Investigating an app compat problem: Part 3: Paydirt

devblogs.microsoft.com/oldnewthing/20160610-00

June 10, 2016

Raymond Chen

Last time, we learned that the proximate cause of failure was that we were trying to set a bit in a bit array, except that the `this` pointer was null. That didn't really bring us any closer to the bug. What we need to do is find out why the calling function tried to invoke the method on a null pointer.

The function that generated the null pointer is kind of long, but let's see what we can get out of it.

```
contoso!ContosoInitialize+0x5620:
31426280 push    ebp
31426281 mov     ebp ,esp
31426283 push    0FFFFFFFFh
31426285 push    offset contoso!ContosoInitialize+0xa57c3 (314c6423)
3142628a mov     eax, dword ptr fs:[00000000h]
31426290 push    eax
31426291 mov     dword ptr fs:[0],esp
31426298 sub     esp, 24h       // 24h bytes of local variables
```

So far, we have the standard prologue for functions that use exception handling when compiled by the Microsoft C++ compiler. The compiler uses `[ebp-4]` to keep track of what objects need to be unwound if an exception is raised, so don't be surprised to see apparent write-only operations to `[ebp-4]`. These writes are actually clues to us that a stack object with a nontrivial destructor was just constructed or destructed.

```
3142629b mov     dword ptr [ebp-2Ch], ecx   // this
3142629e mov     eax, dword ptr [ebp-2Ch]
314262a1 mov     ecx, dword ptr [eax+400h]  // this->m_tlsIndex
314262a7 cmp     ecx, dword ptr [contoso!ContosoInitialize+0xe5d3c (3150699c)] //
some global variable
```

We learned from our first day that the value at offset `0x400` is a TLS slot index. We compare it against some global variable. What's up with that global variable?

Let's search the DLL for all references to that global variable.

```
0:000> s 31410000 3151e000 9c 69 50 31
3142592c  9c 69 50 31 89 08 8b 55-fc c7 42 04 00 00 00 00  .iP1...U..B.....
31425b6d  9c 69 50 31 74 0c 8b 4d-f0 e8 75 01 00 00 85 c0  .iP1t..M..u.....
31425f9b  9c 69 d2 31 74 11 8b 55-fc 8b 84 95 f8 fe ff ff  .iP1t..U........
31425ffe  9c 69 50 31 89 91 00 04-00 00 8b 45 fc 8b e5 5d  .iP1.......E...]
314262a9  9c 69 50 31 75 71 8b 15-ac 69 d2 1f 52 8d 4d e8  .iP1uq...i..R.M.
314262d0  9c 69 50 31 75 3b 6a 00-68 c0 5f c4 1f 8b 55 d4  .iP1u;j.h._...U.
0:000> u 3142592c-2 3142592c
contoso!ContosoInitialize+0x4cca:
3142592a mov     ecx, dword ptr [contoso!ContosoInitialize+0xe5d3c (3150699c)]
0:000> u 31425b6d-2 31425b6d
contoso!ContosoInitialize+0x4f0b:
31425b6b cmp     eax, dword ptr [contoso!ContosoInitialize+0xe5d3c (3150699c)]
0:000> u 31425f9b-2 31425f9b
contoso!ContosoInitialize+0x5339:
31425f99 cmp     ecx, dword ptr [contoso!ContosoInitialize+0xe5d3c (3150699c)]
0:000> u 31425ffe-2 31425ffe
contoso!ContosoInitialize+0x539c:
31425ffc mov     edx, dword ptr [contoso!ContosoInitialize+0xe5d3c (3150699c)]
0:000> u 314262a9-2 314262a9
contoso!ContosoInitialize+0x5647:
314262a7 cmp     ecx, dword ptr [contoso!ContosoInitialize+0xe5d3c (3150699c)]
0:000> u 314262d0-2 314262d0
contoso!ContosoInitialize+0x566e:
314262ce cmp     ecx, dword ptr [contoso!ContosoInitialize+0xe5d3c (3150699c)]
```

It appears that this is a read-only variable. Therefore, its current value is its permanent value. And we saw last time that the permanent value is zero.

And we already found a bug. This code assumes that zero is not a valid TLS index. Actually, the invalid TLS index goes by the name `TLS_OUT_OF_INDEXES`, which is the value that `TlsAlloc` uses to say "Sorry, I couldn't allocate a TLS index for you." If this app ever calls `TlsAlloc` and get zero back, it will think that it hasn't yet assigned a TLS slot.

