# The case of the unhandled exception in a brokered Windows Runtime component

devblogs.microsoft.com/oldnewthing/20230419-00

April 19, 2023

Raymond Chen

An enterprise customer was encountering a crash in their brokered Windows Runtime component. The idea here is that you have a UWP app running in the UWP low-privilege app container. Once in a while, it needs to do something that is not permitted by the low-privilege app container, so it calls out to medium-privilege helper process, known as a *broker*, and the broker does the special thing that requires medium privilege.

Here's the crash in the broker process. I've removed some cluttering namespaces like `Platform::` and `Windows::Foundation::Collections::`.

```
0:005> kn
 # Child-SP          Call Site
00 0000001b`81efbcf0 ucrtbase!abort+0x4e
01 0000001b`81efbd20 ucrtbase!terminate+0x29
02 0000001b`81efbd50 ucrtbase!__crt_state_management::
        wrapped_invoke<void (__cdecl*)(void) noexcept,void>+0x1a
03 0000001b`81efbd80 contoso!__scrt_unhandled_exception_filter+0x5a
04 0000001b`81efbdb0 KERNELBASE!UnhandledExceptionFilter+0x1ec
05 (Inline Function) ntdll!RtlpThreadExceptionFilter+0x9b
06 0000001b`81efbed0 ntdll!RtlUserThreadStart$filt$0+0xac
07 0000001b`81efbf10 ntdll!__C_specific_handler+0x97
08 0000001b`81efbf80 ntdll!RtlpExecuteHandlerForException+0xf
09 0000001b`81efbfb0 ntdll!RtlDispatchException+0x29f
0a 0000001b`81efc710 ntdll!RtlRaiseException+0x195
0b 0000001b`81efce70 KERNELBASE!RaiseException+0x6c
0c 0000001b`81efcf50 msvcrt!_CxxThrowException+0xb7
0d 0000001b`81efcfc0 contoso!`EventSource::DoInvokeVoid<...>::`1'::catch$4+0x49
0e 0000001b`81efcff0 msvcrt!_CallSettingFrame_LookupContinuationIndex+0x20
0f 0000001b`81efd020 msvcrt!__FrameHandler4::CxxCallCatchBlock+0x16e
10 0000001b`81efd110 ntdll!RcFrameConsolidation+0x6
11 (Inline Function) contoso!__abi_ThrowIfFailed+0xb5
12 (Inline Function) contoso!EventSource::InvokeVoid::__l2::
<lambda_...>::operator()+0x11c
13 0000001b`81eff300 contoso!EventSource::DoInvokeVoid<...>+0x1e6
14 0000001b`81eff3c0 contoso!EventSource::InvokeVoid<...>+0xd8
15 (Inline Function) contoso!Vector<Widget ^>::m_wfc_event::raise+0x16
16 0000001b`81eff440 contoso!Vector<Widget ^>::Notify+0xa3
17 (Inline Function) contoso!Vector<Widget ^>::NotifyInserted+0x10
18 0000001b`81eff470 contoso!Vector<Widget ^>::[IVector<Widget ^>]::InsertAt+0x9f
19 0000001b`81eff500 contoso!WidgetManager::[WidgetManager]::OnWidgetAdded+0x706
1a 0000001b`81eff610 contoso!<lambda_...>::operator()+0x204
1b (Inline Function) SHCore!WorkThreadManager::CThread::RunCurrentTaskUnderLock+0xc8
1c 0000001b`81eff6f0 SHCore!WorkThreadManager::CThread::ThreadProc+0x31a
1d 0000001b`81eff960 SHCore!WorkThreadManager::CThread::s_ExecuteThreadProc+0x22
1e (Inline Function) SHCore!<lambda_...>::operator()+0xd
1f 0000001b`81eff9a0 SHCore!<lambda_...>::<lambda_invoker_cdecl>+0x11
20 0000001b`81eff9d0 kernel32!BaseThreadInitThunk+0x1d
21 0000001b`81effa00 ntdll!RtlUserThreadStart+0x28
```

Reading upward from the bottom, we see that the widget manager's `OnWidgetAdded` method was called (frame `19`) presumably to notify it that a widget got added.

In response to this, the widget manager inserts the newly-added widget into what is presumably a collection of active widgets (frame `18`).

