# Debugging walkthrough: Access violation on nonsense instruction, episode 2

devblogs.microsoft.com/oldnewthing/20150313-00

March 13, 2015

Raymond Chen

A colleague of mine asked for help debugging a strange failure. Execution halted on what appeared to be a nonsense instruction.

```
eax=0079f850 ebx=00000000 ecx=00000113 edx=00000030 esi=33ee06ef edi=74b9b8ad
eip=00c0ac74 esp=0079f82c ebp=0079f86c iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
00c0ac74 0000            add     byte ptr [eax],al          ds:002b:0079f850=74
```

If you've been debugging x86 code for a while, you immediately recognize this instruction as "executing a page of zeroes". If you haven't been debugging x86 code for a while, you can see this from the code bytes in the second column.

So how did we end up at this nonsense instruction?

The instruction is not near a page boundary, so we didn't fall through to it. We must have jumped to it or returned to it.

Since debugging is an exercise in optimism, let's assume that we jumped to it via a `call` instruction, and the return address is still on the stack.

```
0:000> dps esp l2
0079f82c  74b9b8b1 user32!GetMessageW+0x4
0079f830  008f108b CONTOSO!MessageLoop+0xe7
0:000> u user32!GetMessageW l3
USER32!GetMessageW:
74b9b8ad cc              int     3
74b9b8ae ff558b          call    dword ptr [ebp-75h]
74b9b8b1 ec              in      al,dx
```

Well, that explains it. The code bytes for the `GetMessageW` function were overwritten, causing us to execute garbage, and one of the garbage instructions was a `call` that took us to page of zeroes.

But look more closely at the overwritten bytes.

The first byte is `cc`, which is a breakpoint instruction. Hm…

Since Windows functions begin with a `MOV EDI, EDI` instruction for hot patching purposes, the first two bytes are always `8b ff`. If we unpatch the `cc` to `8b`, we see that the rest of the code bytes are intact.

```
USER32!GetMessageW:
74b9b8ad 8bff            mov     edi,edi
74b9b8af 55              push    ebp
74b9b8b0 8bec            mov     ebp,esp
```

After a brief discussion, we were able to piece together what happened:

Somebody was trying to debug the `CONTOSO` application, so they connected a user-mode debugger to the application. Meanwhile, they set a breakpoint on `user32!GetMessageW` from the kernel debugger. Setting a breakpoint in a debugger is typically performed by patching an `int 3` at the point where you want the breakpoint. When the `int 3` fires, the debugger regains control and says, "Oh, thanks for stopping. Let me unpatch all the `int 3`'s I put in the program to put things back the way they were."

When the breakpoint hit, it was caught by the user-mode debugger, but since the user-mode debugger didn't set that breakpoint, it interpreted the `int 3` as a hard-coded breakpoint in the application. At this point, the developer saw a spurious breakpoint, didn't know what it meant, and simply resumed execution. This executed the second half of the `MOV EDI, EDI` instruction as the start of a new instruction, and havoc ensued.

That developer then asked his friend what happened, and his friend asked me.

TL;DR: Be careful if you have more than one debugger active. Breakpoints set by one debugger will not be recognized by the other. If the breakpoint instruction is caught by the wrong debugger, things will go downhill fast unless you take corrective action. (In this case, it would be restoring the original byte.)

Raymond Chen

**Follow**