How did the invalid floating point operand exception get raised when I disabled it?

devblogs.microsoft.com/oldnewthing/20080703-00

July 3, 2008

Raymond Chen

Last time, we learned about the dangers of uninitialized floating point variables but left with a puzzle: Why wasn't this caught during internal testing?

I dropped a hint when I described how `SNaN`s work: You have to ask the processor to raise an exception when it encounters a signaling NaN, and the program disabled that exception. Why was an exception being raised when it had been disabled?

The clue to the cause was that the customer that was encountering the crash reported that it tended to happen after they printed a report. It turns out that the customer's printer driver was re-enabling the *invalid operand* exception in its `DLL_PROCESS_ATTACH` handler. Since the exception was enabled, the `SNaN` exception, which was previously masked, was now live, and it crashed the program.

I've also seen DLLs change the floating point rounding state in their `DLL_PROCESS_ATTACH` handler. This behavior can be traced back to old versions of the C runtime library which reset the floating point state as part of their `DLL_PROCESS_ATTACH`; this behavior was corrected as long ago as 2002 (possibly even earlier; I don't know for sure). Obviously that printer driver was even older. Good luck convincing the vendor to fix a bug in a driver for a printer they most likely don't even manufacture any more. If anything, they'll probably just treat it as incentive for you to buy a new printer.

When you load external code into your process, you implicitly trust that the code won't screw you up. This is just another example of how a DLL can inadvertently screw you up.

Sidebar

One might argue that the `LoadLibrary` function should save the floating point state before loading a library and restore it afterwards. This is an easy suggestion to make in retrospect. Writing software would be so much easier if people would just extend the courtesy of coming up with a comprehensive list of "bugs applications will have that you should protect against"

before you design the platform. That way, when a new class of application bugs is found, and they say "You should've protected against this!", you can point to the list and say, "Nuh, uh, you didn't put it on the list. You had your chance."

As a mental exercise for yourself: Come up with a list of "all the bugs that the `LoadLibrary` function should protect against" and how the `LoadLibrary` function would go about doing it.

Raymond Chen

Follow