How is it possible to run Wordpad by just typing its name even though it isn't on the PATH?

devblogs.microsoft.com/oldnewthing/20110725-00

July 25, 2011

Raymond Chen

In a comment completely unrelated to the topic, Chris Capel asks how Wordpad manages to run when you type its name into the Run dialog even though the command prompt can't find it? In other words, the Run dialog manages to find Wordpad even though it's not on the `PATH`. Chris was unable to find anywhere I discussed this issue earlier, but it's there, just with Internet Explorer as the application instead of Wordpad. It's through the magic of `App Paths`. `App Paths` was introduced in Windows 95 to address the *path pollution* problem. Prior to the introduction of `App Paths`, typing the name of a program without a fully-qualified path resulted in a search along the path, and if it wasn't found, then that was the end of that. File not found. As a result, it became common practice for programs, as part of their installation, to edit the user's `AUTOEXEC.BAT` and add the application's installation directory to the path. This had a few problems. First of all, editing `AUTOEXEC.BAT` is decidedlly nontrivial since batch files can have control flow logic like `IF` and `CALL` and `GOTO`. Finding the right `SET PATH=...` or `PATH ...` command is an exercise in code coverage analysis, especially since MS-DOS 6 added multi-config support to `CONFIG.SYS`, so the value of the `CONFIG` environment variable is determined at runtime. If you wanted to avoid hanging your setup program, you would have to solve the Halting Problem. (You can't just stick at `PATH ...` at the beginning because it might get wiped out by a later `PATH` command, and you can't just stick it at the end, because control might never reach last line of the batch file.) And of course, very few uninstall programs would take the time to undo the edits the installer performed, and even if they tried, there's no guarantee that the undo would be successful, since the user (or another installer!) may have edited the `AUTOEXEC.BAT` file in the meantime. Even if you postulate the existence of the *`AUTOEXEC.BAT` editing fairy* who magically edits your `AUTOEXEC.BAT` for you, you still run into the `PATH` length limit. The maximum length of a command line was 128 characters in MS-DOS, and if each program added itself to the `PATH`, it wouldn't be long before the `PATH` reached its maximum length.

Pre-emptive Yuhong Bao irrelevant detail that has no effect on the story: Windows 95 increased the maximum command line length, but the program being launched needed to know where to look for the "long command line". And that didn't help existing installers which were written against the old 128-character limit. Give them an `AUTOEXEC.BAT` with a line longer than 128 characters and you had a good chance that you'd

hit a buffer overflow bug. On top of the difficulty of adding more directories to the `PATH`, there was the recognition that this was another case of using a global setting to solve a local problem. It seemed wasteful to add a directory to the path just so you could find *one file*. Each additional directory on the path slowed down path sarching operations, even the ones unrelated to locating that one program. Enter `App Paths`. The idea here is that instead of adding your application directory to the path, you just create an entry under the `App Paths` key saying, "If somebody is looking to execute `contoso.exe`, I put it over here." Instead of adding an entire directory to the path, you just add a single file, and it's used only for application execution purposes, so it doesn't slow down other path search operations like loading DLLs. (Note that the old documentation on App Paths has been superseded by the new documentation linked above.) Now that there was a place to store information associated with a particular application, you may as well use it for other stuff as well. A secondary source of path pollution came from applications which added not only the application directory to the path, but also a helper directory where the application kept its DLLs. To address this, an additional `Path` value specified which directories your application wanted to be added to the path before it was executed. Over time, additional attributes were added to the `App Paths` key, such as the `UseUrl` value we saw some time ago. When you type the name of a program into the Run dialog (with no path), the `ShellExecute` function checks if the name corresponds to an application registered under `App Paths`. If so, then it uses the registration information to launch the application. Hooray, applications can be run by just typing their name without requiring them to modify the global path. Note that this extra lookup is performed only by the `ShellExecute` family of functions, so if you use `CreateProcess` or `SearchPath`, you'll still get `ERROR_FILE_NOT_FOUND`. Now, the intent was that the registered full path to the application is the same as the registered short name, just with a full path in front. For example, `wordpad.exe` registers the full path of `%ProgramFiles%\Windows NT\Accessories\WORDPAD.EXE`. But there's no check that the two file names match. The Pbrush folks took advantage of this by registering an application path entry for `pbrush.exe` with a full path of `%SystemRoot%\System32\mspaint.exe`: That way, when somebody types `pbrush` into the Run dialog, they get redirected to `mspaint.exe`.

Sneaky.

Raymond Chen

Follow