The COM_INTERFACE_ENTRY must be a COM interface, but nobody actually checks

devblogs.microsoft.com/oldnewthing/20190904-00

September 4, 2019

Raymond Chen

A customer had some code written with the Active Template Library, more commonly known as ATL. Apparently, ATL is still a thing!

Anyway, their problem was that their component that had been working just fine for many years started crashing. Their object went something like this:

```
[uuid("...")]
class ATL_NO_VTABLE CAwesomeWidget :
    public CComObjectRootEx<CComMultiThreadModel>,
    public CComCoClass<CAwesomeWidget, &CLSID_AwesomeWidget>,
    public IWidgetProviderInfo,
    public IWidget
    /// ... other interfaces ...
{
    ...

    BEGIN_COM_MAP(CAwesomeWidget)
      COM_INTERFACE_ENTRY(IWidgetProviderInfo)
      COM_INTERFACE_ENTRY(IWidget)
      // ... other interfaces ...
    END_COM_MAP()

    // IWidgetProviderInfo
    IFACEMETHODIMP GetProviderId(GUID *pguidId);

    // IWidget
    IFACEMETHODIMP Frob();
    ...
};
```

Yes, I'm making you look at ancient ATL code, with all its wacky macros. Product maintenance is like that. Deal with it.

Clients can do

```
void Sample()
{
  CComPtr<IWidget> widget;
  widget.CoCreateInstance(CLSID_AwesomeWidget);
  widget->Frob();
}
```

to create the `CAwesomeWidget` as a widget, and then ask the widget to do something.

The `CAwesomeWidget` did its job very well for over five years, and then suddenly it started crashing in `GetProviderId`, even though nobody called `GetProviderId`.

They traced it back to this call:

```
void Sample2()
{
  CComPtr<IUnknown> unk;
  unk.CoCreateInstance(CLSID_AwesomeWidget);

  CComQIPtr<IWidget> widget(unk); // ← here
  widget->Frob();
}
```

Obviously, this wasn't literally what their code did, but it boils the problem down to its essence.

The idea was that instead of creating the object for its `IWidget` interface, the object was created with the generic `IUnknown` interface, and then it was converted to an `IWidget`. The reason for this extra step isn't important, but you can come up with scenarios where this might happen. (For example, maybe the `CoCreateInstance` is coming from a generic "creation helper" function.)

The problem was in the definition of the `IWidgetProviderInfo`:

```
// widget.h
[uuid("...")]
interface IWidgetProviderInfo
{
    STDMETHOD(GetProviderId)(GUID *pguidId) PURE;
};
```

Do you see something?

Actually, more honestly, I should be asking, "Do you *not* see something?"

The `IWidgetProviderInfo` interface does not derive from `IUnknown`!

Once you realize this, everything unravels.

The `COM_INTERFACE_ENTRY` macro assumes that the thing it's given is indeed a COM interface. This assumption is made in two places:

1. A request for `IUnknown` returns the first interface in the list.
2. Any successful request will be accompanied by a call to `IUnknown::AddRef`, per COM rules.

They listed `IWidgetProviderInfo` as the first "interface", so it was returned as in response to a request for `IUnknown`, even though it wasn't `IUnknown`.

The attempt to convert the `IUnknown` to an `IWidget` involves a call to `IUnknown::QueryInterface`, but remember, that thing which the code thinks is an `IUnknown` is really an `IWidgetProviderInfo`. The call to `IUnknown::QueryInterface` actually called `IWidgetProviderInfo::GetProviderId`. With the wrong number and types of arguments, of course. The crash occurred when the `GetProviderId` method tried to write to what it thought was a `pguidId`, but which was actually a `riid` from the `QueryInterface`.

The customer would have noticed that something was wrong with `IWidgetProviderInfo` had they ever tried to use it!

```
CComPtr<IWidgetProviderInfo> info;
// ^^ error: class "IWidgetProviderInfo" has no "Release" member
```

Merely talking about `IWidgetProviderInfo` causes the compiler to get upset because the `CComPtr` destructor needs to call the `Release` method, which doesn't exist. The customer never noticed this because they never used the `IWidgetProviderInfo` interface at all! It was presumably added in anticipation of a feature that never materialized.

Okay, so now we understand why this crashes and how it eluded compile-time detection, but how did it ever work? After all, the implementation of `CreateInstance` needs to do a `QueryInterface` to return the proper pointer back to the caller.

Here's a simplified version of how ATL implements `CreateInstance`:

```
template <class T1>
class CComCreator
{
  static HRESULT WINAPI CreateInstance(void* pv, REFIID riid, LPVOID* ppv)
  {
    HRESULT hRes = E_OUTOFMEMORY;
    T1* p = NULL;
    ATLTRY(p = new T1(pv))
    if (p != NULL)
    {
      p->SetVoid(pv);
      p->InternalFinalConstructAddRef();
      hRes = p->_AtlInitialConstruct();
      if (SUCCEEDED(hRes))
        hRes = p->FinalConstruct();
      p->InternalFinalConstructRelease();
      if (hRes == S_OK)
        hRes = p->QueryInterface(riid, ppv);
      if (hRes != S_OK)
        delete p;
    }
    return hRes;
  }
}
```

Why doesn't the call to `p->QueryInterface` crash?

Because it's being called from a pointer to a `T1`, not from a pointer to an `IUnknown`. The compiler therefore knows that the `QueryInterface` method is in fact not at slot 0 in vtable 0, but rather is at slot 0 in vtable 1 (taking it from `IWidget`).

So how about fixing the problem?

One fix is to delete the unused `IWidgetProviderInfo` interface. However, when working with legacy code, you may be averse to taking such a drastic step, because there might be somebody who is actually using that interface in a way you failed to detect. Or maybe you want to keep the interface around because you really do plan on using it soon. In that case, you can make the interface derive from `IUnknown`, like it should have in the first place:

```
[uuid("...")]
interface IWidgetProviderInfo : IUnknown
{
    STDMETHOD(GetProviderId)(GUID *pguidId) PURE;
};
```

Customer problem solved.

Next time, we'll look at how to catch this problem at compile time.

Raymond Chen

Follow