The case of the mysterious over-release from deep inside the marshaling infrastructure

devblogs.microsoft.com/oldnewthing/20220624-00

June 24, 2022

Raymond Chen

A customer had a reference-counting bug where a certain scenario caused a COM object to be over-released. We started by taking a Time Travel Trace and gathering all the points where the `AddRef` or `Release` methods were called.

The Time Travel Debugging object model makes it easy to find all the places a method was called and print the results in a nice table.

```
0:040> dx -g @$cursession.TTD.Calls("contoso!CWidget::AddRef").Where(c =>
(__int64)c.Parameters[0]==0x1a792dd0).OrderBy(c => c.TimeStart).Select(c => new {TimeStart =
c.TimeStart, RefCount = c.ReturnValue, ThreadId = c.ThreadId})
```

This expression breaks down as follows:

- `@$cursession.TTD.Calls("contoso!CWidget::AddRef")` — Find all the calls to this function
- `.Where(c => (__int64)c.Parameters[0]==0x1a792dd0)` — Filtered to the calls where the `this` pointer (the invisible first parameter) is a specific value
- `.OrderBy(c => c.TimeStart)` — Sorted chronologically
- `.Select(c => new {TimeStart = c.TimeStart, RefCount = c.ReturnValue, ThreadId = c.ThreadId})` — Print these fields

Therefore, the command finds all the times in the trace where `contoso!CWidget::AddRef` was called for the object that was over-released, sorts them chronologically, and prints the timestamp, the resulting reference count, and the thread that issued the call.

A similar command finds all the calls to `CWidget::Release` .

Sort the two lists together by position to get a chronology of events.

Now we can use the `!tt` command to go to each of those time positions and get a stack trace. Many of the `AddRef` and `Release` calls are easily paired because the calls are made from the same function. Two of the `AddRef` calls are never released beause they correspond to references being held by objects that are still live at the end of the trace.

And then there is the batch of unmatched calls, and we need to match them up and figure out which are the unmatched `Release` calls.

The unmatched `AddRef` calls comes from `combase!CDestObjectWrapper::MarshalInterface`, which is an internal function called from `CoMarshalInterface`.

There are two categories of unmatched `Release` calls. One of them belongs to the `Release` of another object `CNamedWidget` that contains the `CWidget` as a member, and another comes from `combase!CStaticMarshaler::ReleaseMarshalData`, which is an internal function called from `CoReleaseMarshalData`.

And it turns out that the two categories of `Release` calls collectively outnumber the number of unmatched `AddRef` calls. So one of those categories of calls is wrong, but which one?

From what we learned earlier, when a normal-marshaled object is unmarshaled, you do not observe any change to the reference count because the ownership of the reference count is transfered from the stream directly to the unmarshaled object. A breakpoint on the reference count is not going to fire because there is no change to the reference count. It happens behind your back.

If an object is normal-marshaled, it should either be unmarshaled, or the marshal data should be released, but here we're doing both, which is the source of the double-release. Why is it doing both?

The stack for the `AddRef` looks like this:

```
contoso!CWidget::AddRef
contoso!CWidget::QueryInterface+0x31
combase!CStaticMarshaler::MarshalInterface+0x728
combase!CDestObjectWrapper::MarshalInterface+0x2f4
combase!CoMarshalInterface+0x2dc
contoso!CNamedWidget::MarshalInterface+0x208b63
combase!CDestObjectWrapper::MarshalInterface+0x2f4
combase!CoMarshalInterface+0x2dc
combase!GitRegisterHlp+0x2af
combase!CGIPTable::RegisterInterfaceInGlobalHlp+0x292
combase!CGIPTable::RegisterInterfaceInGlobal+0x1b
combase!RoGetAgileReference+0x8a1
```

The marshaling request is coming from the creation of an agile reference, and that is marshaling the `CNamedWidget`, which is in turn marshaling the `CWidget`. After some digging, we noticed an anomaly: For the outer call to `CoMarshalInterface` coming from `GitRegisterHlp`, the flags are

```
rax=00007ffb71634d20 rbx=0000000000000000 rcx=00000000042ea568
rdx=00007ffb5a43cbb0 rsi=0000000000000000 rdi=0000000017c009e8
rip=00007ffb7143ebf0 rsp=00000000042ea4d8 rbp=00000000042ea5b1
 r8=000000001ac11338  r9=0000000000000003 r10=0000000000000000
r11=3ff8000000000003 r12=0000000000000001 r13=00007ffb5a43cbb0
r14=0000000017c00a38 r15=00000000042ea678
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
combase!CoMarshalInterface:
00007ffb`7143ebf0 4055            push    rbp
0:000> dps @rsp+28 L2
00000000`042ea500  00000000`00000000
00000000`042ea508  00007ffb`00000001
```

