeviceFileEvents
| where ActionType == "FileRenamed" and FileName endswith "dll" and
InitiatingProcessIntegrityLevel != "System" and FolderPath contains PreviousFolderPath
| project Timestamp, DeviceName, FileName, PreviousFileName, FolderPath, PreviousFolderPath,
RequestAccountName, InitiatingProcessAccountName
| join kind = inner (
DeviceImageLoadEvents
| where FileName endswith "dll" and  FolderPath has_any ("ProgramData", "Temp", "AppData")
and InitiatingProcessIntegrityLevel == "System" or InitiatingProcessIntegrityLevel ==
"High")| project InitiatingProcessId, FileName, FolderPath, InitiatingProcessIntegrityLevel
) on $left.FileName == $right.FileName
```

> Same directory proxying (pointing to another dll in the same directory, while loading malicious DLL)

```
//Query looks for file creation events from a non-system level user
//Performs a join wtih image load events to see if the file that was created was loaded into
a high or system level context
//Also looks to see if 2 similar DLLs were loaded into the same process within the same
folder.
DeviceFileEvents
| where ActionType == "FileCreated" and FileName endswith "dll" and
InitiatingProcessIntegrityLevel != "System" and FolderPath has_any ("ProgramData", "Temp",
"AppData")
| project Timestamp, DeviceName, FileName, PreviousFileName, FolderPath, PreviousFolderPath,
RequestAccountName, InitiatingProcessAccountName
| join kind = inner (
DeviceImageLoadEvents
| where FileName endswith "dll" and  FolderPath has_any ("ProgramData", "Temp", "AppData")
and (InitiatingProcessIntegrityLevel == "System" or InitiatingProcessIntegrityLevel ==
"High")
| project SplitFileName=split(FileName, '.'), InitiatingProcessId, FileName, FolderPath,
InitiatingProcessIntegrityLevel
| join kind=inner (
DeviceImageLoadEvents
| where FileName endswith "dll" and  FolderPath has_any ("ProgramData", "Temp", "AppData")
and (InitiatingProcessIntegrityLevel == "System" or InitiatingProcessIntegrityLevel ==
"High") and not(InitiatingProcessFileName has_any("devenv.exe", "onedrive.exe", "zoom.exe"))
| project InitiatingProcessId, FileName, FolderPath, InitiatingProcessIntegrityLevel,
InitiatingProcessFileName
) on $left.InitiatingProcessId == $right.InitiatingProcessId
| where FolderPath != FolderPath1 and FileName1 startswith SplitFileName[0]
) on $left.FileName == $right.FileName
```

* Might want to do some tuning on InitiatingProcessFolderPath

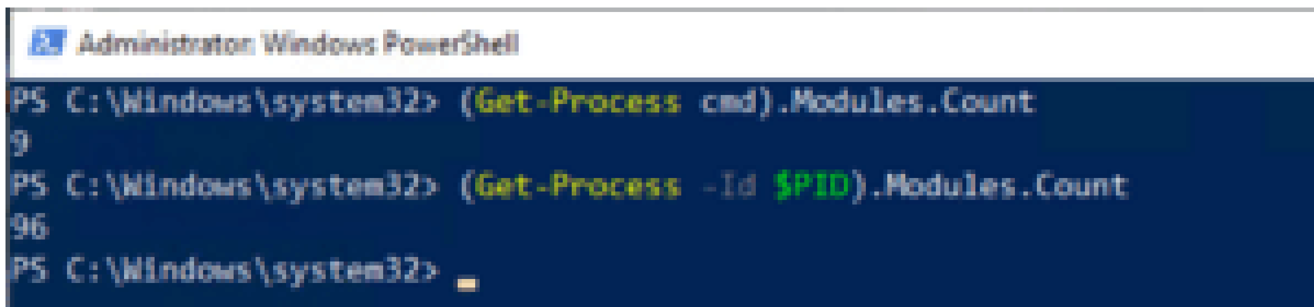
> Different directory proxying (pointing to another dll in a different directory, while loading malicious DLL)

