

ADAPTIVE DLL HIJACKING



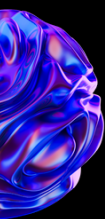
```
A problem has been detected and Windows has been shut down to prevent damage to your computer.
The problem seems to be caused by the following file(s): example.sys
FILE PATH IS CORRUPTED AREA.

If this is the first time you've seen this error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, uninstall any software or hardware
that you haven't tested yet.

If problems continue, disable or remove any newly installed hardware.
If you're unable to do so, you may need to use a system restore point.
If you're unable to do so, contact your hardware or software manufacturer
for help.

Technical information:
*** STOP: 0x0000000A (0x0000000A, 0x00000000, 0x00000000, 0x00000000)
*** example.sys: ADDRESS 0x00000000 base of 0x00000000, operation aborted
```



INTRODUCTION

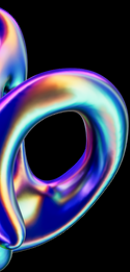
Dynamic-Link Library (DLL) hijacking has been a focal technique for adversarial operations over the years, evolving from a simple exploitation method into a sophisticated and adaptive strategy for compromising software systems. The essence of DLL hijacking lies in the ability to insert malicious code into legitimate processes by manipulating the DLL loading mechanism within Windows operating systems. This technique leverages the inherent trust that software places in DLLs to execute unauthorized code, often bypassing traditional security measures. As defenders have become more vigilant, attackers have responded with increasingly complex implementations, making DLL hijacking a continually evolving battlefield.

Our journey into the depths of DLL hijacking has revealed numerous subtleties that impact its practical application. While the basic concept remains straightforward, achieving reliable and stealthy execution in real-world environments demands a deep understanding of various underlying mechanisms. This includes the intricacies of export table cloning, dynamic Import Address Table (IAT) patching, stack walking, and runtime table reconstruction. Each of these methods addresses specific challenges encountered during hijacking attempts, providing avenues to maintain process stability and evade detection. This introduction aims to shed light on these advanced techniques, distilled from years of operational experience and shared through our Dark Side Ops courses.

For those who have dabbled in DLL hijacking and encountered roadblocks, this discussion will serve as a detailed guide to overcoming common pitfalls. The initial thrill of executing a basic DLL hijack often gives way to frustration when the technique fails to scale to more complex scenarios. This post addresses these frustrations by delving into the reasons why simple hijacks fail and how adaptive methods can be employed to achieve more reliable and covert operations. By dissecting the mechanics of both static and dynamic sinks of DLL execution, we offer insights that can bridge the gap between theoretical understanding and practical success.

The concept of "execution sinks" is crucial for understanding how DLLs are loaded and initialized within a process. Static sinks involve the inclusion of a DLL in a program's dependency graph, resulting in initialization during process startup. In contrast, dynamic sinks occur when a DLL is loaded on demand during runtime through functions like `LoadLibrary`. Each scenario presents unique challenges for hijackers, particularly concerning the handling of export tables and maintaining the stability of the host process. Our exploration will clarify these distinctions and provide strategies for navigating them effectively.

Function proxying emerges as a vital technique for ensuring the stability and continuity of hijacked processes. By redirecting function calls from the hijacked DLL to the legitimate one, attackers can maintain the expected behavior of the target application while executing their malicious payload. This section will cover various methods for implementing function proxying, from simple export forwarding to more dynamic approaches like runtime linking and stack patching. Understanding these techniques is essential for anyone looking to master DLL hijacking in complex environments.



DOCUMENT INFO



To be the vanguard of cybersecurity, HadesS envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish HadesS as a symbol of trust, resilience, and retribution in the fight against cyber threats.

At HadesS, our mission is twofold: to unleash the power of white hat hacking in punishing black hat hackers and to fortify the digital defenses of our clients. We are committed to employing our elite team of expert cybersecurity professionals to identify, neutralize, and bring to justice those who seek to exploit vulnerabilities. Simultaneously, we provide comprehensive solutions and services to protect our client's digital assets, ensuring their resilience against cyber attacks. With an unwavering focus on integrity, innovation, and client satisfaction, we strive to be the guardian of trust and security in the digital realm.

