Adventures in application compatibility: The case of the wild instruction pointer that, upon closer inspection, might not be so wild after all

devblogs.microsoft.com/oldnewthing/20210917-00

September 17, 2021

Raymond Chen

Application compatibility testing as well as Windows Insiders discovered that Windows began crashing randomly if you upgraded to a specific build and had a specific program installed. Uninstalling that program stopped the crashes.

The crash dumps were spread out over a large number of processes unrelated to the program, so it's not that the program itself was crashing, but rather that the presence of the program was causing *other programs* to start crashing. If you looked at the crash dumps, you found that the instruction pointer was just hanging out in the middle of nowhere:

```
rax=00007ffc1f8d0dc0 rbx=0000000000000010 rcx=0000000e194fa970
rdx=0000000000000000 rsi=0000000e194fa728 rdi=0000000e194fa428
rip=00007ffd9d1c5f2c rsp=0000000e194fa3e8 rbp=0000000000000001
 r8=0000011c610f6a30  r9=0000000e194fa150 r10=0000000e194fa760
r11=0000000e194fa9ec r12=0000000000000000 r13=00000000ffffffff
r14=0000000000000000 r15=0000000e194fa650
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010204
00007ffd`9d1c5f2c ??              ???
```

There were some clues on the stack:

```
0:008> dps @rsp
0000000e`194fa3e8  00007ffc`9d1c6219 ntdll!DestroyWidget+0x9
0000000e`194fa3f0  0000007c`a92fb098
0000000e`194fa3f8  00000000`00000000
0000000e`194fa400  0000000e`194fa4c8
0000000e`194fa408  0000011c`6382b440
0000000e`194fa410  00000000`00000246
0000000e`194fa418  00007ffc`763e3573 contoso+0x23573
0000000e`194fa420  0000011c`6102f690
0000000e`194fa428  00000000`00000000
0000000e`194fa430  0000011c`6382b460
0000000e`194fa438  00000000`00000000
0000000e`194fa440  00000000`00000000
0000000e`194fa448  0000000e`194fa4c8
0000000e`194fa450  00000000`00000000
0000000e`194fa458  00000000`00000000
0000000e`194fa460  00000000`00000000
```

According to the stack, the jump-into-space came from `ntdll!DestroyWidget+0x9`, but if you look at the code in `ntdll!DestroyWidget+0x9`, there is no jump into space. It's calling into another nearby function.

```
ntdll!DestroyWidget:
00007ffc`9d1c6210 4883ec28        sub     rsp,28h
00007ffc`9d1c6214 e813fdffff      call    ntdll!DestroyWidgetWorker
(00007ffc`9d1c5f2c)
00007ffc`9d1c6219 85c0            test    eax,eax
```

Notice that the wild instruction pointer differs from the intended jump target by a single bit:

Intended	`00007ffc`9d1c5f2c`
Actual	`00007ffd`9d1c5f2c`

This is not a return address stored on the stack, so it's not rogue memory corruption. The jump target is not stored on the stack at all; it's encoded directly in the instruction stream. So we can rule out a use-after-free bug here.

Hey, it's not much, but it's good to be able to rule out stuff so you can focus on the stuff that is still in play.

Another thought is that this was caused by overclocking. However, the reports were coming from a large number of systems, and the crash was consistent, which is atypical of overclocking, since overclocking crashes tends to be random.

Could something in the code stream be triggering a CPU erratum that caused jump targets to be miscalculated? Perhaps, but the close correlation with a specific program being installed suggests that the problem is in the software, not the hardware.

Inspection of more crash dumps show that the error is not actually a single-bit error after all. It's an "off by 4GB" error.

Intended	`00007ffc`9d1c5f2c`	`00007ff9`33605f2c`
Actual	`00007ffd`9d1c5f2c`	`00007ffa`33605f2c`
XOR	`00000001`00000000`	`00000003`00000000`
Difference	`00000001`00000000`	`00000001`00000000`

There are different levels of crash dumps. Some time ago, I mentioned the triage dump, which is an extremely lightweight dump file that captures only a little bit of stack information, just enough to generate a stack trace but not much else. The dumps we've been looking at here are "minidumps", which contain more complete stack information. But now it's time to bring out the big guns: The full process dump.

