Making sure that a DLL loads only from your application directory

devblogs.microsoft.com/oldnewthing/20250313-00

Rosyna Keller March 13, 2025

Raymond Chen

A customer distributed a program and included its supporting DLLs in the same directory, because the application directory is the application bundle.

They worried about the case that the user deletes one of the supporting DLLs, and then when the program tries to load that DLL, a rogue copy somewhere else on the `PATH` gets loaded instead. They want to reject loading the DLL from anywhere other than the application directory.

You can accomplish this by explicitly calling `LoadLibraryEx` with the `LOAD_LIBRARY_SEARCH_APPLICATION_DIR` flag, which says that the function should look only in the application directory for the DLL. If it's not there, it gives up without searching any other directories. After you load the library, you can use `GetProcAddress` to get the functions.

Unfortunately, this is rather cumbersome since you have to switch from implicit loading to explicit loading, so you don't get the convenience of import libraries.

You might think that you can get the convenience back by using the `/DEPENDENTLOADFLAG` linker option with the value `0x200` (the numeric value of `LOAD_LIBRARY_SEARCH_APPLICATION_DIR`), but the problem is that the dependent load flag applies to *all* DLLs loaded via import tables, and that includes `kernel32` and other DLLs you probably wanted to load from the `system32` directory.

Now, the `system32` directory is writable only by administrators, so we could consider that a "safe" directory, because if somebody attacks that directory, they have already taken over the system. Therefore, you could use the `/DEPENDENTLOADFLAG` linker option with the value `0xA00`, which is the numeric value of `LOAD_LIBRARY_SEARCH_APPLICATION_DIR | LOAD_LIBRARY_SEARCH_SYSTEM32`. Alternatively, you could use the value `0x1000`, which is the

numeric value of `LOAD_LIBRARY_SEARCH_DEFAULT_DIRS`, which includes the application directory, the `system32` directory, and any directories added by `AddDllDirectory` and `SetDllDirectory`.

But wait, what is the issue we are trying to defend against? The stated scenario is "The user deletes a DLL from the application directory." In that case, the user already has write permission into the application directory, so instead of deleting the DLL, they can just replace it with a malicious DLL. Restricting the load to the application directory does not prevent a malicious DLL from being loaded.

But maybe your goal is not to create a security boundary but just to contain the scope of an error. If the user accidentally deletes the DLL from the application directory, at least prevent somebody else from injecting a DLL into the process by planting a DLL on the path.

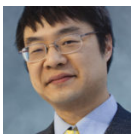
Now, the directories on the path fall into two categories. You have the directories on the global path, and the directories that are specific to a single user. If an attacker can plant a DLL into a directory on the global path, then that means that they have gained write permission onto the global path. To do this without administrator privileges requires that the global path contain a directory writable by non-administrators, which is an insecure configuration, so we are in the case of creating an insecure system and then being surprised that it is insecure. Instead of planting a rogue DLL on the path, the attacker could just plant, say, a rogue `notepad.exe`, and steal all your attempts to run notepad.

The other case is that the directory under attack is a directory on the per-user path. The user chose to add that directory, and if they added a directory that is writable by non-administrators other than the current user, they have once again created an insecure system because they have granted non-administrators the ability to inject things into their path.

The only attacks against rogue DLLs on the path assume that the system has already been compromised. So this issue is not about protecting a secure system but rather trying to protect from an already-compromised system.

Author

Raymond Chen



Raymond has been involved in the evolution of Windows for more than 30 years. In 2003, he began a Web site known as The Old New Thing which has grown in popularity far beyond his wildest imagination, a development which still gives him the heebie-jeebies. The Web site spawned a book, coincidentally also titled The Old New Thing (Addison Wesley 2007). He occasionally appears on the Windows Dev Docs Twitter account to tell stories which convey no useful information.

12 comments

Discussion is closed. Login to edit/delete existing comments.

Newest

- RK

  March 24, 2025

  As Raymond mentioned, using LoadLibraryXX() requires you use GetProcAddress() to hook up functions address pointers. This becomes tedious.

  I am surprised Raymond didn't mention the manifest solution if it can really affect only the specific DLL and automatically connect the function pointers, as loading a DLL by linking to it normally would.

  SK

  Stefan Kanthak March 24, 2025

  "As Raymond mentioned, using LoadLibraryXX() requires you use GetProc-Address() to hook up functions address pointers. This becomes tedious."

