Why don’t Windows functions begin with a pointless MOV EDI,EDI instruction on x86-64?

devblogs.microsoft.com/oldnewthing/20221109-00

November 9, 2022

Raymond Chen

Some time ago, we investigated why Windows functions all begin with a pointless MOV EDI,EDI instruction. The answer was that the instruction was used as a two-byte `NOP` which could be hot-patched to a jump instruction, thereby allowing certain types of security fixes to be applied to a running system. (Those which alter data structures or involve cross-process communication would not benefit from this.)

But you may have noticed that on 64-bit Windows, these pointless instructions are gone. Is hot-patching dead?

No, hot-patching is still alive. But on 64-bit Windows, the hot-patch point is implemented differently.

The idea is that we don’t have to insert a pointless two-byte `nop` instruction into every function. If the first instruction of the function is already a two-byte instruction (or bigger), then that instruction can itself serve as the hot-patch point.

The case where the first instruction of a function is two bytes or larger is by far the dominant one. There are only a few one-byte instructions remaining in x86-64. The ones you’re likely to encounter in user-mode compiler-generated code are

`push r`	`leave`	`cwde`	`int 3`
`pop r`	`ret`	`cdq`	`nop`

where `r` is the 64-bit version of one of the eight named (not numbered) registers.

Some of these instructions are not going to appear naturally at the start of a function.

- `leave` doesn’t make sense because it mutates a callee-preserved register.
- `cwde` and `cdq` don’t make sense because they use `rax` as an input register, but that register is undefined on entry to a function.

- `nop` can just be omitted.
- Starting with a `pop` is disallowed by the Win32 ABI. The return address must stay on the stack.

And then some of the instructions can be worked around if they happen to be the start of a function.

- `push` : If the function pushes any registers `r8` or higher, those can be pushed first, since the push of a high-numbered register is a two-byte instruction. Or the instruction could be re-encoded with a redundant REX prefix `0x48` . Alternatively, the compiler could save the register in the home space, which uses a multi-byte `mov [rsp+n], r` instruction.
- `ret` : This happens if the function is empty and returns no value. The compiler can change this to a 3-byte `ret 0` or a 2-byte `repz ret`.

The last remaining instruction is `int 3` , which is generated by the `__debugbreak` intrinsic.

One option is to use the alternate two-byte encoding `cd 03` (`int nn` , with `nn` = 3). However, the code with the `__debugbreak` may be relying on it being a one-byte instruction, because it intends to patch it with a one-byte `nop` , or it intends to handle the breakpoint exception by stepping over the opcode by incrementing the instruction pointer.

Instead, the compiler plays it safe and begins the function with a two-byte `nop` , which is encoded as if it were `xchg ax, ax` , and in fact the Microsoft debugger disassembles it as such.

The pointless `mov edi, edi` instruction is gone. And most of the time, the compiler can juggle things so that you don't even notice that it arranged for the first instruction of a function to be a multi-byte instruction. The only time it fails is if the first thing your function does is `__debugbreak` , in which case the compiler inserts a pointless `xchg ax, ax` instruction, also known as the two-byte `nop` .

Raymond Chen

Follow