Full process dumps are very large, so Windows Error Reporting doesn't capture them most of the time. But developers can specifically request that the next *N* crashes be captured as full process dumps, and Windows Error Reporting will oblige.

Opening a full process crash dump shows something very telling: The code at `ntdll!DestroyWidget` looks different:

```
0:008> u ntdll!DestroyWidget
ntdll!DestroyWidget:
00007ffc`9d1c6210 e96bab7082      jmp     00007ffc`1f8d0d80
00007ffc`9d1c6215 13fd            adc     edi,ebp
00007ffc`9d1c6217 ff              ???
00007ffc`9d1c6218 ff85c0740bb8    inc     dword ptr [rbp-47F48B40h]
```

The function has been detoured!

Okay, now we're getting somewhere.

When the detour wants to call the original function, it needs to replicate the original instructions that were overwritten and then jump to the first non-overwritten instruction. This is made more complicated by the fact that the last overwritten instruction was a `call` instruction. The replicant is rather messy but it boils down to

```
    ; replicate the "sub rsp,28h"
    sub     rsp,28h

    ; replicate the "call ntdll!DestroyWidgetWorker"
    mov     rax,7FFD9D1C6219h
    push    rax             ; fake return address
    mov     rax,7FFC9D1C5F2Ch
    jmp     rax             ; jump to ntdll!DestroyWidgetWorker
```

To replicate the call instruction, the detour pushes a fake return address and then jumps to the start of the called function. This, of course, messes up the return address predictor since the `call` and `ret` instructions no longer balance. Sorry for your system performance, but hey, at least our program got its detour![1]

Upon looking at the replicated code, you may spot the error: They miscalculated the fake return address.

What happened is that their detour generator incorrectly decoded the `call` instruction and treated the 32-bit immediate as an *unsigned* 32-bit offset rather than a *signed* 32-bit offset. The call to `DestroyWidgetWorker` has a negative offset:

```
00007ffc`9d1c6214 e813fdffff      call    ntdll!DestroyWidgetWorker
(00007ffc`9d1c5f2c)
                      ^^^^^^^^
^^^^^^^^^^^^^^^^^^
         offset = 0xfffffd13                                  lower address than
caller
```

Their instruction decoder zero-extended the offset to a 64-bit value, resulting in a miscalculated jump target that is 4GB too high:

	Correct	Incorrect
Return address	`00007ffc`9d1c6219`	`00007ffc`9d1c6219`
Plus offset	`ffffffff`fffffd13`	`00000000`fffffd13`
Equals target	`00007ffc`9d1c5f2c`	`00007ffd`9d1c5f2c`

My guess is that the instruction decoder was ported from a 32-bit decoder, and in 32-bit code, it doesn't matter whether you treat the offset as signed or unsigned because the sum is truncated to a 32-bit value. But when doing 64-bit decoding, those upper 32 bits are important, and failing to extend negative values correctly results in an off-by-4GB calculation.

Even though this problem has always existed, it requires two triggers:

- The detoured function must have a `call` instruction within the first 5 bytes.
- The destination of the `call` must be at a lower address than the caller.

The program's detour code was lucky, but recently its luck ran out.

We contacted the vendor, who released a patch. The crashes started to abate, but they don't go away completely because not everybody is diligent about installing patches.

Bonus chatter: A reminder that Windows does not support detouring the operating system. This program has wandered into unsupported territory. Not that their customers will know or care.

[1] A version that preserves the return address predictor stack might go something like this:

```
    ; replicate the "sub rsp,28h"
    sub     rsp,28h

    ; replicate the "call ntdll!DestroyWidgetWorker"
    call    @F              ; push a slot onto the return address predictor
@@: mov     rax,7FFC9D1C6219h
    mov     [rsp], rax      ; change the return address to our fake one
    mov     rax,7FFC9D1C5F2Ch
    jmp     rax             ; jump to ntdll!DestroyWidgetWorker
```

The `ret` from `DestroyWidgetWorker` will be mispredicted, but at least all the remaining return addresses will be predicted correctly.

Raymond Chen

Follow