The case of the crash during the release of an object from an unloaded DLL during apartment rundown

devblogs.microsoft.com/oldnewthing/20210108-00

January 8, 2021

Raymond Chen

A Windows component was experiencing a crash in its service. Here's the stack trace:

```
Call Site
ntdll!RtlUnhandledExceptionFilter2+0x364
KERNELBASE!UnhandledExceptionFilter+0x1f1
ntdll!RtlpThreadExceptionFilter+0x65
ntdll!RtlUserThreadStart$filt$0+0x76
ntdll!__C_specific_handler+0x96
ntdll!RtlpExecuteHandlerForException+0xf
ntdll!RtlDispatchException+0x21c
ntdll!KiUserExceptionDispatch+0x2e
combase!CStdMarshal::DisconnectSrvIPIDs::__l35::
<lambda_03ceb3c306c371a8ea5da27fc98e7b7c>::operator()+0x11
combase!ObjectMethodExceptionHandlingAction<<lambda_03ceb3c306c371a8ea5da27fc98e7b7c>
>+0x2e
combase!CStdMarshal::DisconnectSrvIPIDs+0x3fb
combase!CStdMarshal::DisconnectWorker_ReleasesLock+0x757
combase!CStdMarshal::DisconnectSwitch_ReleasesLock+0x1c
combase!CStdMarshal::DisconnectAndReleaseWorker_ReleasesLock+0x32
combase!COIDTable::ThreadCleanup+0x130
combase!FinishShutdown::__l2::
<lambda_eb459d6b43445c5cc6a7489c5b769eeb>::operator()+0x5
combase!ObjectMethodExceptionHandlingAction<<lambda_eb459d6b43445c5cc6a7489c5b769eeb>
>+0x9
combase!FinishShutdown+0x78
combase!NAUninitialize+0x5e
combase!ApartmentUninitialize+0x177
combase!wCoUninitialize+0x1c4
combase!CoUninitialize+0xeb
svchost!SvcHostMain+0x328
svchost!wmain+0x9
svchost!__wmainCRTStartup+0x74
kernel32!BaseThreadInitThunk+0x14
ntdll!RtlUserThreadStart+0x2b
```

First let's understand what the stack is telling us.

Reading from the bottom, we see that the service host calls `CoUninitialize` to uninitialize COM, presumably because the service is shutting down. This goes into `combase` and it's doing a bunch of cleanup work. Eventually, it gets into `DisconnectSrvIPIDs`.

When you go digging into COM, you'll run into a bunch of weird acronymy IDs. Here are the ones you're most likely to bump into:

Term	Meaning
MID	Machine identifier
OXID	Object exporter identifier
OID	Object identifier
IPID	Interface pointer identifier

Object exporter is a fancy name for *COM apartment*.

The tuple of (MID, OXID, OID, IPID) uniquely identify an instance of an interface anywhere in the known COM universe.

When an apartment is shut down, one of the things that COM needs to do is *run down* objects. *Running down* is just a fancy way of saying "shut down in an organized way". In this case, it means that any outstanding clients are disconnected so that they can't call back into the object any more. This in turn causes the underlying object to be released, at which point is is most likely going to destroy itself.

We crashed during this disconnection process. Since we are in `RtlUnhandledException-Filter2`, this suggests that we are in an exception filter, and the first parameter to the exception filter is a pointer to an `EXCEPTION_POINTERS` structure, which is just a pair of pointers, one to the exception record and one to the context.

We are interested in the context, because that lets us see the underlying exception. How can we fish it out?

This is a 64-bit process, and the `EXCEPTION_POINTERS` pointer is the first parameter, so it came in the `rcx` register. Let's see if we can see what the function did with that register:

```
ntdll!RtlUnhandledExceptionFilter2:
    mov     qword ptr [rsp+10h],rdx
    mov     qword ptr [rsp+8],rcx < Went onto the stack
    push    rbx
    push    rsi
    push    rdi
    push    r12
    push    r13
    push    r14
    push    r15
    sub     rsp,40h
    mov     r13,rdx
    mov     r14,rcx < Went into the r14 register
    mov     rax,qword ptr gs:[60h]
    mov     r15,qword ptr [rax+20h]
    xor     esi,esi
    test    r15,r15
    je      ntdll!RtlUnhandledExceptionFilter2+0x39
```

The value in the `rcx` register got saved in two places: It went into the home space on the stack, and it was also stashed into the `r14` register.

