The case of the invalid argument exception from a method that takes no arguments

devblogs.microsoft.com/oldnewthing/20231110-00

November 10, 2023

Raymond Chen

A customer was getting crashes with the following stack:

```
ucrtbase!abort+0x4e
ucrtbase!terminate+0x29
ucrtbase!FindHandler<__FrameHandler4>+0x4f4
ucrtbase!__InternalCxxFrameHandler<__FrameHandler4>+0x276
ucrtbase!__InternalCxxFrameHandlerWrapper<__FrameHandler4>+0x3e
ucrtbase!__CxxFrameHandler4+0xa9
contoso!__GSHandlerCheck_EH4+0x64
ntdll!RtlpExecuteHandlerForException+0xf
ntdll!RtlDispatchException+0x26c
ntdll!RtlRaiseException+0x195
KERNELBASE!RaiseException+0x6c
ucrtbase!_CxxThrowException+0x9a
contoso!winrt::throw_hresult+0xd3
contoso!winrt::check_hresult+0xcf
contoso!winrt::impl::consume_Microsoft_UI_Xaml_Controls_
        Primitives_IFlyoutBase<winrt::Microsoft::UI::Xaml::
        Controls::Primitives::IFlyoutBase>::Hide+0xe9
contoso!MyPanel::CancelCurrentFlyout+0x25e
contoso!MyPanel::IsReadyForNewContextMenu+0x4b
contoso!MyPanel::DoContextMenu+0x1ac
contoso!MyPanel::WndProc+0x7f2
user32!UserCallWinProcCheckWow+0x2ba
user32!DispatchMessageWorker+0x2b0
contoso!CMainWindow::Run+0xde
contoso!wWinMain+0xd69
contoso!invoke_main+0x21
contoso!__scrt_common_main_seh+0x110
kernel32!BaseThreadInitThunk+0x1d
ntdll!RtlUserThreadStart+0x28
```

The block of stack frames at the top is the C++ runtime infrastructure throwing an exception and looking for a handler. Eventually, no handler is found, and we end up terminating due to an unhandled exception.

The next block of stack frames is the code that threw the unhandled exception. From the `winrt::throw_hresult`, we know that this code is written in C++/WinRT, and the `check_hresult` tells us that we are converting a failure `HRESULT` into an exception. The next frame tells us that this failure `HRESULT` came from a call to `IFlyoutBase::Hide`.

The name of that frame breaks down like this:

- `winrt::impl::` C++/WinRT internal method
- `consume_` Wrapper for client code to call into Windows Runtime component (consuming an interface)
- `Microsoft_UI_Xaml_Controls_Primitives_IFlyoutBase` The interface we called is `Microsoft::UI::Xaml::Controls::Primitives::IFlyoutBase`.
- `<winrt::Microsoft::UI::Xaml::Controls::Primitives::IFlyoutBase>` The interface is implemented by an object whose type is `winrt::Microsoft::UI::Xaml::Controls::Primitives::IFlyoutBase`. In this case, we are invoking it directly on the interface, so the interface simply implements itself, and this part looks silly.
- `Hide` is the method being invoked.

Okay, so our call on the `IFlyoutBase::Hide` method failed with an "invalid parameter" exception. How can a method with no parameters have an invalid parameter?

Here's the call in the Contoso app:

```
void MyPanel::CancelCurrentFlyout()
{
    // Cancel any pending action.
    if (!m_actionPending)
    {
        InvokeCancel();
    }

    // Hide any open flyout.
    if (m_menu &&
        ((m_flyoutState == FlyoutState::Opened) ||
         (m_flyoutState == FlyoutState::Opening)))
    {
        m_menu.Hide(); /* we failed here */
    }
    TransitionFlyoutState(FlyoutState::Closed);
    ResetDataModelAndTasks();
}
```

Let's look more closely at the exception that was thrown. We know that it was an `hresult_error`, so we can just dump it straight away.

```
0:000> .frame b
0b 00000000`0107c670 00007ffa`84e2d24b     ucrtbase!_CxxThrowException+0x9a
0:000> dv
pExceptionObject = 0x00000000`0107c700
...
0:000> dps 0x00000000`0107c700 L3
00000000`0107c700  00000000`00000000
00000000`0107c708  80070057`aabbccdd // There's our ERROR_INVALID_PARAMETER
00000000`0107c710  00000000`2c81b7a8 // Here's the IErrorInfo
```

