The case of the UWP application that crashes at launch on Windows 10X

devblogs.microsoft.com/oldnewthing/20210924-00

September 24, 2021

Raymond Chen

Application compatibility testing for Windows 10X identified a program that crashed at launch. We were able to obtain a time travel trace for the application. Let's do some application compatibility debugging.

Even though the Windows 10X project has been shelved, there's still a lesson here for other non-Desktop platforms, like Xbox, so let's dig in.

The application terminates here:

```
 # Call Site
00 SharedLibrary!$8_NativePrimitiveDecoder.DecodeUnsigned+0xe3
01 SharedLibrary!$11_NativeReader.DecodeUnsigned+0x3b
02 SharedLibrary!$14_NativeParser.GetUnsigned+0x23
03
SharedLibrary!$11_ExecutionEnvironmentImplementation.TryGetMethodForOriginalLdFtnResul

04
SharedLibrary!$11_ExecutionEnvironmentImplementation.TryGetMethodForOriginalLdFtnResul

05
SharedLibrary!$11_ReflectionExecutionDomainCallbacksImplementation.GetMethodNameFromSt

06 SharedLibrary!DeveloperExperience.CreateStackTraceString+0x65
07 SharedLibrary!StackTraceHelper.FormatStackTrace+0x8b
08 SharedLibrary!Exception.get_StackTrace+0x47
09 Contoso+0x1f6eaa
0a Contoso+0x5a2dc3
0b Windows_UI_Xaml!GetStringRawBuffer+0x1028
0c Windows_UI_Xaml!GetStringRawBuffer+0x1225
0d Windows_UI_Xaml!GetStringRawBuffer+0x46bc1
0e Windows_UI_Xaml!GetStringRawBuffer+0x46a68
0f Windows_UI_Xaml!GetStringRawBuffer+0x46443
10 Windows_UI_Xaml!GetStringRawBuffer+0x46269
11
twinapi_appcore!GitInvokeHelper<IEventHandler<Windows::ApplicationModel::Core::Unhandl
 *>>::Invoke+0x5a
12 twinapi_appcore!UnhandledErrorInvokeHelper::Invoke+0x1a
13 twinapi_appcore!EventSource<IEventHandler<UnhandledErrorDetectedEventArgs
*>>::InvokeAll::lambda::operator()+0x1d
14 twinapi_appcore!InvokeTraits<2>::InvokeDelegates+0x4f
15 twinapi_appcore!EventSource<IEventHandler<UnhandledErrorDetectedEventArgs
*>>::DoInvoke+0x7b
16 twinapi_appcore!EventSource<IEventHandler<UnhandledErrorDetectedEventArgs
*>>::InvokeAll+0x29
17 twinapi_appcore!CoreApplication::ForwardLocalError+0x73
18 twinapi_appcore!CoreApplicationFactory::ForwardLocalError+0xf0
19 combase!CallErrorForwarder+0x138
1a SharedLibrary!$8_ExceptionHelpers.ReportUnhandledError+0xcd
1b SharedLibrary!$8_InteropCallbacks.ReportUnhandledError+0x9
1c Contoso+0x35d8d
1d SharedLibrary!RuntimeExceptionHelpers.ReportUnhandledException+0x63
1e SharedLibrary!RuntimeAugments.ReportUnhandledException+0x9
1f SharedLibrary!$13_Invoker.InvokeCore$catch$0+0xa
20 mrt100_app!RhpCallCatchFunclet2
21 mrt100_app!RhRethrow+0x4c9
22 mrt100_app!RhThrowEx+0x4f
23 mrt100_app!RhpThrowEx2
24 SharedLibrary!ExceptionServices::ExceptionDispatchInfo.Throw+0x22
25 SharedLibrary!$22_ExceptionDispatchHelper::<>c__DisplayClass0.
<ThrowAsync>b__3+0x1f
26 SharedLibrary!$13_WinRTSynchronizationContext::Invoker.InvokeCore+0x4d
```

```
27 SharedLibrary!$13_WinRTSynchronizationContext::Invoker.Invoke+0x1c
28 SharedLibrary!Func$2<__Canon,TimeSpan>.InvokeOpenStaticThunk+0x1a
29
SharedLibrary!$25_AsyncOperationWithProgressCompletedHandler$2<__Canon,UInt32>.Invoke+

2a Contoso+0x35d78
2b Contoso+0x5a320b
2c Windows_UI!CDispatcher::ProcessInvokeItem+0x2cc
2d Windows_UI!CDispatcher::ProcessMessage+0x347
2e Windows_UI!CDispatcher::WaitAndProcessMessagesInternal+0xc9
2f Windows_UI!CDispatcher::ProcessEvents+0x132
30 Windows_UI_Xaml!DllGetActivationFactory+0xbaee0
31 Windows_UI_Xaml!DllGetActivationFactory+0xbae7f
32 twinapi_appcore!CoreApplicationView::Run+0x3a
33 twinapi_appcore!lambda::operator()+0xf2
34 shcore!_WrapperThreadProc+0xfb
35 ntdll!RtlUserThreadStart+0x2f
```

