Who is this rogue operative that filled my object with 0xDD, then sent me a message?

devblogs.microsoft.com/oldnewthing/20150212-00

February 12, 2015

Raymond Chen

A failure occurred during stress testing, and the component team came to the conclusion that their component was actually the victim of memory corruption and they asked for help trying to see if there was anything still in memory that would give a clue who did the corrupting.

```
/* static */ LRESULT CALLBACK CContoso::WndProc(
    HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    CContoso *pThis = reinterpret_cast<CContoso *>
                          (GetWindowLongPtr(hwnd, GWLP_USERDATA));
    ...
    pThis->... // crash on first dereference of pThis
```

According to the debugger, the value of `pThis` is a valid pointer to memory that is complete nonsense.

```
0: kd> dv
           hwnd = 0xf0040162
           uMsg = 0x219
           ...
          pThis = 0x10938bf0
           ...
0: kd> dt pThis
Type CContoso*
   +0x000 __VFN_table : 0xdddddddd
   +0x004 m_cRef           : 0n-572662307
   +0x008 m_hwnd           : 0xdddddddd HWND__
   ...
```

The `CContoso` object was filled with the byte `0xDD`. Who would do such a thing?

There are a few clues so far, and if you're psychic, you may have picked up on their aura.

But I had a suspicion what happened, so I dug straight into the code to check my theory.

```
BOOL CContoso::StartStuffInBackground()
{
 AddRef(); // DoBackgroundWork will release the reference
 BOOL fSuccess = QueueUserWorkItem(&CContoso::DoBackgroundWork, this, 0);
 if (!fSuccess) Release();
 return fSuccess;
}

/* static */ DWORD CALLBACK CContoso::DoBackgroundWork(void *lpParameter)
{
 CContoso *pThis = static_cast<CContoso *>(lpParameter);
 pThis->DoThis();
 pThis->DoThat();
 pThis->Release();
 return 0;
}
```

So far, we have a standard pattern. An extra reference to the object is kept alive as long as the background thread is still running. This prevents the object from being destroyed prematurely.

(Note that this object is not necessarily a COM object. It could be a plain object that happens to have chosen the names `AddRef` and `Release` for the methods that manipulate the reference count.)

What people often forget to consider is that this means that the final release of the `CContoso` object can occur *on the background thread*. I mean, this is obvious in one sense, because they are adding the extra reference specifically to handle the case where we want to delay object destruction until the background thread completes. But what happens if that scenario actually comes to pass?

```
CContoso::~CContoso()
{
 if (m_hwnd != nullptr) DestroyWindow(m_hwnd);
 ...
}
```

As part of the destruction of the `CContoso` object, it destroys its window. But `DestroyWindow` must be called on the same thread which created the window: "A thread cannot use **DestroyWindow** to destroy a window created by a different thread."

This means that if the final release of the `CContoso` object comes from the background thread, the destructor will run on the background thread, and the destructor will try to destroy the window, but the call will fail because it is on the wrong thread.

The result is that the object is destroyed, but the window still hangs around, and the window has a (now dangling) pointer to the object that no longer exists.

Since the window in question was a hidden helper window, the program managed to survive like this for quite some time: Since the program thought the window was destroyed, there was no code that tried to send it a message, and the normal system-generated messages were not anything the object cared about, so they all fell through to `DefWindowProc` and nobody got hurt. But eventually, some other stress test running on the machine happened coincidentally to broadcast the `WM_SETTINGCHANGE` message `0x0219`, and when the object tried to check what settings changed, that's when it crashed. (That was one of the clues I hinted at above: The message that triggered the crash is `0x0219`. This is a good number to memorize if you spend time studying stress failures because it is often the trigger for crashes like this where a window has been orphaned by its underlying object.)

The root cause is that the object was treated as a free-threaded object even though it actually had thread affinity.

