How important is it nowadays to ensure that all my DLLs have non-conflicting base addresses?

devblogs.microsoft.com/oldnewthing/20170120-00

January 20, 2017

Raymond Chen

Back in the day, one of the things you were exhorted to do was rebase your DLLs so that they all had nonoverlapping address ranges, thereby avoiding the cost of runtime relocation. Is this still important nowadays?

This situation is another demonstration of how it is important for good advice to come with a rationale so you can tell when it becomes bad advice.

The rationale for rebasing goes like this: If a DLL is loaded at its preferred base address, then the image can be paged in directly from backing store without requiring any fixups. This means that the pages can be shared between processes, since each process gets an identical copy. (Of course, the sharing stops once somebody writes to the page and makes their copy different from the shared copy.)

If a DLL cannot be loaded at its preferred address, then the image will be relocated, and the entire relocated DLL is now backed by the page file.[1] This is a relative expensive operation, since the DLL has to be read from disk and fixed up, and a commit charge to the page file is incurred in order to ensure that there is space to write the fixed-up pages. Furthermore, if two processes relocate the DLL and happen by some coincidence to relocate them to the same place, Windows NT does not attempt to share the relocated images. There will be multiple copies in the page file.

The cost of this dynamic relocation is what rebasing attempts to avoid. Let's call this the "relocation penalty."

Enter ASLR.

ASLR causes the DLLs to be loaded at pseudo-random addresses. Consequently, a DLL will load at its preferred base address only in the case of an astonishing coincidence.

Okay, so let's go back to the rationale to see if it still applies.

Does a DLL being loaded away from its preferred base address incur a relocation penalty? If you think about it, ASLR means that no DLL ever loads at its preferred address, but we also saw that the kernel makes accommodations for this so that DLLs subjected to ASLR can still share pages, and it does so without forcing the entire DLL to be relocated on initial load. So there is no relocation penalty in the case where the DLL was relocated by ASLR.

But what if the DLL is relocated for some other reason? For example, it could be that the ASLR-chosen base address is not available in the process, because the process already allocated something else at that location. In that case, a traditional relocation must take place, and you pay the relocation penalty.

Ah, but here's the thing: When a DLL is loaded, ASLR will assign a base address randomly from among the available base addresses that are not already being used.[2] So you're not going to get into the "the ASLR-chosen base address is not available" scenario because ASLR chooses the DLL's base address from among the base address that are available.[3]

Okay, so you can still get into a conflict situation, but you have to really work at it. For example, you could load a DLL into one process, and get an ASLR-assigned base address. You then start a second process, intentionally allocate memory at that address (to force the collision), and then load the DLL. In this case, there will be a relocation because you squatted on the place where ASLR wanted to put the DLL. But this is no worse than what you had before ASLR: In the pre-ASLR world, squatting on a DLL's preferred base address would have forced a relocation penalty anyway.

So, let's see what the story is. To rebase or not to rebase?

In the presence of ASLR, rebasing your DLLs has no effect because ASLR is going to *ignore your base address anyway* and relocate the DLL into a location of its pseudo-random choosing.

Mind you, even though rebasing has no effect, it doesn't hurt either.

If you are on a system without ASLR (either because it predates ASLR, or because ASLR has been disabled for whatever reason), then rebasing will help, for the traditional reasons.

Mind you, systems without ASLR are really hard to find nowadays, so rebasing provides no benefit in the overwhelming majority of cases. But in that vanishingly small percentage of cases where you don't have ASLR, then rebasing helps.

Conclusion: It doesn't hurt to rebase, just in case, but understand that the payoff will be extremely rare. Build your DLL with `/DYNAMICBASE` enabled (and with `/HIGHENTROPYVA` for good measure) and let ASLR do the work of ensuring that no base address collision occurs. That will cover pretty much all of the real-world scenarios. If you happen to fall into one of the very rare cases where ASLR is not available, then your program will still work. It just may run a little slower due to the relocation penalty.

[1] More precisely, all the pages that contained fixups are put into the page file. We discussed this finer point last time.

[2] Okay, there's a third case, which is where ASLR has simply run out of base addresses. But again, this is no worse than what you had before ASLR: If you run out of base addresses, then it's every man for himself. Each time a new DLL loads, the kernel has to scrounge around for a large-enough chunk of available address space into which to load the DLL.

[3] As a result, ASLR actually does a better job of avoiding collisions than manual rebasing, since ASLR can view the system as a whole, whereas manual rebasing requires you to know all the DLLs that are loaded into your process, and coordinating base addresses across multiple vendors is generally not possible.

Raymond Chen

Follow