But that's not the bug that we're chasing, because we got a TLS index of 65. But at least we can come up with a nice name for the variable.

```
DWORD invalidTlsIndex = 0;
```

Back to the function, already in progress.

```
314262ad jne     contoso!ContosoInitialize+0x56c0 (31426320) // if valid, then skip
314262af mov     edx, dword ptr [contoso!ContosoInitialize+0xe5d4c (315069ac)] // get
some other thing
314262b5 push    edx
314262b6 lea     ecx, [ebp-18h]
314262b9 call    contoso!ContosoInitialize+0x5170 (31425dd0) // construct something,
probably
314262be mov     dword ptr [ebp-4], 0                                          //
exception unwinding tracking
```

I'm guessing that we're constructing something because it first this pattern: Put into the `ecx` register the address of some memory never accessed before, then call a function. The object being constructed here doesn't participate in managing the TLS index: We assume this because it doesn't take the TLS slot as a parameter. Therefore, we will ignore it for now. (Although I do know what it is, and you might be able to guess too, after we disassemble a little more.)

```
314262c5 mov     eax, dword ptr [ebp-2Ch]           // this
314262c8 mov     ecx, dword ptr [eax+400h]          // this->m_tlsIndex
314262ce cmp     ecx, dword ptr [contoso!ContosoInitialize+0xe5d3c (3150699c)] //
invalidTlsIndex
314262d4 jne     contoso!ContosoInitialize+0x56b1 (31426311) // jump if not equal
```

This next chunk of code is executed if the TLS index is zero; presumbaly it allocates a TLS slot. Let's see.

```
314262d6 push    0                                  // mystery parameter
314262d8 push    offset contoso!ContosoInitialize+0x5360 (31425fc0) // function
callback
314262dd mov     edx, dword ptr [ebp-2Ch]           // this
314262e0 add     edx, 400h                          // &this->m_tlsIndex
314262e6 push    edx
314262e7 call    contoso!ContosoInitialize+0x6680 (314272e0) // looks promising
314262ec add     esp, 0Ch
314262ef test    eax, eax
314262f1 je      contoso!ContosoInitialize+0x56b1 (31426311) // jump if zero
314262f3 mov     dword ptr [ebp-1Ch], 0             // local1c = 0
314262fa mov     dword ptr [ebp-4], 0FFFFFFFFh
31426301 lea     ecx, [ebp-18h]                     // destruct that thing on the
stack
31426304 call    contoso!ContosoInitialize+0x5200 (31425e60)
31426309 mov     eax, dword ptr [ebp-1Ch]           // return local1c
3142630c jmp     contoso!ContosoInitialize+0x5783 (314263e3)
31426311 mov     dword ptr [ebp-4], 0FFFFFFFFh
31426318 lea     ecx, [ebp-18h]                     // destruct that thing on the
stack
3142631b call    contoso!ContosoInitialize+0x5200 (31425e60)
...
314263e3 mov     ecx, dword ptr [ebp-0Ch]
314263e6 mov     dword ptr fs:[0], ecx
314263ed mov     esp, ebp
314263ef pop     ebp
314263f0 ret
```

So far, we have reverse-compiled the code to look like this:

```
SomeBitArrayClass1* Class2::f_31426280()
{
    if (this->m_tlsIndex == invalidTlsIndex)
    {
        Class3 object3(...);
        if (this->m_tlsIndex == invalidTlsIndex)
        {
            if (f_314272e0(&this->m_tlsIndex, f_31425fc0, 0) != 0)
            {
                return nullptr;
            }
        }
    }
    ... more code ...
```

The `Class3` object is probably some sort of synchronization object, since what we have here looks very much like a double-check-locking pattern.

Anyway, that function at `314272e0` probably allocates the TLS slot, seeing as we pass the address of where we want to put the TLS index.

```
contoso!ContosoInitialize+0x6680:
314272e0 push    ebp
314272e1 mov     ebp, esp
314272e3 push    ecx
314272e4 call    dword ptr [contoso!ContosoInitialize+0xa85bc (314c921c)] // TlsAlloc
314272ea mov     ecx, dword ptr [ebp+8]     // arg1
314272ed mov     dword ptr [ecx], eax       // save it
314272ef mov     edx, dword ptr [ebp+8]
314272f2 cmp     dword ptr [edx], 0FFFFFFFFh // was it invalid?
314272f5 je      contoso!ContosoInitialize+0x66b3 (31427313) // Y: bail
314272f7 mov     eax, dword ptr [ebp+10h]
314272fa push    eax
314272fb mov     ecx, dword ptr [ebp+0Ch]
314272fe push    ecx
314272ff mov     edx, dword ptr [ebp+8]
31427302 mov     eax, dword ptr [edx]
31427304 push    eax
31427305 call    contoso!ContosoInitialize+0x53b0 (31426010) // succeeded, keep going
3142730a mov     ecx, eax
3142730c call    contoso!ContosoInitialize+0x5490 (314260f0)
31427311 jmp     contoso!ContosoInitialize+0x6710 (31427370)
31427370 mov     esp, ebp
31427372 pop     ebp
```

