R&D Log: Analysis of DLL Hijacking for Application Persistence

Project: Persistence POC

Author: Etai Yaron

Date: 24/10/2025

1. Executive Summary

This document details the research and development of a proof-of-concept (POC) for achieving persistence by exploiting a DLL hijacking vulnerability. The investigation began with an attempt to hijack critical, static-load dependencies, which led to predictable application failures.

Through methodical root cause analysis of these failures (deciphering `0xc000007b` errors and missing "Entry Point" exceptions), the methodology was revised. A stable POC was ultimately achieved by simulating and targeting a non-critical, dynamically-loaded dependency. This project validates that successful DLL hijacking is contingent on precise architecture matching and a deep understanding of the target's dependency-loading process (static vs. dynamic).

2. Objective & Scope

The primary objective was to investigate the viability of DLL hijacking as a persistence vector on a modern, compiled 64-bit C++ application.

The research scope included:

- Identifying potential DLL hijacking vulnerabilities in a target executable.
- Developing a minimal payload DLL to signal execution.
- Analyzing common failure modes and crash errors.
- Establishing a stable proof-of-concept that achieves payload execution without compromising host application stability.

3. Methodology & Tools

- **Analysis Tool: Process Monitor (ProcMon)**
  - Used for real-time analysis of file system and process activity to identify DLL load attempts.
- **Development Tool: Visual Studio 2022 (C++)**
  - Used to develop and compile the target executable (`My Cybersecurity toolbox.exe`) and the various payload DLLs.
- **Debugging Tool: Windows Resource Monitor**

    - Used to identify and resolve file-lock linker errors (LNK1104) caused by non-terminated debug processes.

4. R&D Phases

Phase 1: Reconnaissance (Vulnerability Hunting)

The initial phase focused on identifying potential vulnerabilities in the target executable.

1. **Tool:** ProcMon
2. **Filters:**
    - Process Name | is | My Cybersecurity toolbox.exe | Include
    - Path | ends with | .dll | Include
    - Result | is | NAME NOT FOUND | Include
3. **Findings:** Analysis revealed that the executable (compiled in Debug mode) failed to find several debug-runtime libraries in its local directory, including MSVCP140D.dll and ucrtbased.dll. These became the targets for the initial experiment.

Phase 2: Initial Test - Static Dependency Hijacking (Failure Analysis)

This phase tested the hypothesis that a critical dependency could be replaced to execute a payload.

- **Hypothesis:** A payload DLL, named to match a missing static dependency (e.g., ucrtbased.dll), could execute a payload in its DllMain and then return FALSE to signal a "graceful" failure, preventing a hard crash.
- **Payload DLL:** A simple x64 DLL designed to execute a MessageBoxA on DLL_PROCESS_ATTACH and then return FALSE.
- **Result: Total application failure.** The MessageBox payload *did not* execute. The application immediately terminated with one of two errors, depending on the specific payload code:
    1. **"Entry Point Not Found" Error:** This occurred when the payload DLL used C++ Standard Library functions (like std::ofstream). The loader, expecting the *real* C runtime, could not find the necessary C++ function exports (e.g., ?width@ios_base@std...) within our minimal payload DLL and terminated the process.
    2. **0xc000007b ("Application was unable to start") Error:** This occurred consistently, even with a pure C-style Win32 payload.

Phase 3: Root Cause Analysis (Troubleshooting)

The failures from Phase 2 were analyzed to understand the underlying mechanics.

- **Finding 1: The 0xc000007b Error.** This error was consistently traced back to an **architecture mismatch**. The target executable was **x64**, but the initial DLL project default was **x86 (Win32)**. A 64-bit process cannot load a 32-bit DLL. The OS loader identifies this mismatch and terminates the process *before* DllMain is ever called.
- **Finding 2: The "Entry Point" Error.** This demonstrates that even with a matching **x64** architecture, replacing a **critical, static-load dependency** (like the C runtime) is non-trivial. The application *requires* these function exports to initialize. When the loader fails to find them, it terminates the process. This failure occurs *before* DllMain (which runs *after* imports are resolved).
- **Conclusion:** The initial hypothesis was flawed. Targeting critical, static-load dependencies is a fragile and complex method that requires advanced **DLL Proxying** to forward all expected function calls to the real DLL.

Phase 4: Revised Hypothesis & POC - Dynamic Load Hijacking (Success)

Based on the findings from Phase 3, the hypothesis was revised: **A stable hijack must target a non-critical, dynamically-loaded DLL.**

1. **Simulated Vulnerability:** To create a clean and reliable test, the host application (My Cybersecurity toolbox.exe) was modified to simulate this vulnerability. The following code was added to its main() function:
2. // Simulate a 5-second delay
3. std::this_thread::sleep_for(std::chrono::seconds(5));
4.
5. // Simulate a plugin load
6. HMODULE hModule = LoadLibraryA("non_existent_plugin.dll");
7. **Target:** non_existent_plugin.dll (a non-critical, dynamically-loaded target).
8. **Payload DLL:**
   - Compiled as **x64** (to match the host architecture).
   - DllMain payload: MessageBoxA(NULL, "Persistence Succeeded!", "R&D Test", MB_OK);
   - DllMain return: **return TRUE;** (to signal successful initialization).
9. **Final Test:** The compiled payload DLL was renamed to non_existent_plugin.dll and placed in the host's directory.
10. **Result: 100% SUCCESS.**
    - The host application launched and ran normally for 5 seconds.
    - The LoadLibrary call was triggered.
    - The loader found and loaded our payload DLL.
    - The **"Persistence Succeeded!" MessageBox appeared on screen.**
    - After clicking "OK," the host application continued to run without crashing.

5. Key Findings & Conclusion

This research demonstrates a clear methodology for identifying and exploiting DLL hijacking vulnerabilities.

- **1. Architecture is Paramount:** An architecture mismatch (x64 vs. x86) is an immediate-failure condition (`0xc000007b`) that prevents payload execution.
- **2. Static vs. Dynamic Loading:** The distinction is critical. Hijacking static-load dependencies (like runtimes) will fail unless all function exports are proxied. Hijacking dynamic-load dependencies (like plugins) is a simpler and more stable vector.
- **3. `DllMain` Return Value:** `return FALSE;` from `DLL_PROCESS_ATTACH` is not a viable persistence strategy, as it signals a fatal error to the loader and terminates the host application. A successful payload **must `return TRUE;`.**

Final Conclusion: This R&D project successfully developed a stable proof-of-concept for persistence by simulating a dynamic DLL load. The key takeaway is that persistence is not just about placing a file; it's about a deep, system-level understanding of the target's architecture, dependencies, and load order.

6. Future Work & Mitigation

- **Next R&D Step:** The next logical step is to research **DLL Proxying**, which would allow for the hijacking of *legitimate*, *static-load* DLLs without crashing the host.
- **Defensive Mitigation:** Developers can mitigate this attack vector by ensuring all calls to `LoadLibrary` use absolute paths and by enabling "Safe DLL Search Mode."