My code crashed when I asked WIL to convert an exception to an HRESULT, did I throw an improper exception type?

devblogs.microsoft.com/oldnewthing/20211020-00

October 20, 2021

Raymond Chen

A customer used WRL to implement their COM objects, with the help of wil. Meanwhile, they also had client code that instantiated one of those COM objects. An exception was thrown and caught, but the code crashed trying to convert it to an `HRESULT`:

```
// Simplified
int __cdecl wmain()
{
    CoInitialize(nullptr);
    HRESULT result;
    try {
        auto widget = wil::CoCreateInstance<ContosoWidget,
                                            IContosoWidget>();
        result = widget->ReversePolarity();
    } catch (...) {
        result = wil::ResultFromCaughtException(); // crash here
    }

    if (SUCCEEDED(result)) {
        return 0;
    }

    DisplayErrorMessage(result);
    return 1;
}
```

The crash looked like this:

```
ucrtbase!FindHandler<__FrameHandler4>+0x2d3:
00007ff9`c3ffec2f movsxd  r13,dword ptr [rcx+0Ch]
                                  ds:00007ff9`abf00b94=????????

ucrtbase!FindHandler<__FrameHandler4>+0x2d3
ucrtbase!__InternalCxxFrameHandler<__FrameHandler4>+0x278
ucrtbase!__CxxFrameHandler4+0xa0
ntdll!RtlpExecuteHandlerForException+0xf
ntdll!RtlDispatchException+0x244
ntdll!RtlRaiseException+0x185
KERNELBASE!RaiseException+0x69
ucrtbase!_CxxThrowException+0xad
client!wil::details::ResultFromCaughtExceptionInternal+0xad
client!wil::ResultFromCaughtException+0x33
client!wmain$catch$6+0x12
ucrtbase!_CallSettingFrame_LookupContinuationIndex+0x20
ucrtbase!__FrameHandler4::CxxCallCatchBlock+0x10d
ntdll!RcFrameConsolidation+0x6
client!wmain+0x99
client!invoke_main+0x22
client!__scrt_common_main_seh+0x10c
kernel32!BaseThreadInitThunk+0x14
ntdll!RtlUserThreadStart+0x21
```

The default behavior of `wil::ResultFromCaughtException` is to fail fast on an unrecognized exception. But the above looks like a crash, not a fail-fast exception. Is that what fail-fasts look like now? How can we dig out the exception type that went unrecognized?

The crash we see is not a fail-fast exception. What happened is that we crashed while trying to decode the exception. We haven't gotten to the point of rejecting it and failing fast; we don't yet know what it is!

Visual Studio comes with source code for the `FindHandler` function (in `frame.cpp`), so we can use that to help us figure out where things blew up. In fact, all we need is the function prototype:

```
template <class T>
static void FindHandler(
    EHExceptionRecord *pExcept, // Information for this (logical)
                                //   exception
    EHRegistrationNode *pRN,    // ...
    CONTEXT *pContext,          // Context info
    /* other parameters not interesting */
)
```

The valuable information is the `pExcept`, which tells us the exception we're trying to handle, and the `pContext` which tells us who threw it.