Security Researcher

Amir Gholizadeh (@arimaqz), Surya Dev Singh (@kryolite_secure)

TABLE OF CONTENT

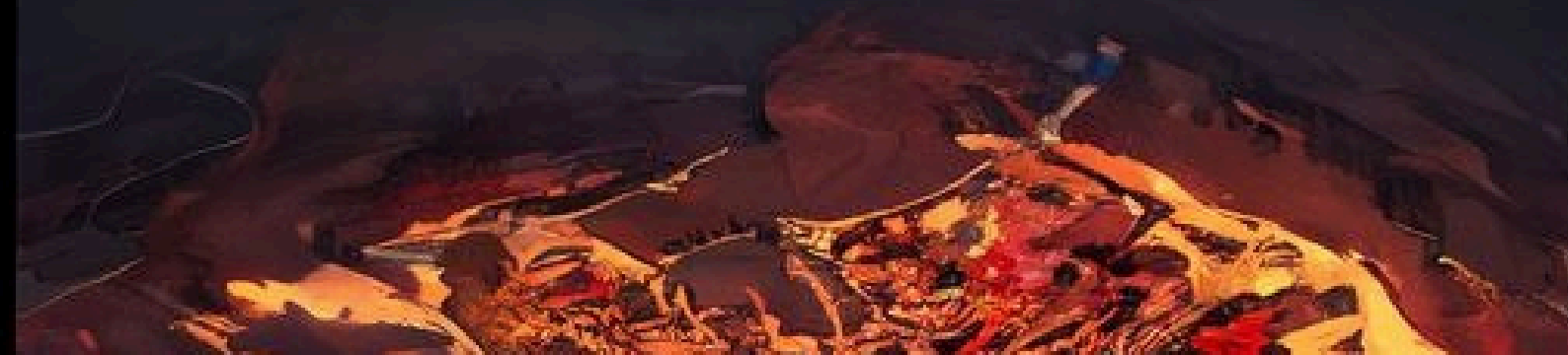
- **Introduction to DLL Hijacking**
 - Brief overview of what DLL hijacking is.
 - Importance of understanding this attack vector.
- **Basic Concepts and Terminology**
 - Explanation of module search order.
 - KnownDLLs and “safe search.”
- **Advanced Techniques**
 - Export table cloning: Cloning export functions.
 - Dynamic IAT patching: Modifying the Import Address Table.
 - Stack walking: Manipulating the call stack.
 - Run-time table reconstruction: Rebuilding tables during execution.
- **Tooling and Exploitation Frameworks**
 - Mentioning tools like Siofra78, DLLSpy9, and Robber10.

EXECUTIVE SUMMARY

DLL hijacking is a critical attack vector where malicious code is injected into legitimate processes by exploiting the DLL loading mechanism in Windows systems. Understanding this technique is essential for robust cybersecurity defense. Key concepts include the module search order and KnownDLLs, which influence how DLLs are located and loaded. Advanced methods such as export table cloning, dynamic IAT patching, stack walking, and runtime table reconstruction enhance the effectiveness of DLL hijacks by ensuring stable and covert execution. Tools like Siofra, DLLSpy, and Robber play a significant role in facilitating these advanced techniques, underscoring the need for comprehensive knowledge and preparation to counteract such sophisticated attacks.

Key Findings

Lastly, we address the critical issue of the loader lock, a synchronization mechanism within the Windows loader that can cause deadlocks or crashes if mishandled during DLL initialization. By discussing the implications of loader lock and providing practical solutions like starting new threads or employing function hooking, we aim to equip practitioners with the knowledge to avoid common pitfalls. Our goal is to ensure that hijacked processes remain stable and functional, thereby enhancing the effectiveness and stealth of the hijack. The culmination of these insights and techniques is encapsulated in our project, Koppeling, which automates advanced DLL hijacking preparations and promises to be a valuable tool for the community.



ABSTRACT

Adaptive DLL hijacking is an advanced attack technique that leverages the dynamic loading mechanism of Windows DLLs to inject malicious code into legitimate processes. This method exploits weaknesses in the module search order and KnownDLLs to achieve unauthorized code execution. Understanding DLL hijacking is crucial for cybersecurity professionals, as it bypasses traditional security measures and can lead to significant system compromise.

The complexity of adaptive DLL hijacking lies in its advanced techniques, including export table cloning, dynamic Import Address Table [IAT] patching, stack walking, and runtime table reconstruction. These methods ensure the malicious DLL maintains functionality and stability within the target process, avoiding crashes and detection. These techniques require a deep understanding of the Windows loader and the ability to manipulate it dynamically, highlighting the sophistication of the attack.