Reading from the bottom, we see that the UI thread is dispatching a work item. This work item is reporting an unhandled exception somewhere else in the application, and the app was trying to build a stack trace when it was terminated, presumably because it had reached some system timeout.

Let's fish out the stowed exception to see what the error was. The unhandled exception is provided in the event arguments.

```
⟦the .frame command changes to a specific stack frame⟧
0:008> .frame 12
12 0000004b`0a1fe9d0 00007fff`be4fa76b
twinapi_appcore!UnhandledErrorInvokeHelper::Invoke+0x1a

⟦the dv command dumps the frame's local variables⟧
0:008> dv
           this = <value unavailable>
         source = <value unavailable>
           args = 0x00000219`7fa93160
⟦the ?? command prints a C++ expression⟧
0:008> ?? args
struct Windows::ApplicationModel::Core::IUnhandledErrorDetectedEventArgs *
0x00000219`7fa93160
   +0x000 __VFN_table : 0x00007fff`be5c1290
```

The `args` is an `IUnhandledErrorDetectedEventArgs`. This is an exclusive interface to the `UnhandledErrorDetectedEventArgs` runtime class, so we can take a shortcut and assume that the underlying object is an `UnhandledErrorDetectedEventArgs` object. (If we wanted to verify the long way, we would dump the vtable to see what concrete class it came from.)

```
⟦the dt command dumps a type⟧
0:008> dt
twinapi_appcore!Windows::ApplicationModel::Core::UnhandledErrorDetectedEventArgs
0x00000219`7fa93160
   +0x000 __VFN_table : 0x00007fff`be5c1290
   +0x008 __VFN_table : 0x00007fff`be5c1270
   +0x010 __VFN_table : 0x00007fff`be5c1228
   +0x028 marshaller_      : Microsoft::WRL::ComPtr<IMarshal>
   +0x038 refCount_        :
Microsoft::WRL::Details::ReferenceCountOrWeakReferencePointer
   +0x040 _error           :
Microsoft::WRL::ComPtr<Windows::ApplicationModel::Core::UnhandledError>
```

I'm guessing that the `_error` contains the error being reported. (It had better, seeing as none of the other members appear to contain anything interesting at all!)

```
⟦the ?? command is handy for for more complex dumping⟧
0:008> ??
((twinapi_appcore!Windows::ApplicationModel::Core::UnhandledErrorDetectedEventArgs*)
0x00000219`7fa93160)->_error
class Microsoft::WRL::ComPtr<Windows::ApplicationModel::Core::UnhandledError>
   +0x000 ptr_             : 0x00000219`7fa93250
Windows::ApplicationModel::Core::UnhandledError
```