This is apparently an observable vector, so we raise the `VectorChanged` event (frame `17` to `15`).

While calling all of the event listeners (frame `14` to `12`), an exception occurs, which is thrown (frame `11`) and goes unhandled, so the process terminates with an unhandled exception.

To diagnose this further, we'd really like to know where that exception came from. Fortunately, there is a debugger extension that helps decode throw Windows Runtime exceptions. If only we could find them.

To find the thrown object, we want an exception record. We can pull this out of `RtlDispatchException`, since that function is going to call each handler with the exception record and context record. So I'll dump the stack between frames `08` and `09`, and hopefully we'll find the exception record in there.

```
0:005> dpp 0000001b`81efbf80
0000001b`81efbf80  00000000`00000000
0000001b`81efbf88  0000001b`81efc510 00000000`00000000
0000001b`81efbf90  0000001b`81efce90 00000081`e06d7363
0000001b`81efbf98  00007fff`fc5d0000 00000003`00905a4d
0000001b`81efbfa0  0000001b`81efc570 00007fff`fc637558 ntdll!RtlUserThreadStart+0x28
0000001b`81efbfa8  00007fff`fc60d0af 2d850fed`854dd08b
0000001b`81efbfb0  00000000`00000000
0000001b`81efbfb8  0000001b`81efce90 00000081`e06d7363
0000001b`81efbfc0  00007fff`fc637558 c88b49d1`8b481feb
0000001b`81efbfc8  00007fff`fc76257c 0006757f`00067530
0000001b`81efbfd0  0000001b`81efc010 0065006c`00690061
0000001b`81efbfd8  00000000`00000000
0000001b`81efbfe0  0000001b`81efc530 00007fff`fc736a7c ntdll!_xmm+0x84bc
0000001b`81efbfe8  0000001b`81efc520 0000001b`81effa00
0000001b`81efbff0  0000001b`81efc528 00007fff`fc661840 ntdll!__C_specific_handler
```

Aha, we found a C++ exception record on the stack. We can tell because it has the special exception code `e06d7363`.

```
0:005> .exr 0000001b`81efce90
ExceptionAddress: 00007ffff9ad5c2c (KERNELBASE!RaiseException+0x000000000000006c)
   ExceptionCode: e06d7363 (C++ EH exception)
  ExceptionFlags: 00000081
NumberParameters: 4
   Parameter[0]: 0000000019930520
   Parameter[1]: 0000001b81eff338
   Parameter[2]: 00007fffebcc9158
   Parameter[3]: 00007fffebc70000
  pExceptionObject: 0000001b81eff338
  _s_ThrowInfo    : 00007fffebcc9158
```

Since we know this was a C++/CX exception, the thrown object will be some kind of `Platform::Exception^` hat pointer, so the `pExceptionObject` will itself be another pointer. Therefore, we will need to `dpp` it to chase through the pointer.

```
0:005> dpp 0000001b81eff338 L1
0000001b`81eff338 000001b4`c1138ec0 00007fff`ebccffa0 wincorlib!
                                          Platform::COMException::`vftable'
```

Now that we've confirmed that it's a `Platform::Exception` (indeed, a `Platform::COMException`), we can dump its contents.

```
0:005> ?? ((contoso!Platform::Exception*)0x000001b4`c1138ec0)
class Platform::Exception * 0x000001b4`c1138ec0
   +0x000 __VFN_table : 0x00007fff`ebccffa0
   +0x008 __VFN_table : 0x00007fff`ebccff58
   +0x010 __VFN_table : 0x00007fff`ebccff00
   +0x018 __VFN_table : 0x00007fff`ebccfeb8
   +0x020 __description    : 0x000001b4`c11641a8 Void
   +0x028 __restrictedErrorString : 0x000001b4`c1164608 Void
   +0x030 __restrictedErrorReference : (null)
   +0x038 __capabilitySid  : (null)
   +0x040 __hresult        : 0n-2147023170
   +0x048 __restrictedInfo : 0x000001b4`bf2c8108 Void
   +0x050 __throwInfo      : 0x00007fff`ebcc9158 Void
   +0x058 __size           : 0x40
   +0x060 __prepare        : Platform::IntPtr
   +0x068 __abi_reference_count : __abi_FTMWeakRefData
   +0x078 __abi_disposed   : 0
```

