How to rescue a broken stack trace: Recovering the EBP chain

devblogs.microsoft.com/oldnewthing/20110309-00

March 9, 2011

Raymond Chen

When debugging, you may find that the stack trace falls apart:

```
ChildEBP RetAddr
001af118 773806a0 ntdll!KiFastSystemCallRet
001af11c 7735b18c ntdll!ZwWaitForSingleObject+0xc
001af180 7735b071 ntdll!RtlpWaitOnCriticalSection+0x154
001af1a8 2f6db1a9 ntdll!RtlEnterCriticalSection+0x152
001af1b4 2fe8d533 ABC!CCriticalSection::Lock+0x12
001af1d0 2fe8d56a ABC!CMessageList::Lock+0x24
001af234 2f6e47ac ABC!CMessageWindow::UpdateMessageList+0x231
001af274 2f6f040e ABC!CMessageWindow::UpdateContents+0x84
001af28c 2f6e4474 ABC!CMessageWindow::Refresh+0x1a8
001af360 2f6e4359 ABC!CMessageWindow::OnChar+0x4c
001af384 761a1a10 ABC!CMessageWindow::WndProc+0xb31
00000000 00000000 USER32!GetMessageW+0x6e
```

This can't possibly be the complete stack. I mean, where's the thread procedure? That should be at the start of the stack for any thread.

What happened is that the EBP chain got broken, and the debugger can't walk the stack any further. If the code was compiled with frame pointer optimization (FPO), then the compiler will not create EBP frames, permitting it to use EBP as a general purpose register instead. This is great for optimization, but it causes trouble for the debugger when it tries to take a stack trace through code compiled with FPO for which it does not have the necessary information to decode these types of stacks.

Begin digression: Traditionally, every function began with the sequence

```
push ebp      ;; save caller's EBP
mov ebp, esp  ;; set our EBP to point to this "frame"
sub esp, n    ;; reserve space for local variables
```

and ended with

```
        mov esp, ebp  ;; discard local variables
        pop ebp       ;; recover caller's EBP
        ret n
```

This pattern is so common that the x86 has dedicated instructions for it. The `ENTER n,0` instruction does the `push / mov / sub`, and the `LEAVE` instruction does the `mov / pop`. (In C/C++, the value after the comma is always zero.)

if you look at what this does to the stack, you see that this establishes a linked list of what are called *EBP frames*. Suppose you have the following code fragment:

```
void Top(int a, int b)
{
 int toplocal = b + 5;
 Middle(a, local);
}

void Middle(int c, int d)
{
 Bottom(c+d);
}

void Bottom(int e)
{
 int bottomlocal1, bottomlocal2;
 ...
}
```

When execution reaches the `...` inside function `Bottom` the stack looks like the following. (I put higher addresses at the top; the stack grows downward. I also assume that the calling convention is `__stdcall` and that the code is compiled with absolutely no optimization.)

`Top`'s stack frame	`0040F8F8`	parameter `b` passed to `Top`	During execution of `Top`, ← `EBP = 0040F8EC`
	`0040F8F4`	parameter `a` passed to `Top`	
	`0040F8F0`	return address of `Top`'s caller	
	`0040F8EC`	EBP of `Top`'s caller	
	`0040F8E8`	`toplocal`	
`Middle`'s stack frame	`0040F8E4`	parameter `d` passed to `Middle`	During execution of `Middle`, ← `EBP = 0040F8D8`
	`0040F8E0`	parameter `c` passed to `Middle`	

	`0040F8DC`	return address of `Middle`'s caller	
	`0040F8D8`	`0040F8EC` = EBP of `Middle`'s caller	
`Bottom`'s stack frame	`0040F8D4`	parameter `e` passed to `Bottom`	During execution of `Bottom`, ← `EBP = 0040F8CC`
	`0040F8D0`	return address of `Bottom`'s caller	
	`0040F8CC`	`0040F8D8` = EBP of `Bottom`'s caller	
	`0040F8C8`	`bottomlocal1`	
	`0040F8C4`	`bottomlocal2`	

Each stack frame is identified by the `EBP` value which the function uses during its execution.

