Debugging walkthrough: Access violation on nonsense instruction

devblogs.microsoft.com/oldnewthing/20141226-00

December 26, 2014

Raymond Chen

A colleague of mine asked for help puzzling out a mysterious crash dump which arrived via Windows Error Reporting.

```
rax=00007fff219c5000 rbx=00000000023c8380 rcx=00000000023c8380
rdx=0000000000000000 rsi=00000000043f0148 rdi=0000000000000000
rip=00007fff21af2d22 rsp=000000000392e518 rbp=000000000392e580
 r8=00000000276e4639  r9=00000000043b2360 r10=00000000ffffffff
r11=0000000000000000 r12=0000000000000001 r13=0000000000000000
r14=000000000237cfc0 r15=00000000023d3ea0
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
nosebleed!CNosebleed::OnFrimble+0x1f891a:
00007fff`21af2d22 30488b xor byte ptr [rax-75h],cl ds:00007fff`219c4f8b=41
```

Well that's a pretty strange instruction. Especially since it doesn't match up with the source code at all.

```
void CNosebleed::OnFrimble(...)
{
    ...
    if (CanFrumble(...))
    {
        ...
    }
    else
    {
        hr = pCereal->AddMilk(pCarton);
        if (SUCCEEDED(hr))
        {
            pCereal->Snap();
            pCereal->Crackle(false);
            if (SUCCEEDED(pCereal->Pop(uId)) // ← crash here
            {
                ....
            }
        }
    }
    ....
}
```

There is no bit-toggling in the actual code. The method calls to `Snap, Crackle, and Pop` are all interface calls and therefore should be vtable calls. We are clearly in a case of a bogus return address, possibly a stack smash (and therefore cause for concern from a security standpoint).

My approach was to try to figure out what was happening just before the crash. And that meant figuring out how we ended up in the middle of an instruction.

Here is the code surrounding the crash point.

```
00007fff`21af2d17 ff90d0020000    call    qword ptr [rax+2D0h]
00007fff`21af2d1d 488b03          mov     rax,qword ptr [rbx]
00007fff`21af2d20 8b5530          mov     edx,dword ptr [rbp+30h]
00007fff`21af2d23 488bcb          mov     rcx,rbx
```

Notice that the code that crashed is actually the last byte of the `mov edx, dword ptr [rbp+30h]` (the `30`) and the first two bytes of the `mov rcx, rbx` (the `488b`).

Disassembling backward is a tricky business on a processor with variable-length instructions, so to get my bearings, I looked for the call to `CanFrumble`:

```
0:011> #CanFrumble nosebleed!CNosebleed::OnFrimble
nosebleed!CNosebleed::OnFrimble+0x1f883b
00007fff`21af2c43 e8e0e40f00 call nosebleed!CNosebleed::CanFrumble
```

The `#` command means “Start disassembling at the specified location and stop when you see the string I passed.” This is an automated way of just hitting `u` until you get to the thing you are looking for.

Now that I am at some known good code, I can disassemble forward:

```
00007fff`21af2c48 488bcb          mov     rcx,rbx
00007fff`21af2c4b 84c0            test    al,al
00007fff`21af2c4d 0f849a000000    je      nosebleed!CNosebleed::OnFrimble+0x1f88e5
(00007fff`21af2ced)
```

The above instructions check whether the `CanFrumble` returned `true`, and if not, it jumps to `00007fff`21af2ced`. Since we know that we are in the `false` path, we follow the jump.

```
// Make a vtable call into pCereal->AddMilk()
00007fff`21af2ced 488b03          mov     rax,qword ptr [rbx] ; vtable
00007fff`21af2cf0 498bd7          mov     rdx,r15 ; pCarton
00007fff`21af2cf3 ff9068010000    call    qword ptr [rax+168h] ; call
00007fff`21af2cf9 8bf8            mov     edi,eax ; save to hr
00007fff`21af2cfb 85c0            test    eax,eax ; succeeded?
00007fff`21af2dfd 0f880dffffff    js      nosebleed!CNosebleed::OnFrimble+0x1f8808
(00007fff`21af2c10)
// Now call Snap()
00007fff`21af2d03 488b03          mov     rax,qword ptr [rbx] ; vtable
00007fff`21af2d06 488bcb          mov     rcx,rbx ; "this"
00007fff`21af2d09 ff9070020000    call    qword ptr [rax+270h] ; Snap
/ Now call Crackle
00007fff`21af2d0f 488b03          mov     rax,qword ptr [rbx] ; vtable
00007fff`21af2d12 33d2            xor     edx,edx ; parameter: false
00007fff`21af2d14 488bcb          mov     rcx,rbx ; "this"
00007fff`21af2d17 ff90d0020000    call    qword ptr [rax+2D0h] ; Crackle
// Get ready to Pop
00007fff`21af2d1d 488b03          mov     rax,qword ptr [rbx] ; vtable
00007fff`21af2d20 8b5530          mov     edx,dword ptr [rbp+30h] ; uId
00007fff`21af2d23 488bcb          mov     rcx,rbx ; "this"
```

But we never got to execute the `Pop` because our return address from `Crackle` got messed up.

