The Intel 80386, part 14: Rescuing a stack trace after the debugger gave up when it reached an FPO function

devblogs.microsoft.com/oldnewthing/20190206-00

February 7, 2019

Raymond Chen

So here you go, minding your own business, taking a stack trace, and then the world stops.

```
ChildEBP RetAddr
0019ec98 5654ef4e combase!CoInitializeEx+0x35
0019ecf8 5654e70b WINSPOOL!GetCurrentNetworkId+0x36
0019ed28 5654e58a WINSPOOL!InternalGetDefaultPrinter+0x8b
0019ed58 75953b77 WINSPOOL!GetDefaultPrinterW+0x5a
0019ed70 7594e6b8 comdlg32!PrintGetDefaultPrinterName+0x17
0019f1b8 7594e520 comdlg32!PrintBuildDevNames+0x60
0019f1d0 75951340 comdlg32!PrintReturnDefault+0x30
0019f628 759a03ab comdlg32!PrintDlgX+0x132
0019fae0 01804a8e comdlg32!PrintDlgA+0x5b
0019fd50 7686196c contoso+0x4a8e
```

The stack trace just gives up. The function in the Cnotoso DLL was compiled with frame pointer omission (FPO), which means that the *ebp* register is being used as a general-purpose register and does not point to the next frame deeper in the stack. And since we don't have symbols for Contoso, the debugger cannot consult the symbol table to get help with unwinding the stack one more level.

We'll have to build the stack trace manually. This is basically the same exercise on every architecture: You look at the code you're returning to, find its function prologue or epilogue, and use that information to unwind another frame.

The last known good stack frame was `0019fae0` from `PrintDlgA`. Let's see what we have there:

```
0:000> dps 0019fae0
0019fae0  0019fd50                    ← saved ebp
0019fae4  01804a8e contoso+0x4a8e ← return address
0019fae8  018083b0 contoso+0x83b0 ← argument to PrintDlgA
0019faec  0000000e
0019faf0  01803b8c contoso+0x3b8c
0019faf4  0019fd50
0019faf8  0000000e
0019fafc  0000000e
0019fb00  00200cce
0019fb04  00000112
0019fb08  0000f095
0019fb0c  0078006b
```

The `PrintDlgA` function takes a single parameter, and it uses the `__stdcall` calling convention, so we know that when `PrintDlgA` returns, the stack pointer will be at `0019faec`, and we will have returned to the code at `01804a8e`. We also see that the *ebp* register will have the value `0019fd50`.

To unwind a level, we need to disassemble at `01804a8e` and look for the code that cleans up the stack and returns to the previous function.

```
contoso+0x4a8e:
01804a8e 833dbc83800100  cmp     dword ptr [contoso+0x83bc (018083bc)],0
01804a95 7509            jne     contoso+0x4aa0 (01804aa0)
01804a97 b8ffffffff      mov     eax,0FFFFFFFFh
01804a9c 5e              pop     esi
01804a9d c3              ret
```

For the purpose of this exercise, we are just looking for any code path that leads to a `ret` instruction. We can assume conditional jumps are taken, or not taken, based on whichever case will get us to a `ret` instruction faster. Along the way to the `ret`, we watch for instructions that affect the *esp* register, because we'll have to simulate them in our head.

In this case, we can pretend that the conditional jump is not taken, and that leads us quickly to a `pop esi` and a `ret`.

So let's simulate those two operations. Since our simulated *esp* register is at `0019faec`, the `pop esi` pops the value `0000000e` into *esi*, and the `ret` returns to `01803b8c`. Since this was a simple `ret` with no parameters, there is no extra cleanup, and the stack pointer is left pointing to `0019faf4`.

```
0019faec  0000000e                    ← saved esi
0019faf0  01803b8c contoso+0x3b8c ← return address
0019faf4  0019fd50                    ← esp points here after ret
0019faf8  0000000e
```

Disassemble at the return address to see how to pop out another level.

```
contoso+0x3b8c:
01803b8c 8bd8            mov     ebx,eax
01803b8e 0bdb            or      ebx,ebx
01803b90 7510            jne     contoso+0x3ba2 (01803ba2)
01803b92 b8fbffffff      mov     eax,0FFFFFFFBh
01803b97 5d              pop     ebp          ← saved ebp
01803b98 5f              pop     edi          ← saved edi
01803b99 5e              pop     esi          ← saved esi
01803b9a 5b              pop     ebx          ← saved ebx
01803b9b 81c4e8000000    add     esp,0E8h   ← adjust esp
01803ba1 c3              ret                ← return, no extra cleanup
```