The structure of each stack frame is therefore

`[ebp+n]`	Offsets greater than 4 access parameters
`[ebp+4]`	Offset 4 is the return address
`[ebp+0]`	Zero offset accesses caller's `EBP`
`[ebp-n]`	Negative offsets access locals

And the stack frames are all connected to each other in the form of a linked list threaded through the `EBP` values. This linked list is known as the *EBP chain*. **End digression**.

To recover from the broken EBP chain, start dumping the stack a little before things go bad (in this case, I would start at `001af384-80`) and then look for something that looks like a valid stack frame. Since the parameters and locals to a function can be pretty much anything, all you have left to work with is the `EBP` and the return address. In other words, you are looking for pairs of values of the form

```
«pointer a little higher up the stack».
«code address»
```

In this case, I got lucky and didn't have to go very far:

```
  001af474  00000000
 -001af478  001af494
/ 001af47c  14f4fba8 DEF!SubclassBase::CallOriginalWndProc+0x1a
| 001af480  2f6e4317 ABC!CMessageWindow::WndProc
| 001af484  00970338
| 001af488  0000000f
| 001af48c  00000000
\ 001af490  00000000
 >001af494  001af4f0
  001af498  14f4fcd6 DEF!SubclassBase::ForwardMessage+0x23
  001af49c  00970338
  001af4a0  0000000f
  001af4a4  00000000
  001af4a8  00000000
  001af4ac  00000000
  001af4b0  2f6e4317 ABC!CMessageWindow::WndProc
  001af4b4  ed758311
  001af4b8  00000000
  001af4bc  15143f70
  001af4c0  00000000
  001af4c4  14f4fb8e DEF!CView::SortItems+0x96
  001af4c8  00000000
  001af4cc  2f6e4317 ABC!CMessageWindow::WndProc
  001af4d0  00000000
```

At stack address `001af478`, we have a pointer to memory higher up the stack followed by a code address. if you follow that pointer, it points to another instance of the same pattern: A pointer higher up the stack followed by the code address.

Once you find where the EBP chain resumes, you can ask the debugger to resume its stack trace from that point with the `=n` option to the `k` command.

```
0:000> k=001af478
ChildEBP RetAddr
001af478 14f4fba8 ntdll!KiFastSystemCallRet
001af494 14f4fcd6 DEF!SubclassBase::CallOriginalWndProc+0x1a
001af4f0 14f4fc8b DEF!SubclassBase::ForwardMessage+0x23
001af514 14f32dd1 DEF!SubclassBase::ForwardChar+0x59
001af530 14f4fcd6 DEF!SubclassBase::OnChar+0x3c
001af58c 14f4fd76 DEF!HelpSubclass::WndProc+0x51
001af5e4 761a1a10 DEF!SubclassBase::s_WndProc+0x1b
001af610 761a1ae8 USER32!GetMessageW+0x6e
001af688 761a1c03 USER32!GetMessageW+0x146
001af6e4 761a3656 USER32!GetMessageW+0x261
001af70c 77380e6e USER32!OffsetRect+0x4d
001af784 761a2a98 ntdll!KiUserCallbackDispatcher+0x2e
001af794 698fd0aa USER32!DispatchMessageW+0xf
001af7a4 2f7bf15c ABC!CThread::DispatchMessageW+0x23
001af7e0 2f7befc9 ABC!CMessageWindow::MessageLoop+0x3a2
001af808 2ff56d20 ABC!CMessageWindow::ThreadProc+0x9f
001af898 75c2384b ABC!CMessageWindow::s_ThreadProc+0x10
001af8a4 7735a9bd kernel32!BaseThreadInitThunk+0x12
001af8e4 00000000 ntdll!LdrInitializeThunk+0x4d
```

When you do this, make sure to ignore the first line of the resumed stack trace, since that is based on your current `EIP`, not the return address stored in the stack frame.

Today was really just a warm-up for another debugging technique that I haven't finished writing up yet, so you're just going to be in suspense for another two years or so, though if you attended my TechEd China talk, you already know where I'm going.

Bonus reading: In Ryan Mangipano's two-part series on kernel mode stack overflows, the second part does a bit of EBP chain chasing. (Feel free to read the first part, as well as earlier discussion on the subject of stack overflows.)

Raymond Chen

Follow