Don't forget to unregister your window classes when your DLL shuts down dynamically

devblogs.microsoft.com/oldnewthing/20060920-07

September 20, 2006

Raymond Chen

If your DLL is unloaded dynamically, you need to make sure you have unregistered your window classes. (You can tell whether the `DLL_PROCESS_DETACH` is due to a dynamic unload or whether it's due to process termination by checking the `lpReserved` parameter to your `DllMain` function.) If you forget to unregister your window classes, all sorts of bad things can happen: First, if you registered any of those classes as a `CS_GLOBALCLASS`, then people will still be able to create a window of that class by passing its class name to the `CreateWindowEx` function (or any other function that leads to `CreateWindowEx`). Since your DLL is no longer in memory, the moment it receives a window message (like, say, `WM_NCCREATE`), the process will crash since the window procedure has been unloaded. This manifests itself in crashes with the instruction pointer in no-man's land—these are typically not easy to debug, and the Windows error reports that are generated by these crashes won't even be assigned to your DLL since your DLL is long gone. Second, even if you registered the classes as private classes, you are still committing namespace pollution, leaking the class into a namespace that you no longer own. If another DLL gets loaded at the same base address as your DLL (thereby receiving the same `HINSTANCE`, it inherits this dirty namespace. If that DLL wants to register its own class that happens to have the same name as the class you leaked, its call to `RegisterClassEx` will fail with `ERROR_CLASS_ALREADY_EXISTS`. This typically leads to the DLL failing to initialize or (if the problem is not detected) an attempt to create a window of that class creating instead a window of **your leaked class**, with a window procedure whose address now resides somewhere in the middle of this new DLL. This is even worse than an instruction pointer in no-man's land; instead, control goes to a random instruction in the new DLL and probably will manage to execute for a little while before finally keeling over. What's worse, not only does the crash not get reported against your DLL (which is no longer in memory), but it gets erroneously reported against the **new DLL** since it is the new DLL's code that was executing when the crash finally occurred. Congratulations, you just created work for somebody you never met. Those poor victims are going to be scratching their heads trying to figure out how control ended up in the middle of a totally random function with completely nonsense values on the stack and in the registers. Third, the namespace you pollute can be your own. Suppose you registered a class as a `CS_GLOBALCLASS`, then your DLL gets unloaded and you forget to unregister the class.

Later, your DLL gets reloaded, but due to changes in the virtual address map, it gets loaded at a new address. Now your DLL attempts to re-register its `CS_GLOBALCLASS` classes, and the call fails with `ERROR_CLASS_ALREADY_EXISTS`. If you're lucky, your DLL detects the error and fails to load, resulting in missing functionality. If you're unlucky, you fail to detect the error and succeed the load anyway. Then the code that did the `LoadLibrary` will try to create a window with that class, but instead of getting your DLL's window class (which failed to register), it gets the window class left over by that first copy of your DLL! Since that DLL no longer exists, you get a crash with the instruction pointer off in no-man's land. This is not a purely theoretical problem. The shell common controls library contained this bug of neglecting to unregister all its classes when dynamically unloaded, and we had to issue a hotfix because the crashes caused by it were actually occurring on real users' machines. Don't be the one responsible for having to issue a hotfix for your product. Unregister your classes if the process is going to continue running after your DLL unloads. Because it's the right thing to do.

(Now, you might notice that this goes against the rule of not calling out to other DLLs during your `DLL_PROCESS_ATTACH`. The solution for this is to have a "cleanup" function that people must call before calling `FreeLibrary` on your library to balance the "initialization" function that they had to call to register your control classes. On the other hand, if you failed to plan ahead for this, such as the shell common control did with its `InitCommonControlsEx` function without a matching `UninitCommonControls` function, then you have to decide between the lesser of two evils.)

Raymond Chen

Follow