To combat this threat, various tools and frameworks like Siofra, DLLSpy, and Robber have been developed to discover and exploit DLL hijacking vulnerabilities. These tools automate the process of identifying vulnerable applications and creating malicious DLLs. For defenders, knowledge of these tools and techniques is essential to identify and mitigate potential hijacking attempts, ensuring robust protection against this adaptive and evolving threat.



01

ATTACKS



What is DLL Hijacking

DLL hijacking is a technique where an attacker exploits the way applications load Dynamic Link Libraries (DLLs) in Windows. When an application is launched, it searches for necessary DLLs in specific directories. If an attacker places a malicious DLL with the same name as a legitimate one in a directory that's searched first, the application may load the malicious DLL instead of the legitimate one, allowing the attacker to execute arbitrary code. 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. There are several Techniques that can be used to Hijack the DLL

Known DLLs and Safe Search

Known DLLs

Windows has a list of "Known DLL" names maintained in the registry. these are a set of system DLLs that Windows applications load by default. The Known DLLs list is maintained in the Windows Registry under the `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs` key. This list includes essential DLLs that the system can trust and ensures they are loaded from a secure, predefined location (typically the system directory).

How does it Work ?

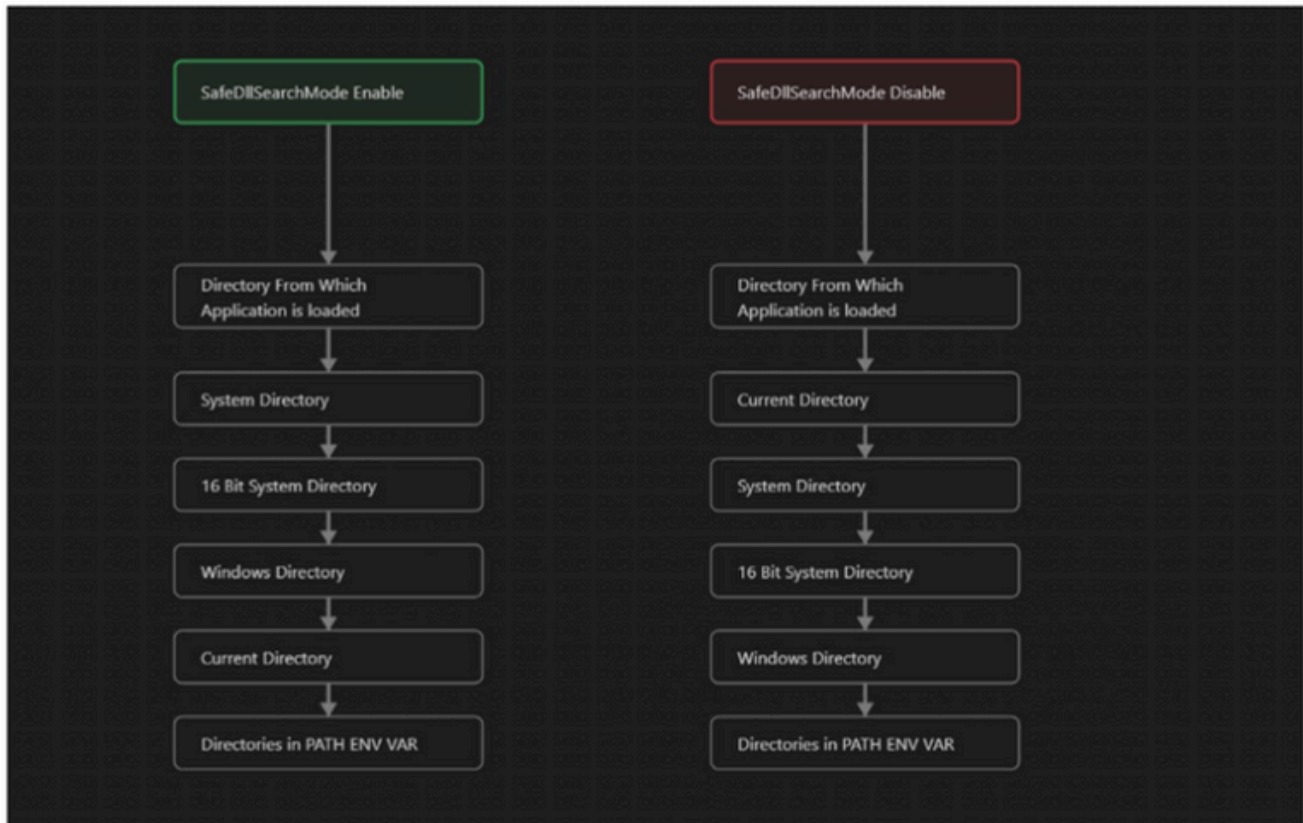
When an application attempts to load a DLL, Windows checks if the DLL name is in the Known DLLs list. If it is, Windows bypasses the usual search order and loads the DLL from the system directory, thus avoiding the risk of DLL hijacking.