  Nobody stops you to link using /DELAYLOAD:‹your.dll› and put the call of LoadLibraryEx() into your own __delayLoadHelper2() routine!

- RK

  March 19, 2025

  The way this is presented, this doesn’t sound like a security issue, it sounds more like a developer wanting to avoid DLL Hell.

  This would especially be a problem if they’re using a third-party DLL that is installed by multiple apps and the user decides to move only the .exe to another location, not realizing the DLL is a necessary component, thereby causing extremely weird (possibly indeterminate) crashes if it happens to grab a DLL from another app’s install that incorrectly installed it on a global search path.

  While a security feature could fix this usability issue (loading only DLLs signed by...

  Read more

  - SK

    Stefan Kanthak March 23, 2025

    Security left aside it is possible with an XML snippet <file loadFrom="%__APPDIR__%‹filename.dll›" name="‹unqualified filename from import directory›"> placed in the applications manifest.

  - SK

    Stefan Kanthak March 23, 2025 · Edited

    "Although I am surprised to learn DLLs can’t embed their own search path when they’re linked to an executable without modifying the global search paths."

    Please inform yourself about Activation Contexts and how these are derived from Application Manifests or Assembly Manifests.
    Additionally pay a look at the "LOAD_WITH_ALTERED_SEARCH_PATH" flag.

    Read more

- MM

  Flux March 17, 2025 · Edited

  The customer request is less about security and more about preventing collateral loading.

  Imagine a user who installs App A, but during installation, chooses Custom Setup, and opts not to install `ffmpeg.dll` (or component that includes `ffmpeg.dll`). FFmpeg, however, is a popular free and open-source solution. Many different apps may ship with custom copies of `ffmpeg.dll`. It is reasonable to assume that one such app, say App B, comes with an incompatible copy of `ffmpeg.dll` that App A wouldn't want to load.

  Cole Tobin March 18, 2025 · Edited

  If `ffmpeg.dll` is required for functionality, the installer shouldn't give the user the option to *not* install it. This is Windows, not Linux. If you need a DLL, provide it yourself. However, if the `ffmpeg.dll` requiring component is optional, the installer should leave a note for the application to not allow that functionality. Your scenario isn't a Windows problem, but an installer/application bug.

- TM

  Tom Mason March 15, 2025

  I've had a somewhat legitimate reason for this before. We were shipping a copy of a newer version of onnxruntime.dll than was included in windows at the time. We got some crash reports that didn't make sense, so we added some code to log the md5 sum of the onnxruntime.dll that we had loaded. Sure enough, the md5 didn't match. We never did figure out why it was loading the wrong one (only for a tiny minority of users, I might add), but explicitly loading it using LoadLibraryW and the full path fixed the issue. Luckily onnxruntime already uses the...

  Read more

- 

  Shawn Van Ness March 14, 2025
  My guess: this is less about security, more about reliability and supportability -- I've seen this flavor of DLL-hell with libraries like DbgHelp.dll, which have changed a lot over the past 20 years, in binary-incompatible ways, but are used and redist'ed fairly ubiquitously -- including by other DLLs you may end up loading, directly or indirectly (thinking of things like FPS counters and VR/XR frameworks).

  It's tough when your app crashes due to an incompatible DLL conflict, and even tougher when it's your crash-handler that's crashing because it can't use DbgHelp.dll to log a stack trace, so your customer has...

  Read more

  JH

  Joshua Hudson March 14, 2025
  That doesn’t work _at all_.

  I filed a vulnerability report after discovering that User32.dll loads additional DLLs dynamically and throws out the value of /DEPENDENTLOADFLAG that the application was linked with. The vulnerability got rejected. So we live in the world where you can’t restrict the DLL load order for security.

  SK

  Stefan Kanthak March 23, 2025
  Ask your administrator to configure SAFER, AppLocker or WDAC for “write XOR execute” in the (NTFS) filesystem.

- 

  Mike Morrison March 14, 2025
  The application bundle is responsible only for the bundle. The app isn’t responsible for insecure system configurations outside the bundle that were not made by the app or installer itself. I don’t see why apps shouldn’t harden their DLL loading (and the directory permissions) due to the possibility of insecure configs outside of the bundle.

Stay informed

Get notified when new posts are published.