The `ComPtr` is a smart pointer class, so we have to dig inside to the `ptr_` to get the raw pointer that it is managing. Different smart pointer classes give different names to the inner raw pointer. There's no point trying to memorize the names; just dump the type and follow your nose.

```
0:008> ??
((twinapi_appcore!Windows::ApplicationModel::Core::UnhandledErrorDetectedEventArgs*)
0x00000219`7fa93160)->_error.ptr_
class Windows::ApplicationModel::Core::UnhandledError * 0x00000219`7fa93250
   +0x000 __VFN_table : 0x00007fff`be5c1bf0
   +0x008 __VFN_table : 0x00007fff`be5c1c38
   +0x010 __VFN_table : 0x00007fff`be5c1c58
   +0x028 marshaller_      : Microsoft::WRL::ComPtr<IMarshal>
   +0x038 refCount_        :
Microsoft::WRL::Details::ReferenceCountOrWeakReferencePointer
   +0x040 _handled         : 0x1 ''
   +0x048 _restrictedError : Microsoft::WRL::ComPtr<IRestrictedErrorInfo>

0:008> ??
((twinapi_appcore!Windows::ApplicationModel::Core::UnhandledErrorDetectedEventArgs*)
0x00000219`7fa93160)->_error.ptr_->_restrictedError
class Microsoft::WRL::ComPtr<IRestrictedErrorInfo>
   +0x000 ptr_             : 0x00000219`777a6888 IRestrictedErrorInfo
```

We land with a raw pointer to a COM interface. It's not clear which class implements this interface, so we'll have to ask the vtable to identify it for us.

```
⟦the dps command dumps pointer-sized data in the style of a stack⟧
0:008> dps 0x00000219`777a6888 L1
00000219`777a6888  00007fff`c2b89208 combase!CRestrictedError::`vftable'
```

Okay, this is a `CRestrictedError` object. We dump the first entry in the vtable to see how much we need to adjust our `this` pointer to get to the start of the object.

```
⟦the dpp command dumps pointer-sized data, and then dereferences it as a pointer⟧
0:008> dpp 0x00000219`777a6888 L1
00000219`777a6888  00007fff`c2b89208 00007fff`c2a85090 combase!
[thunk]:...::QueryInterface`adjustor{8}'
```

The adjustor thunk tells us that the object starts 8 bytes earlier, so we need to subtract 8 from the pointer before dumping it as a `CRestrictedError`. Subtracting 8 from a value that ends in 8 is easy to do in your head: Just change the 8 to a zero.

```
0:008> dt combase!CRestrictedError 00000219`777a6880
   +0x000 __VFN_table : 0x00007fff`c2b89230
...
   +0x050 _pszDescription  : 0x00000219`7faaa650  "Class not registered.."
   +0x058 _pszRestrictedDescription : 0x00000219`7fade760  "Class not registered
(Excep_FromHResult 0x80040154)"
   +0x068 _hrError         : 80040154
   +0x084 _cStackBackTrace : 0x1e
   +0x088 _ppvStackBackTrace : 0x00000219`7fafc040  -> 0x00007fff`c2a3d884 Void
```

