Adventures in application compatibility: The case of the jump into the middle of an instruction from nowhere

devblogs.microsoft.com/oldnewthing/20230324-00

March 24, 2023

Raymond Chen

A spike of Explorer crashes occurred with the release of a particular Windows Insider build. The crash looked like this:

```
explorer!SomeRandomInternalFunction+0x7d4:
00007ffe`27b00720 006639          add     byte ptr [rsi+39h],ah
ds:00000000`0000003a=??
```

This is most likely a nonsense instruction. There's no obvious reason to be adding a partial upper register.

It looks like this is either a corrupted instruction pointer or corrupted code, because the first code byte is suspicious zero. And since the second byte is `66h`, it looks like an off-by-one, since `66h` is not an uncommon initial instruction byte. (It's the operand size override prefix.) Another clue is that the calling function, not shown here, has no reason to be calling the `SomeRandomInternalFunction` function, and in fact, checking the alleged caller, it indeed does not call it.

Disassembling around the instruction shows that the instruction pointer is indeed in the middle of an instruction:

```
00007ffe`27b0071c b820000000      mov     eax,20h
00007ffe`27b00721 6639853e020000  cmp     word ptr [rbp+23Eh],ax
```

The instruction pointer is one less than the actual start of the instruction, causing the zero byte at the end of the immediate of the previous instruction to be misinterpreted as the start of the instruction to be executed.

How did we end up in the middle of an instruction?

I did some bulk analysis of all the crash dumps that we received and observed that one third-party DLL was common to all of them. Further investigation shows that this third-party DLL is part of a "shell enhancement" program. This program patches Explorer in order to accomplish its enhancements, and apparently one of its patches went awry.

The interesting thing here was how the program decided where to install its patches, and in particular how it managed to patch an function that was never exported or stored in a vtable: It found the function by contacting the Microsoft symbol server to get the names of all the functions in Explorer and their addresses!

What happened is that we recently made changes to this internal function, and apparently those changes were enough to cause the patcher to go haywire. This is unfortunately a regular occurrence: Whenever a new build goes out, there's a spike of Explorer crashes because all of these patchers start patching the wrong code, and Explorer starts crashing across all the systems that have these "shell enhancement" programs installed. If you're really unlucky, their rogue patch crashes something in the Explorer startup path, and users finds themselves stuck with an unusable machine due to Explorer crash loops.

This problem is particularly acute with monthly security patches, because we can't roll the fix back. That would expose systems to the security issues that the monthly security update was intended to fix. (And now that the fix went out, all the bad guys have reverse-engineered the security issue and are probably hard at work trying to weaponizing it and take advantage of unpatched systems.) We have to hope that enough of the users whose systems are crashing realize that it's due to the "shell enhancement" program (rather than blaming Windows itself, which is the more likely case), and uninstall or disable the programs in order to get their system working again. Unfortunately, these patchers also cause Windows customer satisfaction numbers to plunge every time an update goes out, particularly among users who don't realize that the problem was caused by that program their computer-savvy nephew installed for them.