Why do Windows functions all begin with a pointless MOV EDI, EDI instruction?

devblogs.microsoft.com/oldnewthing/20110921-00

September 21, 2011

Raymond Chen

If you look at the disassembly of functions inside Windows DLLs, you'll find that they begin with the seemingly pointless instruction `MOV EDI, EDI`. This instruction copies a register to itself and updates no flags; it is completely meaningless. So why is it there?

It's a hot-patch point.

The `MOV EDI, EDI` instruction is a two-byte `NOP`, which is just enough space to patch in a jump instruction so that the function can be updated on the fly. The intention is that the `MOV EDI, EDI` instruction will be replaced with a two-byte `JMP $-5` instruction to redirect control to five bytes of patch space that comes immediately before the start of the function. Five bytes is enough for a full jump instruction, which can send control to the replacement function installed somewhere else in the address space.

Although the five bytes of patch space before the start of the function consists of five one-byte `NOP` instructions, the function entry point uses a single two-byte `NOP`.

Why not use Detours to hot-patch the function, then you don't need any patch space at all.

The problem with Detouring a function during live execution is that you can never be sure that at the moment you are patching in the Detour, another thread isn't in the middle of executing an instruction that overlaps the first five bytes of the function. (And you have to alter the code generation so that no instruction starting at offsets 1 through 4 of the function is ever the target of a jump.) You could work around this by suspending all the threads while you're patching, but that still won't stop somebody from doing a `CreateRemoteThread` after you thought you had successfully suspended all the threads.

Why not just use two `NOP` instructions at the entry point?

Well, because a `NOP` instruction consumes one clock cycle and one pipe, so two of them would consume two clock cycles and two pipes. (The instructions will likely be paired, one in each pipe, so the combined execution will take one clock cycle.) On the other hand, the `MOV EDI, EDI` instruction consumes one clock cycle and one pipe. (In practice, the instruction

will occupy one pipe, leaving the other available to execute another instruction in parallel. You might say that the instruction executes in half a cycle.) However you calculate it, the `MOV EDI, EDI` instruction executes in half the time of two `NOP` instructions.

On the other hand, the five `NOP`s inserted before the start of the function are never executed, so it doesn't matter what you use to pad them. It could've been five garbage bytes for all anybody cares.

But much more important than cycle-counting is that the use of a two-byte `NOP` avoids the Detours problem: If the code had used two single-byte `NOP` instructions, then there is the risk that you will install your patch just as a thread has finished executing the first single-byte `NOP` and is about to begin executing the second single-byte `NOP`, resulting in the thread treating the second half of your `JMP $-5` as the start of a new instruction.

There's a lot of patching machinery going on that most people don't even realize. Maybe at some point, I'll get around to writing about how the operating system manages patches for software that isn't installed yet, so that when you do install the software, the patch is already there, thereby closing the vulnerability window between installing the software and downloading the patches.

Raymond Chen

Follow