So we've learned so far that original exception was a "Class not registered" error. I don't know for sure, but based on the name, it's pretty likely that `_ppvStackBackTrace` points to a stack backtrace, and `_cStackBackTrace` is the number of elements in the backtrace.

```
0:008> dps 0x00000219`7fafc040 l1e
00000219`7fafc040  combase!RoOriginateLanguageException+0x54
00000219`7fafc048
SharedLibrary!$8_::ExceptionHelpers.OriginateLanguageException+0xf8
00000219`7fafc050  SharedLibrary!$8_ExceptionHelpers.ReportUnhandledError+0x7b
00000219`7fafc058  SharedLibrary!$8_InteropCallbacks.ReportUnhandledError+0x9
00000219`7fafc060  Contoso+0x35d8d
00000219`7fafc068
SharedLibrary!SystemRuntimeExceptionHelpers.ReportUnhandledException+0x63
00000219`7fafc070  SharedLibrary!RuntimeAugments.ReportUnhandledException+0x9
00000219`7fafc078
SharedLibrary!$13_WinRTSynchronizationContext::Invoker.InvokeCore$catch$0+0xa
00000219`7fafc080  mrt100_app!RhpCallCatchFunclet2
00000219`7fafc088  mrt100_app!RhRethrow+0x4c9
00000219`7fafc090  mrt100_app!RhThrowEx+0x4f
00000219`7fafc098  mrt100_app!RhpThrowEx2
00000219`7fafc0a0  SharedLibrary!ExceptionDispatchInfo.Throw+0x22
00000219`7fafc0a8  SharedLibrary!$22_ExceptionDispatchHelper::<>c__DisplayClass0.
<ThrowAsync>b__3+0x1f
00000219`7fafc0b0  SharedLibrary!$13_Invoker.InvokeCore+0x4d
00000219`7fafc0b8  SharedLibrary!$13_Invoker.Invoke+0x1c
00000219`7fafc0c0  SharedLibrary!Func$2<__Canon,TimeSpan>.InvokeOpenStaticThunk+0x1a
00000219`7fafc0c8
SharedLibrary!$25_AsyncOperationWithProgressCompletedHandler$2<__Canon,UInt32>.Invoke+

00000219`7fafc0d0  Contoso+0x35d78
00000219`7fafc0d8  Contoso+0x5a320b
00000219`7fafc0e0  Windows_UI!CDispatcher::ProcessInvokeItem+0x2cc
00000219`7fafc0e8  Windows_UI!CDispatcher::ProcessMessage+0x347
00000219`7fafc0f0  Windows_UI!CDispatcher::WaitAndProcessMessagesInternal+0xc9
00000219`7fafc0f8  Windows_UI!CDispatcher::ProcessEvents+0x132
00000219`7fafc100  Windows_UI_Xaml!DllGetActivationFactory+0xbaee0
00000219`7fafc108  Windows_UI_Xaml!DllGetActivationFactory+0xbae7f
00000219`7fafc110  twinapi_appcore!CoreApplicationView::Run+0x3a
00000219`7fafc118  twinapi_appcore!lambda::operator()+0xf2
00000219`7fafc120  shcore!_WrapperThreadProc+0xfb
00000219`7fafc128  ntdll!RtlUserThreadStart+0x2f
```

Okay, that stack trace isn't interesting, since it's the same stack we are on right now. What we want is the stack that generated the original exception.

So let's find the work item that was queued onto this thread and see who queued it.

At this point, we take advantage of the fact that we have a time travel trace: Execute backward to the point that the work item was pulled off the queue.

We want to go backward to this point in the stack trace:

```
2b 0000004b`0a1ff4c0 00007fff`ad9f9140 Contoso+0x5a320b
2c 0000004b`0a1ff510 00007fff`ad9f790b
Windows_UI!Windows::UI::Core::CDispatcher::ProcessInvokeItem+0x2cc
```

The address we want is the instruction immediately before return address of the next function deeper on the stack.

```
⟦the u command disassembles ("unassembles")⟧
0:008> u 00007fff`ad9f9140-20 00007fff`ad9f9140
00007fff`ad9f9122 488b01          mov     rax,qword ptr [rcx]
00007fff`ad9f9125 488b4068        mov     rax,qword ptr [rax+68h]
00007fff`ad9f9129 ff15e1a70900    call    qword ptr
[Windows_UI!__guard_dispatch_icall_fptr (00007fff`ada93910)]
00007fff`ad9f912f 488b4b10        mov     rcx,qword ptr [rbx+10h]
00007fff`ad9f9133 488b01          mov     rax,qword ptr [rcx]
00007fff`ad9f9136 488b4018        mov     rax,qword ptr [rax+18h]
00007fff`ad9f913a ff15d0a70900    call    qword ptr
[Windows_UI!__guard_dispatch_icall_fptr (00007fff`ada93910)]
00007fff`ad9f9140 8be8            mov     ebp,eax
```