```
//Query will perform search for DLL Proxying where the file was created by a non-system level
account but two of the same DLLs from different paths were loaded into a SYSTEM level or High
IL applicaiton.
DeviceFileEvents
| where ActionType == "FileCreated" and FileName endswith "dll" and
InitiatingProcessIntegrityLevel != "System" and FolderPath has_any ("ProgramData", "Temp",
"AppData")
| project Timestamp, DeviceName, FileName, PreviousFileName, FolderPath, PreviousFolderPath,
RequestAccountName, InitiatingProcessAccountName
| join kind = inner (
DeviceImageLoadEvents
| where FileName endswith "dll" and  FolderPath has_any ("ProgramData", "Temp", "System32",
"Program Files", "AppData") and (InitiatingProcessIntegrityLevel == "System" or
InitiatingProcessIntegrityLevel == "High")  //and InitiatingProcessIntegrityLevel != "Low"
| project InitiatingProcessId, FileName, FolderPath, InitiatingProcessIntegrityLevel,
InitiatingProcessCommandLine
| join kind=inner (
DeviceImageLoadEvents
| where FileName endswith "dll" and  FolderPath has_any ("ProgramData", "Temp", "System32",
"Program Files" "AppData") and (InitiatingProcessIntegrityLevel == "System" or
InitiatingProcessIntegrityLevel == "High")
| project InitiatingProcessId, FileName, FolderPath, InitiatingProcessIntegrityLevel,
InitiatingProcessCommandLine
) on $left.InitiatingProcessId == $right.InitiatingProcessId and $left.FileName ==
$right.FileName
| where FolderPath != FolderPath1
) on $left.FileName == $right.FileName
```

I'd like to point out I would like the above queries a lot more if MDE exposed if the PE was signed or not. I think this would help narrow down on some of this behavior a bit more. However, Sysmon is a good alternative for this.

Looking solely at module loads for this activity is going to be tough for detection, so looking at other activity like file creation and process creation events will help tremendously with identification. I also can't stress enough how important it is to look at integrity levels when it comes to the file creation, module loads, and process creation events. If you don't know about integrity levels I talk about them in my previous post: Better Know a Data Source: Process Integrity Levels. However, one of the biggest goals for an attacker when it comes to hijacking is privilege escalation not just initial access and persistence. Identifying code execution flow changes from a lower to higher integrity level will help identify malicious behavior.

One big pain point I see often with detection is that not many people collect module loads because of how loud they are. As an example, here are 2 examples where I spawned cmd.exe and powershell.exe and got their loaded module counts:

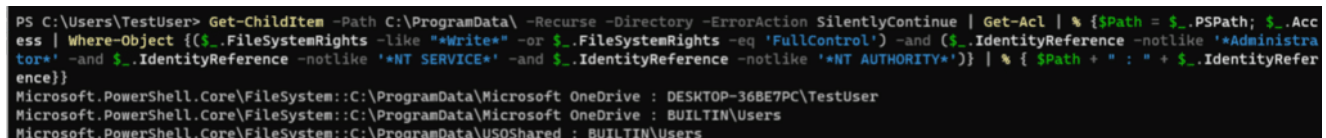


As we can see, there is a huge difference in the number of DLLs one process loads versus another. That doesn't include when that process starts executing other code either. So, it is hard to say, "this process should only load x number of DLLs". If that was something we could track, that would be awesome. So, I can understand why many don't want to collect module loads, I think there is a good middle ground where we can target on known bad or trending behaviors to get at least a percentage of visibility. We need to look at these things from a behavioral perspective and see if we can pick up on the common actions either performed during the hijack or immediately afterwards.

We also need to push back on the number of 3rd party applications that are allowed in the organization. Applications that change the folder's DACL to allow write access to lower-level users shouldn't be allowed.

If you'd like to see the access given to different folder paths, feel free to run this PowerShell one-liner. This isn't the greatest solution, things like NtObjectManager or a PowerShell script that would parse out the security descriptor and mandatory labels of the folder path would be better, but this gives a good idea of directories that give Write or FullControl access to non-Administrator or System users:

```
Get-ChildItem -Path C:\ProgramData\ -Recurse -Directory -ErrorAction SilentlyContinue | Get-Acl | % {$Path = $_.PSPath; $_.Access | Where-Object {($_.FileSystemRights -like "*Write*" -or $_.FileSystemRights -eq 'FullControl') -and ($_.IdentityReference -notlike '*Administrator*' -and $_.IdentityReference -notlike '*NT SERVICE*' -and $_.IdentityReference -notlike '*NT AUTHORITY*')} | % { $Path + " : " + $_.IdentityReference}}
```



Conclusion

DLL hijacking has always been a difficult technique to detect, but I wanted to share my methodology and thought process for detecting these attacks. Often, we focus on creating fixed rules that trigger alerts when a known threat is detected. However, in doing so, we may

miss out on potential attacks that employ unknown or evolving techniques. Therefore, we need to strike a balance between being mindful of the imperfect nature of detection and not forsaking all efforts because of it.

Although there are various hijacking techniques available, we focused on three techniques based on DLL hijacking: search order hijacking, sideloading, and proxying. There are plenty of other hijacking techniques like COM hijacking, but we'll cover those in future blogs.

Resources