The case of the fail-fast trying to log a caught exception

devblogs.microsoft.com/oldnewthing/20240119-00

January 19, 2024

Raymond Chen

A customer had a bunch of crash dumps that showed that their code had caught an exception but then crashed anyway while trying to log the exception.

Here is the stack trace at the point the crash dump was captured:

```
00 KERNELBASE!RaiseFailFastException+0x152
01 contoso!wil::details::WilDynamicLoadRaiseFailFastException+0x49
02 contoso!wil::details::WilRaiseFailFastException+0x18
03 contoso!wil::details::WilFailFast+0xa8
04 contoso!wil::details::ReportFailure_NoReturn<3>+0x7e
05 contoso!wil::details::ReportFailure_Base<3,0>+0x30
06 contoso!wil::details::ReportFailure_CaughtExceptionCommon<2>+0x11c
07 contoso!wil::details::ReportFailure_CaughtException<2>+0x52
08 contoso!wil::details::in1diag3::Log_CaughtException+0x13
09 contoso!`Contoso::SpeechRecognition::Initialize'::`1'::catch$258+0x3f
0a VCRUNTIME140_1_APP!_CallSettingFrame_LookupContinuationIndex+0x20
0b VCRUNTIME140_1_APP!__FrameHandler4::CxxCallCatchBlock+0x115
0c ntdll!RcFrameConsolidation+0x6
0d contoso!Contoso::SpeechRecognition::Initialize+0x64
...
```

Reading the stack trace from the bottom up gives us the timeline.

We are running the function `SpeechRecognition::Initialize`. We must have encountered an exception because we reached `CxxCallCatchBlock`, which is a C++ runtime function whose job is to, well, call the `catch` block.

We are inside the `catch` block, which is named after the function, adding a `catch` suffix.

The `catch` block is trying to log the caught exception, and that's where we ran into trouble. Here's the code for the `Initialize` method:

```
bool SpeechRecognition::Initialize()
{
    ⟦ ... ⟧
    try
    {
        ⟦ ... lots of code ... ⟧
        return true;
    }
    catch (...)
    {
        LOG_CAUGHT_EXCEPTION();
        return false;
    }
}
```

We are crashing at the `LOG_CAUGHT_EXCEPTION()` macro that is part of the Windows Implementation Library (WIL). We learned that WIL fails fast if asked to handle an unrecognized exception. Is that what happened here?

Let's find out by looking at what was thrown.

```
0:008> .frame b
0b VCRUNTIME140_1_APP!__FrameHandler4::CxxCallCatchBlock+0x115
0:008> dv
              pExcept = 0x00000011`cd17ba40
continuationAddresses = void *[2]
      TranslatedCatch = 0n0
    pForeignException = 0x00000000`00000000
              rethrow = 0n0
            FrameInfo = struct FrameInfo
   HandlerSearchState = 0n111
       pSaveException = 0x00000000`00000000
  continuationAddress = 0x00000000`00000000
    pEstablisherFrame = 0x00000011`cd17bba8
         pSaveContext = 0x00000000`00000000
           pFrameInfo = 0x00000011`cd17ad18
       handlerAddress = 0x00007fff`486d1eeb
             FuncInfo = struct FH4::FuncInfo4
             pContext = 0x00000011`cd17c670
       UnwindTryState = 0n7
       pThisException = 0x00000011`cd17ce30
```

Now we have to play a guessing game: Find the exception!

You would think that `pExcept` would be the exception. But it doesn't seem to be:

```
0:008> ?? pExcept
struct _EXCEPTION_RECORD * 0x00000011`cd17ba40
   +0x000 ExceptionCode    : 0x80000029
   +0x004 ExceptionFlags   : 0x22
   +0x008 ExceptionRecord  : (null)
   +0x010 ExceptionAddress : (null)
   +0x018 NumberParameters : 0xf
   +0x020 ExceptionInformation : [15] 0x00007fff`52af2590
```

Exception code `0x80000029` is `STATUS_UNWIND_CONSOLIDATE`, which isn't interesting to us. That's telling us that we are unwinding due to a caught exception, but we knew that already. We really want to find the exception that triggered the unwinding.