The return address is `00007fff`ad9f9140`, so we want to go back to the instruction before it, which is `00007fff`ad9f913a`.

```
⟦the g- command executes backward until a breakpoint is hit⟧
0:008> g- 00007fff`ad9f913a
Time Travel Position: 1D6637:68
Windows_UI!Windows::UI::Core::CDispatcher::ProcessInvokeItem+0x2c6:
00007fff`ad9f913a ff15d0a70900    call    qword ptr
[Windows_UI!__guard_dispatch_icall_fptr (00007fff`ada93910)]

0:008> dv
                this = 0x00000219`74134a30
pbInvokeItemProcessed = 0x0000004b`0a1ff691
                  hr = 0x00000000
          pInvokeItem = 0x00000219`7f41a170
             dwStatus = <value unavailable>
     requiredPriority = <value unavailable>
                  msg = {msg=0x270 wp=0x0 lp=0x1}
         bucketAssist = 0x00007fff`7d731330
 spIdleDispatchedArgs = {...}
   WPP_GLOBAL_Control = <value unavailable>
0:008> ?? pInvokeItem
struct Windows::UI::Core::_InvokeEntry * 0x00000219`7f41a170
   +0x000 pNext            : 0x00000219`7f41ab30 Windows::UI::Core::_InvokeEntry
   +0x008 dwTickCount      : 0x115a777
   +0x00c Priority         : 0 ( CoreDispatcherPriority_Normal )
   +0x010 spHandler        :
Microsoft::WRL::ComPtr<Windows::UI::Core::IDispatchedHandler>
   +0x018 spIdleHandler    :
Microsoft::WRL::ComPtr<Windows::UI::Core::IIdleDispatchedHandler>
   +0x020 spCoreAsyncInfo  :
Microsoft::WRL::ComPtr<Windows::UI::Core::ICoreAsyncInfo>
```

Now that we've executed back to the point where we're about to dispatch the work item, we can look at the local variables to get the item. (If that hadn't worked, we could have pulled it out of the `rcx` register, since the code is set up to make a call.)

The `pInvokeItem` looks like it's the work item that got queued and is about to be dispatched. I want to go back to when the work item was created, so I picked a field that is probably set only at construction: The priority.

```
0:008> ?? pInvokeItem->Priority
Windows::UI::Core::CoreDispatcherPriority CoreDispatcherPriority_Normal (0n0)
0:008> ?? &pInvokeItem->Priority
Windows::UI::Core::CoreDispatcherPriority * 0x00000219`7f41a17c
0:008> ba w4 0x00000219`7f41a17c
```

We ask the debugger for the address of the `Priority` member and set a 4-byte write breakpoint on it. Then execute backward some more to see who sets it.

```
0:008> g-
Breakpoint 6 hit
Time Travel Position: 1CE288:87
Windows_UI!Windows::UI::Core::CDispatcher::EnqueueAsyncWork+0x390:
00007fff`ad9f88bc 48ff15eda90900  call    qword ptr [Windows_UI!_imp_GetTickCount
(00007fff`ada932b0)]
```

