Your program loads libraries by their short name and you don't even realize it

devblogs.microsoft.com/oldnewthing/20110505-00

May 5, 2011

Raymond Chen

In the discussion of the problems that occur if you load the same DLL by both its short and long names, Xepol asserted that any program which loads a DLL by its short name "would have ONLY itself to blame for making stupid, unpredictable, asinine assumptions" and that Windows should "change the loader to NOT load any dll with a short name where the short name and long name do not match." The problem with this rule, well, okay, one problem is that you're changing the rules after the game has started. Programs have been allowed to load DLLs by their short name in the past; making it no longer possible creates an application compatibility hurdle. Second, it may not be possible to obtain the long name for a DLL. If the user has access to the DLL but not to one of the directories in the path to the DLL, then the attempt to get its long name will fail. The "directory you can access hidden inside a directory you cannot access" directory structure is commonly used as a less expensive alternative to Access Based Enumeration. Maybe your answer to this is to say that this is not allowed; people who set up their systems this way deserve to lose. Third, the application often does not control the path being passed to `LoadLibrary`, so it doesn't even know that it's making a stupid, unpredictable, asinine assumption. For example, the program may call `Get-ModuleFileName` to obtain the directory the application was installed to, then attempt to load satellite DLLs from that same directory. If the current directory was set by passing a short name to `SetCurrentDirectory` (try it: `cd C:\Progra~1`) then the program will unwittingly be making a stupid, unpredictable, asinine decision. (But since the decision is being made consistently, there was no problem up until now.) When you call `CoCreate-Instance`, there is nearly always a `LoadLibrary` happening behind the scenes, because the object you are trying to create is coming from a DLL that somebody else registered. If they registered it with a short name, then any application that wanted to use that object runs afoul of the new rule, even though the application did nothing wrong. Now, you might argue that this just means that the component provider is the one who made a stupid, unpredicable, asinine assumption. But that may not have been a stupid assumption at the time: 16-bit applications see only short names, so it might have been that that's the only thing they can do. Or the component may have made a conscious decision to register under short names; this was common in the Windows 95 era because file names often passed through components that didn't understand long file names. (Examples: 16-bit applications, backup applications.) Even if you decide that what was once a reasonable decision is now asinine, you have to get there from here. Do you declare all 16-bit applications asinine (because they can only use those asinine short file name)? Even for the 32-bit components, how do you convince them to switch over? Can you even get them to switch over? Until they do, their component will not be accessible: The shell extension won't be loaded until they fix their registration. A program which anticipated this problem and always loaded DLLs by their short names is now broken. (After all, you had to make an arbitrary decision to use long names exclusively or short names exclusively, and someone may have chosen the other branch of the decision tree.) And long name/short name is really just a special case of a single file having multiple names. Other ways of creating multiple names include hard links, symbolic links, and junction points. If a file has two names due to hard links, which one is the "preferred" name and which is the "asinine" name? (And what if the "preferred" name is not available to the user? Suppose you decide that the preferred name is the one that comes first alphabetically. Can I launch a denial-of-service attack by creating a hard link to `C:\Windows\system32\shell32.dll` called `C:\A\A.DLL`? All attempts to link to `C:\Windows\system32\shell32.dll` will fail because it is now the "asinine" name.) Does this rule against loading DLLs by "asinine" names negate part of the functionality of the compatibility paths introduced in Windows Vista (since, under this new rule, you can't use them to load DLLs or execute programs because they entail traversing a symbolic link).

Perhaps these concerns could be addressed in a manner that didn't have all these problems, but it's an awful lot of complexity. In general, simple rules are preferred to complex rules. Especially if the complexity is to address an issue that was not a serious problem to begin with.

Raymond Chen

Follow