# How did real-mode Windows fix up jumps to functions that got discarded?

devblogs.microsoft.com/oldnewthing/20120622-00

June 22, 2012

Raymond Chen

In a discussion of how real-mode Windows walked stacks, commenter Matt wonders about fixing jumps in the rest of the code to the discarded functions.

I noted in the original article that "there are multiple parts to the solution" and that stack-walking was just one piece. Today, we'll look at another piece: Inter-segment fixups.

Recall that real-mode Windows ran on an 8086 processor, a simple processor with no memory manager, no CPU privilege levels, and no concept of task switching. Memory management in real-mode Windows was handled manually by the real-mode kernel, and the way it managed memory was by loading code from disk on demand, and discarding code when under memory pressure. (It didn't discard data because it wouldn't know how to regenerate it, and it can't swap it out because there was no swap file.)

There were a few flags you could attach to a segment. Of interest for today's discussion are *movable* (and it was spelled without the "e") and *discardable*. If a segment was not movable (known as *fixed*), then it was loaded into memory and stayed there until the module was unloaded. If a segment was movable, then the memory manager was allowed to move it around when it needed to defragment memory in order to satisfy a large memory allocation. And if a segment was discardable, then it could even be evicted from memory to make room for other stuff.

| Movable | Discardable | Meaning |
|---|---|---|
| No | No | Cannot be moved or discarded |
| No | Yes | (invalid combination) |
| Yes | No | Can be moved in memory |
| Yes | Yes | Can be moved or purged from memory |

I'm going to combine the movable and discardable cases, since the effect is the same for the purpose of today's discussion, the difference being that with discardable memory, you also have the option of throwing the memory out entirely.

First of all, let's get the easy part out of the way. If you had an intra-segment call (calling a function in your own segment), then there was no work that needed to be done. Real-mode Windows always discarded full segments, so if your segment was running code, it was by definition present, and therefore any other code in that segment was also present. The hard part is the inter-segment calls.

As it happens, an old document on the 16-bit Windows executable file format gives you some insight into how things worked, if you sit down and puzzle it out hard enough.

Let's start with the `GetProcAddress` function. When you call `GetProcAddress`, the kernel needs to locate the address of the function inside the target module. The loader consults the *Entry Table* to find the function you're asking for. As you can see, there are three types of entries in the Entry Table. Unused entries (representing ordinals with no associated function), fixed segment entries, and movable segment entries. Obviously, if the match is in an unused entry, the return value is `NULL` because there is no such function. If the match is in a fixed entry, that's pretty easy too: Look up the segment number in the target module's segment list and combine it with the specified offset. Since the segment is fixed, you can just return the raw pointer directly, since the code will never move.

The tricky part is if the function is in a movable segment. If you look at the document, it says that "a moveable segment entry is 6 bytes long and has the following format." It starts with a byte of flags (not important here), a two-byte `INT 3Fh` instruction, a one-byte segment number, and the offset within the segment.

What's the deal with the `INT 3Fh` instruction? It seems rather pointless to specify that a file format requires some `INT 3Fh` instructions scattered here and there. Why not get rid of it to save some space in the file?

If you called `GetProcAddress` and the result was a function in a movable segment, the `GetProcAddress` didn't actually return the address of the target function. It returned the address of the `INT 3Fh` instruction! (Thankfully, the Entry Table is always a fixed segment, so we don't have to worry about the Entry Table itself being discarded.)

(Now you see why the file format includes these strange `INT 3Fh` instructions: The file format was designed to be loaded directly into memory. When the loader loads the entry table, it just slurps it into memory and bingo, it's ready to go, `INT 3Fh` instructions and all!)

Since `GetProcAddress` returned the address of the `INT 3Fh` instruction, calls to imported functions didn't actually go straight to the target. Instead, you called the `INT 3Fh` instruction, and it was the `INT 3Fh` handler which said, "Gosh, somebody is trying to call

code in another segment. Is that segment loaded?" It took the return address of the interrupt and used it to locate the segment number and offset. If the segment in question was already in memory, then the handler jumped straight to the segment at the specified offset. You got the function call you wanted, just in a roundabout way.

If the segment wasn't loaded, then the `INT 3Fh` handler loaded it (which might trigger a round of discarding and consequent stack patching), then jumped to the newly-loaded segment at the specified offset. An even more roundabout function call.

Okay, so that's the case where a function pointer is obtained by calling `GetProcAddress`. But it turns out that a lot of stuff inside the kernel turns into `GetProcAddress` at the end of the day.

Suppose you have some code that calls a function in another segment within the same module. As we saw earlier, fixups are threaded through the code segment, and if you scroll down to the *Per Segment Data* section of that old document, you'll see a description of the way the relocation records are expressed. A call to a function to a segment within the same module requires an `INTERNALREF` fixup, and as you can see in the document, there are two types of `INTERNALREF` fixups, ones which refer to fixed segments and ones which refer to movable segments.

The easy case is a reference to a fixed segment. In that case, the kernel can just look up where it put that segment, add in the offset, and patch that address into the code segment. Since it's a fixed segment, the patch will never have to be revisited.

The hard case is a reference to a movable segment. In that case, you can see that the associated information in the fixup table is the "ordinal number index into [the] Entry Table."

Aha, you now realize that the Entry Table is more than just a list of your exported functions. It's also a list of all the functions in movable segments that are called from other segments. In a sense, these are "secret exports" in your module. (However, you can't get to them by `GetProcAddress` because `GetProcAddress` knows how to keep a secret.)

To fix up a reference to a function in a movable segment, the kernel calls the `SecretGetProcAddress` (not its real name) function, which as we saw before, returns not the actual function pointer but rather a pointer to the magic `INT 3Fh` in the Entry Table. It is that pointer which is patched into your code segment, so that when your code calls what it thinks is a function in another segment, it's really calling the Entry Table, which as we saw before, loads the code in the target segment if necessary before jumping to it.

Matt wrote, "If the kernel wants to discard that procedure, it has to find that jump address in my code, and redirect it to a page fault handler, so that when my process gets to it, it will call the procedure and fault the code back in. How does it find all of the references to that

function across the program, so that it can patch them all up?" Now you know the answer: It finds all of those references because it already had to find them when applying fixups. It doesn't try to find them at discard time; it finds them when it loads your segment. (Exercise: Why doesn't it need to reapply fixups when a segment moves?)

All inter-segment function pointers were really pointers into the Entry Table. You passed a function pointer to be used as a callback? Not really; you really passed a pointer to your own Entry Table. You have an array of function pointers? Not really; you really have an array of pointers into your Entry Table. It wasn't actually hard for the kernel to find all of these because you had to declare them in your fixup table in the first place.

It is my understanding that the `INT 3Fh` trick came from the overlay manager which was included with the Microsoft C compiler. (The Zortech C compiler followed a similar model.)

**Note**: While the above discussion describes how things worked *in principle*, there are in fact a few places where the actual implementation differs from the description above, although not in any way that fundamentally affects the design.

For example, real-mode Windows did a bit of optimization in the `INT 3Fh` stubs. If the target segment was in memory, then it replaced the `INT 3Fh` instruction with a direct `jmp xxxx:yyyy` to the target, effectively precalculating the jump destination when a segment is loaded rather than performing the calculation each time a function in that segment is called.

By an amazing coincidence, the code sequence

```
int 3fh
db  entry_segment
dw  entry_offset
```

is five bytes long, which is the exact length of a `jmp xxxx:yyyy` instruction. Phew, the patch just barely fits!

Raymond Chen

**Follow**