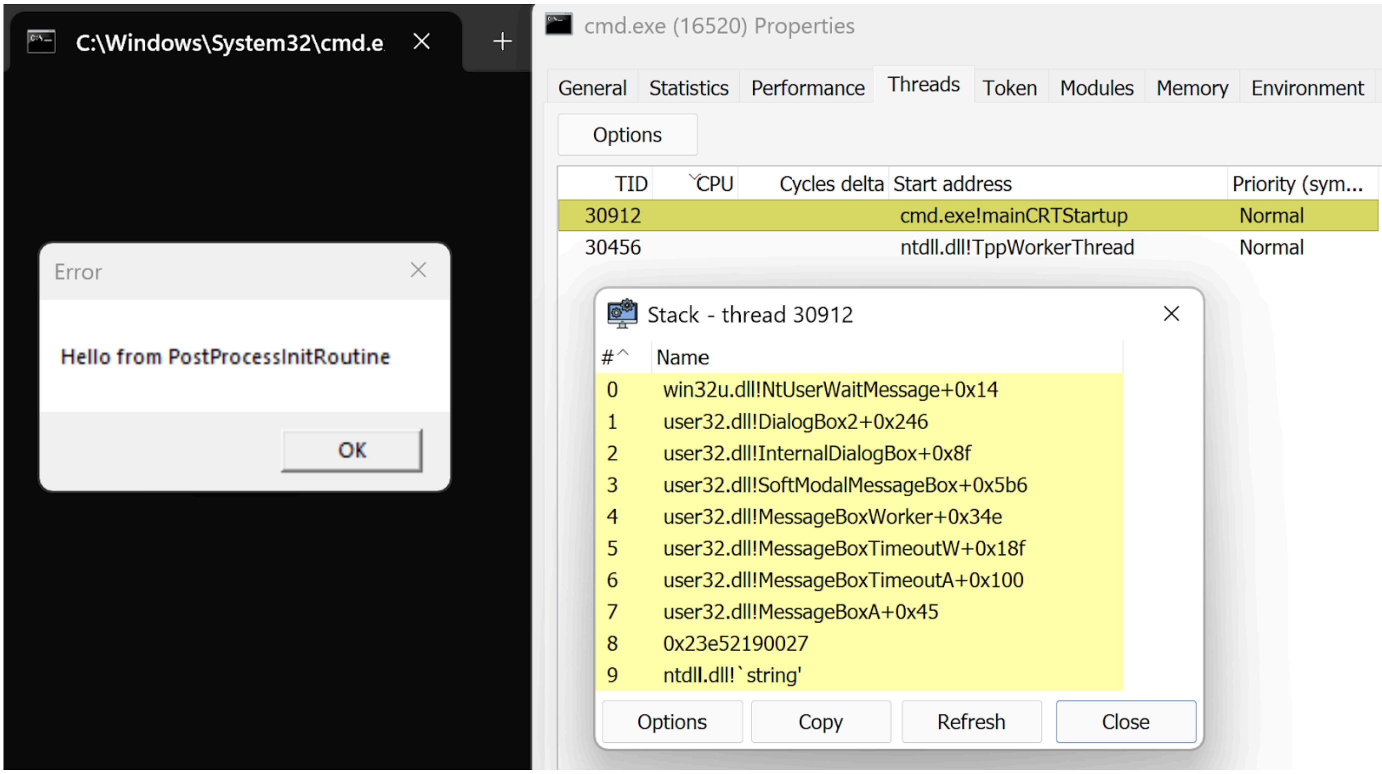
Escaping Loader Locks with PostProcessInitRoutine

originhq.com/blog/escaping-loader-locks-with-postprocessinitroutine

John Uhlmann

October 29, 2025



Next Generation
Endpoint Security

Execution in DllMain while DLL sideloading is constrained by the loader lock, potentially leading to deadlock for some payloads. `PostProcessInitRoutine` *provides a simple way to execute free of these restrictions.*

I was recently looking at the official documentation for the [Process Environment Block](https://) (PEB) structure and `PostProcessInitRoutine` caught my eye.

```
typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged;
    BYTE Reserved2[21];
    PPEB_LDR_DATA Ldr;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    BYTE Reserved3[520];
    PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
    BYTE Reserved4[136];
    ULONG SessionId;
} PEB;
```

There is more unofficial detail available from [Microsoft's public debugging symbols] which is what I usually work with - and especially the [annotated version] published by the System Informer team. But, on this day, I was only looking at the officially supported fields. I was familiar with `BeingDebugged`, `Ldr->InMemoryOrderModuleList`, `ProcessParameters` and `SessionId`—but I hadn't noticed `PostProcessInitRoutine` before.

