Getting MS-DOS games to run on Windows 95: Working around the iretd problem

devblogs.microsoft.com/oldnewthing/20160404-00

April 4, 2016

Raymond Chen

Today's story is the story of *Speed Racer in the Challenge of Racer X*. Here goes. The really scary thing is that *I still remember the details*.

To this day, I can't bear to listen to the Speed Racer theme song because I spent over a week debugging why the program froze up right after the title sequence music. The crashes were completely nonsensical and random.

Windows 95 uses the `iretd` instruction to return from the kernel back to the application. After days of frustrating head-scratching, I eventually discovered that if you use the instruction to return from the kernel back to the application, and the application is running 32-bit protected-mode code on a 16-bit stack, then only the bottom 16 bits of the `esp` register are updated by the `iretd` instruction. The upper 16 bits remain unchanged and continue to hold the value they had while you were in kernel mode. This behavior doesn't appear to be documented anywhere in Intel's reference books.[1]

The effect of this is that 32-bit protected-mode code running on a 16-bit stack will observe that the upper 16 bits of the `esp` register are spontaneously corrupted randomly. (Sound familiar?) Unfortunately, *Speed Racer* was counting on the upper 16 bits of the `esp` register remaining zero.

To fix this, I had to counter insanity with more insanity.

At the last moment before restoring all the general purpose registers and executing the `iretd` instruction, Windows 95 does a check to see whether the troublesome scenario is about to occur. If so, the kernel sets up a temporary stack selector whose base linear address matches the high 16 bits of the kernel `esp` register, then switches to that stack while simultaneously zeroing out the high 16 bits of its own `esp` register. This double-switch rewrites the `ss:esp` value such that it points to the same memory, but shuffles the bits around to arrange for the high 16 bits of `esp` to be zero. In other words, it rewrote `SS:ESP = 00000000 + xxxxyyyy` as `SS:ESP = xxxx0000 + 0000yyyy`. (Sound familiar?)

At this point, the kernel is set up to restore the general purpose registers and perform the `iretd`. This returns control back to the application with the high 16 bits of the `esp` register set to zero, as the application expects.

Now, this may seem like an awful lot of work just to get a single game to work, and it's not like *Speed Racer* was a blockbuster game like *DOOM*. However, this particular problem was not intrinsic to *Speed Racer*. Rather, it was a problem in the client-side library code that came with the MS-DOS extender they were using, and that MS-DOS extender was one of the major players in the MS-DOS extender market, so fixing this issue actually fixed a lot of programs. It's just that *Speed Racer* was the first one discovered to exhibit the problem, so it was the one I ended up debugging.

[1]Maybe I'm missing it. You tell me if you see it in there. The pseudocode at the `RETURN-TO-OUTER-PRIVILEGE-LEVEL` label talks about raising an exception if the stack doesn't have at least 8 bytes of data in it, but it doesn't appear to discuss what happens to the `esp` register. The discussion says "If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack," but it doesn't mention what happens if the destination stack pointer is a different size from the current stack pointer.

Raymond Chen

Follow