Although we don't know the internal layout of the `IErrorInfo`, we can dump it to see if anything jumps out at us.

```
0:000> dc 0x00000000`2c81b7a8
00000000`2c81b7a8  6b70cca8 00007ffa 6b70cb30 00007ffa  ..pk....0.pk....
00000000`2c81b7b8  6b70cad0 00007ffa 6b70ccd0 00007ffa  ..pk......pk....
00000000`2c81b7c8  6b70cb78 00007ffa 6b70cc50 00007ffa  x.pk....P.pk....
00000000`2c81b7d8  6b70cc08 00007ffa 6b70caf8 00007ffa  ..pk......pk....
00000000`2c81b7e8  00000000 00000001 28a6a1a0 00000000  ...........(....
00000000`2c81b7f8  290bc760 00000000 00000000 00000000  `..).............
00000000`2c81b808  80070057 00000000 00000000 00000000  W...............
00000000`2c81b818  00000000 00000000 00010002 00000039  ............9...
00000000`2c81b828  283dbe90 00000000 00000008 00000000  ..=(............
00000000`2c81b838  00000038 53453032 80070057 000056dd  8...20ESW....V..
00000000`2c81b848  6b4d1bf8 00007ffa 00000008 00000039  ..Mk........9...
00000000`2c81b858  283dbe90 00000000 00000000 00000000  ..=(............
00000000`2c81b868  00000000 00000000 00000000 00000000  ................
00000000`2c81b878  43832534 35316d47 108a28cb 09f94d9d  4%.CGm15.(...M..
00000000`2c81b888  00000000 00000000 2c81ab08 00000000  ...........,....
00000000`2c81b898  00000000 00000000 00000002 00000000  ................
```

After a lot of introductory stuff, we recognize the failure `HRESULT` of `0x80070057`, but more interestingly, the signature value `53453032` which decodes to the ASCII characters `SE02`, the signature for `STOWED_EXCEPTION_INFORMATION_V2`. There's a good chance this is a `STOWED_EXCEPTION_INFORMATION_V2` structure. Let's ask the `!pde.dse` extension to dump it.

The `!pde.dse` extension optionally takes a pointer to an array of stowed exception pointers. To dump a single stowed exception, we'll have to create one of these arrays and ask `!pde.dse` to dump it.

```
0:000> eq @rsp-8 0x00000000`2c81b7a8
0:000> !pde.dse @rsp-8
Stowed Exception Array @ 0x00000000035477e8

Stowed Exception #1 @ 0x000000002c81b838
        0x80070057: E_INVALIDARG - One or more arguments are not valid

Stack    : 0x283dbe90

combase!RoOriginateErrorW+0x131
Microsoft_UI_Xaml!DirectUI::ErrorHelper::OriginateError+0x172
Microsoft_UI_Xaml!DirectUI::ErrorHelper::OriginateError+0x28
Microsoft_UI_Xaml!DirectUI::VisualTreeHelper::GetOpenPopupsForXamlRoot+0x63
contoso!<lambda_⟦…⟧>::operator()+0x3e
contoso!winrt::⟦…⟧::VisualTreeHelper::GetOpenPopupsForXamlRoot+0x43
contoso!<lambda_⟦…⟧>::operator()+0x7f
contoso!winrt::impl::delegate<winrt::⟦…⟧::TypedEventHandler<
    winrt::⟦…⟧::FlyoutBase,
    winrt::⟦…⟧::FlyoutBaseClosingEventArgs>,
    <lambda_⟦…⟧> >::Invoke+0x22
Microsoft_UI_Xaml!DirectUI::CEventSourceBase<⟦…⟧>::Raise+0xa0
Microsoft_UI_Xaml!DirectUI::FlyoutBase::OnClosing+0x7c
Microsoft_UI_Xaml!DirectUI::FlyoutBase::HideImpl+0x3b
Microsoft_UI_Xaml!DirectUI::FlyoutBaseGenerated::Hide+0x52
contoso!MyPanel::CancelCurrentFlyout+0x25e
contoso!MyPanel::IsReadyForNewContextMenu+0x4b
contoso!MyPanel::DoContextMenu+0x1ac
contoso!MyPanel::WndProc+0x7f2
user32!UserCallWinProcCheckWow+0x2ba
user32!DispatchMessageWorker+0x2b0
contoso!CMainWindow::Run+0xde
contoso!wWinMain+0xd69
contoso!invoke_main+0x21
contoso!__scrt_common_main_seh+0x110
kernel32!BaseThreadInitThunk+0x1d
ntdll!RtlUserThreadStart+0x28
```

Notice that the deeper part of the stack matches our current stack. This is expected because the stowed exception captures the error origination slightly deeper in the stack, so the stuff outside `MyPanel::CancelCurrentFlyout` will still be the same.