The official documentation for this field just states "Not supported" and System Informer's unofficial annotation that it is a "Pointer to the post-process initialization routine available for use by the application" isn't much more illuminating. 🤔

Wikipedia provides a more informative definition of "A pointer to a callback function called after DLL initialization but before the main executable code is invoked" along with a very intriguing note that "This callback function is used on Windows 2000, but is not guaranteed to be used on later versions of Windows NT." Unfortunately, the Microsoft documentation version that this information originated from was not archived.

Windows Internals (7th Edition, p170) describes a step at the end of process initialisation as "Run the associated subsystem DLL post-process initialization routine registered in the PEB. For Windows applications, this does Terminal Services-specific checks, for example." This subsystem reference aligns with the timeline in the Wikipedia comment. Windows 2000 included POSIX and OS/2 subsystems, but these were removed for Windows XP and Windows Server 2003 leaving just the Windows (Console and GUI) subsystem until the Windows Subsystem for Linux was added in Windows 10.

The next clue came from ReactOS - an open-source operating system that is intended to be binary-compatible with computer programs developed for Windows Server 2003 and later versions of Microsoft Windows. At the end of `LdrpInitializeProcess` it calls `PEB->PostProcessInitRoutine`, though it describes it as "user-defined".

If we disassemble the Windows 11 25H2 version of `ntdll.dll` version we can still see that a call to `PEB->PostProcessInitRoutine` at the very end of `LdrpInitializeProcess` still exists.

```
NTSTATUS LdrpInitializeProcess(...) {
  // ... highly abridged function structure reconstruction
  PPEB peb = NtCurrentTeb()->ProcessEnvironmentBlock;

  // load and initialise kernel32.dll and kernelbase.dll
  LdrpInitializeKernel32Functions(...);

  // check for application compatibility shims
  LdrpInitShimEngine(...);

  // map all static DLL dependancies in parallel
  LdrpEnableParallelLoading(...);

  // check for application verifier
  if ((peb->NtGlobalFlag & 0x100) != 0 || (AvrfAppVerifierMode & 2) != 0 )
    AVrfInitializeVerifier(...);

  // break for debugger
  if( peb->BeingDebugged )
    LdrpDoDebuggerBreak();

  LdrpAcquireLoaderLock();
  // in dependency order, call the init routines of the static imports
  LdrpInitializeGraphRecurse(...);
  LdrpReleaseLoaderLock();

  PostProcessInitRoutine = peb->PostProcessInitRoutine;
  if( PostProcessInitRoutine )
    PostProcessInitRoutine();

  return status;
}
```

So this function appears to only ever be called during the process initialisation which occurs on the main thread. It is called after the entry points of all static imports and after the AppVerifier and ShimEngine callbacks, but before the APC queue is drained and before the main executable's entry point.

