Your program assumes that COM output pointers are initialized on failure; you just don't realize it yet

devblogs.microsoft.com/oldnewthing/20100101-00

January 1, 2010

Raymond Chen

We saw last time that the COM rules for output pointers are that they must be initialized on return from a function, even if the function fails. The COM marshaller relies on this behavior, but then again, so do you; you just don't realize it yet.

If you use a smart pointer library (be it ATL or boost or whatever), you are still relying on output pointers being `NULL` when not valid, regardless of whether or not the call succeeded. Let's look at this line of code from that article about `IUnknown::QueryInterface`:

```
CComQIPtr<ISomeInterface> spsi(punkObj);
...
// spsi object goes out of scope
```

If the `IUnknown::QueryInterface` method puts a non-`NULL` value in `spsi` on failure, then when `spsi` is destructed, it's going to call `IUnknown::Release` on itself, and something bad happens. If you're lucky, you will crash because the thing lying around in `spsi` was a garbage pointer. But if you're not lucky, the thing lying around in `spsi` might be a pointer to a COM object:

```
// wrong!
HRESULT CObject::QueryInterface(REFIID riid, void **ppvObj)
{
  *ppvObj = this; // assume success since it almost always succeeds
  if (riid == IID_IUnknown || riid == IID_IOtherInterface) {
    AddRef();
    return S_OK;
  }
  // forgot to set *ppvObj = NULL
  return E_NOINTERFACE;
}
```

Notice that this code optimistically sets the output pointer to itself, but if the interface is not supported, it changes its mind and returns `E_NOINTERFACE` *without setting the output pointer to* `NULL`. Now you have an elusive reference counting bug, because the destruction of `spsi` will call `CObject::Release`, which will manifest itself by `CObject` object being destroyed prematurely because you just over-released the object. If you're lucky, that'll happen relative soon; if you're not lucky, it won't manifest itself for another half hour.

Okay, sure, maybe this is too obvious a mistake for `CObject::QueryInterface`, but any method that has an output parameter can suffer from this error, and in those cases it might not be quite so obvious:

```
// wrong!
HRESULT CreateSurface(const SURFACEDESC *psd,
                      ISurface **ppsf)
{
 *ppsf = new(nothrow) CSurface();
 if (!*ppsf) return E_OUTOFMEMORY;
 HRESULT hr = (*ppsf)->Initialize(psd);
 if (SUCCEEDED(hr)) return S_OK;
 (*ppsf)->Release(); // throw it away
 // forgot to set *ppsf = NULL
 return hr;
}
```

This imaginary function takes a surface description and tries to create a surface that matches it. It does this by first creating a blank surface, and then initializing the surface. If that succeeds, then we succeed; otherwise, we clean up the incomplete surface and fail.

Except that we forgot to set `*ppsf = NULL` in our failure path. If initialization fails, then we destroy the surface, and the pointer returned to the caller points to the surface that we abandoned. But the caller shouldn't be looking at that pointer because the function failed, right?

Well, unless the caller called you like this:

```
CComPtr<ISurface> spsf;
if (SUCCEEDED(CreateSurface(psd, &spsf))) {
 ...
}
```

If the surface fails to initialize, then `spsf` contains a pointer to a surface that has already been deleted. When the `spsf` is destructed, it's going to call `ISurface::Release` on some point that is no longer valid, and bad things are going to happen. This can get particularly insidious when `spsf` is not a simple local variable but rather a member of class which itself doesn't get destroyed for a long time. The bad pointer sits in `m_spsf` like a time bomb.

Although all the examples I gave here involve COM interface pointers, the rule applies to all output parameters.

```
CComBSTR bs;
if (SUCCEEDED(GetName(&bs)) { ... }
// -or-
CComVariant var;
if (SUCCEEDED(GetName(&var)) { ... }
```

In the first case, the the `GetName` method had better not leave garbage in the output `BSTR` on failure, because the `CComBSTR` is going to `SysFreeString` in its destructor. Similarly in the second case with `CComVariant` and `VariantClear`.

So remember, if your function doesn't want to return a value in an output pointer, you still have to return something in it.

Raymond Chen

Follow