This function allocates the TLS slot, and if successful, it does something ambiguous. The code seqeuence at `314272f7` could be any of

```
    f_31426010(*tlsIndex, callbackFunction, arg3)->f_314260f0()
    f_31426010(*tlsIndex, callbackFunction)->f_314260f0(arg3)
    f_31426010(*tlsIndex)->f_314260f0(callbackFunction, arg3)
    f_31426010()->f_314260f0(*tlsIndex, callbackFunction, arg3)
```

If the `f_31426010` and `f_314260f0` functions were `__cdecl`, then there would be `add esp, N` instructions after each call, and that would tell us how many parameters each function consumes. But there isn't, which means that these functions are `__stdcall`.

To find out which of the above four cases is the one we have, we need to look at the function epilogue for `f_31426010`. That will tell us how many bytes of parameters it consumes, and that will tell us which of the parameters belong to `f_31426010` and which belong to `f_314260f0`.

```
contoso!ContosoInitialize+0x53b0:
31426010 push    ebp
31426011 mov     ebp,esp
...
314260ea mov     esp,ebp
314260ec pop     ebp
314260ed ret
```

Okay, the function ends with a plain `ret`, which combined with the lack of `add esp, N` in the calling code means that it consumes zero parameters from the stack.

Therefore, we are in this case:

```
    f_31426010()->f_314260f0(*tlsIndex, callbackFunction, arg3)
```

The parameters (including the TLS slot index that we are tracking very closely) all go to the `f_314260f0` function.

```
contoso!ContosoInitialize+0x5490:
314260f0 push    ebp
314260f1 mov     ebp, esp
314260f3 push    0FFFFFFFFh
314260f5 push    offset contoso!ContosoInitialize+0xa5778 (314c63d8)
314260fa mov     eax, dword ptr fs:[00000000h]
31426100 push    eax
31426101 mov     dword ptr fs:[0], esp
31426108 sub     esp, 28h
3142610b mov     dword ptr [ebp-34h], ecx       // save "this"
3142610e mov     eax, dword ptr [contoso!ContosoInitialize+0xe5d4c (315069ac)]
31426113 push    eax
31426114 lea     ecx, [ebp-18h]
31426117 call    contoso!ContosoInitialize+0x5170 (31425dd0) // construct Class3
3142611c mov     dword ptr [ebp-4], 0
31426123 mov     ecx, dword ptr [ebp+8]         // tlsIndex
31426126 mov     dword ptr [ebp-10h], ecx       // local10 = tlsIndex
31426129 cmp     dword ptr [ebp-10h], 40h       // tlsIndex compared with 64
3142612d jae     contoso!ContosoInitialize+0x551f (3142617f) // Jump if tlsIndex >=
64

3142617f mov     dword ptr [ebp-30h], 0FFFFFFFFh // local30 = -1
31426186 mov     dword ptr [ebp-4], 0FFFFFFFFh
3142618d lea     ecx, [ebp-18h]
31426190 call    contoso!ContosoInitialize+0x5200 (31425e60) // destruct Class3
31426195 mov     eax, dword ptr [ebp-30h]       // return local30
31426198 mov     ecx, dword ptr [ebp-0Ch]
3142619b mov     dword ptr fs:[0], ecx
314261a2 mov     esp, ebp
314261a4 pop     ebp
314261a5 ret     0Ch
```

Aha! We see that the code checks the numeric value of the TLS index. The only meaningful value to compare the index against is `TLS_OUT_OF_INDEXES`. Once you verify that you have a valid TLS index, the actual numeric value is opaque. Changing behavior based on the numeric value of the slot index is highly suspect.

The partially-reverse-compiled function looks like this:

```
int f_314260f0(DWORD tlsIndex, callback, x)
{
    Class3 object3(...);
    if (tlsIndex < 64)
    {
        ...
    }
    else
    {
        return -1;
    }
}
```

Holy cow, this function simply rejects TLS slot indices that are greater than or equal to 64! Returning `-1` causes the calling function `f_31426280` to return `nullptr`, which leads to our null pointer crash.

Now we understand why the program is crashing. It mishandles TLS slot indices that are 64 or higher. It tries to reject them, but notice that when function `f_314260f0` returns `-1`, the calling function does not free the TLS slot or reset `this->m_tlsIndex` back to `invalidTlsIndex`. Instead, it leaves the TLS slots index allocated, and the next time the code wants to use the object, it sees that the TLS slot index is valid and tries to use the value stored in that slot.

Except we never stored anything there.

What's so special about the number 64? We'll dig into that next time.

Raymond Chen

Follow