Safe Search

Safe DLL search is a mechanism in Windows to improve the security of DLL loading processes. By default, Windows uses a specific search order to locate and load DLLs. Safe DLL search mode changes the search order to prioritize secure locations over potentially vulnerable directories.

How Does it Work ?

These are the search Order that is being followed if `SafeDllSearchMode` is Enable / Disable:



Safe DLL search mode is enabled by default in modern versions of Windows (e.g., Windows 10, Windows Server 2016 and later). but can also be Enabled if its not it can be enable by creating on DWORD value name `SafeDllSearchMode` at `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager` and set it to 1 to enable it .

Export table cloning : Cloning Export Function

What is Export Table

Every DLL has an export table which lists all the functions and data that other modules (executables or other DLLs) can call. Each function in the export table has a name and an address pointing to its location in memory.

How Export Table Cloning Process Work

1. Identifying Target DLL

The attacker identifies a DLL that is loaded by a target application and contains functions they wish to hijack.

2. Creating the Malicious DLL

They create a malicious DLL with the same filename as the legitimate one.

3. Extracting Export Table

Using tools (like [NetClone](#) from [Koppeling](#) project) or manual analysis, the attacker extracts the export table from the legitimate DLL. This includes copying the EAT, ENPT, and ordinal table entries to malicious dll

4. Modifying Function Addresses

The attacker replaces the addresses in the EAT with addresses pointing to their malicious code. This step is crucial as it redirects legitimate function calls to malicious code.

Attacker can implement proxying also to maintain the original functionality of the application while also running the malicious code.

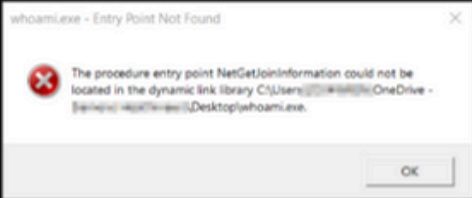
Implementation

Prepare a hijack scenario with an incorrect DLL, which will give intentional error :

```
C:\Users\johndoe\OneDrive - Siemens Healthineers\Desktop>copy C:\windows\system32\whoami.exe .\whoami.exe
1 file(s) copied.

C:\Users\johndoe\OneDrive - Siemens Healthineers\Desktop>copy C:\windows\system32\kernel32.dll .\wkscli.dll
1 file(s) copied.

C:\Users\johndoe\OneDrive - Siemens Healthineers\Desktop>whoami.exe
```



Now use the NetClone.exe to clone the wkscli.dll functions to kernel32.dll and output them as wkscli.dll . which will act as as proxy to actual wkscli.dll.

```
C:\Users\johndoe\OneDrive - Siemens Healthineers\Desktop>Koppeling\Bin\NetClone.exe --target C:\windows\system32\kernel32.dll --reference C:\windows\system32\wkscli.dll --output wkscli.dll
[+] Done.

C:\Users\johndoe\OneDrive - Siemens Healthineers\Desktop>whoami.exe
ad0051r004aren

C:\Users\johndoe\OneDrive - Siemens Healthineers\Desktop>
```

Dynamic IAT patching: Modifying the Import Address Table

Dynamic Import Address Table (IAT) patching is a technique used in DLL hijacking to intercept and modify function calls made by an application to a DLL. The goal is to redirect these calls to malicious functions while maintaining the application's original functionality. But unlike Export Function Cloning, this technique works after the DLL is loaded in memory.

How Dynamic IAT Patching works ?

1. Application Startup

- The vulnerable application starts and begins loading its required DLLs.

2. DLL Search Order and Loading

- The application finds and loads `malicious_example.dll` instead of the legitimate `example.dll` due to search order and naming conventions.

3. Load Original DLL within Malicious DLL

- Inside `malicious_example.dll`, use `LoadLibrary` to load the original `example.dll`.
- Store the handle and use `GetProcAddress` to retrieve the addresses of the original functions.

