DLL Search Order Hijacking

Context's Intelligence and Response teams have seen DLL Search Order being abused as a means of conducting network intrusions in real environments. Abusing the DLL Search Order and taking advantage of this mechanism in order for an application to load a rogue DLL instead of the legitimate one is known as DLL preloading, or (in the MITRE ATT&CK framework) hijacking.

In this blog post, you will find out more about the fundamentals of DLL Search Order and how legitimate binaries can be weaponized, and introduce a tool to automate the discovery of binaries suitable for payload execution via DLL hijacking.

By Lampros Noutsos & Oliver Fay

01 Jul 2020

Security, Tools

About Dynamic Link Libraries

A Dynamic Link Library (DLL) is a module that contains functions and data that can be used by another module (application or DLL). These functions are exported from library files in order to be available for use by the applications or DLLs that rely on them. In order to use these functions, the applications have to import them from the library files. There are two ways an application imports functions from modules: implicitly (load-time dynamic linking) and explicitly (run-time dynamic linking). Let's have a look at each.

Implicit Linking (Load-Time Dynamic Library Linking)

When an application is opened, the Windows loader takes steps in order to map the application's executable image in memory and eventually start up a process that hosts and executes its code. During the loading process, the loader parses the import table of the executable image in order to map the imported module(s) (Dynamic Link Library) in the address space of this process.

An executable image may have embedded a manifest that describes dependencies on Windows side-by-side assemblies. Before loading the imported DLL(s), the Side-by-Side Manager (SxS Manager) checks if any of the dependencies that exist in the manifest file of this executable are met. If yes, the required modules are loaded from the path C:\Windows\WinSxS\.

For the rest of the imported modules, the loader first checks if each one of the DLLs that are to be imported exists on the list set by the KnownDLLs registry key (HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs). If yes, the loader uses the copy of the DLL pointed to by this registry key. Otherwise, it searches for the module by applying the DLL search order. More on the DLL search order to follow on in this article.

Explicit Linking (Run-Time Dynamic Library Linking)

Applications may need to make use of functions within DLLs that are loaded dynamically while the application is running. Loading DLLs in this way is called Run-Time DLL Dynamic Linking. This type of loading occurs when the application calls the functions LoadLibrary or LoadLibraryEx.

These two functions accept the name of the module to be loaded. The name can be either the filename of the module or a full path to the module. In case the application does not specify a full path to the module, the Windows loader is looking for the DLL applying the Dynamic Library Search Order.

DLL Search Order hijacking and how to identify it

Abusing the DLL Search Order and taking advantage of this mechanism in order for an application to load a rogue DLL instead of the legitimate one is known as DLL preloading. It is called preloading because the attackers can place their DLL earlier in the search order and thus the application loads this instead of the legitimate one. This technique is documented as DLL Search Order Hijacking in the MITRE ATT&CK framework (T1038). In the following sections of this article we demonstrate how legitimate applications can be abused to load and execute a Cobalt Strike beacon payload via search order hijacking.

The DLL Search Order hijacking mainly offers two advantages that make it an effective technique. The first advantage is that it can be used to evade detections. The weaponized application is often a legitimate signed binary that loads the malicious DLL in its address space by calling a function that this DLL exports. Thus, in order to execute the malicious DLL, an automated sandbox has to first identify which exported function to call or execute the innocuous binary and set up the environment so it loads the malicious DLL. This requirement makes DLL search order hijacking a suitable vector for defence evasion.

The other advantage this technique offers is the potential to escalate privileges. When the legitimate application which loads a DLL in its address space runs with elevated privileges, any code that executes in its context is executed with the same level of privileges. This means that the payload DLL, and as a result the functionality it implements, gets executed with the same privileges as the process that loads it.

There are two different approaches in identifying a candidate DLL for hijacking: static analysis and dynamic analysis, just like the approach taken in malware analysis. The dynamic analysis consists of executing the application and monitoring the libraries it loads. This can be achieved with tools such as Procmon and API Monitor, which monitor API calls. The static analysis approach consists of using disassembler tools such as IDA and Ghidra to identify - without executing the application - if calls to specific APIs occur.

The main difference between the two approaches is that with employing dynamic analysis, a larger number of candidate DLLs for both load-time and run-time dynamic linking is revealed, whereas the static analysis only reveals candidate DLLs that are loaded in run-time.

DLL Search Order

Since Windows XP, when the SafeDllSearchMode option is enabled a module is loaded in the address space of an application - either during load-time or run-time - and the full path to the module is not explicitly specified or the manifest file does not state where the dependency can be looked up, the operating system searches for the DLL in a defined search order, that consists of the following:

1. The directory from which the application is loaded
2. The system directory
3. The 16-bit system directory
4. The Windows directory
5. The current directory (same as 1 unless otherwise specified)
6. The directories that are listed in the PATH environment variable

The above search order is slightly different when the SafeDllSearchMode is not enabled. For additional information on this, please consult the Microsoft documentation.