The Windows x86-64 calling convention passes the first parameter in `rcx` and the third parameter in `r8`, so those are the ones we need to track.

```
0:000> u .-2d3
ucrtbase!FindHandler<__FrameHandler4>
00007ff9`c3ffe95c push    rbp
00007ff9`c3ffe95e push    rbx
00007ff9`c3ffe95f push    rsi
00007ff9`c3ffe960 push    rdi
00007ff9`c3ffe961 push    r12
00007ff9`c3ffe963 push    r13
00007ff9`c3ffe965 push    r14
00007ff9`c3ffe967 push    r15
00007ff9`c3ffe969 lea     rbp,[rsp-88h]
00007ff9`c3ffe971 sub     rsp,188h
00007ff9`c3ffe978 mov     rax,qword ptr [ucrtbase!__security_cookie
(00007ff9`c40af450)]
00007ff9`c3ffe97f xor     rax,rsp
00007ff9`c3ffe982 mov     qword ptr [rbp+70h],rax
00007ff9`c3ffe986 mov     rax,qword ptr [rbp+108h]
00007ff9`c3ffe98d mov     r12,rdx
00007ff9`c3ffe990 mov     r14,qword ptr [rbp+0F0h]
00007ff9`c3ffe997 mov     rbx,rcx // exception
00007ff9`c3ffe99a mov     qword ptr [rbp-60h],rdx
00007ff9`c3ffe99e xor     r13b,r13b
00007ff9`c3ffe9a1 mov     rcx,r14
00007ff9`c3ffe9a4 mov     qword ptr [rsp+70h],r8 // context
00007ff9`c3ffe9a9 mov     rdx,r9
00007ff9`c3ffe9ac mov     qword ptr [rbp-80h],rax
00007ff9`c3ffe9b0 mov     rsi,r9
```

Okay, so the exception pointer is in `rbx` and the context pointer is on the stack at `rsp+70h`.

```
0:000> .exr @rbx
ExceptionAddress: 00007ff9c3a64ed9 (KERNELBASE!RaiseException+0x0000000000000069)
   ExceptionCode: e06d7363 (C++ EH exception)
  ExceptionFlags: 00000001
NumberParameters: 4
   Parameter[0]: 0000000019930520
   Parameter[1]: 000000638477c910
   Parameter[2]: 00007ff9abf00b88
   Parameter[3]: 00007ff9abec0000
  pExceptionObject: 000000638477c910
  _s_ThrowInfo    : 00007ff9abf00b88
```

The `.exr` command was kind enough to decode the parameters of the thrown C++ exception and give us the exception object. Let's look at that exception object:

```
0:000> dps 000000638477c910
00000063`8477c910  00007ff9`abef3110 <Unloaded_contoso.dll>+0x33110
00000063`8477c918  00000000`00000000
00000063`8477c920  00000000`00000000
00000063`8477c928  80070005`00000000
```

Uh-oh.

What happened here is that `contoso.dll` threw an exception, and it escaped the module and was caught by `client!wmain`. And as the stack unwound, at some point the DLL got unloaded, so when `client!wmain` tried to inspect the object, it crashed trying to figure out *what it was*.

The `ucrtbase!FindHandler<__FrameHandler4>` appears to be going through the same exercise I described some time ago when I explained how to decode the parameters of a thrown C++ exception. But it crashed before it could get to the type information.

Notice the value `0x80070005`, which corresponds to `E_ACCESS_DENIED`. There's a good chance that that is the exception being thrown.

We hope that the exception that was thrown was a WIL exception. (Remember, debugging is an exercise in optimism.) Since `client.dll` also uses WIL, we can use `client.dll` debugging information to parse the `wil::ResultException`:

```
0:000> dt client!wil::ResultException
   +0x000 __VFN_table : Ptr64
   +0x008 _Data            : __std_exception_data
   +0x018 m_failure        : wil::StoredFailureInfo
   +0x0b8 m_what           : wil::details::shared_buffer
0:000> ?? ((client!wil::ResultException*)0x00000063`8477c910)
class wil::ResultException * 0x00000063`8477c910
   +0x000 __VFN_table : 0x00007ff9`abef3110
   +0x008 _Data            : __std_exception_data
   +0x018 m_failure        : wil::StoredFailureInfo
   +0x0b8 m_what           : wil::details::shared_buffer
0:000> ?? ((client!wil::ResultException*)0x00000063`8477c910)->m_failure
class wil::StoredFailureInfo
   +0x000 m_failureInfo    : wil::FailureInfo
   +0x090 m_spStrings      : wil::details::shared_buffer
0:000> ?? ((client!wil::ResultException*)0x00000063`8477c910)-
>m_failure.m_failureInfo
struct wil::FailureInfo
   +0x000 type             : 0 ( Exception )
   +0x004 hr               : 80070005
   +0x008 failureId        : 0n1
   +0x010 pszMessage       : (null)
   +0x018 threadId         : 0x7d830
   +0x020 pszCode          : (null)
   +0x028 pszFunction      : (null)
   +0x030 pszFile          : 0x000001d5`679582e4  "contoso\widget\connection.cpp"
   +0x038 uLineNumber      : 44
   +0x03c cFailureCount    : 0n1
   +0x040 pszCallContext   : (null)
   +0x048 callContextOriginating : wil::CallContextInfo
   +0x060 callContextCurrent : wil::CallContextInfo
   +0x078 pszModule        : 0x000001d5`67958314  "contoso.dll"
   +0x080 returnAddress    : 0x00007ff9`abed698c Void
   +0x088 callerReturnAddress : 0x00007ff9`abed6654 Void
```

