When can you free the memory backing the HSTRING you created with WindowsCreateStringReference?

devblogs.microsoft.com/oldnewthing/20160928-00

September 28, 2016

Raymond Chen

A little while back, I posted my complete guide to HSTRING semantics (a rip-off of Eric's complete guide to BSTR semantics). A discussion of security descriptor lifetime somehow triggered the question "When can you free the memory backing the `HSTRING` you created with `WindowsCreateStringReference`?"

You can free the memory backing the `HSTRING` after you destroy the `HSTRING`, and since this is a fast-pass string, you destroy the `HSTRING` by simply abandoning it. Therefore, you can free the memory when you know that nobody should have a copy of the fast-pass `HSTRING` handle any more.

(For the purpose of terminology, I'm going to say that you have a "copy" of an `HSTRING` handle if you merely copied the `HSTRING` handle. E.g., `HSTRING copy = hstr;` On the other hand, I'm going to say that you have a "duplicate" of an `HSTRING` if you passed it to `WindowsDuplicateString`.)

Okay, so how do you know that nobody has a copy of the fast-pass `HSTRING` handle any more?

Recall the rules for `HSTRING`s: If a function is passed an `HSTRING` and it wants to save the `HSTRING` for future use, it must use `WindowsDuplicateString` to increment the reference count on the string (and possibly convert it from fast-pass to standard). Therefore, if you pass the `HSTRING` to another function, you know that there are no copies of that `HSTRING` handle when the function returns, because creating a copy is not allowed. The only place where a literal copy of the `HSTRING` handle is allowed is in the function that created it, and therefore you know when there are no more copies of the `HSTRING` handle because all of the copies belong to you.

The question sort of acknowledges this rule, but notes, "All it takes is one bug somewhere in all of WinRT where someone forgets to duplicate a input string if they need said string later after the function has returned."

That's true. But it's true of C-style string pointers, too! If you pass a C-style string to another function, and that other function wants to retain the string, it's going to need to call `strdup` or some other string duplication function so it can have its own private copy of the string. The value received as a function parameter is not valid once the function returns; if you need to use it after the function returns, you need to duplicate the string.

Similarly, if you receive a COM interface pointer, and you want to continue using it after the function returns, you need to call `IUnknown::AddRef` to increase the reference count on the interface, corresponding to the copy of the pointer you retained. When you're done with the pointer, you call `IUnknown::Release`.

In both of these cases, you are relying on people writing code to respect these rules. All it takes is one bug somewhere in all of C where someone forgets to duplicate a input string if they need said string later after the function has returned.

Somehow, we've managed to survive working with C-style strings and with COM interface pointers with these rules. Maybe it's with the help of things like smart pointers, or maybe it's just through good discipline. Whatever the reason, keep up the good work.

Bonus chatter: One of the rules for fast-pass strings is that you cannot change the contents of the string as long as the `HSTRING` is still in use. One commenter interpreted this to mean that string references aren't thread-safe. Not true. Rather, the statement is a direct reflection of the fact that an `HSTRING` is immutable. If you changed the contents of the buffer that backs the `HSTRING`, then you break the immutability rule. Thread safety is not at issue here. You can use a fast-pass string from any thread you like, as long as you stop using it before your function returns. (This means that your function cannot return until the other thread has definitely finished with the fast-pass string. In practice, this is not commonly done; instead, the function uses `WindowsDuplicateString` to create a standard `HSTRING` and passes that `HSTRING` to the other thread, which can then `WindowsDeleteString` the `HSTRING` when it is done.)

Raymond Chen

Follow