That `HRESULT` is

```
C:\> certutil /error -2147023170
0x800706be (WIN32: 1726 RPC_S_CALL_FAILED) -- 2147944126 (-2147023170)
Error message text: The remote procedure call failed.
CertUtil: -error command completed successfully.
```

You can also ask the `!pde.err` command to decode it.

```
0:005> !pde.err -0n2147023170
0x800706BE (FACILITY_WIN32 - Win32 Undecorated Error Codes): The remote procedure
call failed.
```

This agrees with the `__restrictedErrorString`:

```
0:005> dc 0x000001b4`c1164608
000001b4`c1164608  00680054 00200065 00650072 006f006d  T.h.e. .r.e.m.o.
000001b4`c1164618  00650074 00700020 006f0072 00650063  t.e. .p.r.o.c.e.
000001b4`c1164628  00750064 00650072 00630020 006c0061  d.u.r.e. .c.a.l.
000001b4`c1164638  0020006c 00610066 006c0069 00640065  l. .f.a.i.l.e.d.
```

which is a nice reassurance that we haven't gotten ourselves all turned around.

The `__restrictedInfo` contains additional details about the exception. Let's see what it holds.

```
0:005> dps 0x000001b4`bf2c8108
000001b4`bf2c8108  00007fff`fba2ff80 combase!CRestrictedError::`vftable'
000001b4`bf2c8110  00007fff`fba30040 combase!CRestrictedError::`vftable'
000001b4`bf2c8118  00007fff`fba2ffa8 combase!CRestrictedError::`vftable'
000001b4`bf2c8120  00007fff`fba2ffd0 combase!CRestrictedError::`vftable'
000001b4`bf2c8128  00007fff`fba2ff60 combase!CRestrictedError::`vftable'
000001b4`bf2c8130  00007fff`fba300d0 combase!CRestrictedError::`vftable'
000001b4`bf2c8138  00007fff`fba30088 combase!CRestrictedError::`vftable'
000001b4`bf2c8140  00007fff`fba30008 combase!CRestrictedError::`vftable'
000001b4`bf2c8148  00000001`00000000
000001b4`bf2c8150  000001b4`bf2ab4b0
000001b4`bf2c8158  000001b4`c1164a10
000001b4`bf2c8160  00000000`00000000
000001b4`bf2c8168  00000000`800706be
000001b4`bf2c8170  00000000`00000000
000001b4`bf2c8178  00000000`00000000
000001b4`bf2c8180  00000015`00010002
000001b4`bf2c8188  000001b4`c1145310
000001b4`bf2c8190  00000000`00000008
000001b4`bf2c8198  53453032`00000038
000001b4`bf2c81a0  00003a69`800706be
000001b4`bf2c81a8  00007fff`fb84c3b3 combase!UseSetErrorInfo+0x1c3
000001b4`bf2c81b0  00000015`00000008
000001b4`bf2c81b8  000001b4`c1145310
```

Aha, we found a stowed exception signature. I recognized it from time code 5:30 of Andrew Richard's presentation.

```
0:005> dt combase!STOWED_EXCEPTION_INFORMATION_V2 000001b4`bf2c8198
   +0x000 Header           : _STOWED_EXCEPTION_INFORMATION_HEADER
   +0x008 ResultCode       : 800706be
   +0x00c ExceptionForm    : 0y01
   +0x00c ThreadId         : 0y00000000000000000111010011010 (0xe9a)
   +0x010 ExceptionAddress : 0x00007fff`fb84c3b3 Void
   +0x018 StackTraceWordSize : 8
   +0x01c StackTraceWords  : 0x15
   +0x020 StackTrace       : 0x000001b4`c1145310 Void
   +0x010 ErrorText        : 0x00007fff`fb84c3b3  "????"
   +0x028 NestedExceptionType : 0
   +0x030 NestedException  : (null)
```

The `STOWED_EXCEPTION_INFORMATION_V2` structure is documented on MSDN docs.microsoft.com learn.microsoft.com.