Maybe `pThisException` will tell us.

```
0:008> ?? pThisException
struct EHExceptionRecord * 0x00000011`cd17ce30
   +0x000 ExceptionCode    : 0xe06d7363
   +0x004 ExceptionFlags   : 0x81
   +0x008 ExceptionRecord  : (null)
   +0x010 ExceptionAddress : 0x00007fff`6d2b520c Void
   +0x018 NumberParameters : 4
   +0x020 params           : EHExceptionRecord::EHParameters
0:008> .exr 0x00000011`cd17ce30
ExceptionAddress: 00007fff6d2b520c (KERNELBASE!RaiseException+0x000000000000006c)
   ExceptionCode: e06d7363 (C++ EH exception)
  ExceptionFlags: 00000081
NumberParameters: 4
   Parameter[0]: 0000000019930520
   Parameter[1]: 00000011cd17cf70
   Parameter[2]: 00007fff486fb558
   Parameter[3]: 00007fff48670000
  pExceptionObject: 00000011cd17cf70
  _s_ThrowInfo    : 00007fff486fb558
```

Okay, this looks a lot more like it. The exception code is `0xe06d7363`, which is the magic value used by Visual C++ for C++ exceptions. And the debugger kindly extracted the thing that was thrown, the `pExceptionObject`. But first, we want to find out what the type is, because that's the thing that WIL was unable to recognize.

We can follow the cookbook for decoding the parameters of a thrown exception.

```
0:008> * dump the fourth DWORD at the _s_ThrowInfo
0:008> dc 00007fff486fb558 L4
00007fff`486fb558  00000000 00000000 00000000 0008b548  ...........H...
0:008> * Add that to Parameter[3] and dump the second DWORD
0:008> dc 00007fff48670000+0008b548 L2
00007fff`486fb548  00000001 0008b520                    .... ...
0:008> * Add that to Parameter[3] and dump the second DWORD (again)
0:008> dc 00007fff48670000+0008b520 L2
00007fff`486fb520  00000001 000920f0                    ..... ..
0:008> * Add that to Parameter[3] and dump the string at offset 0x10
0:008> dc 00007fff48670000+000920f0+10
00007fff`48702100  004b5f2e 00000000 486db370 00007fff  ._K.....p.mH....
```

Huh, the type name is `._K`. That's weird.

My guess is that this is a primitive type, because C++ name mangling uses special short names for primitive types.[1]

Let's see what was thrown.

```
0:008> dps 00000011cd17cf70
00000011`cd17cf70  00000000`00000003
00000011`cd17cf78  00000000`00000003
00000011`cd17cf80  00000000`00000003
```

It's the number three.

Just the number three.

We can see more about the code that threw the exception by using the `pContext`:

```
0:008> .cxr 0x00000011`cd17c670
rax=00007fff3a9ab31d rbx=00007fff486fb558 rcx=00000011cd17c750
rdx=00007fff3a951de6 rsi=00000011cd17cf70 rdi=0000000019930520
rip=00007fff6d2b520c rsp=00000011cd17ce10 rbp=000001fef78d2308
 r8=00007fff3aa5bf64  r9=00007fff3aae3c50 r10=00007fff3aa5cb31
r11=00000011cd17c7d0 r12=000001fef78d2298 r13=000001fef78d2278
r14=000001fef78d22f8 r15=00000000000000ea
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
KERNELBASE!RaiseException+0x6c:
00007fff`6d2b520c 0f1f440000      nop     dword ptr [rax+rax]
0:008> kn
  *** Stack trace for last set context - .thread/.cxr resets it
 # Call Site
00 KERNELBASE!RaiseException+0x6c
01 VCRUNTIME140_APP!_CxxThrowException+0x90
02 contoso!__azac_handle_native_ex+0x50
03 contoso!__azac_rethrow+0x8
04 contoso!Microsoft::CognitiveServices::Speech::Intent::IntentRecognizer::FromConfig+0xaa
05 contoso!Contoso::SpeechRecognition::Initialize+0x64
...
```

The exception hasn't unwound yet, so we can still access the stack of the `throw`.

```
0:008> * This is the C++ runtime support for throwing an exception
0:008> .frame 1
01 VCRUNTIME140_APP!_CxxThrowException+0x90
0:008> dv
pExceptionObject = 0x00000011`cd17cf70
      pThrowInfo = <value unavailable>
     magicNumber = 0x19930520
      parameters = unsigned int64 [4]
             pTI = 0x00007fff`486fb558
  throwImageBase = 0x00007fff`48670000
            pWei = <value unavailable>
           ppWei = <value unavailable>
0:008> * And here comes the function that did the throwing
0:008> .frame 2
02 contoso!__azac_handle_native_ex+0x50
0:008> dv
              hr = 3
           error = 0
       errorMsg = class std::basic_string<char,std::char_traits<char>,std::allocator<char> >
      callstack = <value unavailable>
           what = <value unavailable>
Unimplemented error for throwException
```