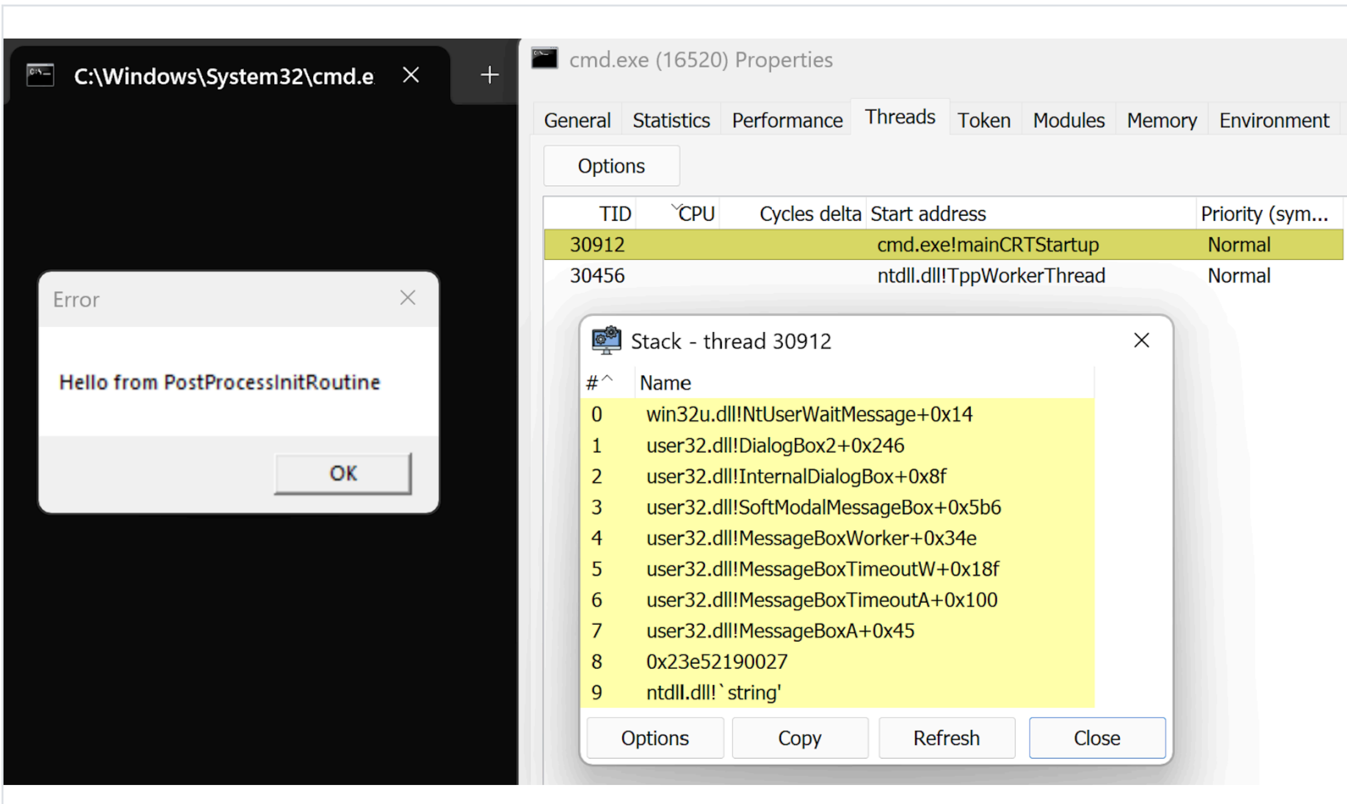
So we have a function pointer at a known PEB offset that should be called during process initialisation. This is potentially an alternate early execution trigger for child process injection. This would provide reliable pre-application execution, though would still be after the insertion of user-mode hooks by some products. Further information on the process initialisation flow and security product behavior is available in the following blogs -

- [Bypassing EDRs With EDR-Preloading | MalwareTech]
- [Introducing Early Cascade Injection | Outflank Blog]

It was now time to test the theory. I created a suspended process, wrote a function pointer into the `PostProcessInitRoutine` field, resumed execution and…nothing happened. I read the value back—and it had been zeroed. So I repeated the experiment with a hardware breakpoint on the field to discover that `user32.dll!UserClientDllInitialize` was clearing it. The Windows subsystem was explicitly **not** calling a subsystem post-process initialisation routine—but only if `user32.dll` is statically imported.

There are hundreds of binaries that do not statically import `user32.dll` in System32. For each of these, code execution in a child process can be hijacked with a simple pointer write to the PEB and **writing to the PEB of a child process is completely normal process creation behavior.** For example, this is how some application compatibility data is passed.

For this set of binaries, we can trigger execution during process initialization:



So far we have a minor variant of child process injection. What about binaries that *do* statically import `user32.dll`? We can't set `PostProcessInitRoutine` **before** the first thread starts, but we can set it **after** the `user32.dll` entrypoint has been called.

This is the exact set of conditions under which many [DLL sideload] entry points are called. However, these entry points are called while the loader lock is held which significantly constrains the actions that can be safely taken.

- [DllMain entry point - Win32 apps | Microsoft Learn]
- [What is Loader Lock?] by Elliot Killick

These limitations have complicated DLL proxying approaches for many years -

- Adaptive DLL Hijacking by Nick Landers
- Perfect DLL Hijacking by Elliot Killick