From the function name `EnqueueAsyncWork` it appears that we found the code that creates the work item. Let's see why it's being created.

```
〖the k command takes a stack trace〗
0:008> k
 # Call Site
00 Windows_UI!Windows::UI::Core::CDispatcher::EnqueueAsyncWork+0x390
01 Windows_UI!Windows::UI::Core::CDispatcher::RunAsyncWorker+0xb0
02 Windows_UI!Windows::UI::Core::CDispatcher::RunAsync+0x58
03 Contoso+0x150d0d
04 Contoso+0x150c07
05 Contoso+0x150b98
06 Contoso+0x150b79
07 Contoso+0x35d46
08 SharedLibrary!$13_System::Threading::WinRTSynchronizationContext.Post+0x68
09
SharedLibrary!$13_System::Runtime::CompilerServices::AsyncMethodBuilderCore.ThrowAsync

0a
SharedLibrary!$13_System::Runtime::CompilerServices::AsyncVoidMethodBuilder.SetExcepti

0b Contoso+0x45a79c
0c mrt100_app!RhpCallCatchFunclet2
0d mrt100_app!RhRethrow+0x4c9
0e mrt100_app!RhThrowEx+0x4f
0f mrt100_app!RhpThrowEx2
10 SharedLibrary!$8_Interop::WinRT.RoGetActivationFactory+0x19c
11
SharedLibrary!$8_System::Runtime::InteropServices::FactoryCache.GetActivationFactoryIn

12
SharedLibrary!$8_System::Runtime::InteropServices::FactoryCache.GetActivationFactory+0

13
SharedLibrary!$8_System::Runtime::InteropServices::McgMarshal.GetActivationFactory+0x4

14
SharedLibrary!$8_System::Runtime::InteropServices::McgModuleManager.GetActivationFactc

15 Contoso+0x3fe578
16 Contoso+0x3fdb71
17 Contoso+0x454fab
18 Contoso+0x454f10
19 Contoso+0x45a446
1a Contoso+0x45a3a7
1b Contoso+0x45a36c
1c Contoso+0x45a351
1d Contoso+0x5a2593
1e Windows_UI_Xaml!GetErrorContextIndex+0x40f58
1f Windows_UI_Xaml!GetErrorContextIndex+0x40cc4
20 Windows_UI_Xaml!DllCanUnloadNow+0x26484
21 Windows_UI_Xaml!DllCanUnloadNow+0x27e91
22 Windows_UI_Xaml!DllCanUnloadNow+0x27a9d
23 Windows_UI_Xaml!DllGetActivationFactory+0x243ea
24 Windows_UI_Xaml!GetErrorContextIndex+0x51cf2
```

```
25 Windows_UI_Xaml!GetErrorContextIndex+0x51b1e
26 Windows_UI_Xaml!GetErrorContextIndex+0x519fa
27 minuser!Core::Yield::WndProc+0x6b
28 minuser!Core::Window::DeliverMessage+0x30f
29 minuser!Core::Window::SendCommon+0x82
2a minuser!Core::Window::Send+0x3a
2b minuser!Core::Api::SendMessageAW+0x48
2c minuser!minSendMessageAW+0x5d
2d Windows_UI_Xaml!GetErrorContextIndex+0x24de7
2e Windows_UI_Xaml!GetErrorContextIndex+0x4cc7c
2f Windows_UI_Xaml!GetErrorContextIndex+0x47b05
30 Windows_UI_Xaml!GetErrorContextIndex+0x47a0f
31 Windows_UI_Xaml!GetErrorContextIndex+0x2c29f
