What were ShellExecute hooks designed for?

devblogs.microsoft.com/oldnewthing/20080910-00

September 10, 2008

Raymond Chen

Windows 95 introduced (and Windows Vista removed) the concept of ShellExecute hooks. These are objects which implemented the `IShellExecuteHook` interface. That interface had just one method: `IShellExecuteHook::Execute`, which took a `SHELLEXECUTEINFO` structure and returned `S_OK` to indicate that the item was executed, `S_FALSE` to allow processing to continue, and an error code to halt processing.

The intended purpose of the shell execute hook was to allow you to extend the set of strings that can be executed. For example, Internet Explorer 1 used a shell execute hook so that you could type `http://www.microsoft.com/` into the Run dialog and invoke the Web browser. This was necessary because the original version of Windows 95 had no special knowledge of URLs. Without the hook, typing a URL into the Run dialog would have generated an error.

Let's look at those return values again. The simplest return value is `S_FALSE`, which means that you didn't recognize the item and wish to continue normal processing. Returning `S_OK` means that you recognized the item and successfully executed it. Returning an error code means that you recognized the item but an error occurred when you tried to execute it. Returning an error stops further processing, which is important in this case, because you do want to stop processing: You recognized what the user was trying to do and you couldn't do it. If you had returned `S_FALSE`, then the shell (and other shell execute hooks) would take their shot at executing the item. Since it's something only you recognize, they would fail, and the user would get some sort of error like "File not found" instead of an error that accurately describes why you couldn't execute the item.

That's the theory, but I've seen a few applications that use shell execute hooks for completely different purposes. And they are doomed to failure. One application installs a shell execute hook in order to implement some sort of weird concept of "security". When their hook is called, they check whether the user is "allowed" to run the program; if allowed, they return `S_FALSE` and if blocked, they return `E_ACCESSDENIED`. Another application uses their hook to log all the programs that the user runs, presumably so the logs can be inspected for auditing purposes. After logging the action, they return `S_FALSE` to allow normal execution to continue.

Both of these methods are really an abuse of the shell execute hook model since they aren't extending the set of items that can be executed by the shell. They're just using the hook model to sneak some code into the execute path. But that's also the problem.

First, they are intercepting only actions that go through the `ShellExecuteEx` function. If somebody just calls `CreateProcess` directly, then the shell execute hooks won't run, and whoever it is just snuck past your "security" and "auditing" system.

Second, these hooks both assume that they are the only shell execute hook in the system. (Remember, there can be more than one.) Suppose there are three hooks installed, the hook installed by Internet Explorer to handle URLs, the "auditing" hook, and the "security" hook, and suppose that the shell decides to invoke them in that order. (The order in which hooks are executed is unspecified because, if you use them as intended, the order doesn't matter.) If the user types a URL into the Run dialog, the URL hook launches the Web browser and returns `S_OK` to say, "I took care of this one." Your "auditing" hook never got to see the URL, and your "security" hook never got a chance to reject the operation. Hooks that sit ahead of the "security" hook not only get to bypass security, but they also elude the "auditing" hook.

So much for "auditing" and "security".

Sidebar

Since shell execute hooks effectively become part of the `ShellExecute` function, they are unwittingly subject to all the application compatibility constraints that `ShellExecute` must follow. Welcome to the operating system. Your world has just gotten a lot more complicated.

There is no "list of application compatibility constraints" (oh how life would be simple if there were); basically, anything that `ShellExecute` did in the past is a de facto compatibility constraint. Some programs crash if you create a worker thread inside `ShellExecute`; other programs crash if you call `PeekMessage` inside `ShellExecute`. Some programs call `ShellExecute` and immediately exit, which means that your shell execute hook had better not return before its work is done (or at least before the work has been given to another process to finish). Some programs call `ShellExecute` on an unusually small stack, so you have to watch your stack usage. And even though it is called out in the documentation that programs should not call shell functions from inside `DllMain`, that doesn't stop people from trying, anyway. Your shell execute hook had better behave in a sane manner when called under such "impossible" conditions.

Raymond Chen

Follow