Why do I get ERROR_INVALID_HANDLE from GetModuleFileNameEx when I know the process handle is valid?

devblogs.microsoft.com/oldnewthing/20150716-00

July 16, 2015

Raymond Chen

Consider the following program:

```
#define UNICODE
#define _UNICODE
#include <windows.h>
#include <psapi.h>
#include <stdio.h> // horrors! mixing C and C++!

int __cdecl wmain(int, wchar_t **)
{
 STARTUPINFO si = { sizeof(si) };
 PROCESS_INFORMATION pi;
 wchar_t szBuf[MAX_PATH] = L"C:\\Windows\\System32\\notepad.exe";

 if (CreateProcess(szBuf, szBuf, NULL, NULL, FALSE,
                   CREATE_SUSPENDED,
                   NULL, NULL, &si, &pi)) {
  if (GetModuleFileNameEx(pi.hProcess, NULL, szBuf, ARRAYSIZE(szBuf))) {
   wprintf(L"Executable is %ls\n", szBuf);
  } else {
   wprintf(L"Failed to get module file name: %d\n", GetLastError());
  }
  TerminateProcess(pi.hProcess, 0);
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);
 } else {
  wprintf(L"Failed to create process: %d\n", GetLastError());
 }

 return 0;
}
```

This program prints

```
Failed to get module file name: 6
```

and error 6 is `ERROR_INVALID_HANDLE`. "How can the process handle be invalid? I just created the process!"

Oh, the process handle is valid. The handle that isn't valid is the `NULL`.

"But the documentation says that `NULL` is a valid value for the second parameter. It retrieves the path to the executable."

In Windows, processes are initialized in-process. (In other words, processes are self-initializing.) The `CreateProcess` function creates a process object, sets the initial state of that object, copies some information into the address space of the new process (like the command line parameters), and sets the instruction pointer to the process startup code inside `ntdll.dll`. From there, the startup code in `ntdll.dll` pulls the process up by its bootstraps. It creates the default heap. It loads the primary executable and the associated bookkeeping that says "Here is the module information for the primary executable, in case anybody asks." It identifies all the DLLs referenced by the primary executable, the DLLs referenced by those DLLs, and so on. It loads each of the DLLs in turn, creating the module information that says "Here is another module that this process loaded, in case anybody asks," and then it initializes the DLLs in the proper order. Once all the process bootstrapping is complete, `ntdll.dll` calls the executable entry point, and the program takes control.

An interesting take-away from this is that modules are a user-mode concept. Kernel mode does not know about modules. All kernel mode sees is that somebody in user mode asked to map sections of a file into memory.

Okay, so if the process is responsible for managing its modules, how do functions like `Get-ModuleFileNameEx` work? They issue a bunch of `ReadProcessMemory` calls and manually parse the in-memory data structures *of another process*. Normally, this would be considered "undocumented reliance on internal data structures that can change at any time," and in fact those data structures do change quite often. But it's okay because the people who maintain the module loader (and therefore would be the ones who change the data structures) are also the people who maintain `GetModuleFileNameEx` (so they know to update the parser to match the new data structures).

With this background information, let's go back to the original question. Why is `GetModule-FileNameEx` failing with `ERROR_INVALID_HANDLE`?

Observe that the process was created suspended. This means that the process object has been created, the initialization parameters have been injected into the new process's address space, but no code in the process has run yet. In particular, the startup code inside `ntdll.dll` hasn't run. This means that the code to add a module information entry for the main executable *hasn't run*.

Now we can connect the dots. Since the module information entry for the main executable hasn't been added to the module table, the call to `GetModuleFileNameEx` is going to try to parse the module table from the suspended Notepad process, and it will see that the table is empty. Actually, it's worse than that. The module table *hasn't been created yet*. The function then reports, "There is no module table entry for `NULL`," and it tells you that the handle `NULL` is invalid.

Functions like `GetModuleFileNameEx` and `CreateToolhelp32Snapshot` are designed for diagnostic or debugging tools. There are naturally race conditions involved, because the process you are inspecting is certainly free to load or unload a module immediately after the call returns, at which point your information may be out of date. What's worse, the process you are inspecting may be in the middle of updating its module table, in which case the call may simply fail with a strange error like `ERROR_PARTIAL_COPY`. (Protecting the data structures with a critical section isn't good enough because critical sections do not cross processes, and the process doing the inspecting is going to be using `ReadProcessMemory`, which doesn't care about critical sections.)

In the particular example above, the code could avoid the problem by using the `QueryFull-ProcessImageName` function to get the path to the executable.

Bonus chatter: The `CreateToolhelp32Snapshot` function extracts the information in a different way from `GetModuleFileNameEx`. Rather than trying to parse the information via `ReadProcessMemory`, it injects a thread into the target process and runs code to extract the information from within the process, and then marshals the results back. I'm not sure whether this is more crazy than using `ReadProcessMemory` or less crazy.

Second bonus chatter: A colleague of mine chose to describe this situation more directly. "Let's cut to the heart of the matter. These APIs don't really work by the normally-accepted definitions of 'work'." These snooping-around functions are best-effort, so use them in situations where best-effort is better than nothing. For example, if you have a diagnostic tool, you're probably happy that it gets information at all, even if it may sometimes be incomplete. (Debuggers don't use any of these APIs. Debuggers receive special events to notify them of modules as they are loaded and unloaded, and those notifications are generated by the loader itself, so they are reliable.)

Exercise: Diagnose this customer's problem: "If we launch a process suspended, the `Get-ModuleInformation` function fails with `ERROR_INVALID_HANDLE`."

```
#include <windows.h>
#include <psapi.h>
#include <iostream>

int __cdecl wmain(int, wchar_t **)
{
 STARTUPINFO si = { sizeof(si) };
 PROCESS_INFORMATION pi;
 wchar_t szBuf[MAX_PATH] = L"C:\\Windows\\System32\\notepad.exe";

 if (CreateProcess(szBuf, szBuf, NULL, NULL, FALSE,
                   CREATE_SUSPENDED,
                   NULL, NULL, &si, &pi)) {
  DWORD addr;
  std::cin >> std::hex >> addr;
  MODULEINFO mi;
  if (GetModuleInformation(pi.hProcess, (HINSTANCE)addr,
                           &mi, sizeof(mi))) {
   wprintf(L"Got the module information\n");
  } else {
   wprintf(L"Failed to get module information: %d\n", GetLastError());
  }
  TerminateProcess(hProcess, 0);
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);
 } else {
  wprintf(L"Failed to create process: %d\n", GetLastError());
 }

 return 0;
}
```

Run Process Explorer, then run this program. When the program asks for an address, enter the address that Process Explorer reports for the base address of the module.

Raymond Chen

Follow