4. Locate and Patch IAT

- Parse the PE headers of the application to locate the IAT.
- Overwrite the entries in the IAT that point to the original `example.dll` functions with addresses pointing to functions in `malicious_example.dll`.

```
Example: IAT Entry for FunctionA originally -> Address: 0x12345678  
(example.dll!FunctionA)
```

```
Patched IAT Entry for FunctionA -> Address: 0xabcdef12  
(malicious_example.dll!FunctionA)
```



5. Function Call Redirection

- When the application calls `FunctionA`, it now jumps to `malicious_example.dll!FunctionA`.

6. Malicious Code Execution and Proxying

- Inside `malicious_example.dll!FunctionA`:
 - Execute any malicious actions (e.g., logging data, modifying parameters).
 - Call the original `FunctionA` in `example.dll` using the stored address obtained from `GetProcAddress`.

7. **Return Results to Application**

- The results from the original `FunctionA` are returned to the application, maintaining expected behavior and functionality.

Advantages of Dynamic IAT Patching

- **Stealth:** By modifying the IAT at runtime, the technique is less likely to be detected by static analysis tools.
- **Persistence:** The malicious DLL remains active, continually intercepting and redirecting function calls.
- **Functionality:** Ensures the application continues to operate normally, reducing the chance of detection by users or security mechanisms.

Example Code Snippet (Simplified)

Here is a simplified conceptual example in C++ to illustrate the process:

```
// Inside malicious_example.dll C

// Load the original DLL
HMODULE hOriginal = LoadLibrary("example.dll");

// Get the address of the original FunctionA
typedef void (*OriginalFunctionA)();
OriginalFunctionA pOriginalFunctionA =
(OriginalFunctionA)GetProcAddress(hOriginal, "FunctionA");

// Malicious implementation of FunctionA
extern "C" __declspec(dllexport) void FunctionA() {
    // Execute malicious actions
    // ...

    // Call the original FunctionA
    pOriginalFunctionA();
}

// Code to patch the IAT (simplified for illustration purposes)
void PatchIAT(HMODULE hModule, const char* originalDLLName, const char*
functionName, void* newFunction) {
    // Locate and parse the IAT of hModule
    // Overwrite the IAT entry for functionName to point to newFunction
    // ...
}
```

Common Tool for Used in DLL

Siofra

Siofra is a tool designed to identify and exploit DLL hijacking vulnerabilities :

It is able to simulate the Windows loader in order to give visibility into all of the dependencies (and corresponding vulnerabilities) of a PE on disk, or alternatively an image file in memory corresponding to an active process.

More significantly, the tool has the ability to easily generate DLLs to exploit these types of vulnerabilities via PE infection with dynamic shellcode creation.

These infected DLLs retain the code (DllMain, exported functions) as well as the resources of a DLL to seamlessly preserve the functionality of the application loading them, while at the same time allowing the researcher to specify an executable payload to be either run as a separate process or loaded into the target as a module.

The tool contains automated methods of combining UAC auto-elevation criteria with the aforementioned functionality in order to scan for UAC bypass vulnerabilities.

Here is one example for Windows Defender Binary on windows 10 x64 Home/Pro :


```
PowerShell
Siofra64.exe --mode file-scan -f "c:\Program Files\Windows
Defender\MpCmdRun.exe"
--enum-dependency --dll-hijack

===== c:\Program Files\Windows Defender\MpCmdRun.exe [64-bit PE] =====
MpCmdRun.exe
  msvcrt.dll [KnownDLL]
  KERNEL32.dll [KnownDLL]
  OLEAUT32.dll [KnownDLL]
    msvcp_win.dll [Base]
      api-ms-win-crt-string-l1-1-0.dll [API set]
        ucrtbase.dll [Base]
      combase.dll [KnownDLL]
        RPCRT4.dll [KnownDLL]
        bcryptPrimitives.dll [Base]
  ADVAPI32.dll [KnownDLL]
    api-ms-win-eventing-controller-l1-1-0.dll [API set]
      sechost.dll [KnownDLL]
  OLE32.dll [KnownDLL]
    GDI32.dll [KnownDLL]
      api-ms-win-gdi-internal-uap-l1-1-0.dll [API set]
        gdi32full.dll [Base]
          USER32.dll [KnownDLL]
            win32u.dll [Base]
  SspiCli.dll [!]
  mpclient.dll [!]
    CRYPT32.dll [Base]
      MSASN1.dll [Base]
  WINTRUST.dll [Base]

[!] Module SspiCli.dll vulnerable at c:\Program Files\Windows
Defender\SspiCli.dll
(real path: C:\WINDOWS\system32\SspiCli.dll)
```

