ReadProcessMemory is not a preferred IPC mechanism

devblogs.microsoft.com/oldnewthing/20060117-14

January 17, 2006

Raymond Chen

Occasionally I see someone trying to use the `ReadProcessMemory` function as an inter-process communication mechanism. This is ill-advised for several reasons.

First, you cannot use `ReadProcessMemory` across security contexts, at least not without doing some extra work. If somebody uses “runas” to run your program under a different identity, your two processes will not be able to use `ReadProcessMemory` to transfer data back and forth.

You could go to the extra work to get `ReadProcessMemory` by adjusting the privileges on your process to grant `PROCESS_VM_READ` permission to the owner of the process you are communicating with, but this opens the doors wide open. Any process running with that identity read the data you wanted to share, not just the process you are communicating with. If you are communicating with a process of lower privilege, you just exposed your data to lower-privilege processes other than the one you are interested in.

What’s more, once you grant `PROCESS_VM_READ` permission, you grant it to your **entire** process. Not only can that process read the data you’re trying to share, it can read anything else that is mapped into your address space. It can read all your global variables, it can read your heap, it can read variables out of your stack. It can even corrupt your stack!

What? Granting read access can corrupt your stack?

If a process grows its stack into the stack guard page, the unhandled exception filter catches the guard exception and extends the stack. But when it happen inside a private “catch all exceptions” handler, such as the one that the `IsBadReadPtr` Function uses, it is handled privately and doesn’t reach the unhandled exception filter. As a result, the stack is not grown; a new stack guard page is not created. When the stack normally grows to and then past the point of the prematurely-committed guard page, what would normally be a stack guard exception is now an access violation, resulting in the death of the thread and with it likely the process.

You might think you could catch the stack access violation and try to shut down the thread cleanly, but that is not possible for multiple reasons. First, structured exception handling executes on the stack of the thread that encountered the exception. If that thread has a corrupted stack, it becomes impossible to dispatch that exception since the stack that the exception filters want to run on is no longer viable.

Even if you could somehow run these exception filters on some sort of “emergency stack”, you still can’t fix the problem. At the point of the exception, the thread could be in the middle of anything. Maybe it was inside the heap manager with the heap lock held and with heap data structures in a state of flux. In order for the process to stay alive, the heap data structures need to be made consistent and the heap lock released. But you don’t know how to do that.

There are plenty of other inter-process communication mechanisms available to you. One of them is anonymous shared memory, which I discussed a few years ago. Anonymous shared memory still has the problem that any process running under the same token as the one you are communicating with can read the shared memory block, but at least the scope of the exposure is limited to the data you explicitly wanted to share.

(In a sense, you can’t do any better than that. The process you are communicating with can do anything it wants with the data once it gets it from you. Even if you somehow arranged so that only the destination process can access the memory, there’s nothing stopping that destination process from copying it somewhere outside your shared memory block, at which point your data can be read from the destination process by anybody running with the same token anyway.)

Raymond Chen

Follow