Finding the leaked object reference by scanning memory: Example

devblogs.microsoft.com/oldnewthing/20150109-00

January 9, 2015

Raymond Chen

An assertion failure was hit in some code.

```
// There should be no additional references to the object at this point
assert(m_cRef == 1);
```

But the reference count was 2. That's not good. Where is that extra reference and who took it?

This was not code I was at all familiar with, so I went back to first principles: Let's hope that the reference was not leaked but rather that the reference was taken and not released. And let's hope that the memory hasn't been paged out. (Because debugging is an exercise in optimism.)

```
1: kd> s 0 0fffffff 00 86 ec 00
04effacc  00 86 ec 00 c0 85 ec 00-00 00 00 00 00 00 00 00  ................ // us
0532c318  00 86 ec 00 28 05 00 00-80 6d 32 05 03 00 00 00  ....(....m2..... // rogue
```

The first hit is the reference to the object from the code raising the assertion. The second hit is the interesting one. That's probably the rogue reference. But who is it?

```
1: kd> ln 532c318
1: kd>
```

It does not report as belong to any module, so it's not a global variable.

Is it a reference from a stack variable? If so, then a stack trace of the thread with the active reference may tell us who is holding the reference and why.

```
1: kd> !process -1 4
PROCESS 907ef980  SessionId: 2  Cid: 06cc    Peb: 7f4df000  ParentCid: 0298
    DirBase: 9e983000  ObjectTable: a576f560  HandleCount: 330.
    Image: contoso.exe

        THREAD 8e840080  Cid 06cc.0b78  Teb: 7f4de000 Win32Thread: 9d04b3e0 WAIT
        THREAD 91e24080  Cid 06cc.08d8  Teb: 7f4dd000 Win32Thread: 00000000 WAIT
        THREAD 8e9a3580  Cid 06cc.09f8  Teb: 7f4dc000 Win32Thread: 9d102cc8 WAIT
        THREAD 8e2be080  Cid 06cc.0878  Teb: 7f4db000 Win32Thread: 9d129978 WAIT
        THREAD 82c08080  Cid 06cc.0480  Teb: 7f4da000 Win32Thread: 00000000 WAIT
        THREAD 90552400  Cid 06cc.0f5c  Teb: 7f4d9000 Win32Thread: 9d129628 WAIT
        THREAD 912c9080  Cid 06cc.02ec  Teb: 7f4d8000 Win32Thread: 00000000 WAIT
        THREAD 8e9e8680  Cid 06cc.0130  Teb: 7f4d7000 Win32Thread: 9d129cc8 READY on
processor 0
        THREAD 914b8b80  Cid 06cc.02e8  Teb: 7f4d6000 Win32Thread: 9d12d568 WAIT
        THREAD 9054ab00  Cid 06cc.0294  Teb: 7f4d5000 Win32Thread: 9d12fac0 WAIT
        THREAD 909a2b80  Cid 06cc.0b54  Teb: 7f4d4000 Win32Thread: 00000000 WAIT
        THREAD 90866b80  Cid 06cc.0784  Teb: 7f4d3000 Win32Thread: 93dbb4e0 RUNNING
on processor 1
        THREAD 90cfcb80  Cid 06cc.08c4  Teb: 7f3af000 Win32Thread: 93de0cc8 WAIT
        THREAD 90c39a00  Cid 06cc.0914  Teb: 7f3ae000 Win32Thread: 00000000 WAIT
        THREAD 90629480  Cid 06cc.0bc8  Teb: 7f3ad000 Win32Thread: 00000000 WAIT
```

Now I have to dump the stack boundaries to see whether the address in question lies within the stack range.

```
1: kd> dd 7f4de000 l3
7f4de000  ffffffff 00de0000 00dd0000
1: kd> dd 7f4dd000 l3
7f4dd000  ffffffff 01070000 01060000
...
1: kd> dd 7f4d7000 l3
7f4d7000  ffffffff 04e00000 04df0000 // our stack
...
```

The rogue reference did not land in any of the stack ranges, so it's probably on the heap. Fortunately, since it's on the heap, it's probably part of some larger object. And let's hope (see: optimism) that it's an object with virtual methods.

```
0532c298  73617453
0532c29c  74654d68
0532c2a0  74616461
0532c2a4  446e4961
0532c2a8  00007865
0532c2ac  00000000
0532c2b0  76726553 USER32!_NULL_IMPORT_DESCRIPTOR  (USER32+0xb6553)
0532c2b4  44497265
0532c2b8  45646e49
0532c2bc  41745378 contoso!CMumble::CMumble+0x4c
0532c2c0  00006873
0532c2c4  00000000
0532c2c8  4e616843
0532c2cc  79546567
0532c2d0  4e496570
0532c2d4  00786564
0532c2d8  2856662a
0532c2dc  080a9b87
0532c2e0  00f59fa0
0532c2e4  05326538
0532c2e8  00000000
0532c2ec  00000000
0532c2f0  0000029c
0532c2f4  00000001
0532c2f8  00000230
0532c2fc  fdfdfdfd
0532c300  45ea1370 contoso!CFrumble::`vftable'
0532c304  45ea134c contoso!CFrumble::`vftable'
0532c308  00000000
0532c30c  05b9a040
0532c310  00000002
0532c314  00000001
0532c318  00ec8600
```

Hooray, there is a vtable a few bytes before the pointer, and the contents of the memory do appear to match a `CFrumble` object, so I think we found our culprit.

I was able to hand off the next stage of the investigation (why is a Frumble being created with a reference to the object?) to another team member with more expertise with Frumbles.

(In case anybody cared, the conclusion was that this was a variation of a known bug.)

Raymond Chen

Follow