Investigating an app compat problem: Part 2: Digging in

devblogs.microsoft.com/oldnewthing/20160609-00

June 9, 2016

Raymond Chen

We left our story with the conclusion that the program crashed because its TLS slot was null. But how can we figure out who sets the TLS slot and why it failed to set the TLS slot?

Let's hope that the reason is close to the failure (because debugging is an exercise in optimism) and see if we can find the code that is supposed to set the TLS value and figure out why it failed.

This is where we roll up our sleeves and get our hands dirty.

Here is the function that crashed. Let's do some reverse-compilation. My personal convention is as follows:

- Register-sized variables are left untyped until I figure out what type it really is. If I must specify a type for a variable declaration, I use `int` or `void*`. (If the type turns out really to be an `int`, I use `int32_t`.)
- Local variables are named `localXX` where `XX` is the offset of the variable relative to the frame pointer.
- Member variables are named `m_XX` where `XX` is the offset of the member relative to the start of the object.
- Functions are named `f_XXXXXXXX` where `XXXXXXXX` is the address of the first instruction.

```
contoso!ContosoInitialize+0x4d40:
314259a0 push    ebp
314259a1 mov     ebp, esp
314259a3 sub     esp, 10h                   // 16 bytes of local variables
314259a6 mov     dword ptr [ebp-10h], ecx   // local10 = this
314259a9 mov     eax, dword ptr [ebp+8]     // arg1
314259ac mov     dword ptr [ebp-8], eax     // local8 = arg1
314259af lea     ecx, [ebp-0Ch]             // &localc
314259b2 push    ecx
314259b3 lea     edx, [ebp-4]               // &local4
314259b6 push    edx
314259b7 mov     eax, dword ptr [ebp-8]     // local8
314259ba push    eax
314259bb call    contoso!ContosoInitialize+0x4db0 (31425a10)
314259c0 add     esp, 0Ch
314259c3 mov     edx, 1
314259c8 mov     ecx, dword ptr [ebp-0Ch]   // localc
314259cb shl     edx, cl                    // 1 << localc
314259cd mov     eax, dword ptr [ebp-4]     // local4
314259d0 mov     ecx, dword ptr [ebp-10h]   // this
314259d3 mov     eax, dword ptr [ecx+eax*4] // this->m_0[local4]
314259d6 and     eax, edx                   // this->m_0[local4] & (1 << localc)
314259d8 test    eax, eax
314259da je      contoso!ContosoInitialize+0x4d83 (314259e3) // jump if bit was clear
314259dc mov     eax, 1                     // return 1
314259e1 jmp     contoso!ContosoInitialize+0x4da3 (31425a03)
314259e3 mov     edx, 1
314259e8 mov     ecx, dword ptr [ebp-0Ch]   // localc
314259eb shl     edx, cl                    // 1 << localc
314259ed mov     eax, dword ptr [ebp-4]     // local4
314259f0 mov     ecx, dword ptr [ebp-10h]   // this
314259f3 mov     eax, dword ptr [ecx+eax*4] // this->m_0[local4]
314259f6 or      eax, edx                   // this->m_0[local4] | (1 << localc)
314259f8 mov     ecx, dword ptr [ebp-4]     // local4
314259fb mov     edx, dword ptr [ebp-10h]   // this
314259fe mov     dword ptr [edx+ecx*4], eax // this->m_0[local4] = this->m_0[local4]
| (1 << localc)
31425a01 xor     eax, eax                   // return 0
31425a03 mov     esp, ebp
31425a05 pop     ebp
31425a06 ret     4
0:000>
```

The lack of common subexpression elimination and the frequent spilling and reloading of registers tells me that this code was compiled with optimizations disabled. Bad for performance, but it makes reverse-engineering so much easier. We end up with this, after renaming some variables and propagating stores.

```
BOOL Class1::f_314259a0(int arg1)
{
    int elementIndex;
    int relativeBitIndex;
    f_31425a10(arg1, &elementIndex, &relativeBitIndex);
    if (this->m_0[elementIndex] & (1 << relativeBitIndex))
    {
        return TRUE;
    }
    else
    {
        this->m_0[elementIndex] =
        this->m_0[elementIndex] | (1 << relativeBitIndex);
        return FALSE;
    }
}
```

This function calculates a bit in a buffer, and if the bit is not set, it sets the bit. The function then returns the previous state of the bit. Let's look at the function that calculates which bit to set.

```
contoso!ContosoInitialize+0x4db0:
31425a10 push    ebp
31425a11 mov     ebp,esp
31425a13 mov     eax,dword ptr [ebp+8]      // arg1
31425a16 shr     eax,5                      // arg1 / 32 (unsigned)
31425a19 mov     ecx,dword ptr [ebp+0Ch]    // arg3
31425a1c mov     dword ptr [ecx],eax        // *arg3 = arg1 / 32
31425a1e mov     eax,dword ptr [ebp+8]      // arg1
31425a21 xor     edx,edx                    // zero-extend to 64 bits
31425a23 mov     ecx,20h
31425a28 div     eax,ecx                    // arg1 / 32
31425a2a mov     eax,dword ptr [ebp+10h]    // arg2
31425a2d mov     dword ptr [eax],edx        // *arg2 = arg1 / 32
31425a2f pop     ebp
31425a30 ret
```

Okay, so the bit index is nothing fancy. The buffer at `m_0` is treated as a giant bit array, and this function figures out which element holds that bit and where that bit is. We also learned that the incoming and outgoing parameters are unsigned 32-bit integers because the arithmetic operations are consistent with unsigned operations rather than signed. We don't know how big the bit array is, but at least we can give the function a nicer name.

We can capture what we've learned as follows:

```
class SomeBitArrayClass1
{
public:
    BOOL SetBit(uint32_t bitIndex);

private:
    static void CalcBitPosition(
        uint32_t bitIndex,
        uint32_t* elementIndex,
        uint32_t* relativeBitIndex);

    uint32_t buffer[unknown_size];
};

BOOL SomeBitArrayClass1::SetBit(uint32_t bitIndex)
{
    uint32_t elementIndex;
    uint32_t relativeBitIndex;
    CalcBitPosition(bitIndex, &elementIndex, &relativeBitIndex);
    if (this->buffer[elementIndex] & (1 << relativeBitIndex))
    {
        return TRUE;
    }
    else
    {
        this->buffer[elementIndex] =
        this->buffer[elementIndex] | (1 << relativeBitIndex);
        return FALSE;
    }
}
```

Sure, the code that sets the bit could have been written as

```
this->buffer[elementIndex] |= (1 << relativeBitIndex);
```

but I’m just repeating the code that was written, and what they wrote calculates the indexed element address twice.

We’re off to a good start, but we haven’t really learned much yet. Much more interesting is the function that produced the null pointer that caused us to crash.

We’ll pick that up next time.

Raymond Chen

Follow