If you're going to write your own allocator, you need to respect the MEMORY_ALLOCATION_ALIGNMENT

devblogs.microsoft.com/oldnewthing/20121115-00

November 15, 2012

Raymond Chen

This time, I'm not going to set up a story. I'm just going to go straight to the punch line.

A customer overrode the `new` operator in order to add additional instrumentation. Something like this:

```
struct EXTRASTUFF
{
    DWORD Awesome1;
    DWORD Awesome2;
};
// error checking elided for expository purposes
void *operator new(size_t n)
{
  EXTRASTUFF *extra = (EXTRASTUFF)malloc(sizeof(EXTRASTUFF) + n);
  extra->Awesome1 = get_awesome_1();
  extra->Awesome2 = get_awesome_2();
  return ((BYTE *)extra) + sizeof(EXTRASTUFF);
}
// use your imagination to implement
// operators new[], delete, and delete[]
```

This worked out okay on 32-bit systems because in 32-bit Windows, `MEMORY_ALLOCATION_ALIGNMENT` is 8, and `sizeof(EXTRASTUFF)` is also 8. If you start with a value that is a multiple of 8, then add 8 to it, the result is still a multiple of 8, so the pointer returned by the custom `operator new` remains properly aligned.

But on 64-bit systems, things went awry. On 64-bit systems, `MEMORY_ALLOCATION_ALIGNMENT` is 16, As a result, the custom `operator new` handed out *guaranteed-misaligned* memory.

The misalignment went undetected for a long time, but the sleeping bug finally woke up when somebody allocated a structure that contained an `SLIST_ENTRY`. As we saw earlier, the `SLIST_ENTRY` really does need to be aligned according to the `MEMORY_ALLOCATION_ALIGNMENT`, especially on 64-bit systems, because 64-bit Windows

takes advantage of the extra “guaranteed to be zero” bits that 16-byte alignment gives you. If your `SLIST_ENTRY` is not 16-byte aligned, then those “guaranteed to be zero” bits are not actually zero, and then the algorithm breaks down.

Result: Memory corruption and eventually a crash.

Raymond Chen

Follow