Identifying an object whose underlying DLL has been unloaded

devblogs.microsoft.com/oldnewthing/20070425-00

April 25, 2007

Raymond Chen

Okay, so I gave it away in the title, but follow along anyway.

Your program chugs along and then suddenly it crashes like this:

```
eax=06bad8e8 ebx=00000000 ecx=1e1cfdf0 edx=00000000 esi=06b9a680 edi=01812950
eip=1180ab57 esp=001178b4 ebp=001178c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
ABC!FunctionX+0x1f:
1180ab57 ff5108          call    dword ptr [ecx+8]    ds:0023:1e1cfdf8=????????
0:000>>
```

Instantly you recognize the following:

- This is a virtual method call. (Call indirect through register plus offset.) — Very high confidence.
- The vtable is in `ecx`. (That is the base register of the indirect call.) — Very high confidence.
- The underlying DLL for this object has been unloaded. (The memory that contains the vtable is not valid and its address is consistent with once having been in valid code.) — High confidence.
- This is a `IUnknown::Release` call. (`Release` is the third function of `IUnknown` and therefore resides at offset 8 on x86.) — High confidence.

Of course, all of the above "instant conclusions" are merely "highly-educated guesses", but life is full of highly-educated guesses. (Every morning, I guess that my plates are still in the cupboard.)

Let's run with our theory that the object was in an unloaded DLL and look for confirmation.

```
0:000> lm
start    end        module name
...
Unloaded modules:
10340000 10348000   DEF.DLL
1e1c0000 1e781000   GHI.DLL
25a90000 25a96000   JKL.DLL
0:000>
```

Aha, our presumed vtable address lies right inside the address space where `GHI.DLL` used to be loaded. Let's see what used to be loaded at that address. For this, I borrow a trick from Doron, namely loading a module as a dump file. This "virtually loads" the library so you can poke around inside it.

```
C:\Program Files\ABC> ntsd -z GHI.DLL
Microsoft (R) Windows Debugger
Copyright (c) Microsoft Corporation. All rights reserved.
Loading Dump File [C:\Program Files\ABC\GHI.DLL]
...
ModLoad: 15800000 15dc1000   C:\Program Files\ABC\GHI.DLL
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=15807366 esp=00000000 ebp=00000000 iopl=0         nv up di pl nz na pe nc
cs=0000  ss=0000  ds=0000  es=0000  fs=0000  gs=0000             efl=00000000
GHI!_DllMainCRTStartup:
15807366 8bff            mov     edi,edi
0:000>
```

That module-load notification tells you where the DLL got virtually-loaded; in our case, it got loaded to 0x15800000. This isn't the same address as it was in our crashed process, so we'll have to do some mental arithmetic to account for the discrepancy.

Going back to the original register dump, we see that our putative vtable is at `ecx=1e1cfdf0` relative to the load address `1e1c0000`. Since our DLL-loaded-as-a-dump-file was loaded at `0x1580000` we need to adjust the address to be relative to the new location.

```
// working with the second copy of ntsd
0:000> ln 0x1580fdf0
(1580fdf0)   GHI!CAlphaStream::`vftable'
```

That magic number `0x1580fdf0` is just the result of some mental arithmetic. First:

```
  0x1e1cfdf0
- 0x1e1c0000
------------
  0x0000fdf0
```

This is the address of the vtable in the crashed process relative to the load address of the DLL in the crashed process. Next:

```
  0x15800000
+0x0000fdf0
  0x1580fdf0
```

This is the address of the vtable in the DLL-loaded-as-a-dump-file relative to the load address of the DLL in the DLL-loaded-as-a-dump-file. The math really isn't that hard, as you can see, since a lot of things cancel out. This happens a lot.

When we asked the debugger to tell us what symbol is nearest to that address, we hit the jackpot: It is exactly a vtable for the `CAlphaStream` object. This confirms our original theory. We can even confirm the `IUnknown::Release` theory by dumping the vtable.

```
0:000> dds 1580fdf0
1580fdf0  159234b3 GHI!CAlphaStream::QueryInterface
1580fdf4  15810539 GHI!CBetaState::AddRef
1580fdf8  15923cfc GHI!CAlphaStream::Release
1580fdfc  15923d30 GHI!CAlphaStream::Read
...
```

Yup, that's a `CAlphaStream` vtable all right.

Since I'm not familiar with the `GHI.DLL` file, let's ask the debugger where the source code is so we can take a closer look:

```
0:000> .lines
Line number information will be loaded
0:000> dds 1580fdf0
1580fdf0  159234b3 GHI!CAlphaStream::QueryInterface
                   [c:\dev\fabricam\synergy\proactive\winwin.cpp @ 2624]
1580fdf4  15810539 GHI!CBetaState::AddRef
                   [c:\dev\fabricam\leverage\paradigm\initiative.cpp @ 427]
1580fdf8  15923cfc GHI!CAlphaStream::Release
                   [c:\dev\fabricam\synergy\proactive\winwin.cpp @ 2638]
1580fdfc  15923d30 GHI!CAlphaStream::Read
                   [c:\dev\fabricam\synergy\proactive\winwin.cpp @ 2649]
```

Now that we know where the source code to `CAlphaStream` is, we can hop on over to take a quick peek and confirm that, oh look, the object doesn't increment the DLL object count when it is constructed (or decrement it when it is destructed). As a result, when COM calls `DllCanUnloadNow`, the `GHI.DLL` says, "Sure, go ahead!" The DLL is unloaded even though `ABC` still has a reference to it, and then when `ABC` goes to release that reference, we crash because `GHI` is already gone.

After I wrote this up, I discovered that Tony Schreiner went through pretty much the same exercise with a third-party Internet Explorer toolbar, except he had the extra bonus challenge of not having source code for the plug-in!

Raymond Chen

Follow