# When you think you found a problem with a function, make sure you're actually calling the function, episode 2

devblogs.microsoft.com/oldnewthing/20150722-00

July 22, 2015

Raymond Chen

A customer reported that the `DuplicateHandle` function was failing with `ERROR_INVALID_HANDLE` even though the handle being passed to it seemed legitimate:

```
  // Create the handle here
  m_Event =
    ::CreateEvent(NULL, FALSE/*bManualReset*/,
                        FALSE/*bInitialState*/, NULL/*lpName*/));
  ... error checking removed ...


// Duplicate it here
HRESULT MyClass::CopyTheHandle(HANDLE *pEvent)
{
 HRESULT hr = S_OK;

 if (m_Event != NULL) {
  BOOL result = ::DuplicateHandle(
                GetCurrentProcess(),
                m_Event,
                GetCurrentProcess(),
                pEvent,
                0,
                FALSE,
                DUPLICATE_SAME_ACCESS
                );
  if (!result) {
    // always fails with ERROR_INVALID_HANDLE
    return HRESULT_FROM_WIN32(GetLastError());
  }
 } else {
  *pEvent = NULL;
 }

 return hr;
}
```

The handle in `m_Event` appears to be valid. It is non-null, and we can still set and reset it. But we can't duplicate it.

Now, before claiming that a function doesn't work, you should check what you're passing to it and what it returns. The customer checked the `m_Event` parameter, but what about the other parameters? The function takes *three* handle parameters, after all, and they checked only one of them. According to the debugger, `DuplicateHandle` was called with the parameters

| | | |
|---|---|---|
| `hSourceProcessHandle` | `= 0x0aa15b80` | |
| `hSourceHandle` | `= 0x00000ed8` | ← `m_Event`, appears to be valid |
| `hTargetProcessHandle` | `= 0x0aa15b80` | |
| `lpTargetHandle` | `= 0x00b0d914` | |
| `dwDesiredAccess` | `= 0x00000000` | |
| `bInheritHandle` | `= 0x00000000` | |
| `dwOptions` | `= 0x00000002` | |

Upon sharing this information, the customer immediately saw the problem: The other two handle parameters come from the `GetCurrentProcess` function, and that function was returning `0x0aa15b80` rather than the expected pseudo-handle (which is currently `-1`, but that is not contractual).

The customer explained that their `MyClass` has a method with the name `GetCurrent-Process`, and it was that method which was being called rather than the Win32 function `GetCurrentProcess`. They left off the leading `::` and ended up calling the wrong `Get-CurrentProcess`.

By default, Visual Studio colors member functions and global functions the same, but you can change this in the *Fonts and Colors* options dialog. Under *Show settings for*, select *Text Editor*, and then under *Display items* you can customize the colors to use for various language elements. In particular, you can choose a special color for static and instance member functions.

Or, as a matter of style, you could have a policy of not giving member functions the same name as global functions. (This has the bonus benefit of reducing false positives when grepping.)

**Bonus story**: A different customer reported a problem with visual styles in the common tab control. After a few rounds of asking questions, coming up with theories, testing the theories, disproving the theories, the customer wrote back: "We figured out what was happening when

we tried to step into the call to `CreateDialogIndirectParamW`. Someone else in our code base redefined all the dialog creation functions in an attempt to enforce a standard font on all of them, but in doing so, they effectively made our code no longer isolation aware, because in the overriding routines, they called `CreateDialogIndirectParamW` instead of `Isolation-AawreCreateDialogIndirectParamW`. Thanks for all the help, and apologies for the false alarm."

Raymond Chen

**Follow**