The time one of my colleagues debugged a line-of-business application for a package delivery service

devblogs.microsoft.com/oldnewthing/20140718-00

July 18, 2014

Raymond Chen

Back in the days of Windows 95 development, one of my colleagues debugged a line-of-business application for a major delivery service. This was a program that the company gave to its top-tier high-volume customers, so that they could place and track their orders directly. And by *directly*, I mean that the program dialed the modem (since that was how computers communicated with each other back then) to contact the delivery service's mainframe (it was all mainframes back then) and upload the new orders and download the status of existing orders.[1]

Version 1.0 of the application had a notorious bug: Ninety days after you installed the program, it stopped working. They forgot to remove the beta expiration code. I guess that's why they have a version 1.01.

Anyway, the bug that my colleague investigated was that if you entered a particular type of order with a particular set of options in a particular way, then the application crashed your system. Setting up a copy of the application in order to replicate the problem was itself a bit of an ordeal, but that's a whole different story.

Okay, the program is set up, and yup, it crashes exactly as described when run on Windows 95. Actually, it also crashes exactly as described when run on Windows 3.1. This is just plain an application bug.

Here's why it crashed: After the program dials up the mainframe to submit the order, it tries to refresh the list of orders that have yet to be delivered. The code that does this assumes that the list of undelivered orders is the control with focus. But if you ask for labels to be printed, then the printing code changes focus in order to display the "Please place the label on the package exactly like this" dialog, and as a result, the refresh code can't find the undelivered order list and crashes on a null pointer. (I'm totally making this up, by the way. The details of the scenario aren't important to the story.)

Okay, well, that's no big deal. A null pointer fault should just put up the Unrecoverable Application Error dialog box and close the program. Why does this particular null pointer fault crash the entire system?

The developers of the program saw that their refresh code sometimes crashed on a null pointer, and instead of fixing it by actually fixing the code so it could find the list of undelivered orders even if it didn't have focus, or fixing it by adding a null pointer check, they fixed it by adding a null pointer exception handler. (I wish to commend myself for resisting the urge to put the word *fixed* in quotation marks in that last sentence.)

Now, 16-bit Windows didn't have structured exception handling. The only type of exception handler was a global exception handler, and this wasn't just global to the process. This was global to the entire system. Your exception handler was called for every exception everywhere. If you screwed it up, you screwed up the entire system. (I think you can see where this is going.)

The developers of the program converted their global exception handler to a local one by going to every function that had a "We seem to crash on a null pointer and I don't know why" bug and making these changes:

```
extern jmp_buf caught;
extern BOOL trapExceptions;
void scaryFunction(...)
{
 if (setjmp(&caught)) return;
 trapExceptions = TRUE;
 ... body of function ...
 trapExceptions = FALSE;
}
```

Their global exception handler checks the `trapExceptions` global variable, and if it is `TRUE`, they set it back to `FALSE` and do a `longjmp` which sends control back to the start of the function, which detects that something bad must have happened and just returns out of the function.

Yes, things are kind of messed up as a result of this. Yes, there is a memory leak. But at least their application didn't crash.

On the other hand, if the global variable is `FALSE`, because their application crashed in some other function that didn't have this special protection, or because some other totally unrelated application crashed, the global exception handler decided to exit the application by running around freeing all the DLLs and memory associated with their application.

Okay, so far so good, for certain values of *good*.

These system-wide exception handlers had to be written in assembly code because they were dispatched with a very strange calling convention. But the developers of this application didn't write their system-wide exception handler in assembly language. Their application was written in MFC, so they just went to Visual C++ (as it was then known), clicked through some *Add a Windows hook* wizard, and got some generic `HOOKPROC`. (I don't know if Visual C++ actually had an *Add a Windows hook* wizard; they could just have copied the code from somewhere.) Nevermind that these system-wide exception handlers are not `HOOKPROC`s, so the function has the wrong prototype. What's more, the code they used marked the hook function as `__loadds`. This means that the function saves the previous value of the `DS` register on entry, then changes the register to point to the application's data, and on exit, the function restores the previous value of `DS`.

Okay, now we're about to enter the set piece at the end of the movie: Our hero's fear of spiders, his girlfriend's bad ankle from an old soccer injury, the executive toy on the villain's desk, and all the other tiny little clues dropped in the previous ninety minutes come together to form an enormous chain reaction.

The application crashes on a null pointer. The system-wide custom exception handler is called. The crash is not one that is being protected by the global variable, so the custom exception handler frees the application from memory. The system-wide custom exception handler now returns, but wait, what is it returning to?

The crash was in the application, which means that the `DS` register it saved on entry to the custom exception handler points to the application's data. The custom exception handler freed the application's data and then returned, declaring the exception handled. As the function exited, it tried to restore the original `DS` register, but the CPU said, "Nice try, but that is not a valid value for the `DS` register (because you freed it)." The CPU reported this error by (dramatic pause) raising an exception.

That's right, the system-wide custom exception handler crashed with an exception.

Okay, things start snowballing. This is the part of the movie where the director uses quick cuts between different locations, maybe with a little slow motion thrown in.

Since an exception was raised, the custom exception handler is called recursively. Each time through the recursion, the custom exception handler frees all the DLLs and memory associated with the application. But that's okay, right? Because the second and subsequent times, the memory was already freed, so the attempts to free them again will just fail with an invalid parameter error.

But wait, their list of DLLs associated with the application included `USER`, `GDI`, and `KERNEL`. Now, Windows is perfectly capable of unloading dependent DLLs when you unload the main DLL, so when they unloaded their main program, the kernel already decremented

the usage count on `USER`, `GDI`, and `KERNEL` automatically. But they apparently didn't trust Windows to do this, because after all, it was Windows that was causing their application to crash, so they took it upon themselves to free those DLLs manually.

Therefore, each time through the loop, the usage counts for `USER`, `GDI`, and `KERNEL` drop by one. *Zoom in on the countdown clock on the ticking time bomb.*

Beep beep beep beep beep. The reference count finally drops to zero. The window manager, the graphics subsystem, and the kernel itself have all been unloaded from memory. There's nothing left to run the show!

Boom, bluescreen. Hot flaming death.

The punch line to all this is that whenever you call the company's product support line and describe a problem you encountered, their response is always, "Yeah, we're really sorry about that one."

Bonus chatter: What is that *whole different story* mentioned near the top?

Well, when the delivery service sent the latest version of the software to the Windows 95 team, they also provided an account number to use. My colleague used that account number to try to reproduce the problem, and since the problem occurred only after the order was submitted, she would have to submit delivery requests, say for a letter to be picked up from 221B Baker Street and delivered to 62 West Wallaby Street, or maybe for a 100-pound package of radioactive material to be picked up from 1600 Pennsylvania Avenue and delivered to 10 Downing Street.

After about two weeks of this, my colleague got a phone call from Microsoft's shipping department. "What the heck are you doing?"

It turns out that the account number my colleague was given was Microsoft's own corporate account number. As in a *real live account*. She was inadvertently prank-calling the delivery company and sending actual trucks all over the country to pick up nonexistent letters and packages. Microsoft's shipping department and people from the delivery service's headquarters were frantic trying to trace where all the bogus orders were coming from.

[1] Mind you, this sort of thing is the stuff that average Joe customers can do while still in their pajamas, but back in those days, it was a feature that only top-tier customers had access to, because, y'know, mainframe.

Raymond Chen

Follow