Okay, so now we can get a stack trace.

```
0:005> dpS 0x000001b4`c1145310 L15
 00007fff`fb84b431 combase!RoOriginateError+0x51
 00007fff`ebc73264 wincorlib!Platform::Details::ReCreateException+0x6c
 00007fff`ebc72859 wincorlib!__abi_WinRTraiseCOMException+0x9
 00007fff`9b517ee5 contoso!__abi_WinRTraiseException+0xb5
 00007fff`9b5219a6 contoso!TypedEventHandler<...>::__abi_IDelegate::Invoke+0x26
 00007fff`9b519746 contoso!EventSource::DoInvokeVoid<...>+0xea
 00007fff`9b530b22 contoso!DialWidgetManager::DialWidgetChanged::raise+0x72
 00007fff`9b53d5b7 contoso!DialWidgetManager::OnWidgetVectorChanged+0x387
 00007fff`9b53bb5b contoso!`VectorChangedEventHandler>...>::::`2'::
                   __abi_PointerToMemberWeakRefCapture::Invoke+0xcb
 00007fff`9b52e9ed contoso!?__abi_...+0xd
 00007fff`881f2071 contoso!EventSource::DoInvokeVoid<...>+0x131
 00007fff`881f2948 contoso!EventSource::InvokeVoid<...>+0xd8
 00007fff`88217553 contoso!Vector<Widget ^>::Notify+0xa3
 00007fff`8821644f contoso!Vector<Widget ^>::[IVector<Widget ^>]::InsertAt+0x9f
 00007fff`88243256 contoso!WidgetManager::[WidgetManager]::OnWidgetAdded+0x706
 00007fff`8823dd94 contoso!<lambda_...>::operator()+0x204
 00007fff`fa58fd0a SHCore!WorkThreadManager::CThread::ThreadProc+0x31a
 00007fff`fa58c712 SHCore!WorkThreadManager::CThread::s_ExecuteThreadProc+0x22
 00007fff`fa5abff1 SHCore!<lambda_...>::<lambda_invoker_cdecl>+0x11
 00007fff`fa9f458d kernel32!BaseThreadInitThunk+0x1d
 00007fff`fc637558 ntdll!RtlUserThreadStart+0x28
```

The bottom part of this stack trace matches up with the non-inline functions in the stack trace that crashed, which is another confirmation that we haven't gone too far astray.

The top of this stack trace tells us more information: The vector change handler that crashed is `DialWidgetManager::OnWidgetVectorChanged`. It appears that it looked at the newly-added widget and realized that it was a `DialWidget` (whatever that is), and it raised the `DialWidgetChanged` event, and the handler of *that* event threw an exception.

So we have to find that second event handler.

How are we going to find it? All we have is a stack trace. We don't have any local variables.

We'll have to work forward from what we *do* have, which is the `WidgetManager`.

```
0:005> .frame 13
13 0000001b`81eff300 contoso!EventSource::DoInvokeVoid<...>+0x1e6
0:005> dv
           this = 0x000001b4`c1146098
      __lockArg = 0x000001b4`c11460b8
 __invokeOneArg = 0x0000001b`81eff3e0
    __dummylock = struct Platform::Details::EventLock
   __targetsLoc = 0x000001b4`c19aec70
 __EvSrcGTA_ret = 0x000001b4`c19aec70
         __size = 1
        __index = 0
        __token = class Windows::Foundation::EventRegistrationToken
      __element = 0x000001b4`c1159210
              e = 0x000001b4`c1138ec0 "The remote procedure call failed.
"
```