Things seem to line up pretty well. Line 44 of `connection.cpp` could indeed throw an exception:

```
Connection::Connection()
{
    ...
    session = wil::CoCreateInstance<ContosoUserSession,
                                    IContosoUserSession>();
    ...
}
```

Let's tell the debugger to load symbols for `contoso.dll` based on its last known address. That will make those return addresses decodable.

```
0:000> !reload /unl contoso.dll

0:000> ln 0x00007ff9`abed698c
(00007ff9`abed6864)   contoso!Microsoft::WRL::Details::
             MakeAndInitialize<ContosoWidget,IUnknown>+0x128

0:000> ln 0x00007ff9`abed6654
(00007ff9`abed6600)   contoso!Microsoft::WRL::
    SimpleClassFactory<ContosoWidget,0>::CreateInstance+0x54
```

The exception was thrown from `MakeAndInitialize`, which strongly suggests that came from the inlined constructor of `ContosoWidget`. The `pContext` will help us confirm this theory. Recall that we learned that the context pointer is on the stack at `rsp+70h`.

```
0:000> .cxr poi(@rsp+70)
rax=00007ff9abef55c8 rbx=00007ff9abf00b88 rcx=000000638477c690
rdx=0000000700000020 rsi=000000638477f0e0 rdi=000000638477c910
rip=00007ff9c3a64ed9 rsp=000000638477c7a0 rbp=000000638477c8e0
 r8=000001d567966d52  r9=0000000000000000 r10=000000638477c179
r11=0000000000000003 r12=0000000000000000 r13=0000000000000000
r14=000000000000002c r15=00007ff9abef6710
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
KERNELBASE!RaiseException+0x69:
00007ff9`c3a64ed9 0f1f440000      nop     dword ptr [rax+rax]
0:000> k
  *** Stack trace for last set context - .thread/.cxr resets it
Child-SP          Call Site
00000063`8477c7a0 KERNELBASE!RaiseException+0x69
00000063`8477c880 ucrtbase!_CxxThrowException+0xad
00000063`8477c8f0 contoso!wil::details::ThrowResultExceptionInternal+0x25
00000063`8477c9f0 contoso!wil::ThrowResultException+0x16
00000063`8477ca20 contoso!wil::details::ReportFailure+0x174
00000063`8477df90 contoso!wil::details::ReportFailure_Hr+0x44
00000063`8477dff0 contoso!wil::details::in1diag3::_Throw_Hr+0x26
(Inline Function) contoso!wil::details::in1diag3::Throw_IfFailed+0x6a
(Inline Function) contoso!Connection::{ctor}+0x98
00000063`8477e040
contoso!Microsoft::WRL::Details::MakeAndInitialize<ContosoWidget,IUnknown>+0x128
00000063`8477e0b0
contoso!Microsoft::WRL::SimpleClassFactory<ContosoWidget,0>::CreateInstance+0x54
00000063`8477e0f0 combase!CServerContextActivator::CreateInstance+0x1d4
00000063`8477e270 combase!ActivationPropertiesIn::DelegateCreateInstance+0x90
00000063`8477e300 combase!CApartmentActivator::CreateInstance+0x9c
00000063`8477e3b0 combase!CProcessActivator::CCICallback+0x58
00000063`8477e400 combase!CProcessActivator::AttemptActivation+0x40
00000063`8477e450 combase!CProcessActivator::ActivateByContext+0x91
00000063`8477e4e0 combase!CProcessActivator::CreateInstance+0x80
00000063`8477e530 combase!ActivationPropertiesIn::DelegateCreateInstance+0x90
00000063`8477e5c0 combase!CClientContextActivator::CreateInstance+0x17f
00000063`8477e870 combase!ActivationPropertiesIn::DelegateCreateInstance+0x90
00000063`8477e900 combase!ICoCreateInstanceEx+0x90a
00000063`8477f7f0 combase!CComActivator::DoCreateInstance+0x169
(Inline Function) combase!CoCreateInstanceEx+0xd1
00000063`8477f950 combase!CoCreateInstance+0x10c
00000063`8477f9f0
client!wil::CoCreateInstance<ContosoWidget,IContosoWidget,wil::err_exception_policy>+0

00000063`8477fa40 client!wmain+0x99
(Inline Function) client!invoke_main+0x22
00000063`8477fac0 client!__scrt_common_main_seh+0x10c
00000063`8477fb00 kernel32!BaseThreadInitThunk+0x14
00000063`8477fb30 ntdll!RtlUserThreadStart+0x21
```

