Why does ShellExecute return SE_ERR_ACCESSDENIED for nearly everything?

devblogs.microsoft.com/oldnewthing/20121018-00

October 18, 2012

Raymond Chen

We saw a while ago that the `ShellExecute` function returns `SE_ERR_ACCESSDENIED` at the slightest provocation. Why can't it return something more meaningful? The short-term answer is that the return value from `ShellExecute` is both a success code and an error code, and you check whether the value is greater than 32 to see which half you're in. In particular, the error code case is if the value you got is less than or equal to 32. This already demonstrates that the error codes are limited to values less than or equal to 32. And all those error codes are already accounted for, so there's nowhere to stick "an error not on the original list of 32 possible error codes." Therefore, any error that wasn't on the original list of error codes gets turned into `SE_ERR_ACCESSDENIED`, in the same way that MS-DOS turned any error that didn't map to one of its original errors into 5 (access denied). Okay, but why was 32 chosen as the cut-off? The `ShellExecute` function didn't come up with that number. That number came from the kernel folks, who decided that `WinExec` function returned the handle to the application that was executed on success, or an error code less than 32 on failure. And back in the old days, `ShellExecute` was just a function that called `FindExecutable` and then passed the result to `WinExec`, so following the `WinExec` pattern made sense. (You may have noticed a tiny discrepancy there. The shell folks decided to add a new error code `SE_ERR_DLLNOTFOUND` with a numeric value of 32, thereby making the return value from `ShellExecute` behave subtly differently from that of `WinExec`. The people who made this decision probably regretted it once it became clear that lots of applications were checking the return value incorrectly, but it's too late to fix it now.) Okay, so let's peel back another layer: Why did the `WinExec` function overload the return value? Well, overloaded return values were all the rage back then. A lot of functions to create something return the created object on success, or null on failure. The kernel folks said, "Well, we can do even better than that. Not only can we tell you that we failed to create the application, we can even tell you why! You see, MS-DOS has a maximum of 31 error codes, so we can just return the error code directly if we can ensure that no values less than 32 are valid segments. And we can make that guarantee because the 8086 processor reserves the first 1024 bytes of memory (the first 64 segments) for its interrupt vector table, so no application could possibly be loaded there. Hooray! We're such over-achievers!" This weird way of reporting errors from `ShellExecute` has been preserved for compatibility. New

applications would probably better served to switch to the `ShellExecuteEx` function instead, since it reports errors by calling `SetLastError` with the *real* error code before returning. (In other words, you can call `GetLastError` to get the real error code.) **Bonus chatter**: Wait a second, if `GetLastError` gets you the real error code, how come the original report was that the `ShellExecuteEx` function also returns `SE_ERR_ACCESSDENIED`? Because it depends on what you mean by "returns". Technically speaking, the `ShellExecuteEx` function returns `FALSE` for all errors, since it is prototyped as returning a `BOOL`. When somebody says that it returns an error code, you first have to ask where they got that error code from. If they got it from `GetLastError`, then they'll get a meaningful error code, or at least something more meaningful than `SE_ERR_ACCESSDENIED`.

But if instead they look at the `hInstApp` member of the `SHELLEXECUTEINFO` structure, then they'll get that useless `SE_ERR_ACCESSDENIED` value again. Because the `hInstApp` is where the legacy return value is recorded. If you look there, you're going to see the old lame error code. So don't look there.

Raymond Chen

Follow