Here's the code from IntentRecognizer:

```
class IntentRecognizer :
    public AsyncRecognizer<
        IntentRecognitionResult,
        IntentRecognitionEventArgs,
        IntentRecognitionCanceledEventArgs>
{
    ⟦ ... ⟧

    static std::shared_ptr<IntentRecognizer>
        FromConfig(
            std::shared_ptr<SpeechConfig> speechConfig,
            std::shared_ptr<Audio::AudioConfig> audioInput = nullptr)
    {
        SPXRECOHANDLE hreco;
        SPX_THROW_ON_FAIL(::recognizer_create_intent_recognizer_from_config(
            &hreco,
            HandleOrInvalid<SPXSPEECHCONFIGHANDLE, SpeechConfig>(speechConfig),
            HandleOrInvalid<SPXAUDIOCONFIGHANDLE, Audio::AudioConfig>(audioInput)));
        return std::make_shared<IntentRecognizer>(hreco);
    }

    ⟦ ... ⟧
};
```

The `recognizer_create_intent_recognizer_from_config` function failed, and the `SPX_THROW_ON_FAIL` macro threw an exception. But what is `SPX_THROW_ON_FAIL`? We find it in `spxdebug.h`:

```
#define SPX_THROW_ON_FAIL(hr) __SPX_THROW_ON_FAIL(hr)
```

which is in turn defined as:

```
#define __SPX_THROW_ON_FAIL(hr) \
        __AZAC_THROW_ON_FAIL(hr)
```

which we then chase into `azac_debug.h`:

```
#define __AZAC_THROW_ON_FAIL(hr)  \
    do {                          \
        AZACHR x = hr;            \
        if (AZAC_FAILED(x)) {     \
            __AZAC_THROW_HR(x);   \
    } } while (0)
```

And `__AZAC_THROW_HR` is

```
#ifndef __AZAC_THROW_HR
#define __AZAC_THROW_HR(hr) __AZAC_THROW_HR_IMPL(hr)
#endif
```

Keep chasing:

```
#ifndef __AZAC_THROW_HR_IMPL
#define __AZAC_THROW_HR_IMPL(hr) __azac_rethrow(hr)
#endif
```

Just a little further:

```
inline void __azac_rethrow(AZACHR hr)
{
    __azac_handle_native_ex(hr, true);
}
```

And finally, we reach the code that throws something.

```
inline void __azac_handle_native_ex(AZACHR hr, bool throwException)
{
    AZAC_TRACE_SCOPE(__FUNCTION__, __FUNCTION__);

    auto handle = reinterpret_cast<AZAC_HANDLE>(hr);
    auto error = error_get_error_code(handle);
    if (error == AZAC_ERR_NONE)
    {
        if (throwException)
        {
            throw hr;
        }
        ⟦ ... ⟧
    }
    ⟦ ... ⟧
}
```

If we look at the disassembly, we can see the mysterious `_K`.

```
00007fff`486913b3 lea     rdx,[contoso!TI1_K (00007fff`486fb558)]
00007fff`486913ba lea     rcx,[rsp+20h]
00007fff`486913bf call    contoso!CxxThrowException (00007fff`48692524)
00007fff`486913c4 int     3
```

And we can look up what a `AZACHR` is:

```
typedef uintptr_t AZACHR;
```

So we threw error 3. Just out of curiosity, I looked up what that is.

```
#define AZAC_ERR_UNHANDLED_EXCEPTION  __AZAC_ERRCODE_FAILED(0x003)
```

So it was some unhandled exception.

The problem, therefore, is that the Azure Speech Cognitive Services header files throw a `uintptr_t`, which is not something WIL understands.

One option is to teach WIL how to convert these `AZACHR` exceptions into `HRESULT`s. I'm not a fan of this for a few reasons.

- You are catching thrown integers, and who knows what other code just throws integers, seeing as integers aren't strongly typed. Maybe somebody else is throwing integers that represent something that isn't an `AZACHR`.
- You can install only one WIL "exception converter" per DLL, and maybe there's some other library that is used by this DLL that also throws some custom exception.

Instead, I would use what appears to be an extension point to let me define my own `__AZAC_THROW_HR` macro to throw a WIL exception.

```
// Do this before including Azure Cognitive Services headers
#define __AZAC_THROW_HR(hr) \
    THROW_HR(MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF, hr))
```

Note: This is just me reading the headers. You should check the documentation to confirm that providing a custom exception thrower is supported. I tried to read the Azure Cognitive Services SDK documentation and I couldn't find any discussion of exceptions at all! Maybe you will have better luck.

[1] Somebody appears to have gone to some effort to reverse-engineer the Visual Studio name-mangling system, and according to this table, the code `_K` means `unsigned __int64`, which agrees with what we discovered by other means.