When I enable page heap, why is my one-byte buffer overrun not detected immediately?

devblogs.microsoft.com/oldnewthing/20170410-00

April 10, 2017

Raymond Chen

Page Heap is a mode of the heap manager[1] that can be enabled by the Debugging Tools for Windows. When enabled, each memory allocation is placed at the end of a dedicated page, and the page that follows is left invalid. That way, if you overrun the allocation, you will crash with an access violation because you are reaching into the next (invalid) page.[2] When the memory is freed, the entire page is decommitted, so that any use-after-free bugs will result in an access violation. (Eventually, the heap manager will reuse the address space.)

A customer noted that when they enabled page heap, the crash did not occur on their one-byte buffer overrun. They were able to overrun the buffer by 13 bytes before the crash occurred. "We thought full page heap was supposed to catch buffer overruns immediately."

Page heap places the allocation as close to the end of the page as it can, but it may not able to push it right up to the edge because the heap is contractually obligated to respect the `MEMORY_ALLOCATION_ALIGNMENT`. Without these alignment guarantees, the heap would be much harder to use because every allocation would have to be manually aligned to the desired boundary. (In practice, nearly all data structures have nontrivial alignment requirements, because they will almost always contain at least one member that is larger than a byte.)

On 64-bit systems, the allocation alignment is 16, which means that `HeapAlloc` promises to return a value that is evenly divisibly by 16. This contractual obligation means that if you make an allocation request for 3 bytes on a 64-bit system, then the allocation will be placed at an address of the form `xxxxxxxx`xxxxFFF0`, with three bytes of actual data and 13 padding bytes.

That's where the 13 bytes of slop are coming from. The heap manager cannot give you a pointer of the form `xxxxxxxx`xxxxFFFD`, because that would violate the alignment contract. However, the heap manager does put canary bytes in those extra 13 bytes, and when you free or reallocate the memory block, the heap manager verifies that the canary bytes have not been tampered with. So the write overrun is detected eventually.

If you want to break the alignment contract and make the memory go right up to the edge, you can ask for `/unaligned /full`. Note, however, that handing back unaligned memory is likely to result in other problems, because one of the ground rules of programming is that in the absence of explicit permission to the contrary, pointers must be properly aligned. The consequences of breaking this rule vary depending on how strictly your platform enforces alignment. The code might take alignment faults. Or the code might simply operate on the wrong memory. The x86 architecture is mostly alignment-forgiving, but there are still places (such as interlocked operations and SIMD instructions) where alignment is still important.

[1] I'm using the definite article on "the heap manager" because I'm referring to the system-provided heap manager. The one that you are using when you call functions like `HeapAlloc`. If your program uses a custom heap library, then the page heap settings have no effect on that custom heap library. (This sounds obvious, but sometimes customers expect the page heap settings to somehow be able to alter the behavior of code it didn't write.)

[2] Formally, this model is known as "full page heap". There's also a "standard page heap" which places canary bytes after the end of each allocation. When you free or reallocate the memory, the heap manager checks whether the canary bytes have been tampered with; if so, then it informs you of a heap buffer overrun. I don't know why they call this "standard page heap" because there are no pages involved.

Raymond Chen

Follow