How do I free the pointers returned by functions like GetTokenInformation?

devblogs.microsoft.com/oldnewthing/20230517-00

May 17, 2023

Raymond Chen

There are functions like `GetTokenInformation` which return structures that in turn contain pointers. How are you supposed to free the pointers?

There are two common patterns for returning complex structures.

One pattern is used by functions that return allocated memory which you have to free with a dedicated memory-freeing function. For example, the networking management functions in the `lm.h` family all follow the dedicated memory-freeing pattern: When you call the function, the implementation allocates memory and returns a pointer to the caller. The caller can use the pointed-to data, as well as anything that is in turn pointed to. To free the memory, call the dedicated memory-freeing function with the original pointer that was returned by the function.

This mechanism works because the embedded pointers just point to other parts of the same allocation. That's why a single allocation will free everything.

Other functions that follow the "dedicated memory-free function" pattern are the Lsa functions like `LsaQueryInformationPolicy` and `LsaQueryTrustedDomainInfo`.

Another pattern is used by functions that fill an existing buffer. Examples of these functions are the registry functions, `GetProcessInformation` `GetThreadInformation` `GetTokenInformation`, and `QueryInformationJobObject`. (Somebody in the job object world decided not to follow the existing pattern, I guess.)

In the second pattern, the caller allocates the memory, and the function fills it in. If the caller's buffer is not large enough, than an error is returned like `ERROR_INSUFFICIENT_BUFFER`, and the caller is expected to allocate a bigger buffer and try again.

On success, the buffer contains the desired information, and pointers in the returned buffer point to other parts of the caller's buffer. It's basically, the same as the previous pattern, except that the caller is responsible for allocating the buffer by whatever means they desire.

And since the caller allocated the buffer by whatever means they desire, they also are responsible for freeing the buffer by whatever corresponding means they desire.

The second pattern is commonly used when the caller is in user mode and the implementation is in kernel mode. Kernel mode cannot allocate heap memory in user mode (since the heap is entirely a user-mode concept), so it requires the caller to provide the memory.

You can now answer this customer's question:

> When I call `GetTokenInformation` to obtain the `TOKEN_GROUPS_AND_PRIVILEGES`, what is the lifetime of the array of `SID_AND_ATTRIBUTES` structures returned in the `Sids` member, and what is the lifetime of the `SID`s that are pointed by each `Sids[n].Sid` pointer? Are they valid for the lifetime of the token? (Like, is it a pointer back into the token's internal data structures?)

The various pointers reachable from the `TOKEN_GROUPS_AND_PRIVILEGES` structure are all pointers into various parts of the memory buffer you passed to `GetTokenInformation`. That's why you have to call the function twice: The first time gets the required size of the buffer, which will be the size of the `TOKEN_GROUPS_AND_PRIVILEGES`, plus the size of the `SID_AND_ATTRIBUTES` arrays, plus the sizes of all the `SID`, and so on. After you allocate the required memory, the second call fills it in.

Therefore, the memory will remain valid until you free that buffer. If you free the buffer right away, then those pointers become invalid right away. Closing the token handle has no effect on the buffer.

The pointers in the `TOKEN_GROUPS_AND_PRIVILEGES` certainly cannot point into the token's internal data structures, because the token is a kernel object, and therefore even if you had a pointer directly into that internal data, you couldn't access it because kernel pointers are not usable from user mode.

(And naturally the internal data has to be kept in kernel mode: If it were kept in user mode, then the application could modify it and spoof the token!)