You can find more details at : <https://github.com/Cybereason/siofra>

Robbers

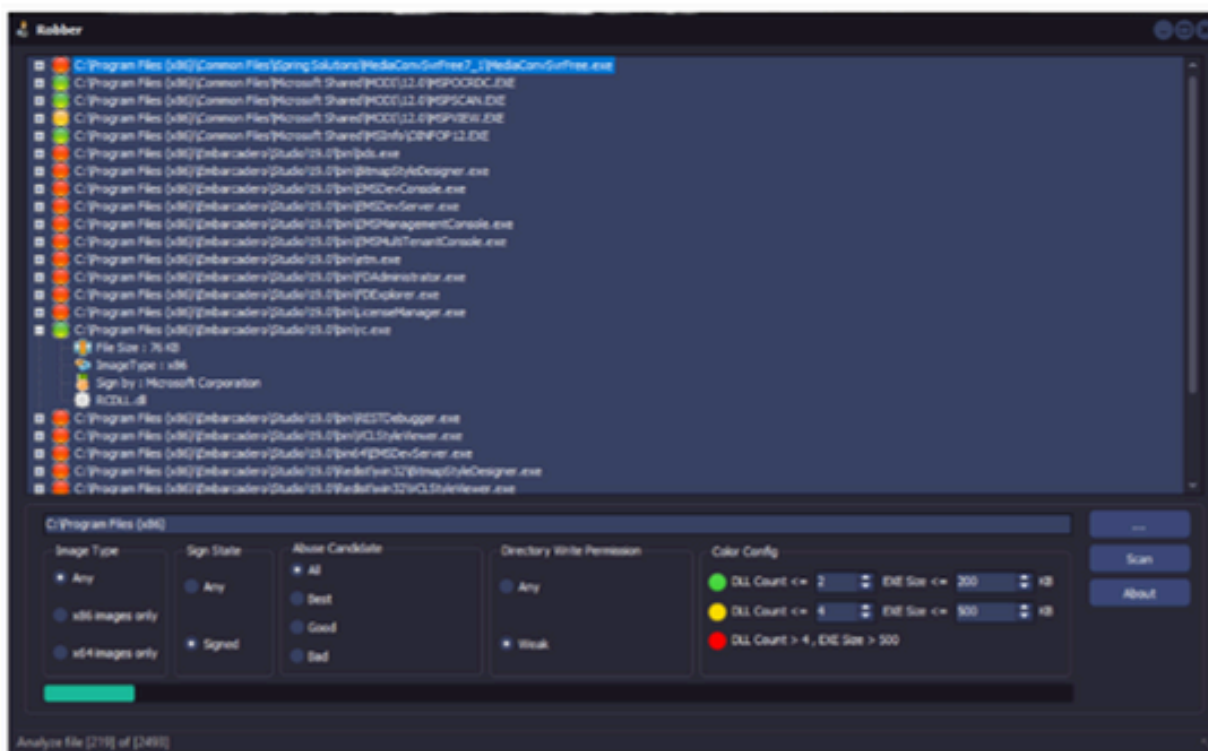
Robber is open source tool for finding executables prone to DLL hijacking.

Robber use simple mechanism to figure out DLLs that prone to hijacking :

1. Scan import table of executable and find out DLLs that linked to executable
2. Search for DLL files placed inside executable that match with linked DLL (current working directory of the executable has highest priority)
3. If any DLL found, scan the export table of theme
4. Compare import table of executable with export table of DLL and if any matching was found, the executable and matched common functions flag as DLL hijack candidate.

Here are its features :

- Ability to select scan type (signed/unsigned applications)
- Determine executable signer
- Determine wich referenced DLLs candidate for hijacking
- Determine exported method names of candidate DLLs
- Configure rules to determine which hijacks is best or good choice for use and show theme in different colors
- Ability to check write permission of executable directory that is a good candidate for hijacking



Importance of understanding this attack vector

This attack method allows malicious actors to introduce harmful code into legitimate processes, granting them persistent and often elevated access to compromised systems. Defense against this attack vector requires a deep understanding of how it actually works. Furthermore this knowledge helps develop mitigation strategies against DLL hijacking.

Explanation of module search order

In order to find the appropriate DLL when importing it in the program, the system searches for it in the following order:

1. HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs
2. Application Directory
3. C:\Windows\System32
4. C:\Windows\System
5. C:\Windows
6. Current Directory
7. PATH variables directory

Stack walking: Manipulating the call stack

Stack walking is the process of traversing the call stack to examine the sequence of function calls.