Note also that the stack trace from the stowed exception doesn't include inlined functions, because the stack capturing code doesn't have access to symbols. It just walks the stack frames.

So let's look at the new information we gained from the stowed exception:

```
combase!RoOriginateErrorW+0x131
Microsoft_UI_Xaml!DirectUI::ErrorHelper::OriginateError+0x172
Microsoft_UI_Xaml!DirectUI::ErrorHelper::OriginateError+0x28
Microsoft_UI_Xaml!DirectUI::VisualTreeHelper::GetOpenPopupsForXamlRoot+0x63
contoso!`winrt::〚…〛::VisualTreeHelper::GetOpenPopupsForXamlRoot`::
    <lambda_1>::operator()+0x3e
contoso!winrt::〚…〛::VisualTreeHelper::GetOpenPopupsForXamlRoot+0x43
contoso!`winrt::〚…〛::MenuFlyoutControl::CreateMenu`::<lambda_1>::operator()+0x7f
contoso!winrt::impl::delegate<winrt::〚…〛::TypedEventHandler<
    winrt::〚…〛::FlyoutBase,
    winrt::〚…〛::FlyoutBaseClosingEventArgs>,
    `winrt::〚…〛::MenuFlyoutControl::CreateMenu`::<lambda_1> >::Invoke+0x22
Microsoft_UI_Xaml!DirectUI::CEventSourceBase<〚…〛>::Raise+0xa0
Microsoft_UI_Xaml!DirectUI::FlyoutBase::OnClosing+0x7c
Microsoft_UI_Xaml!DirectUI::FlyoutBase::HideImpl+0x3b
Microsoft_UI_Xaml!DirectUI::FlyoutBaseGenerated::Hide+0x52
```

Reading from the bottom: Our code called the `Hide` method, which went into `HideImpl`, which called `OnClosing`. Based on the name of the method and the fact that it calls `CEventSourceBase::Raise`, it's apparent that this code is raising the `Closing` event. The delegate registered for this event is a lambda in `MenuFlyoutControl::CreateMenu`, and it called `GetOpenPopupsForXamlRoot`, which decided to return an "invalid argument" error.

Let's see why.

```
0:000> u 7ffa266ad663-63 7ffa266ad663
DirectUI::VisualTreeHelper::GetOpenPopupsForXamlRoot:
00007ffa`266ad600 mov     qword ptr [rsp+20h],rbx
00007ffa`266ad605 push    rbp
00007ffa`266ad606 push    rsi
00007ffa`266ad607 push    rdi
00007ffa`266ad608 sub     rsp,30h
00007ffa`266ad60c mov     rax,qword ptr [_security_cookie (00007ffa`26bef010)]
00007ffa`266ad613 xor     rax,rsp
00007ffa`266ad616 mov     qword ptr [rsp+28h],rax
00007ffa`266ad61b and     qword ptr [rsp+20h],0
00007ffa`266ad621 mov     rsi,r8
00007ffa`266ad624 mov     rdi,rdx
00007ffa`266ad627 mov     rbp,rcx
00007ffa`266ad62a test    r8,r8
00007ffa`266ad62d jne     00007ffa`266ad64a
00007ffa`266ad62f lea     r8,[`string' (00007ffa`26904470)]
00007ffa`266ad636 mov     ecx,80070057h
00007ffa`266ad63b lea     edx,[rsi+0Bh]
00007ffa`266ad63e call    DirectUI::ErrorHelper::OriginateError
00007ffa`266ad643 mov     ebx,eax
00007ffa`266ad645 jmp     00007ffa`266ad6f3
00007ffa`266ad64a test    rdi,rdi
00007ffa`266ad64d jne     00007ffa`266ad675
00007ffa`266ad64f lea     r8,[`string' (00007ffa`26b663b0)]
00007ffa`266ad656 mov     ecx,80070057h
00007ffa`266ad65b lea     edx,[rdi+8]
00007ffa`266ad65e call    DirectUI::ErrorHelper::OriginateError
00007ffa`266ad663 mov     ebx,eax
```