This gives us a much better view of what's going on. The `ContosoWidget` object has a member variable that is a `Connection`, and construction of the `Connection` failed with an exception. The exception propagated out of the constructor, which destructed the partially-constructed `ContosoWidget` and then propagates the exception past `MakeAnd-Initialize`, the COM infrastructure, and was finally caught back in the client.

The WRL library operates at the COM ABI layer, which means that it generally requires that nothing throws exceptions. (There are some places where it does support exceptions, but in general it doesn't.) You can see that it slaps `throw()` around all its COM methods, meaning "These COM method don't throw any exceptions (and I'm trusting you to honor that rule, no enforcement)."

In this case, the exception escaped `contoso.dll` and unwound all the way across `combase.dll`, which violates the rule against throwing exceptions across stack frames you don't control.

What seems to have happened is that as the exception propagated out of COM, COM realized that something bad happened in `contoso.dll`. "Hey there buddy, you feeling okay? Do you want to go home?" COM called `contoso.dll`'s `DllCanUnloadNow` function, and since there were no active COM objects in `contoso.dll`, it said, "Yeah, I'm not needed here. You can unload me."

But there *was* an active object in `contoso.dll`: the exception object that it just threw!

COM unloaded `contoso.dll` so it could go home and get some rest. And then when `client!wmain` caught the exception and tried to interrogate it, it crashed because the exception source had already been unloaded.

The fix here is not to throw exceptions from constructors of COM objects that use `WRL` as their factory, because the WRL factory is just going to wave good-bye to the exception as it leaves the DLL. (It doesn't have much choice, seeing as it has no way of interpreting the exception andto convert it to an `HRESULT`.)

If you want to fail the creation of an object, you can do so by moving all the potentially-failable things out of the constructor and into the `RuntimeClassInitialize` method. You can have that method return a failure `HRESULT` when it is not happy.

We went back to the code and found that in the time since this crash was identified, somebody else had already fixed the bug by accident! The member variable type was changed from `Connection` to `std::unique_ptr<Connection>`, and the `Connection` itself was created on demand rather than in the constructor. The reason for the change was commented as

```
// Establish a connection on first use. This is done here instead of the
// constructor so we can return a meaningful HRESULT to the caller.
```

Moving the failure out of the constructor also has the nice benefit of not crashing.

Raymond Chen

Follow