In the product end game, every change carries significant risk

devblogs.microsoft.com/oldnewthing/20091104-01

November 4, 2009

Raymond Chen

One of the things I mentioned in my talk the other week comparing school with Microsoft is that in school, as the deadline approaches, the work becomes increasingly frantic. On the other hand, in commercial software, as the deadline approaches, the rate of change slows down, because the risk of regression outweighs the benefit of the fix. A colleague of mine offered up this example from Windows 3.1: To fix a bug in GDI, the developers made a very simple fix. It consisted of setting a global flag when a condition was detected and checking the flag in another place in the code and executing a few lines of code if it was set. The change was just a handful of lines, it was very tightly scoped, and it did not affect the behavior of GDI if the flag was not set. They tested the code, it fixed the problem, everything looked good. What could possibly go wrong? A few days after the fix went in, the GDI team started seeing weird crashes that made no sense in code completely unrelated to the places where they made the change. What is going on? After some investigation, they discovered a memory corruption bug. In 16-bit Windows, the local heap came directly after the global variables, and local heap memory was managed in the form of local handles. A common error when working with the local heap was using a local handle as a pointer rather than passing it to the `LocalLock` function to convert the handle to a pointer. The developers found a place where the code forgot to perform this conversion before using a local handle. (In Windows 3.1, most of GDI was written in assembly language, so you didn't have a compiler to do type checking and complain that you're using a handle as a pointer.) Using the handle as a pointer resulted in a global variable being corrupted. Investigation of the code history revealed that this bug had existed in the code since the day it was first written. Why hadn't anybody encountered this bug before? The handle that was being used incorrectly was allocated at boot time, so its value was consistent from run to run. The corruption took the form of writing a zero into memory at the wrong location, and it so happened that the variable that was accidentally being set to zero was not used often, and at the time the corruption occurred, it *happened to have the value zero already*. Adding a new global variable shifted the other global variables around in memory, and now the accidental write of zero hit an important variable whose value was usually *not* zero.

In the product end game, every change carries significant risk. It's often a more prudent decision to live with the bug you understand than to fix it and risk exposing an even worse bug whose existence may not come to light until after you ship.

Raymond Chen

Follow