We can look at the code for `EventSource::DoInvokeVoid`, since it's included with the Visual C++ compiler, in the header `vccorlib.h`. Here's the code, though I've cleaned it up a bit to make it easier to read.

```
template <typename TDelegate, typename TInvokeMethod>
void DoInvoke(EventLock* __lockArg, TInvokeMethod __invokeOneArg)
{
    Object^ __targetsLoc;
    EventLock __dummylock = { nullptr, nullptr };
    AcquireSRWLockShared(&__lockArg->targetsLock);
    void* __EvSrcGTA_ret = EventSourceGetTargetArray(targets, &__dummylock);
    ReleaseSRWLockShared(&__lockArg->targetsLock);
    *reinterpret_cast<void**>(&targetsLoc) = __EvSrcGTA_ret;

    if (__targetsLoc != nullptr)
    {
        const unsigned int __size = EventSourceGetTargetArraySize(__targetsLoc);

        for (unsigned int __index = 0; __index < __size; __index++)
        {
            EventRegistrationToken __token = {};

            try
            {
                TDelegate^ __element = EventSourceGetTargetArrayEvent(
                        __targetsLoc,
                        __index,
                        &__uuidof(__TDelegate^),
                        &__token.Value
                    );

                (__invokeOneArg)(__element);
            }
            catch(::Platform::Exception^ e)
            {
                int __hr = e->HResult;
                if (__hr == 0x800706BA /*
HRESULT_FROM_WIN32(RPC_S_SERVER_UNAVAILABLE) */
                 || __hr == 0x80010108 /* RPC_E_DISCONNECTED */
                 || __hr == 0x89020001 /* JSCRIPT_E_CANTEXECUTE */)
                {
                    EventSourceRemove(&__targets, __lockArg, __token);
                }
                else
                {
                    throw e;
                }
            }
        }
    }
}
```

The code loops through the registered event handlers (which it calls “targets”) and calls each one (which it keeps in `__element`). If an exception occurs, the code checks if it’s one of the three special exceptions, and if not, it rethrows the exception, which means that all

subsequent handlers are skipped, and an exception propagates out of the event's `raise` method.

This means that in our stack trace, the `__element` is the failed delegate.

```
0:005> dps 0x000001b4`c1159210
000001b4`c1159210 00007fff`9b5a00e8 contoso!VectorChangedEventHandler<Widget
^>::`vftable'
000001b4`c1159218 00007fff`9b59e9c0 contoso!VectorChangedEventHandler<Widget
^>::`vftable'
000001b4`c1159220 00007fff`9b59e988 contoso!VectorChangedEventHandler<Widget
^>::`vftable'
000001b4`c1159228 000001b4`c1162c60
000001b4`c1159230 ffffffff`ffffffff
000001b4`c1159238 00007fff`9b5a4710 contoso!`VectorChangedEventHandler<...>::
                                    VectorChangedEventHandler<...>
<DialWidgetManager,...>

::`2'::__abi_PointerToMemberWeakRefCapture::`vftable'
000001b4`c1159240 000001b4`bf25b300
000001b4`c1159248 00007fff`9b53d230 contoso!DialWidgetManager::OnDevicesVectorChanged
```

There is the delegate. We see the member function pointer, and right in front of it is highly like to be `this` pointer.

```
0:005> dps 000001b4`bf25b300
000001b4`bf25b300  00007fff`ebcd0670
wincorlib!Platform::Details::ControlBlock::`vftable'
000001b4`bf25b308  00000004`00000005
000001b4`bf25b310  000001b4`bf23ed90
```

Okay, I was close. It's not the `DialWidgetManager`, but rather a control block that presumably represents a weak pointer to the `DialWidgetManager`.

In practice, all weak pointer control blocks work basically the same way. You have a weak reference count, a strong reference count, and a pointer to the object (which is valid only if the strong reference count is nonzero).

Channel9 had a video by Stephan T. Lavavej that explained how control blocks work, but that video seems to have been lost as part of Channel9 being folded into Microsoft Learn, so I'll have to point you to a different video. Here's a different description in text form if you're the sort of person who learns better by reading.

I covered the details of the C++/CX weak reference earlier, but the idea is the same.

After the strong and weak reference counts is probably a pointer to the object.

```
0:005> dps 000001b4`bf23ed90
000001b4`bf23ed90  00007fff`9b5a3278 contoso!DialWidgetManager::`vftable'
000001b4`bf23ed98  00007fff`9b5a3230 contoso!DialWidgetManager::`vftable'

0:005> ?? ((contoso!DialWidgetManager*)0x000001b4`bf23ed90)
class DialWidgetManager * 0x000001b4`bf23ed90
   +0x000 __VFN_table : 0x00007fff`9b5a3278
   +0x008 __VFN_table : 0x00007fff`9b5a3230
   +0x010 __VFN_table : 0x00007fff`9b5a31e8
   +0x018 __VFN_table : 0x00007fff`9b5a31a0
   +0x020 __VFN_table : 0x00007fff`9b5a3088
   +0x028 __VFN_table : 0x00007fff`9b5a3030
   +0x030 __VFN_table : 0x00007fff`9b5a2fd8
   +0x038 __VFN_table : 0x00007fff`9b5a2fa0
   +0x040 __VFN_table : 0x00007fff`9b5a2f70
   +0x048 __VFN_table : 0x00007fff`9b5a2f38
   +0x050 <backing_store>DialWidgetChanged : Platform::EventSource
   ... other members ...
```

