How did real-mode Windows patch up return addresses to discarded code segments?

devblogs.microsoft.com/oldnewthing/20120629-00

June 29, 2012

Raymond Chen

Last week, I described how real-mode Windows fixed up jumps to functions that got discarded. But what about return addresses to functions that got discarded?

The naïve solution would be to allocate a special "return address recovery" function for each return address you found, but that idea comes with its own problems: You are patching addresses on the stack because you are trying to free up memory. It would be a bad idea to try to allocate memory while you're trying to free memory! What if in order to satisfy the allocation, you had to discard still *more* memory? You would start moving and patching stacks before they were fully patched from the previous round of memory motion. The stack patcher would get re-entered and see an inconsistent stack because the previous stack patcher didn't get a chance to finish. The result would be rampant memory corruption.

Therefore, you have to preallocate your "return address recovery" functions. But you don't know how many return addresses you're going to need until you walk the stack (at which point it's too late), and you definitely don't want to preallocate the worst-case scenario since each stack can be up to 64KB in size, and can hold up to 16384 return addresses. You'd end up allocating nearly all your available memory just for return address recovery stubs!

How did real-mode Windows solve this problem?

It magically found a way to put ten pounds of flour in a five-pound bag.

For each segment, there was a special "return thunk" that was shared by all return addresses which returned back into that segment. Since there is only one per segment, you can preallocate it as part of the segment bookkeeping data. To patch the return address, the original return address was overwritten by this shared return thunk. Okay, so you have 32 bits of information you need to save (the original return address, which consists of 16 bits for the segment and 16 bits for the offset), and you have a return thunk that captures the segment information. But you still have 16 bits left over. Where do you put the offset?

We saw some time ago that the BP register associated with far calls was incremented before being pushed on the stack. This was necessary so that the stack patcher knew whether to decode the frame as a near frame or a far frame. But that wasn't the only rule associated with far stack frames. On entry to a far function,

- The first thing you do is increment the BP register and push it onto the stack.
- The second thing you do is push the DS register. (DS is the data segment register, which holds a segment containing data the caller function wanted to be able to access.)

Every far call therefore looks like this on the stack:

xxxx+6	IP of return address
xxxx+4	CS of return address
xxxx+2	pointer to next stack frame (bottom bit set)
xxxx+0	DS of caller

The stack patcher overwrote the saved CS:IP with the address of the return thunk. The return thunk describes the segment that got discarded, so the CS is implied by your choice of return thunk, but the stack patcher still needed to save the IP somewhere. So it saved it where the DS used to be.

Wait, you've just traded one problem for another. Sure, you found a place to put the IP, but now you have to find a place to put the DS.

Here comes the magic.

Recall that on the 8086, the combination segment:offset corresponds to the physical address segment×16 + offset. For example, the address 1234:0005 refers to physical byte 0x1234 * 16 + 0x0005 = 0x12345.

Since the segment and offset were both 16-bit values, there were multiple ways to refer to the same physical address. For example, 1000:2345 would also resolve to physical address 0x12345. But there are other ways to refer to the same byte, like the not entirely obvious 0FFF:2355. In fact, there's a whole range of values you can use, starting from 0235:FFF5 and ending at 1234:0005. In general, there are 4096 different ways of referring to the same address.

There's a bit of a problem with very low addresses, though. To access byte 0x00400, for example, you could use 0000:0400 through 0040:0000, but that's as far as you could go, so these very low addresses do not have the full complement of aliases.

Aha, but they do *if you take advantage of wraparound*. Since the 8086 had only 20 address lines, any overflow in the calculations was simply taken mod 0x100000. Therefore, you could also use F041:FFF0 to refer to the address, because 0xF041 × 16 + 0xFFF0 = 0x100400 ≡ 0x00400. Wraparound allowed the full range of 4096 aliases since you could use F041:FFF0 to FFFF:0410, and then 0000:0400 to 0040:0000.

Related reading: *The story of the mysterious WINA20.386 file*.

Okay, back to stack patching.

Once you consider aliasing, you realize that the 32 bits of flour actually had a lot of air in it. By rewriting the address of the return thunk into the form XXXX:000Y, you can see the 12 bits of air, and to stash the 12-bit value N into that air pocket, you would set the segment to XXXX−N and the offset to N×16+Y.

Recall that we were looking for a place to put that saved DS value, which is a 16-bit value, and we have found 12 bits of air in which to save it. We need to find 4 more bits of air somewhere.

The next trick is realizing that DS is not an arbitrary 16-bit value. It's a 16-bit segment value that was obtained from the Windows memory manager. Therefore, if the Windows memory manager imposed a generous artificial limit of 4096 segments, it could convert the DS segment value into an integer segment index.

That index got saved in the upper 12 bits of the offset.

Okay, let's see what happens when the code tries to unwind to the patched return address.

The function whose return address got patched goes through the usual function epilogue. It pops what it thinks is the original DS off the stack, even though that DS has been secretly replaced by the original return address's IP. The epilogue then pops the old BP, decrements it, and returns to the return thunk. The return thunk now knows where the real return address is (it knows which segment it is responsible for, and it can figure out the IP from the incoming DS register). It can also study its own IP to extract the segment index and from that recover the original DS value. Now that it knows what the original code was trying to do, it can reload the segment, restore the registers to their proper values, and jump to the original return address inside the newly-loaded segment.

I continue to be amazed at how real-mode Windows managed to get so much done with so little.

Exercise: The arbitrary limit of 4096 segments was quite generous, seeing as the maximum number of selectors in protected mode was defined by the processor to be 8191. What small change could you make to expand the segment limit in real mode to match that of protected mode?

Raymond Chen

Follow