32 Windows_UI_Xaml!GetErrorContextIndex+0x9e01d
33 Windows_UI_Xaml!GetErrorContextIndex+0x9deaf
34 Windows_UI_Xaml!GetErrorContextIndex+0x9d7e6
35 Windows_UI_Xaml!DllGetActivationFactory+0x3d26d
36 Windows_UI_Xaml!DllGetActivationFactory+0x3d183
37 Windows_UI_Xaml!GetErrorContextIndex+0x51a89
38 Windows_UI_Xaml!GetErrorContextIndex+0x9dc1b
39 Windows_UI_Xaml!GetErrorContextIndex+0x9db0c
3a CoreMessaging!Microsoft__CoreUI__Dispatch__TimeoutHandler$CallbackThunk+0x11b
3b CoreMessaging!Microsoft::CoreUI::Dispatch::TimeoutHandler::Invoke+0x1a
3c
CoreMessaging!Microsoft::CoreUI::Dispatch::TimeoutManager::Callback_OnDispatch+0x18b
3d CoreMessaging!Microsoft::CoreUI::Dispatch::Dispatcher::DispatchNextItem+0x885
3e CoreMessaging!Microsoft::CoreUI::Dispatch::Dispatcher::Callback_DispatchLoop+0x9e3
3f CoreMessaging!Microsoft::CoreUI::Dispatch::EventLoop::Callback_RunCoreLoop+0xc45
40
CoreMessaging!Microsoft::CoreUI::Dispatch::UserAdapterBase::DrainCoreMessagingQueue+0x

41 CoreMessaging!Microsoft::CoreUI::Dispatch::UserAdapter::OnUserDispatch+0x1d7
42 CoreMessaging!Microsoft::CoreUI::Dispatch::UserAdapter::OnUserDispatchRaw+0x9c
43 CoreMessaging!Microsoft::CoreUI::Dispatch::UserAdapter_DoWork+0xe9
44 CoreMessaging!Microsoft::CoreUI::Dispatch::UserAdapter_WindowProc+0xa3
45 minuser!Core::Yield::WndProc+0x6b
46 minuser!Core::Window::DeliverMessage+0x30f
47 minuser!Input::EventMessageDeliveryManager::OnDispatchNotify+0x27
48 minuser!Input::EventMessageDeliveryManager$R::Delegate0+0x2b
49 minuser!Core::OnWindowEventMessage::Invoke+0x65
4a minuser!Input::EventMessageDeliveryManager::CheckAndConsumeMinQEventMessage+0x193
4b minuser!Input::InputQueue::PeekForInput+0x2f0
4c minuser!Core::ThreadInfo::ReadMessageEntryWorker+0x155
4d minuser!Core::ThreadInfo::ReadMessageEntry+0x50
4e minuser!Core::Api::PeekMessageAW+0x88
4f minuser!minPeekMessageAW+0x66
50 Windows_UI!Windows::UI::Core::CDispatcher::ProcessMessage+0xdf
51 Windows_UI!Windows::UI::Core::CDispatcher::WaitAndProcessMessagesInternal+0xc9
52 Windows_UI!Windows::UI::Core::CDispatcher::ProcessEvents+0x132
53 Windows_UI_Xaml!DllGetActivationFactory+0xbaee0
54 Windows_UI_Xaml!DllGetActivationFactory+0xbae7f
55 twinapi_appcore!Windows::ApplicationModel::Core::CoreApplicationView::Run+0x3a
```

```
56 twinapi_appcore!<lambda_643db08282a766b00cec20194396f531>::operator()+0xf2
57 shcore!_WrapperThreadProc+0xfb
58 ntdll!RtlUserThreadStart+0x2f
```