Again, we pretend that the conditional jump is not taken, and that leads us quickly to the function epilogue. We pop four values off the stack, then add `0e8h` to the *esp* register before executing the `ret`. Let's simulate those operations on our stack.

```
0019faf4  0019fd50       ← saved ebp
0019faf8  0000000e       ← saved edi
0019fafc  0000000e       ← saved esi
0019fb00  00200cce       ← saved ebx
0019fb04  00000112       ← esp points here after pop ebx
```

After popping *ebx*, the code adds `0E8h` to *esp*, so let's ask the debugger to skip ahead `0xe8` bytes.

```
0:000> dps 0019fb04+e8
0019fbec  01801325 contoso+0x1325 ← return address
0019fbf0  0000000e                ← esp points here after ret
```

Just keep swimming.

```
01801325 0bc0            or      eax,eax
01801327 0f8d74040000    jge     contoso+0x17a1 (018017a1)
0180132d 83f8fd          cmp     eax,0FFFFFFFDh
01801330 0f846b040000    je      contoso+0x17a1 (018017a1)
01801336 83f8fb          cmp     eax,0FFFFFFFBh
01801339 740d            je      contoso+0x1348 (01801348)
0180133b 83f8fc          cmp     eax,0FFFFFFFCh
0180133e 7410            je      contoso+0x1350 (01801350)
```

Okay, we're not so lucky this time. We don't see the end of the function right away. The code does a bunch of stuff with the value returned by this function, but if the return value is nonnegative, it jumps ahead to `018017a1`. I'm guessing that that jump forward will take us closer to the end of the function, so let's continue disassembling there.

```
018017a1 b801000000      mov     eax,1
018017a6 5f              pop     edi
018017a7 5e              pop     esi
018017a8 81c404010000    add     esp,104h
018017ae c20c00          ret     0Ch
```

My hunch paid off. We pop two registers, adjust *esp*, and then return with 12 bytes of extra cleanup.

```
0019fbf0  0000000e            ← pop edi
0019fbf4  00000111            ← pop esi
0019fbf8  00000000            ← esp points here after pop esi
0:000> dps 0019fbf8+0x104  ← simulate "add esp, 104h"
0019fcfc  01801fea contoso+0x1fea ← return address
0019fd00  00200cce            ← first four bytes of stack arguments
0019fd04  0000000e            ← next four bytes of stack arguments
0019fd08  00000000            ← last four bytes of stack arguments
0019fd0c  00000111            ← esp points here after ret 0Ch
```

Okay, that was a little trickier because the `ret 0Ch` means that after popping the return address, we also have to add `0Ch` to the *esp* register, leaving it at `0019fd0c`.

On to the next function.

```
contoso+0x1fea:
01801fea 0bc0            or      eax,eax
01801fec 0f85d6010000    jne     contoso+0x21c8 (018021c8)
01801ff2 8b44242c        mov     eax,dword ptr [esp+2Ch]
01801ff6 50              push    eax
01801ff7 57              push    edi
01801ff8 56              push    esi
01801ff9 53              push    ebx
01801ffa e831060000      call    contoso+0x2630 (01802630)
01801fff 5f              pop     edi
01802000 5e              pop     esi
01802001 5b              pop     ebx
01802002 83c410          add     esp,10h
01802005 c21000          ret     10h
```

This one is a little trickier, for even though the `ret` is in sight, there's another function call in between.

I'm going to assume that the function at `01802630` ends with a `ret 10h`, matching the 16 bytes of parameters pushed immediately prior to the `call`. This is generally a safe bet with the Microsoft C compiler, which prefers to create its entire stack frame at function entry and leave it alone until the function epilogue.

That means that the epilogue starts with the `pop edi`, and we can simulate those instructions as well.

```
0019fd0c  00000111                ← saved edi
0019fd10  00000000                ← saved esi
0019fd14  01801b90 contoso+0x1b90 ← saved ebx
0019fd18  00000070                                     \
0019fd1c  ffffffff                                      \ skipped by
0019fd20  ffffffff                                      / add esp, 10h
0019fd24  768617bb USER32!UserCallWinProcCheckWow+0x1fb /
0019fd28  7688311b USER32!_InternalCallWinProc+0x2b ← return address
0019fd2c  00200cce
0019fd30  00000111
0019fd34  0000000e
0019fd38  00000000
0019fd3c  00000000                ← esp points here after return
```

Hooray, we finally returned to a function we have symbols for! That means we can use the `k=` command to resume our stack trace.

