On proper handling of buffers in COM and RPC methods

devblogs.microsoft.com/oldnewthing/20210715-00

July 15, 2021

Raymond Chen

Suppose you have a function method that accepts a sized buffer in the form of a pointer plus a length. How defensive do you need to be about validating the parameters? A customer wanted to know, "How can I validate that caller passed a buffer of the declared size? We can check if the pointer is null, but what about the other cases where the client passes a pointer to one element, but claims that it's a pointer to ten? How do we prevent a buffer overflow?"

There are a few cases here.

If your function does not cross a security boundary, then there is no need to validate. Go ahead and access ten elements if that's what the client told you it has. If you overflow the buffer, you're overflowing the client's buffer, and they get what they deserve. There's no way of knowing the length of the buffer anyway. At the ABI level, all you get is a pointer and a purported length.

If your function crosses a security boundary, then the amount of care you need to provide depends on what level of protection the boundary provides automatically.

In the case of a cross-process COM method (or RPC method), the marshaling infrastructure does most of the work.

If the buffer is an `in` or `in_out` buffer, then the contents of the client buffer are read and transmitted to the server. If the pointer is invalid, the crash occurs here on the client. If the buffer size is incorrect, the overflow occurs on the client, and the marshaling infrastructure either reads beyond the buffer or crashes trying. Either way, it's all happening in the client process. (If the buffer is `out`, then the initial contents of the client buffer are not transmitted to the server.)

On the server side, a corresponding buffer is allocated and initialized either with the data received from the client (for `in` and `in_out` buffers) or with zeroes (for `out` buffers).

The method is then called with the server-side buffer pointer.

When the method returns, the server-side buffer is freed (for `in` buffers) or transferred to the client (for `out` and `in_out` buffers). If transferred to the client side, then the client copies the results back to the original client buffer.

If the caller provides an invalid pointer or an incorrect size for an `out` or `in_out` buffer, the buffer overflow or crash occurs in the client process when the results are copied back. No security boundary is crossed. The crash or corruption occurs at the same security level that passed the invalid parameter.

Zero-initializing the server-side `out` buffer means that if the method writes to only part of the buffer, the unwritten part still contains zeroes rather than uninitialized data from the server's heap, avoiding an information disclosure vulnerability.[1]

Either way, your COM or RPC method can just write to the buffer for the desired length, and the right thing will happen, provided you've annotated your buffers properly.

[1] Of course, if your method copies data into the buffer *from* uninitialized server data, then you still have the information disclosure vulnerability, but that was an error created explicitly by you. The marshaling infrastructure does what it can, but it can't prevent you from driving through the stop sign.

Also, your method should still make sure to initialize any data it reports as having been initialized, even if really you did want to initialize it with zero. If the call is made from within the same process, then no security boundary is crossed, and it is operating directly on the caller-provided buffer. If you say that you wrote 10 bytes to the buffer without actually doing so, then cross-process callers will get zeroes, but in-process callers will get uninitialized garbage. COM and RPC are protecting against security issues, but correctness is still up to you.

Raymond Chen

Follow