IsBadXxxPtr should really be called CrashProgramRandomly

devblogs.microsoft.com/oldnewthing/20060927-07

September 27, 2006

Raymond Chen

Often I'll see code that tries to "protect" against invalid pointer parameters. This is usually done by calling a function like `IsBadWritePtr`. But this is a bad idea. `IsBadWritePtr` should really be called `CrashProgramRandomly`.

The documentation for the `IsBadXxxPtr` functions presents the technical reasons why, but I'm going to dig a little deeper. For one thing, if the "bad pointer" points into a guard page, then probing the memory will raise a guard page exception. The `IsBadXxxPtr` function will catch the exception and return "not a valid pointer". But guard page exceptions are raised only once. **You just blew your one chance.** When the code that is managing the guard page accesses the memory for what it thinks is the first time (but is really the second), it won't get the guard page exception but will instead get a normal access violation.

Alternatively, it's possible that your function was called by some code that intentionally passed a pointer to a guard page (or a `PAGE_NOACCESS` page) and was expecting to receive that guard page exception or access violation exception so that it could dynamically generate the data that should go onto that page. (Simulation of large address spaces via pointer-swizzling is one scenario where this can happen.) Swallowing the exception in `IsBadXxxPtr` means that the caller's exception handler doesn't get a chance to run, which means that your code rejected a pointer that would actually have been okay, if only you had let the exception handler do its thing.

"Yeah, but my code doesn't use guard pages or play games with `PAGE_NOACCESS` pages, so I don't care." Well, for one thing, just because your code doesn't use these features pages doesn't mean that no other code in your process uses them. One of the DLLs that you link to might use guard pages, and your use of `IsBadXxxPtr` to test a pointer into a guard page will break that other DLL.

And second, your program does use guard pages; you just don't realize it. The dynamic growth of the stack is performed via guard pages: Just past the last valid page on the stack is a guard page. When the stack grows into the guard page, a guard page exception is raised,

which the default exception handler handles by committing a new stack page and setting the **next** page to be a guard page.

(I suspect this design was chosen in order to avoid having to commit the entire memory necessary for all thread stacks. Since the default thread stack size is a megabyte, this would have meant that a program with ten threads would commit ten megabytes of memory, even though each thread probably uses only 24KB of that commitment. When you have a small pagefile or are running without a pagefile entirely, you don't want to waste 97% of your commit limit on unused stack memory.)

"But what should I do, then, if somebody passes me a bad pointer?"

You should crash.

No, really.

In the Win32 programming model, exceptions are truly exceptional. As a general rule, you shouldn't try to catch them. And even if you decide you want to catch them, you need to be very careful that you catch exactly what you want and no more.

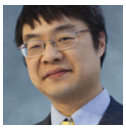
Trying to intercept the invalid pointer and returning an error code creates nondeterministic behavior. Where do invalid pointers come from? Typically they are caused by programming errors. Using memory after freeing it, using uninitialized memory, that sort of thing. Consequently, an invalid pointer might actually point to valid memory, if for example the heap page that used to contain the memory has not been decomitted, or if the uninitialized memory contains a value that when reinterpreted as a pointer just happens to be a pointer to memory that is valid right now. On the other hand, it might point to truly invalid memory. If you use `IsBadWritePtr` to "validate" your pointers before writing to them, then in the case where it happens to point to memory that is valid, you end up corrupting memory (since the pointer is "valid" and you therefore decide to write to it). And in the case where it happens to point to an invalid address, you return an error code. In both cases, **the program keeps on running**, and then that memory corruption manifests itself as an "impossible" crash two hours later.

In other words `IsBadWritePtr` is really `CorruptMemoryIfPossible`. It tries to corrupt memory, but if doing so raises an exception, it merely fails the operation.

Many teams at Microsoft have rediscovered that `IsBadXxxPtr` causes bugs rather than fixes them. It's not fun getting a bucketful of crash dumps and finding that they are all of the "impossible" sort. You hunt through your code in search of this impossible bug. Maybe you find somebody who was using `IsBadXxxPtr` or equivalently an exception handler that swallows access violation exceptions and converts them to error codes. You remove the `IsBadXxxPtr` in order to let the exception escape unhandled and crash the program. Then you run the scenario again. And wow, look, the program crashes **in that function**, and

when you debug it, you find the code that was, say, using a pointer after freeing it. That bug has been there for years, and it was manifesting itself as an “impossible” bug because the function was trying to be helpful by “validating” its pointers, when in fact what it was doing was taking a straightforward problem and turning it into an “impossible” bug.

There is a subtlety to this advice that you should just crash when given invalid input, which I’ll take up next time.



Raymond Chen

Follow