The parameters to the `k=` command are

- The value to pretend is in *ebp*.
- The value to pretend is in *esp*.
- The value to pretend is in *eip*.

We will pretend that we are just about to execute the `ret 10h` instruction. From our calculations, therefore, immediately after the `ret 10h` instruction, the stack pointer is at `0019fd3c`, the instruction pointer is at `7688311b`, and the *ebp* register has the value… um, what's the value?

Look back through our notes for the most recent simulated `pop ebp`.

```
0019faf4  0019fd50       ← saved ebp
```

Ah, there it is. Okay, let's go for it.

```
0:000> k=0019fd50 0019fd28 768617bb
ChildEBP RetAddr
0019fd50 7686196c USER32!_InternalCallWinProc+0x2b
0019fe34 76860abe USER32!UserCallWinProcCheckWow+0x3ac
0019fea8 7687d750 USER32!DispatchMessageWorker+0x20e
0019feb0 018022d1 USER32!DispatchMessageA+0x10
0019ff70 765b60c9 contoso+0x22d1 ← UH-OH
0019ff80 77d43814 KERNEL32!BaseThreadInitThunk+0x19
0019ffdc 77d437e4 ntdll!__RtlUserThreadStart+0x2f
0019ffec 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Okay, this seems to look good, but there's that `contoso` on the stack again. However, this time, the debugger was able to walk the stack past that function. It could mean that the function was compiled with frame pointers enabled, in which case we have a valid stack trace.

Or it could mean that the function was compiled with frame pointers omitted, but the value in the *ebp* register happened to point to another frame, which is probably the next *ebp*-based frame.

Since debugging is an exercise in optimism, we'll assume that the stack trace is "good enough". It certainly looks reasonable. The *ebp* chain looks reasonable. The next frame is only slightly deeper on the stack. And even if there were some FPO functions in there, we can defer trying to tease them out until our investigation tells us that we need to do so.

So here's the stack trace we ended up with at the point we decided we had something "good enough":

```
ChildEBP RetAddr
0019ec98 5654ef4e combase!CoInitializeEx+0x35
0019ecf8 5654e70b WINSPOOL!GetCurrentNetworkId+0x36
0019ed28 5654e58a WINSPOOL!InternalGetDefaultPrinter+0x8b
0019ed58 75953b77 WINSPOOL!GetDefaultPrinterW+0x5a
0019ed70 7594e6b8 comdlg32!PrintGetDefaultPrinterName+0x17
0019f1b8 7594e520 comdlg32!PrintBuildDevNames+0x60
0019f1d0 75951340 comdlg32!PrintReturnDefault+0x30
0019f628 759a03ab comdlg32!PrintDlgX+0x132
0019fae0 01804a8e comdlg32!PrintDlgA+0x5b
0019fd50 7686196c contoso+0x4a8e
0019faf0 01803b8c contoso+0x3b8c \ we reconstructed these
0019fbec 01801325 contoso+0x1325  > three stack
0019fcfc 01801fea contoso+0x1fea / frames
0019fd50 7686196c USER32!_InternalCallWinProc+0x2b
0019fe34 76860abe USER32!UserCallWinProcCheckWow+0x3ac
0019fea8 7687d750 USER32!DispatchMessageWorker+0x20e
0019feb0 018022d1 USER32!DispatchMessageA+0x10
0019ff70 765b60c9 contoso+0x22d1 ← UH-OH
0019ff80 77d43814 KERNEL32!BaseThreadInitThunk+0x19
0019ffdc 77d437e4 ntdll!__RtlUserThreadStart+0x2f
0019ffec 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Now, sure, digging out those three stack frames doesn't look that useful because we don't have any symbols for Contoso at all, but you may be in a case where you do have symbols for Contoso, but those symbols lack FPO information. In that case, reconstructing stack frames gives you a proper stack trace as if you had FPO information all along.

And those extra stack frames may be the difference between a "How did we get here?" and a "Oh, this is how we got here."

Next time, we'll look at some compiler code generation idioms.

Raymond Chen

Follow