From this we see that the parameters passed to `CoMarshalInterface` are

Parameter	Passed in	Value	Notes
`pStm`	`rcx`	`0x42ea568`	
`riid`	`rdx`	`0x00007ffb5a43cbb0`	The interface being marshaled
`pUnk`	`r8`	`0x1ac11338`	The `CNamedWidget` being marshaled
`dwDestContext`	`r9`	`3`	`MSHCTX_INPROC`
`pvDestContext`	`[rsp+28]`	`nullptr`	
`mshlflags`	`[rsp+30]`	`1`	`MSHLFLAGS_TABLESTRONG`

However, when the `CNamedWidget` goes to marshal the inner `CWidget`, we see this:

```
rax=0000000000000000 rbx=000000001ac11330 rcx=00000000042ea568
rdx=00007ffb6d31c4b0 rsi=0000000000000003 rdi=00000000042ea568
rip=00007ffb7143ebf0 rsp=00000000042ea178 rbp=00000000042ea2d0
 r8=000000001a792dd0  r9=0000000000000003 r10=00000000161a3080
r11=000000000cf59986 r12=00000000042ea568 r13=0000000000000004
r14=0000000000000001 r15=00007ffb5a43cbb0
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
combase!CoMarshalInterface:
00007ffb`7143ebf0 4055            push    rbp
0:000> dps @rsp+28 L2
00000000`042ea1a0  00000000`00000000
00000000`042ea1a8  00000000`00000000
```

Parameter	Passed in	Value	Notes

`pStm`	`rcx`	`0x42ea568`	
`riid`	`rdx`	`0x00007ffb6d31c4b0`	The interface being marshaled
`pUnk`	`r8`	`0x1a792dd0`	The `CWidget` being marshaled
`dwDestContext`	`r9`	`3`	`MSHCTX_INPROC`
`pvDestContext`	`[rsp+28]`	`nullptr`	
`mshlflags`	`[rsp+30]`	`0`	`MSHLFLAGS_NORMAL` ← huh?

Okay, that seems awfully strange. The outer object is being strong-marshaled but the inner object is only normal-marshaled.

That explains why we are seeing a double-release of the inner object: The `RoGetAgile-Reference` function strong-marshaled the object, which means that it is going to call *both* `UnmarshalInterface` *and* `ReleaseMarshalData`. However, the `CWidget` was normal-marshaled, which means that it expected to receive *either* `UnmarshalInterface` *or* `ReleaseMarshalData`, but not both. If you call both, then the marshal data gets double-destroyed, and that's where the double-release is coming from.

Here's a sketch of the marshaling code for the `CNamedWidget`:

```
HRESULT CNamedWidget::MarshalInterface(...)
{
  if (⟦want to marshal by shallow copy⟧) {
    RETURN_IF_FAILED(IStream_WriteStr(pstm, m_name));
    RETURN_IF_FAILED(CoMarshalInterface(pstm, __uuidof(m_widget.Get()), m_widget.Get(),
                          dwDestCtx, pvDestCtx, MSHLFLAGS_NORMAL));
    return S_OK;
  }
  ⟦delegate to standard marshaler⟧
}
```

Observe that the `MarshalInterface` method always marshals the `CWidget` with `MSHLFLAGS_NORMAL` instead of using the same marshal flags that it was given. That's the source of the problem.

The customer confirmed that making that one change fixed their problem.

Bonus chatter: Here are the other marshaling and unmarshaling methods:

```
HRESULT CNamedWidget::UnmarshalInterface(...)
{
  *ppv = nullptr;
  RETURN_IF_FAILED(IStream_ReadStr(pstm, &m_name));
  RETURN_IF_FAILED(CoUnmarshalInterface(pstm, IID_PPV_ARGS(&m_widget)));
  RETURN_IF_FAILED(QueryInterface(riid, ppv));
  return S_OK;
}

HRESULT CNamedWidget::ReleaseMarshalData(...)
{
  if (!m_widget) {
    // Read the string and throw it away.
    CComHeapPtr<wchar_t> name;
    RETURN_IF_FAILED(IStream_ReadStr(pstm, &name));
    RETURN_IF_FAILED(CoReleaseMarshalData(pstm));
  }
  return S_OK;
}
```

You might notice that there's also a bug in the `ReleaseMarshalData` method: That method is supposed to clean up the marshal data unconditionally, but this version does so only if the unmarshaler hasn't yet been used to unmarshal anything: It releases the marshal data only if the `m_widget` is still empty. Furthermore, in the case where it decides not to release the marshal data, it doesn't even bother to move the stream pointer past the marshal data. It just leaves the stream pointer where it was, causing the next object in the stream to receive the `CNamedWidget` 's unmarshal data instead of the data it expects.

Fortunately, in practice, the unmarshaler is nearly always empty, because `CoRelease-MarshalData` will create a brand new unmarshaler in order to call its `ReleaseMarshalData` . So this bug ends up masked, but they made a note to fix it anyway.

Raymond Chen

Follow