Let's follow the call into `Crackle`.

```
0:011> dps @rbx l1
00000000`02b4b790  00007fff`219c50a0 nosebleed!CCereal::`vftable'
0:011> dps 00007fff`219c50a0+2d0 l1
00007fff`219c5370  00007fff`21aa5c28 nosebleed!CCereal::Crackle
0:011> u 00007fff`21aa5c28
nosebleed!CCereal::Crackle:
00007fff`21aa5c28 889163010000    mov     byte ptr [rcx+163h],dl
00007fff`21aa5c2e c3              ret
```

So at least the `pCereal` pointer seems to be okay. It has a vtable and the slot in the vtable points to the function we expect. The `Crackle` method merely stashes the `bool` parameter into a member variable. No stack corruption here because `rbx` is nowhere near `rsp`.

```
0:012> db @rbx+163 l1
00000000`02b4b8f3  ??                                              ?
```

Sadly, the byte in question was not captured in the dump, so we cannot verify whether the call actually was made. Similarly, the members of `CCereal` manipulated by the `Snap` method were also not captured in the dump, so we can't verify that either. (The only member of `CCereal` captured in the dump is the vtable itself.)

So we can't find any evidence one way or the other as to whether any of the calls leading up to `Pop` actually occurred. Maybe we can try to figure out how many misaligned instructions we managed to execute before we crashed, see if that reveals anything. To do this, I'm going to disassemble at varying incorrect offsets and see which ones lead to the instruction that crashed.

```
0:011> u .-1 l2
nosebleed!CNosebleed::OnFrimble+0x1f8919:
00007fff`21af2d21 55              push    rbp
00007fff`21af2d22 30488b          xor     byte ptr [rax-75h],cl
// ^^ this looks interesting; we'll come back to it
0:011> u .-3 l2
nosebleed!CNosebleed::OnFrimble+0x1f8917:
00007fff`21af2d1f 038b5530488b    add     ecx,dword ptr [rbx-74B7CFABh]
00007fff`21af2d25 cb              retf
// ^^ this doesn't lead to the crashed instruction
0:011> u .-4 l2
nosebleed!CNosebleed::OnFrimble+0x1f8916:
00007fff`21af2d1e 8b03            mov     eax,dword ptr [rbx]
00007fff`21af2d20 8b5530          mov     edx,dword ptr [rbp+30h]
// ^^ this doesn't lead to the crashed instruction
0:012> u .-5 l3
nosebleed!CNosebleed::OnFrimble+0x1f8914:
00007fff`21af2d1c 00488b          add     byte ptr [rax-75h],cl
00007fff`21af2d1f 038b5530488b    add     ecx,dword ptr [rbx-74B7CFABh]
00007fff`21af2d25 cb              retf
// ^^ this doesn't lead to the crashed instruction
0:012> u .-6 l3
nosebleed!CNosebleed::OnFrimble+0x1f8913:
00007fff`21af2d1b 0000            add     byte ptr [rax],al
00007fff`21af2d1d 488b03          mov     rax,qword ptr [rbx]
00007fff`21af2d20 8b5530          mov     edx,dword ptr [rbp+30h]
// ^^ this doesn't lead to the crashed instruction
```

Exercise: Why didn't I bother checking `.-2`?

You only need to test as far back as the maximum instruction length, and in practice you can give up much sooner because the maximimum instruction length involves a lot of prefixes which are unlikely to occur in real code.

The only single-instruction rewind that makes sense is the `push rbp`. Let's see if it matches.

```
0:011> ?? @rbp
unsigned int64 0x453e700
0:011> dps @rsp l1
00000000`0453e698  00000000`0453e700
```

Yup, it lines up. This wayward push is also consistent with the stack frame layout for the function.

```
nosebleed!CNosebleed::OnFrimble:
00007fff`218fa408 48895c2410      mov     qword ptr [rsp+10h],rbx
00007fff`218fa40d 4889742418      mov     qword ptr [rsp+18h],rsi
00007fff`218fa412 55              push    rbp
00007fff`218fa413 57              push    rdi
00007fff`218fa414 4154            push    r12
00007fff`218fa416 4156            push    r14
00007fff`218fa418 4157            push    r15
00007fff`218fa41a 488bec          mov     rbp,rsp
00007fff`218fa41d 4883ec60        sub     rsp,60h
```

The values of `rbp` and `rsp` should differ by `0x60`.

```
0:012> ?? @rbp-@rsp
unsigned int64 0x68
```

The difference is in error by 8 bytes, exactly the size of the `rbp` register that was pushed.

It therefore seems highly likely that the `push rbp` was executed.

Repeating the exercise to find the instruction before the `push rbp` shows that no instruction fell through to the `push rbp`. Therefore, execution jumped to `00007fff`21af2d21` somehow.

Another piece of data is that `rax` matches the value we expect it to have, sort of. Here are some selected lines from earlier in the debug session:

```
// What we expected to have executed
00007fff`21af2d1e 8b03            mov     eax,dword ptr [rbx]
// The value we expected to have fetched
0:011> dps @rbx l1
00000000`02b4b790  00007fff`219c50a0 nosebleed!CCereal::`vftable'
// The value in the rax register
rax=00007fff219c5000 ...
```

The value we expect is `00007fff`219c50a0`, but the value in the register has the bottom eight bits cleared.

Putting this all together, my theory is that the CPU executed the instruction at `00007fff`21af2d1e`, and then due to some sort of hardware failure, instead of incrementing the `rip` register by two, it (1) incremented it by three, and then (2) as part of its confusion, zeroed out the bottom byte of `rax`. The erroneous `rip` led to the rogue `push rbp` and the crash on the nonsensical `xor`.

It's not a great theory, but it's all I got.

As to what sort of hardware failure could have occurred: This particular failure was reported twice, so a cosmic ray is less likely to be the culprit (because you have to get lightning to strike twice) than overheating or overclocking.

Raymond Chen

Follow