The list of heaps returned by GetProcessHeaps is valid when it returns, but who knows what happens later

devblogs.microsoft.com/oldnewthing/20110701-00

July 1, 2011

Raymond Chen

A customer had a problem involving heap corruption.

> In our code, we call `GetProcessHeaps` and then for each heap, we call `HeapSetInformation` to enable the low fragmentation heap. However, the application crashes due to an invalid heap handle.
>
> ```
> HANDLE heaps[1025];
> DWORD nHeaps = GetProcessHeaps(heaps, 1024);
> for (DWORD i = 0; i < nHeaps; i++) {
>  ULONG HeapFragValue = HEAP_LFH;
>  HeapSetInformation(heaps[i], HeapCompatibilityInformation,
>                     &HeapFragValue, sizeof(HeapFragValue));
> }
> ```
>
> My question is, why do we need to allocate an array of size 1025 even though we pass 1024 to `GetProcessHeaps`?

Ha, faked you out with that question, eh? (It sure faked *me* out.)

It's not clear why the code under-reports the buffer size to `GetProcessHeaps`. So let's ignore the customer's stated question and move on to the more obvious question: Why does this code crash due to an invalid heap handle?

Well, for one thing, the code mishandles the case where there are more than 1024 heaps in the process. But as it happens, the value returned by `GetProcessHeaps` was well below 1024, so that wasn't the reason for the crash.

Unlike kernel objects, heaps are just chunks of user-mode-managed memory. A heap handle is not reference-counted. (Think about it: If it were, how would you release the reference count? There is no `HeapCloseHandle` function.) When you destroy a heap, all existing handles to that heap become invalid.

The consequence of this is that there is a race condition inherent in the use of the `GetProcessHeaps` function: Even though the list of heaps is correct when it is returned to you, another thread might sneak in and destroy one of those heaps out from under you.

This didn't explain the reported crash, however. "We execute this code during application startup, before we create any worker threads, so there should be no race condition."

While it may be true that the code is executed before the program calls `CreateThread`, a study of the crash dump reveals that some sneaky DLLs had paid a visit to the process and had even unloaded themselves!

```
0:001> lm
start    end        module name
75b10000 75be8000   kernel32   (deferred)
77040000 7715e000   ntdll      (deferred)
...
Unloaded modules:
775e0000 775e6000   NSI.dll
76080000 760ad000   WS2_32.dll
71380000 713a2000   COEC23~1.DLL
```

"Well, that explains how a heap could have been destroyed from behind our back. That `COEC23~1.DLL` probably created a private heap and destroyed it when it was unloaded. But how did that DLL get injected into our process in the first place?"

Given the presence of some networking DLLs, the customer guessed that `COEC23~1.DLL` was injected by network firewall security software, but given that these were Windows Error Reporting crash dumps, there was no way to get information from the user's machine about how that `COEC23~1.DLL` ended up loaded in the process, and then spontaneously unloaded.

Even though we weren't able to find the root cause, we were still able to make some suggestions to avoid the crash.

Instead of trying to convert every heap to a low fragmentation heap, just convert the process heap. The process heap remains valid for the lifetime of the process, so you won't see it destroyed out from under you. (Or if you do, then you have bigger problems than a crash in `HeapSetInformation`.)

In fact, you can remove the code entirely when running on Windows Vista or higher, because all heaps default to the low fragmentation heap starting in Windows Vista.

Running around and changing settings on heaps you didn't create is not a good idea. Somebody else owns that heap; who knows what they're going to do with it?

Okay, so if `GetProcessHeaps` is so fragile, why does it even exist?

Well, it's not really intended for general use. It exists primarily for diagnostics. You might be chasing down a memory corruption bug, so you sprinkle into your code some calls to a helper function that calls `GetProcessHeaps` to get all the heaps and then calls `HeapValidate` on each one to check for corruption. Or maybe you're chasing down a memory leak in a particular scenario, so you have a function which calls `GetProcessHeaps` and `HeapWalk` once before entering the scenario, and then again after the scenario completes, and then compares the results looking for leaks. In both cases, you're using the facility for debugging and diagnostic purposes. If there's a race condition that destroys a heap while you're studying it, you'll just throw away the results of that run and try again.

Bonus chatter: While writing up this story, I went back and did some more Web searching for that mysterious `COEC23~1.DLL`. (Tracking it down is hard because all you really know about the file name is that it begins with "CO"; the rest is a hashed short file name.) And I found it: It's not an antivirus program. It's one of those "desktop enhancement" programs that injects itself into every process with the assistance of `AppInit_DLLs`, or as I prefer to call it `Deadlock_Or_Crash_Randomly_DLLs`. (You may have noticed that I anonymized the program as "CO", short for *Contoso*, a fictional company used throughout Microsoft literature.)

Raymond Chen

Follow