One way to fix this is to isolate the parts with thread affinity so that they are used only on the UI thread. The one we identified is the destructor due to its use of `DestroyWindow`. So at a minimum, we could marshal destruction to the UI thread.

```
LONG CContoso::Release()
{
 LONG cRef = InterlockedDecrement(&this->m_cRef);
 if (cRef == 0)
 {
  if (m_hwnd == nullptr) {
    delete this;
  } else {
    PostMessage(m_hwnd, CWM_DESTROYTHIS, 0, 0);
  }
 }
 return cRef;
}

/* static */ LRESULT CALLBACK CContoso::WndProc(
    HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    CContoso *pThis = reinterpret_cast<CContoso *>
                          (GetWindowLongPtr(hwnd, GWLP_USERDATA));
    ...

    case CWM_DESTROYTHIS:
     delete pThis;
     return 0;
    ...
```

(The original code had better have been using an interlocked operation on `Release` because it was releasing from a background thread already.)

If the final `Release` happens before we have a window, then we just destruct in-place, on the theory that if no window is created, then we are being destroyed due to failed initialization and are still on the original thread. Otherwise, we post a message to the window to ask it to destroy the object.

Note that this design does have its own caveats:

- Even if the final `Release` happens on the UI thread, we still post a message, even though we could have destroyed it inline.
- Posting a message assumes that the message pump will continue to run after the object is released. If somebody releases the object and then immediately exits the thread, the posted message will never arrive and the object will be leaked.
- Posting a message makes destruction asynchronous. There may be some assumptions that destruction is synchronous with the final release.

As for the first problem, we could do a thread check and destruct in-place if we are on the UI thread. This would most likely solve the second problem because the exiting thread is not the one that will process the message. It will still be a problem if the background thread does something like

```
Release();
DoSomethingThatCausesTheUIThreadToExitImmediately();
```

For the second problem, we could change the `PostMessage` to a `SendMessage`, but this creates its own problems because of the risk of deadlock. If the UI thread is blocked waiting for the background thread, and the background thread tries to send the UI thread a message, the two threads end up waiting for each other and nobody makes any progress. On the other hand, making the destruction synchronous would fix the third problem.

Another approach is to push the affinity out one more step:

```
/* static */ DWORD CALLBACK CContoso::DoBackgroundWork(void *lpParameter)
{
 CContoso *pThis = static_cast<CContoso *>(lpParameter);
 pThis->DoThis();
 pThis->DoThat();
 pThis->AsyncRelease();
 return 0;
}

void CContoso::AsyncRelease()
{
 PostMessage(m_hwnd, CWM_RELEASE, 0, 0);
}

/* static */ LRESULT CALLBACK CContoso::WndProc(
    HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    CContoso *pThis = reinterpret_cast<CContoso *>
                          (GetWindowLongPtr(hwnd, GWLP_USERDATA));
    ...
    case CWM_RELEASE:
     pThis->Release();
     return 0;

    ...
```

In this design, we make the asynchronicity explicit in the name of the function, and we require all background threads to use the asynchronous version. This design again assumes that the only reason the window wouldn't exist is that something went wrong during initialization before any background tasks were created.

Unfortunately, this design also retains the original constraint that `Release` can be called only from the UI thread. That makes the object rather fragile, because it is not obvious that `Release` has such constraints. If you go this route, you probably should rename `Release` to `ReleaseFromUIThread`.

If this object is a COM object, then another option is to use COM marshaling to marshal the `IUnknown` to the background thread and use `IUnknown::Release` to release the object. Since you used COM to marshal the object, it knows that `CoUninitialize` should wait for all outstanding references marshaled to other threads, thereby avoiding the "lost message" problem.

Anyway, those are a few ideas for addressing this problem. None of them are particularly beautiful, though. Maybe you can come up with something better.

(The component team fixed this problem by taking advantage of a detail in the usage pattern of the `CContoso` object: The client of the `CContoso` object is expected to call `CContoso::Stop` before destroying the object, and after calling `CContoso::Stop`, the

only valid thing you can do with the object is destroy it. Furthermore, that call to `CContoso::Stop` must occur on the UI thread. Therefore, they moved the part of the cleanup code that must run on the UI thread into the `Stop` method. The object's background tasks already knew to abandon work once they detected that the object had been stopped.)

Raymond Chen

Follow