The case of the exception that a catch (…) didn't catch

devblogs.microsoft.com/oldnewthing/20240405-00

April 5, 2024

Raymond Chen

A customer thought they fixed a bug, but they were still getting crashes from it.

According to the `!analyze` output, the problem was coming from this stack:

```
contoso!winrt::hresult_error::hresult_error+0x143
contoso!winrt::throw_hresult+0x132
contoso!winrt::impl::consume_LitWare_IIconProvider
    <winrt::LitWare::IIconProvider>::LoadIcon+0x3b
contoso!winrt::Contoso::implementation::IconDataModel::
    ReloadIcon$_ResumeCoro$1+0x214
contoso!winrt::impl::resume_background_callback+0x10
ntdll!TppSimplepExecuteCallback+0xa3
ntdll!TppWorkerThread+0x8f6
kernel32!BaseThreadInitThunk+0x1d
ntdll!RtlUserThreadStart+0x28
```

This was puzzling because "We already fixed that bug!" The file version number and timestamp confirm that the code for `ReloadIcon` catches the exception:

```
    try
    {
        icon = m_provider.LoadIcon(); // ⇐ blamed frame
    }
    catch(...)
    {
        // There was a problem getting the new icon.
        // Just stick with the old one.
        LOG_CAUGHT_EXCEPTION();
        co_return;
    }
```

Let's look at the stack at the point of the crash:

```
KERNELBASE!RaiseFailFastException+0x152
combase!RoFailFastWithErrorContextInternal2+0x4d9
contoso!wil::details::FailfastWithContextCallback+0xc1
contoso!wil::details::WilFailFast+0x47
contoso!wil::details::ReportFailure_NoReturn<3>+0x2df
contoso!wil::details::ReportFailure_Base<3,0>+0x30
contoso!wil::details::ReportFailure_CaughtExceptionCommonNoReturnBase<3>+0xa7
contoso!wil::details::ReportFailure_CaughtExceptionCommon+0x22
contoso!wil::details::ReportFailure_CaughtException<3>+0x40
contoso!wil::details::in1diag3::FailFast_CaughtException+0x13
contoso!`<lambda_f370031fe3623a0b308de0bbdeb2db76>::operator()'::`1'::catch$2+0x22
ucrtbase!_CallSettingFrame_LookupContinuationIndex+0x20
ucrtbase!__FrameHandler4::CxxCallCatchBlock+0x115
ntdll!RcFrameConsolidation+0x6
contoso!<lambda_f370031fe3623a0b308de0bbdeb2db76>::operator()+0x1a
contoso!std::invoke+0x24
contoso!std::_Invoker_ret<void,1>::_Call+0x24
contoso!std::_Func_impl_no_alloc<<lambda_f370031fe3623a0b308de0bbdeb2db76>,
    void,Concurrency::task<void> >::_Do_call+0x28
contoso!std::_Func_class<void,Concurrency::task<void> >::operator()+0x31
contoso!Concurrency::details::_MakeTToUnitFunc::__l2::
    <lambda_64124396551846798083ef48cd389b4a>::operator()+0x46
contoso!std::invoke+0x66
contoso!std::_Invoker_ret<unsigned char,0>::_Call+0x66
contoso!std::_Func_impl_no_alloc<<lambda_64124396551846798083ef48cd389b4a>,
    unsigned char,Concurrency::task<void> >::_Do_call+0x72
contoso!std::_Func_class<unsigned char,Concurrency::task<void> >::
    operator()+0x32
contoso!Concurrency::task<void>::_ContinuationTaskHandle<void,
    void,std::function<void __cdecl(Concurrency::task<void>)>,
    std::integral_constant<bool,1>,Concurrency::details::_TypeSelectorNoAsync>::
    _LogWorkItemAndInvokeUserLambda<std::function<unsigned char __cdecl(
    Concurrency::task<void>)>,Concurrency::task<void> >+0x8b
contoso!Concurrency::task<void>::_ContinuationTaskHandle<void,
    void,std::function<void __cdecl(Concurrency::task<void>)>,
    std::integral_constant<bool,1>,Concurrency::details::_TypeSelectorNoAsync>::
    _Continue+0x8c
contoso!Concurrency::task<void>::_ContinuationTaskHandle<void,
    void,std::function<void __cdecl(Concurrency::task<void>)>,
    std::integral_constant<bool,1>,Concurrency::details::_TypeSelectorNoAsync>::
    _Perform+0x8
contoso!Concurrency::details::_PPLTaskHandle<unsigned char,Concurrency::task<
    void>::_ContinuationTaskHandle<void,void,std::function<
    void __cdecl(Concurrency::task<void>)>,std::integral_constant<bool,1>,
    Concurrency::details::_TypeSelectorNoAsync>,
    Concurrency::details::_ContinuationTaskHandleBase>::invoke+0x37
contoso!Concurrency::details::_TaskProcHandle::_RunChoreBridge+0x25
contoso!Concurrency::details::_DefaultPPLTaskScheduler::_PPLTaskChore::
    _Callback+0x26
msvcp140!Concurrency::details::`anonymous namespace'::
    _Task_scheduler_callback+0x5d
```

```
ntdll!TppWorkpExecuteCallback+0x13a
ntdll!TppWorkerThread+0x8f6
kernel32!BaseThreadInitThunk+0x1d
ntdll!RtlUserThreadStart+0x28
```