This is a huge stack, but it breaks down neatly into three parts.

The part near the top of the stack is the runtime recording the failure in `RoGetActivation-Factory`.

The part near the bottom is the code that led up to the failure in `RoGetActivation-Factory`. It looks like it was triggered by a timer.

Let's see what we were trying to activate.

```
0:008> .frame 11
11 0000004b`0a1fd8d0 00007fff`7c209536
SharedLibrary!$8_FactoryCache.GetActivationFactoryInternal+0x5e
0:008> dv
       typeName = 0x00007fff`7ce64060
       typeInfo = 0x0000004b`0a1fdb58
 currentContext = 0x00000219`000109b0
       pFactory = struct System::IntPtr
        itfGuid = struct System::Guid
0:008> ?? typeName
class System::String * 0x00007fff`7ce64060
   +0x000 __VFN_table : 0x00007fff`7c554030
   +0x000 m_pEEType        : System::IntPtr
   +0x008 m_stringLength   : 0n38
   +0x00c m_firstChar      : 0x57 'W'
0:008> du 0x00007fff`7ce64060+c
00007fff`7ce6406c  "Windows.Devices.Portable.Storage"
00007fff`7ce640ac  "Device"
```

This program is trying to create a `Windows.Devices.Portable.StorageDevice`. I bet it fails.

```
⟦g- with a parameter sets a temporary breakpoint before executing backward⟧
0:008> g- combase!RoGetActivationFactory
Time Travel Position: 1CD829:AE
combase!RoGetActivationFactory:
00007fff`c29ca2f0 4d8bc8          mov     r9,r8
0:008> dv
activatableClassId = 0x0000004b`0a1fd860 "Windows.Devices.Portable.StorageDevice"
               iid = 0x0000004b`0a1fd920 {5ECE44EE-1B23-4DD2-8652-BC164F003128}
           factory = 0x0000004b`0a1fd900

⟦gu executes until the function returns ("go up")⟧
0:008> gu
Time Travel Position: 1CE201:20
SharedLibrary!$8_Interop::WinRT.RoGetActivationFactory+0x136:
00007fff`7c209836 488b4db0        mov     rcx,qword ptr [rbp-50h]
ss:0000004b`0a1fd7f0=000002197408c8e0
0:008> r
rax=0000000080040154 rbx=00007fff7ce64060 rcx=fff83836ff080000
rdx=0000000000000006 rsi=0000004b0a1fd920 rdi=0000004b0a1fd900
rip=00007fff7c209836 rsp=0000004b0a1fd7c0 rbp=0000004b0a1fd840
 r8=0000000000000001  r9=0000000000000001 r10=0000000000005dc0
r11=0000004b0a1fcee0 r12=00000000e409abda r13=000002197fabec70
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
SharedLibrary!$8_Interop::WinRT.RoGetActivationFactory+0x136:
00007fff`7c209836 488b4db0        mov     rcx,qword ptr [rbp-50h]
ss:0000004b`0a1fd7f0=000002197408c8e0
```

The function return value is placed in the `rax` register, and we see that it is indeed `0x80040154`, which is "Class not registered".

The `Windows.Devices.Portable.StorageDevice` class is not part of the Universal contract. It is in a separate PortableDevice contract. The app marked itself as Universal in its manifest, but it used classes outside the Universal contract without first validating that the contract is available. The PortableDevice contract is available on Desktop Windows, which is why they get away with it when running on Desktop Windows. But the PortableDevice contract is not present on Windows 10X, which is why the app crashes there.

This program probably also crashes on Xbox, since Xbox doesn't support the PortableDevice contract either.

So this app gets a special compatibility flag in the Store that says, "Yeah, when this app says that it's Universal, they're lying. Don't install it on non-Desktop systems."

Bonus chatter: Now that we know that the answer is, we realize in retrospect that we could have found it faster. The error message "Class not registered" suggests that a call to `CoCreateInstance`, `CoGetClassFactory`, `RoActivateInstance`, or `CoGetActivation-`

`Factory` failed. We could have asked the time travel tracing object model to find all such calls that returned that error code:

```
0:008> dx -r2 @$currsession.TTD.Calls("combase!CoCreateInstance",
"combase!CoGetClassFactory", "combase!RoActivateInstance",
"combase!RoGetActivationFactory").Where(c => c.ReturnValue == (HRESULT)0x80040154)

    [0x97]
        EventType        : 0x0
        ThreadId         : 0x2fc0
        UniqueThreadId   : 0xd
        TimeStart        : 1CD829:AE
        TimeEnd          : 1CE201:20
        Function         : combase!RoGetActivationFactory
        FunctionAddress  : 0x7fffc29ca2f0
        ReturnAddress    : 0x7fff7c209836
        ReturnValue      : 0x80040154 (Class not registered) [Type: HRESULT]
        Parameters
```

Boom, there's the failure, at call number 0x97.

Now we can use the `!tt` command to jump to that timecode to see what the object was.

```
0:008> !tt 1CD829:AE
Setting position: 1CD829:AE
(4420.3130): Break instruction exception - code 80000003 (first/second chance not
available)
Time Travel Position: 1CD829:AE
combase!RoGetActivationFactory:
00007fff`c29ca2f0 4d8bc8          mov     r9,r8
```

Raymond Chen

Follow