Before talking about it in detail, let's recap the call stack:

A stack data structure uses the LIFO mechanism and stores information about functions. When a function is executed, the local variables, parameters, and return addresses are pushed onto the stack creating a stack frame. When a function returns, its stack frame is popped off as well.

Manipulating the Call Stack

Manipulating the call stack can lead to various outcomes, including:

- **Return-Oriented Programming (ROP):**
 - Reusing existing code snippets (gadgets) to execute arbitrary code.
 - Involves carefully crafting a sequence of return addresses to control program flow.
 - Often used in exploitation scenarios.
- **Stack Buffer Overflows:**
 - Overwriting data on the stack to modify return addresses or corrupt other data.
 - Can lead to arbitrary code execution.
- **Exception Handling Hijacking:**
 - Interfering with the normal exception handling process.
 - Can be used to bypass security checks or execute malicious code.
- **Debugger and Profiler Functionality:**
 - These tools manipulate the call stack to inspect program state and performance.

Run-time table reconstruction: Rebuilding tables during execution

To reflectively load DLLs in memory, the program must reconstruct and rebuild the necessary tables like IAT and relocation.

The IAT is a crucial component of a DLL as it maps function calls within the DLL to their corresponding addresses in external libraries. When a DLL is loaded reflectively, it lacks a predefined base address. Therefore, the DLL must dynamically resolve these addresses. This involves:

- **Locating the IAT:** The DLL's PE header contains information about the IAT's location within the DLL's data section.
- **Parsing Import Descriptors:** The IAT consists of import descriptors, which contain information about imported libraries and function names. These descriptors are parsed to identify the required libraries and their exported functions.
- **Loading Dependent Libraries:** The reflective DLL must explicitly load the necessary libraries using functions like `LoadLibrary`.
- **Resolving Import Addresses:** For each imported function, the DLL must obtain the function's address from the loaded library using functions like `GetProcAddress`. The resolved addresses are then written back to the IAT.

Relocation Handling: Relocations are adjustments made to memory addresses within a DLL to accommodate different loading addresses. In a reflectively loaded DLL, the loading address is unknown beforehand, necessitating relocation processing. This involves:

- **Locating the Relocation Table:** The PE header points to the relocation table, which contains information about memory locations that require adjustments.
- **Applying Relocations:** The DLL iterates through the relocation table, calculating the address offset based on the DLL's actual loading address. The necessary adjustments are made to the target memory locations.

By successfully rebuilding the IAT and handling relocations, the reflective DLL becomes self-sufficient, capable of functioning independently within the target process's memory space. This level of control makes it a potent tool for both legitimate and malicious purposes. However, it also significantly increases the complexity of the DLL's implementation and makes it a challenging target for analysis and detection.

Adaptive DLL Hijacking Techniques

DLL hijacking has been a cornerstone in the arsenal of many penetration testers and malicious actors for years. Its effectiveness lies in its ability to manipulate how applications load DLLs, often leading to code execution in privileged contexts. This post dives deep into advanced DLL hijacking techniques, addressing common pitfalls and providing solutions for stability and execution control. If you've struggled with DLL hijacking in the real world, this guide is for you.

Refresher

This guide assumes familiarity with the basics of DLL hijacking, such as module search order, KnownDLLs, and "safe search." If you need a refresher, check out these resources:

- [DLL Hijacking Attacks Revisited](#)
- [DLL Hijacking](#)
- [Understanding How DLL Hijacking Works](#)
- [Lateral Movement: SCM and DLL Hijacking Primer](#)

Tools for discovering and exploiting DLL hijacks:

- [Siofra](#)
- [DLLSpy](#)
- [Robber](#)

Basic DLL Hijack Example

Here's a simple DLL hijack example:

```
void BeUnsafe() {
    HMODULE module = LoadLibrary("functions.dll");
    // ...
}

BOOL WINAPI DllMain(HINSTANCE instance, DWORD reason, LPVOID reserved)
{
    if (reason != DLL_PROCESS_ATTACH)
        return TRUE;
    // Execute malicious code
    system("start calc.exe");
    return TRUE;
}
```

This basic example is easy to exploit but often fails in real-world scenarios due to process instability and the complexity of maintaining proper functionality.