The code that leads up to the error breaks up into three parts:

```
00007ffa`266ad600 mov     qword ptr [rsp+20h],rbx
00007ffa`266ad605 push    rbp
00007ffa`266ad606 push    rsi
00007ffa`266ad607 push    rdi
00007ffa`266ad608 sub     rsp,30h
00007ffa`266ad60c mov     rax,qword ptr [_security_cookie (00007ffa`26bef010)]
00007ffa`266ad613 xor     rax,rsp
00007ffa`266ad616 mov     qword ptr [rsp+28h],rax
00007ffa`266ad61b and     qword ptr [rsp+20h],0
00007ffa`266ad621 mov     rsi,r8
00007ffa`266ad624 mov     rdi,rdx
00007ffa`266ad627 mov     rbp,rcx
```

This first section is function prologue stuff and initializing local variables. The inbound parameters are stored in `rbp`, `rdi`, and `rsi`. Therefore we have this so far:

`rbp` = `rcx`	`this`

`rdi` = `rdx`	`xamlRoot`
`rsi` = `r8`	`result` (output)

Next up is this section:

```
00007ffa`266ad62a test    r8,r8
00007ffa`266ad62d jne     00007ffa`266ad64a
00007ffa`266ad62f lea     r8,[`string' (00007ffa`26904470)]
00007ffa`266ad636 mov     ecx,80070057h
00007ffa`266ad63b lea     edx,[rsi+0Bh]
00007ffa`266ad63e call    DirectUI::ErrorHelper::OriginateError
00007ffa`266ad643 mov     ebx,eax
00007ffa`266ad645 jmp     00007ffa`266ad6f3
```

If `r8` is zero, then we return `0x80070057` = `E_INVALIDARG` with a custom message:

```
0:000> du 00007ffa`26904470
00007ffa`26904470  "returnValue"
```

Okay, so first we verify that the result parameter (which I guess this function calls `returnValue`) is not null; if it is, we fail with `E_INVALIDARG`.

And then we have this:

```
00007ffa`266ad64a test    rdi,rdi
00007ffa`266ad64d jne     00007ffa`266ad675
00007ffa`266ad64f lea     r8,[`string' (00007ffa`26b663b0)]
00007ffa`266ad656 mov     ecx,80070057h
00007ffa`266ad65b lea     edx,[rdi+8]
00007ffa`266ad65e call    DirectUI::ErrorHelper::OriginateError
00007ffa`266ad663 mov     ebx,eax
```

This returns `E_INVALIDARG` with a custom message if the `rdi` register is zero, which our table above tells us corresponds to the `xamlRoot` parameter. The error message confirms this:

```
0:000> du 00007ffa`26b663b0
00007ffa`26b663b0  "xamlRoot"
```

So we got an “invalid argument” error because `xamlRoot` was null. Working backward, we can look at the `Closing` event handler:

```
void MenuFlyoutControl::CreateMenu(const winrt::IIterable& menuItems)
{
    ⟦…⟧

    m_closingRevoker = flyout.Closing(winrt::auto_revoke,
        [](auto&& sender, auto&&) {
        if (auto flyout = sender.try_as<xaml::CommandBarFlyout>()) {
            auto popups =
xaml::VisualTreeHelper::GetOpenPopupsForXamlRoot(flyout.XamlRoot());
            ⟦…⟧
        }
    });

    ⟦…⟧
}
```

We infer therefore that `flyout.XamlRoot()` was null. And that's what generates the "invalid parameter" exception: Null is not a valid parameter for `GetOpenPopupsForXamlRoot`.

This invalid parameter error projects as an exception, which causes the `Closing` event handler to fail with an exception, which is then transformed back into `E_INVALIDARG` at the ABI. This error presumably propagates out of the `Raise` method back into `OnClosing` and then to `HideImpl`, and then back to the caller, which causes `m_menu.Hide()` to fail with an invalid argument exception, which is where our crash dump was generated.

The customer fixed the problem by revising the `Closing` event handler to skip the call to `GetOpenPopupsForXamlRoot` if `flyout.XamlRoot()` is null, and just assume that there are no popups.

Bonus viewing: Stowed Exception C000027B: A video with the creator of the `!pde.dse` debugger extension. The array of pointers shows up at time code 5:24.