When the operating system starts the search for a DLL that the application is importing, unless a full path is specified the directory from which the application was loaded is searched first. If the DLL does not exist there, the system directory is searched next and so on. The abuse occurs when the attackers, having observed the behaviour of an application, place their own DLL earlier in the search order so the application loads that DLL instead of continuing the search. The threat group tracked by Context's intelligence team as AVIVORE have used this technique in order to achieve payload execution within victim environments.

A case of Load-Time Dynamic Linking

To illustrate the Load-Time loading, we analysed a signed Google executable (GoogleCrashHandler - MD5: 83bb030c71c9727dcfb2737005772c4e) that Context's Intelligence team observed being used in intrusions.

The reason we focus on this binary, is that it makes a suitable candidate. And what do we mean by this? First of all, it is a legitimate and signed binary from a trusted vendor. Once this application is executed it presents no visible window and it terminates upon execution, so the user upon executing this binary is not made aware that the binary was indeed run and the process does not stay in a running state. Last by not least, even though the application loads a rogue DLL, it does not generate any other activity - like a pop up window - that would potentially raise user's suspicion. So, it can be used as a leverage to execute a payload DLL.

In order to identify the modules this executable attempts to load and could potentially be hijacked, we use SysInternal's Procmon with the following filters:

After applying the filters, we get the following candidate DLLs:

Weaponize the DLL search order

In order to weaponize the GoogleCrashHandler.exe, we created a custom DLL that executes the beacon shellcode generated from CobaltStrike. We named the output DLL wkscli.dll, placed it in the same directory the GoogleCrashHandler executable image exists. We then opened GoogleCrashHandler. The next picture shows the rogue DLL was mapped in the address space of GoogleCrashHandler and a beacon was launched.

A case of Run-Time Dynamic Linking

To illustrate the Run-Time loading, we analysed a signed Microsoft executable (OleView - MD5: d1e6767900c85535f300e08d76aac9ab). The signed executable was used to load a DLL that subsequently decrypted a PlugX payload. For more information, please see the 'Indicators of Compromise' section of this article.

Let's have a look with Procmon at the DLLs this executable attempts to load once it is launched:

To confirm that this DLL is loaded dynamically and not during load-time, we use API Monitor. By hooking the LoadLibrary and LoadLibraryEx APIs, we observe that the application attempts to dynamically load multiple DLLs:

Correlating the data we get from Procmon and API monitor, we determine the DLL ACLUI.DLL is searched by applying the DLL search order. We can do the same by analysing the executable on Ghidra. In the Defined Strings section (Window -> Defined Strings) we search for the string ACLUI. Three results show up:

We then follow the reference to the string ACLUI.DLL and we end up where the call to LoadLibraryW is made:

We observe that the argument to the LoadLibraryW API is not a full path and thus we confirm that the DLL search order is applied.

If we create a payload DLL, place it in the same directory as the executable and then run the executable, the following message box is shown:

This means the executable requests a function from our payload DLL that does not exist. In order to allow our payload to execute from the application, we have to export the function EditSecurity. A way to do this in C/C++ code is shown in the following screenshot:

Weaponize the DLL search order

Likewise as we did for the GoogleCrashHandler image, we created a custom DLL that executes the beacon shellcode generated from CobaltStrike. We named the output DLL ACLUI.DLL and placed it in the same directory as OleView.exe executable image. We then opened OleView. The next picture shows the rogue DLL was mapped in the address space of OleView and a beacon was launched.

Mitigation Strategies

Based on the incidents we have observed and how DLL Search Order has been abused in the past, we can recommend specific best practices to be applied in the environments. The recommendations aim to reduce attack surface, limit consequences of a potential attack and reduce the amount of time it takes related attacks to be detected.

Companies are generally recommended to deploy capabilities and policies within their environments. As part of the capabilities, it is highly recommended that Event Detection and Response (EDR) software is deployed. On the policies side, it is a good practice to populate and maintain a list of all the applications that are authorized to run within the environment and implement application white-listing to prevent applications that do not belong in this list from running. In order to detect rogue DLLs, it is recommended to

conduct frequency analysis of the observed DLLs within the environment. Low frequency occurrence of a DLL would probably be an indicator worth further analysis. One more thing that can protect against other DLL hijacking attack scenarios is to enable the SafeDllSearchMode and configure the CWDIllegalInDLLSearch registry on systems.