Now we can look at the handlers of the `DialWidgetChanged` event.

```
0:005> ?? ((contoso!Platform::EventSource*)(0x000001b4`bf23ed90+0x50))
class Platform::EventSource * 0x000001b4`bf23ede0
   +0x000 __targets        : 0x000001b4`c19aea00 Void

0:005> dps 0x000001b4`c19aea00
000001b4`c19aea00 00007fff`ebccfc10 wincorlib!EventTargetArray::`vftable'
000001b4`c19aea08 00007fff`ebccfbe8 wincorlib!EventTargetArray::`vftable'
000001b4`c19aea10 00007fff`ebccfb98 wincorlib!EventTargetArray::`vftable'
000001b4`c19aea18 00000000`00000000
000001b4`c19aea20 00000000`00000000
000001b4`c19aea28 00007fff`fbae44c8 combase!CStaticMarshaler::s_Instance
000001b4`c19aea30 00000000`00000000
000001b4`c19aea38 00000000`00000001
000001b4`c19aea40 000001b4`bf3e2088
000001b4`c19aea48 000001b4`bf3e2098
```

Unfortunately, we don't know the internal structure of an `EventTargetArray`, but we can guess: From the last three values we see a `1` and two pointers close together.

One possibility is that the `1` is the count of registered handlers, and the first pointer points to the first such handler.

Another possibility is that the `1` is the reference count of the `EventTargetArray`, the first pointer points to the first handler, and the second pointer points to one past the last handler (in the style of a `std::vector`).

Under both of these theories, the first pointer points to the handler, so let's go with it.

```
0:005> dps 000001b4`bf3e2088 L2
000001b4`bf3e2088  000001b4`bf3e71e0
000001b4`bf3e2090  000001b4`bf3e71d0
```

These two pointers are very close to each other, with the second one coming numerically before the first, so that smells like a `std::shared_ptr` control block.

```
0:005> dps 000001b4`bf3e71d0
000001b4`bf3e71d0 00007fff`ebccfa38 wincorlib!std::_Ref_count_obj<
        Platform::Agile<Platform::Object,1> >::`vftable'
000001b4`bf3e71d8  00000001`00000001
000001b4`bf3e71e0  000001b4`bf240678
```

Looks like we guessed right. We have a vtable that suggests that this is a control block, and we have two small integers (the weak and strong reference counts), and the pointer is probably the actual thing we care about.

```
0:005> dps 000001b4`bf240678
000001b4`bf240678  00007fff`fba23e90 combase!g_StublessClientVtbl
000001b4`bf240680  00000000`00000001
000001b4`bf240688  000001b4`bf23eb88
000001b4`bf240690  000001b4`bf23eed0
000001b4`bf240698  00000000`00000000
000001b4`bf2406a0  00000000`00000000
000001b4`bf2406a8  00007fff`f2b30db0 OneCoreUAPCommonProxyStub!gPFactory
000001b4`bf2406b0  00000000`00000000
000001b4`bf2406b8  00000000`00000000
000001b4`bf2406c0  00007fff`fba22348
combase!CStdProxyBuffer_ReleaseMarshalBuffersVtbl
000001b4`bf2406c8  00007fff`f25e6e48 OneCoreUAPCommonProxyStub!
    ___FITypedEventHandler_2_IInspectable_IInspectableProxyVtbl
```

And here at last we have located the COM proxy that failed.

The customer shared this crash dump with the Windows team, who were able to confirm that this is indeed a COM proxy to their UWP app. And the customer's logging told them that the UWP app was suspended and subsequently terminated shortly before this crash occurred. That explains why we got an `RPC_S_CALL_FAILED`: The process was terminated before it could respond to the event.

Next time, we'll look at what the customer can do to address this problem.