How were DLL functions imported in 16-bit Windows?

devblogs.microsoft.com/oldnewthing/20060717-13

July 17, 2006

Raymond Chen

Last time, we looked at the way functions were exported from 16-bit DLLs. Today, we'll look at how they were imported.

When each segment is loaded into memory, the raw contents are loaded from disk, and then relocation fixups are applied. A fixup for an imported function consists of the name of the target DLL, the target function (either a name or ordinal), and the position of the first location in the segment where the fixup needs to be applied. All imported addresses are far addresses since they reside in another segment. (If they were in the same segment, then they would be in the same DLL, so you wouldn't be importing it!) On 16-bit Windows, a far address is four bytes (a two-byte selector and a two-byte offset), and since the target address is not known when the DLL is generated, those four bytes are just placeholders, waiting to be filled in with the actual target address when the import is resolved. And it is those placeholder bytes that serve double duty.

All the calls within a segment that import the same function are chained in a linked list, where the relocation record points to the first entry. The items in the linked list? The four-byte placeholders. And the "next" pointer in the linked list? The placeholder itself! For example, suppose we have a segment that requires two fixups for the function `GetPrivateProfileInt`, which happens to be kernel function 127. The relocation table entry would say "This segment needs function 127 from KERNEL; start at offset 01D1". The on-disk copy of the segment might go something like this:

…	
01D0	9A
01D1	FE
01D2	01
01D3	00
01D0	00

…	
01FD	9A
01FE	FF
01FF	FF
0200	00
0201	00
…	

To apply the fixup, we first call `GetProcAddress` to get the address of kernel function 127. Then we go to the first fixup location (`0x01D1`), write the address there, then look at the value we overwrote. That value was `0x01FE` , so we now go to offset `0x01FE` and write the address there, too. The value we overwrote was `0xFFFF` , which marks the end of the fixup chain.

But what if the call to `GetProcAddress` fails? (Say, there is no such function 127 in `KERNEL` .) Then instead of writing the address of the target function, the loader wrote the address of a function that displayed the "Call to Undefined Dynalink" fatal error dialog.

Okay, that's a quick introduction to how functions are imported and exported on 16-bit Windows. Next time, we'll look at the transition to 32-bit Windows and the design decisions that went into the new model.

Raymond Chen

Follow