Hey, wait a second, this doesn’t look anything like the stack reported by `!analyze`! What’s going on?

The `!analyze` used the stack from the first stowed exception. You can dump all of the stowed exceptions with the `!pde.dse` command.

```
0:076> !pde.dse
Stowed Exception Array @ 0x000000002b1ef170

Stowed Exception #1 @ 0x000000001ce068e8
    0x80070005 (FACILITY_WIN32 - Win32 Undecorated Error Codes):
    E_ACCESSDENIED - General access denied error

    Stack    : 0x2b214de0
    contoso!winrt::hresult_error::hresult_error+0x143
    contoso!winrt::throw_hresult+0x132
    contoso!winrt::impl::consume_LitWare_IIconProvider
        <winrt::LitWare::IIconProvider>::LoadIcon+0x3b
    contoso!winrt::Contoso::implementation::IconDataModel::
        ReloadIcon$_ResumeCoro$1+0x214
    contoso!winrt::impl::resume_background_callback+0x10
    ntdll!TppSimplepExecuteCallback+0xa3
    ntdll!TppWorkerThread+0x8f6
    kernel32!BaseThreadInitThunk+0x1d
    ntdll!RtlUserThreadStart+0x28

Stowed Exception #2 @ 0x000000001ce02378
    0x80070005 (FACILITY_WIN32 - Win32 Undecorated Error Codes):
    E_ACCESSDENIED - General access denied error

    Stack    : 0x12cda890
    litware!winrt::hresult_error::hresult_error+0x12c
    litware!winrt::throw_hresult+0x83
    litware!winrt::LitWare::implementation::IconProvider::LoadIcon+0x90
    litware!winrt::impl::produce<winrt::LitWare::implementation::IconProvider,
        winrt::LitWare::IIconProvider>::LoadIcon+0x1b
    contoso!winrt::impl::consume_LitWare_IIconProvider
        <winrt::LitWare::IIconProvider>::LoadIcon+0x3b
    contoso!winrt::Contoso::implementation::IconDataModel::
        ReloadIcon$_ResumeCoro$1+0x214
    contoso!winrt::impl::resume_background_callback+0x10
    ntdll!TppSimplepExecuteCallback+0xa3
    ntdll!TppWorkerThread+0x8f6
    kernel32!BaseThreadInitThunk+0x1d
    ntdll!RtlUserThreadStart+0x28

Stowed Exception #3 @ 0x000000001ce04fa8
    0x80070005 (FACILITY_WIN32 - Win32 Undecorated Error Codes):
    E_ACCESSDENIED - General access denied error

    Stack    : 0x1d94b410
    combase!RoOriginateError+0x51
    contoso!wil::details::RaiseRoOriginateOnWilExceptions+0x137
    contoso!wil::details::ReportFailure_Return<1>+0x1b8
    contoso!wil::details::ReportFailure_Win32<1>+0x70
    contoso!wil::details::in1diag3::Return_Win32+0x18
    contoso!Internal::ContosoSettingsStorage::Save+0xdc729
```

```
contoso!Internal::ContosoSettings::SaveToDefaultLocalStorage+0xf1
contoso!Internal::ContosoSettings::Save+0x4ef
contoso!Contoso::AppSettings::save+0x4ef
contoso!std::_Func_impl_no_alloc<<lambda_f4300885c0b58e31cf789c4999ed9d7a>,
    void>::_Do_call+0x2b
contoso!std::_Func_impl_no_alloc<<lambda_052e919cc0e5399df76dff3972c0cac1>,
    unsigned char>::_Do_call+0x28
contoso!Concurrency::task<unsigned char>::_InitialTaskHandle<void,
    <lambda_f4300885c0b58e31cf789c4999ed9d7a>,
    Concurrency::details::_TypeSelectorNoAsync>::_Init+0xc3
contoso!Concurrency::details::_PPLTaskHandle<unsigned char,
    Concurrency::task<unsigned char>::_InitialTaskHandle<void,
    <lambda_f4300885c0b58e31cf789c4999ed9d7a>,
    Concurrency::details::_TypeSelectorNoAsync>,
    Concurrency::details::_TaskProcHandle>::invoke+0x55
contoso!Concurrency::details::_TaskProcHandle::_RunChoreBridge+0x25
contoso!Concurrency::details::_DefaultPPLTaskScheduler::_PPLTaskChore::
    _Callback+0x26
msvcp140!Concurrency::details::`anonymous namespace'::
    _Task_scheduler_callback+0x5d
ntdll!TppWorkpExecuteCallback+0x13a
ntdll!TppWorkerThread+0x686
kernel32!BaseThreadInitThunk+0x10
ntdll!RtlUserThreadStart+0x2b
```