Best practices have to be followed by software developers, as well. For example, when an application is loading modules in run-time, it is recommended to specify an absolute path for modules provided to LoadLibrary and LoadLibraryEx APIs or use the API SetDllDirectory to specifically set the path from where the Windows loader will load the provided module. By doing this, the normal search order is not being looked up. One more recommendation in accordance to best practises would be to implement integrity checks for the loaded modules via Application Manifests. By doing this the application will prevent the load of rogue DLLs and potentially inform the users of identified anomalies.

Introducing DLLHSC

DLLHSC is an application designed to automate the scan of a provided executable image, generate leads - that can later be manually assessed - and report potential paths of taking advantage of the DLL search order with the ultimate goal to load a payload DLL in the address space of the provided image via search order hijacking.

Modes of operation

The tool implements 3 modes of operation which are described below.

Lightweight Mode

Loads the executable image in memory, parses the Import table and then replaces any DLL referred in the Import table with a payload DLL. The tool places in the application directory a module (DLL) that does not already exist in the application directory, does not belong to WinSxS and does not belong to the KnownDLLs.

Then, it launches the application and reports if the payload DLL was executed. The payload DLL upon execution creates a temporary file in the path C:\Users\%USERNAME%\AppData\Local\Temp\DLLHSC.tmp. If the temporary file exists, this indicates the scanned application can be abused. As some executables import functions from the DLLs they load, error message boxes may be shown up when the provided DLL fails to export these functions and thus meet the dependencies of the provided image.

However, the message boxes indicate the DLL may be a good candidate for payload execution if the dependencies are met. In this case, additional analysis is required. The title of these message boxes may contain the strings: ‘Ordinal Not Found’ or ‘Entry Point Not Found’. DLLHSC looks for windows that contains these strings, closes them as soon as they shown up and reports the results.

List Modules Mode

Creates a process with the provided executable image, enumerates the modules that are loaded in the address space of this process and reports the results after applying filters.

The tool only reports the modules loaded from the System directory and do not belong to the KnownDLLs. The results are leads that require additional analysis. The analyst can then place the reported modules in the application directory and check if the application loads the provided module instead.

Run-Time Mode

Hooks the LoadLibrary and LoadLibraryEx APIs via Microsoft Detours and reports the modules that are loaded in run-time.

Each time the scanned application calls LoadLibrary and LoadLibraryEx, the tool intercepts the call and writes the requested module in the file C:\Users\%USERNAME%\AppData\Local\Temp\DLLHSCRTLOG.tmp. If the LoadLibraryEx is specifically called with the flag LOAD_LIBRARY_SEARCH_SYSTEM32, no output is written to the file. After all interceptions have finished, the tool reads the file and prints the results. Of interest for further analysis are modules that do not exist in the KnownDLLs registry key, modules that do not exist in the System directory and modules with no full path (for these modules loader applies the normal search order).

You can find the source code of DLLHSC as well as compiled binaries for x86 and x64 architecture on our GitHub page.

References:

Indicators of Compromise

Indicator	Type	Filepath	Comment

b38703090659bc5c297f74e0da3d5d75 f8d089ed3ab422f7bf5776bc180001d1e7e83d1c dba03177d6612c9117970de0696e167fbe1c42dc3e9b68663969e88ad59a0082	MD5 SHA-1 SHA-256	-	Dropper
d1e6767900c85535f300e08d76aac9ab 4a0f328e7672ee7ba83f265d48a6077a0c9068d4 91f6547bceddfb2f241570ac82c00de700e311e4a38dea60d8619638f1ed3520	MD5 SHA-1 SHA-256	C:\ProgramData\Microsoft Help\OleView.exe	Legitimate Executable
4b62274eb48440e2d080e111124d9d04 f98e12eedfe451220a5765a0e808f48c66d9433d c82556bfc26fa5b38839c361aa11651177c2c35cf9e985debf0faec6aa789b87	MD5 SHA-1 SHA-256	C:\ProgramData\Microsoft Help\ACLUI.DLL	Loader DLL
73d13e8c38778d0cc62ff29f89c1c337 2170fb614ad2e04081adc9fbcfa8ff6f9f000787 016b269293abcd607f476aebe18524df7d7044f3ef99f3b3e94344243c0e955c	MD5 SHA-1 SHA-256	C:\ProgramData\Microsoft Help\ACLUI.DLL.UI	Encrypted PlugX Payload
yy[.]h365[.]net:443	Domain	-	PlugX command and control (C2) domain
yy[.]u1e[.]net:443	Domain	-	PlugX command and control (C2) domain
yy[.]wwdlq[.]com:443	Domain	-	PlugX command and control (C2) domain

About Lampros Noutsos & Oliver Fay

Lampros Noutsos is a Consultant in our Assurance team and Oliver Fay is Technical Lead in our Threat Intelligence & Incident Response team.