Let's see what's there:

```
0:000> dps @rsp+40+38+8 L1
000000a5`a7c8e470  000000a5`a7c8df40
0:000> dps @r14 L2
000000a5`a7c8df40  000000a5`a7c8ebb0
000000a5`a7c8df48  000000a5`a7c8e6c0
```

Reading the disassembly, we see that the stack pointer is adjusted by seven pushes and an explicit `sub rsp, 40h`, so we need to add `38h + 40h` to the current stack pointer to get back to what the stack pointer was at the start of the function, and then we can add the offset of 8 to that result. The value stored there matches what's in `r14`, which is a nice little confirmation that things are not too far gone.

Dumping the two pointers at `r14` gives us the exception record and the context record. Let's switch to the context in the context record:

```
0:000> .cxr 000000a5`a7c8e6c0
rax=00007fff3a1c4480 rbx=00007fff3a1c4480 rcx=00000285a6c5e6e0
rdx=000000a5a7c8f570 rsi=000000a5a7c8f500 rdi=00007fff40e6c6a8
rip=00007fff40c2f2ca rsp=000000a5a7c8f480 rbp=000000a5a7c8f530
 r8=000000a5a7c8f538  r9=0000000000000003 r10=baac1f10365eb170
r11=4201100001040004 r12=0000000000000008 r13=00000285a6c4b708
r14=0000000000000001 r15=deaddeaddeaddead
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
combase!CStdMarshal::DisconnectSrvIPIDs::__l35::
<lambda_03ceb3c306c371a8ea5da27fc98e7b7c>::operator()+0x11:
00007fff`40c2f2ca 488b4010        mov     rax,qword ptr [rax+10h]
ds:00007fff`3a1c4490=????????????????
0:000>
```

We are calling the Release method on a COM object because the proxy is disconnecting from it. (Since this a 64-bit system, the offsets are `08h` for `AddRef` and `10h` for `Release`.)

We can look at the vtable address to see what this object is.

```
0:000> ln @rax
(00007ff8`1a464480)   <Unloaded_windows.serviceframework.widget>+0x14480
```

As expected, this vtable came from an unloaded module. The system keeps track of the most recently unloaded modules, but the buffer for the file name is fixed in size (to avoid memory allocations). If the name of the DLL were reasonably short, it would all fit into the buffer, and you could use `!reload /unl name.dll` to tell the debugger to pretend that `name.dll` were still in memory so you could resolve addresses within it.

Unfortunately, `windows.serviceframework.widgetservice.dll` is too long to fit in the buffer, so its name gets truncated, and the debugger can't recover it.

We'll have to resolve the symbol manually[1] using a technique I discussed some time ago: Loading the module as if were a dump file and fixing up the addresses.

```
C:\> ntsd -z windows.serviceframework.widgetservice.dll
...
ModLoad: 00000001`80000000 00000001`8001f000
windows.serviceframework.widgetservice.dll
windows_serviceframework_widgetservice!_DllMainCRTStartup:
00000001`80010c70 48895c2408      mov     qword ptr [rsp+8],rbx
ss:00000000`00000008=????????????????
0:000> ln 00000001`80000000+14480
(00000001`80014480)   windows_serviceframework_widgetservice!winrt::impl::produce

<winrt::Windows::ServiceFramework::WidgetService::implementation::ColorChangedEventArg


winrt::Windows::ServiceFramework::WidgetService::IColorChangedEventArgs>::`vftable'
```

Aha, so this object is a `ColorChangedEventArgs` object, and we see that it is implemented in C++/WinRT.

Services that are also COM servers use COM custom contexts so they can disconnect all their objects prior to being unloaded. For this trick to work, all the interfaces they expose to clients must be marshalable, and the objects themselves must not be free-threaded. If the objects are free-threaded (also known as *agile*, short for apartment-agile), then the request for a marshaler would produce the free-threaded marshaler, which says, "Don't worry about marshaling me. You can just take me from any context to any other context without having to do anything special." But this is the opposite of what you want with an object provided by a service DLL, since you want those objects to stay inside the custom context so you can disconnect them at unload.

C++/WinRT objects are free-threaded by default. This particular component was careful to mark its main object with the `non_agile` marker type, thereby preventing it from being free-threaded. However, it forgot to mark some of its helper classes as `non_agile`, and it is one of those helper classes that escaped the custom COM context and therefore escaped being run down when all objects in the context were disconnected.

The fix was to make another pass through the objects offered by the DLL and make sure all of the ones used by the service are marked as non-agile. The unit tests for these helper classes were updated to verify that they are not agile, with the hope that if somebody introduces a new helper class, they will copy an existing unit test to use as a starting point and therefore will copy the agility test.

[1] In retrospect, I probably could have done

```
.reload windows.serviceframework.widgetservice.dll=0x00007ff8`1a450000
```

to tell the debugger to pretend that a DLL was loaded in memory at a particular address.

Raymond Chen

Follow