Advanced Techniques

Execution Sinks

Two primary sinks from which DLL execution can originate:

- **Static Sink (IAT):** Occurs during process initialization or dynamic loading. The subsystem calculates dependencies and initializes them sequentially, verifying the export table.
- **Dynamic Sink (LoadLibrary):** Active code requests a new module without specifying required functions, often followed by `GetProcAddress`.

Function Proxying

To maintain process stability, proxy functionality to the real DLL:

- **Export Forwarding:** Redirect exports to another module.

```
#pragma comment(linker, "/export:ReadThing=real.ReadThing")  
#pragma comment(linker, "/export:WriteThing=real.WriteThing")
```

- **Stack Patching:** Walk backward from `DllMain` and replace the return value for `LoadLibrary` with a different module handle.

```
HMODULE hRealDll = LoadLibrary("real.dll");  
HMODULE hCurrentDll = NULL;  
__asm {  
    mov eax, [ebp+4]  
    mov hCurrentDll, eax  
}  
if (hCurrentDll == hOurDll) {  
    __asm {  
        mov eax, hRealDll  
        mov [ebp+4], eax  
    }  
}
```

- **Run-Time Linking:** Remap function pointers dynamically in *DllMain*.

```
BOOL WINAPI DllMain(HINSTANCE instance, DWORD reason, LPVOID reserved)
{
    if (reason != DLL_PROCESS_ATTACH)
        return TRUE;

    HMODULE hRealDll = LoadLibrary("real.dll");
    if (!hRealDll)
        return FALSE;

    FARPROC realFunction = GetProcAddress(hRealDll, "FunctionName");
    if (!realFunction)
        return FALSE;

    // Code to patch function pointers dynamically
    return TRUE;
}
```

- **Run-Time Generation:** Rebuild the entire export address table at runtime.

```
BOOL WINAPI DllMain(HINSTANCE instance, DWORD reason, LPVOID reserved)
{
    if (reason != DLL_PROCESS_ATTACH)
        return TRUE;

    HMODULE hRealDll = FindModule(instance);
    if (!hRealDll)
        return FALSE;

    ProxyExports(hRealDll);
    return TRUE;
}
```

Loader Lock

The loader lock can cause deadlocks or crashes if not handled correctly. Avoid calling functions like `LoadLibrary`, `CreateThread`, or synchronization functions within `DllMain`. Use threading to execute complex code outside `DllMain`.

```
BOOL WINAPI DllMain(HINSTANCE instance, DWORD reason, LPVOID reserved)
{
    if (reason != DLL_PROCESS_ATTACH)
        return TRUE;

    DWORD dwThreadId;
    HANDLE hThread = CreateThread(NULL, 0, ThreadFunc, NULL, 0, &dwThreadId);
    if (hThread == NULL)
        return FALSE;

    CloseHandle(hThread);
    return TRUE;
}

DWORD WINAPI ThreadFunc(LPVOID lpParam)
{
    // Complex code here
    return 0;
}
```

Function Hooking for Stability

Implement hooks to maintain stability and ensure execution control after the loader finishes.

```
BOOL WINAPI DllMain(HINSTANCE instance, DWORD reason, LPVOID reserved)
{
    if (reason != DLL_PROCESS_ATTACH)
        return TRUE;

    // Implement hooks to maintain stability
    HookFunction();

    return TRUE;
}


void HookFunction()
{
    // Code to hook functions and ensure execution control
}
```



Conclusion

In conclusion, adaptive DLL hijacking represents a sophisticated and evolving threat in the cybersecurity landscape. By exploiting the Windows DLL loading mechanism, attackers can inject malicious code into legitimate processes, bypassing traditional security measures. The advanced techniques of export table cloning, dynamic IAT patching, stack walking, and runtime table reconstruction not only enhance the effectiveness of these attacks but also ensure the stability and functionality of the compromised process. This underscores the importance of a deep understanding of the Windows loader and the need for continuous vigilance and adaptation in cybersecurity practices.

To counter these sophisticated threats, it is imperative for cybersecurity professionals to be equipped with the knowledge of tools and frameworks like Siofra, DLLSpy, and Robber, which facilitate the identification and exploitation of DLL hijacking vulnerabilities. By leveraging these tools, defenders can better understand the mechanisms of adaptive DLL hijacking, enabling them to develop robust mitigation strategies. As the threat landscape continues to evolve, staying ahead requires a commitment to ongoing education and the implementation of advanced defensive measures to protect against these complex attacks.





cat ~/.hades

"Hades" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:

WWW.HADESS.IO

Email

MARKETING@HADESS.IO