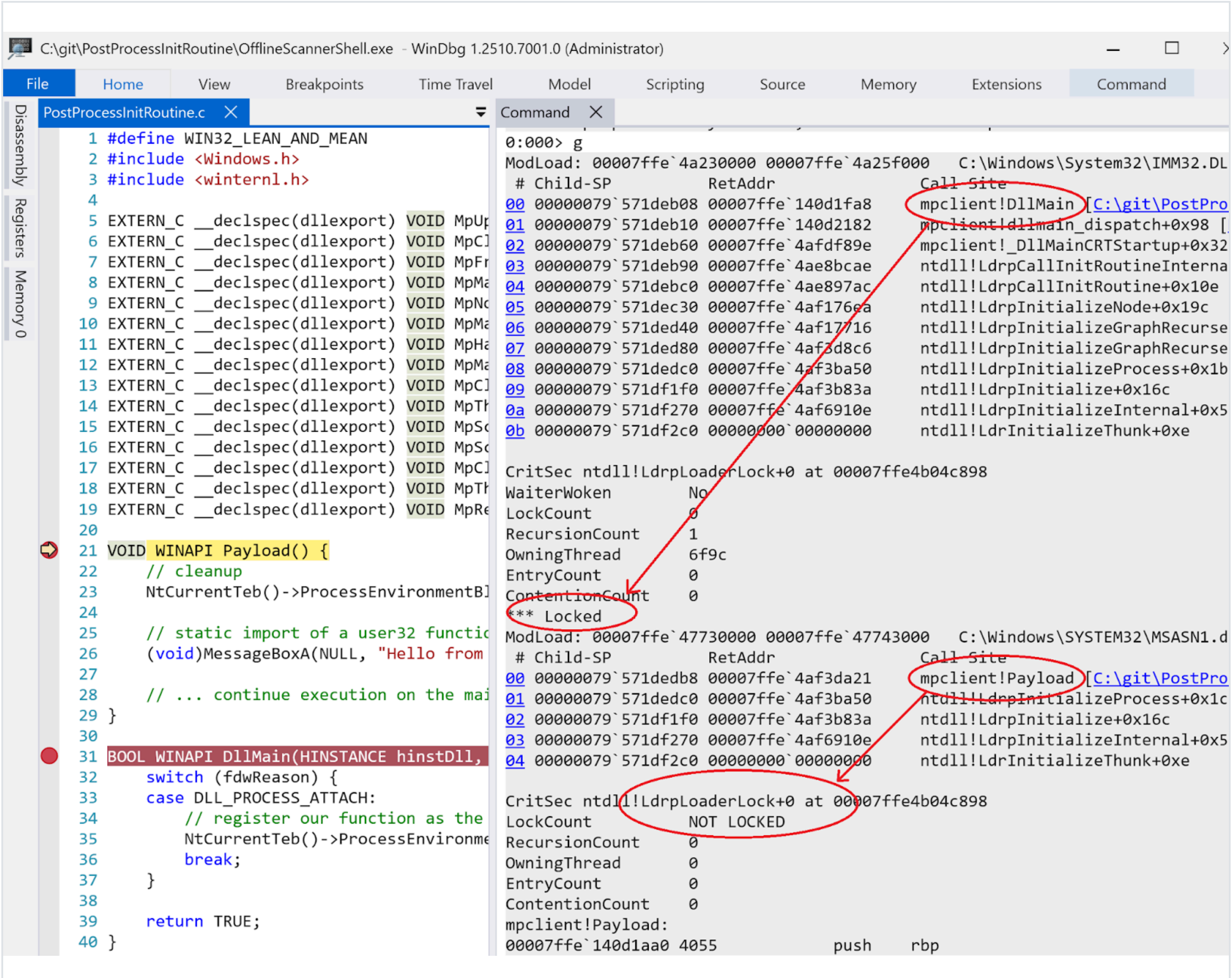
However, this process post-initialization routine is called **outside of the loader lock**. Presumably this was the original mechanism for complex subsystem-specific initialisation to occur prior to the user application entry point.

Our loader lock escape is then very, very simple:

```
#define WIN32_LEAN_AND_MEAN
#include <Windows.h>
#include <winternl.h>

VOID Payload() {
    // cleanup
    NtCurrentTeb()->ProcessEnvironmentBlock->PostProcessInitRoutine = NULL;

    // ... continue execution on the main thread prior to APCs and entrypoint
    MessageBoxA(NULL, "Hello from PostProcessInitRoutine", "Hijack", MB_OK);
}

BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, LPVOID lpvReserved) {
    switch (fdwReason) {
    case DLL_PROCESS_ATTACH:
        // register our function as the subsystem post-init routine
        // note - ensure that we staticly import user32 to ensure call ordering
        NtCurrentTeb()->ProcessEnvironmentBlock->PostProcessInitRoutine = Payload;
        break;
    }

    return TRUE;
}
```

Since this deprecated feature isn't used by the Windows subsystem, it can be "borrowed" by a single statically imported DLL to provide a mechanism for reliable unconstrained execution during process launch.

This gives the DLL simple, stable race-free execution irrespective of target application:



This technique primarily serves as a historical curiosity. It does not enable any new functionality, as workarounds for the loader lock have always existed. This workaround is just far simpler than was previously understood to be possible.