Now things are starting to come together.

The rule of thumb for throwing Windows Runtime exceptions is that before you throw the exception or return the failure `HRESULT`, you call `RoOriginateError` to capture the stack and other context. It is common when working with the Windows Runtime that the exception is caught and saved (“stowed”), usually in an `IAsyncAction` or similar interface, and then later, when the caller does a `co_await` or similar operation. the exception is rethrown.

When the exception is rethrown, the original stack has already unwound, so there is nothing on the stack to trace. Calling `RoOriginateError` captures the stack at the point of failure before it’s too late. This information can then be used to “stitch together” the exception lifetime, starting with the code that threw the exception and ending with the code that tried (and failed) to catch it.

The system does this stitching by storing error history in per-thread data, allowing components to capture that history and transfer it to another thread when a task’s error state moves between threads, and looking for errors with the same `HRESULT`s.[1] If there is a recent captured stack for an `HRESULT` that matches the `HRESULT` of the exception that went unhandled, then the system says, “I bet these two belong together.”

Usually, all of this stack-stitching works out well because our API design principles say that exceptions should not be thrown for recoverable errors. This means that there generally not a lot of exception traffic, so the rate of false positives is low.

But in this case, we had a false positive: The `IconDataModel` called `IconProvider::LoadIcon()`, which failed with `E_ACCESSDENIED`. This exception was then caught and handled. We see this from the top two stowed exceptions, using what we learned a little while ago about stitching together multiple error stacks get a fuller picture of what led to a failure.

In this case, the `IconProvider::LoadIcon()` explicitly threw an exception with `throw_hresult` (Stowed Exception #2), which then was converted from a C++ exception to an `HRESULT` at the ABI boundary, and then on the other side, C++/WinRT turned the `HRESULT` back into an exception and rethrew it (Stowed Exception #1). This rethrown exception was then caught by the `catch (...)`, and that's the end of that exception.

That's not what caused us to crash.

The current active stack shows that we raised a fail-fast exception from a lambda. The debugger tells us that it's this lambda:

```
void ViewPreferences::SaveChanges()
{
    m_settings.save_async()
    .then([](concurrency::task<void> precedingTask) {
        try
        {
            precedingTask.get();
        }
        CATCH_FAIL_FAST();
    });
}
```

The code saves the settings and fails fast if the operation failed.

And we see that failure in the *third* stack, the one with `ContosoSettingsStorage::Save`. That `Save` operation failed with `E_ACCESSDENIED`, and it was logged in the failure history.

What happened is that there were *two* `E_ACCESSDENIED` errors that occurred at roughly the same time, and `!analyze`'s attempt to figure out which stacks belonged to which sequence was not completely successful, and it thought that the current failure matched up with the `m_provider.LoadIcon()` failure. But we, using our human brains, saw that the `m_provider.LoadIcon()` exception was handled, and the real culprit was the Stowed Exception #3.

[1] You can call the function `RoTransformError` if your code receives one error code and returns a different one. This tells COM error-tracking that these two error sequences should be stitched together to form one large error sequence.