How to rescue a broken stack trace on x64: Recovering the stack pointer

devblogs.microsoft.com/oldnewthing/20130906-00

September 6, 2013

Raymond Chen

Recovering a broken stack on x64 machines on Windows is trickier because the x64 uses unwind codes for stack walking rather than a frame pointer chain. When you dump the stack, all you're going to see is return addresses sprinkled in amongst the stack data.

Begin digression: According to the x64 ABI, each function must begin with a prologue which sets up the stack frame. It traditionally goes something like this:

```
    push rbx ;; save registers
    push rsi ;; save registers
    push rdi ;; save registers
    sub rsp, 0x20 ;; allocate space for local variables and outbound calls
```

Suppose we have functions

```
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 code is compiled with absolutely no optimization.)

`0040F8E8`	parameter 4 (unused)

	`0040F8E0`	parameter 3 (unused)	
	`0040F8D8`	parameter `b` passed to `Top`	
	`0040F8D0`	parameter `a` passed to `Top`	
`Top`'s stack frame	`0040F8C8`	return address of `Top`'s caller	During execution of `Top`, ← `rsp = 0040F8A0`
	`0040F8C0`	`toplocal`	
	`0040F8B8`	parameter 4 (unused)	
	`0040F8B0`	parameter 3 (unused)	
	`0040F8A8`	parameter `d` passed to `Middle`	
	`0040F8A0`	parameter `c` passed to `Middle`	
`Middle`'s stack frame	`0040F898`	return address of `Middle`'s caller	During execution of `Middle`, ← `rsp = 0040F870`
	`0040F890`	padding for alignment	
	`0040F888`	parameter 4 (unused)	
	`0040F880`	parameter 3 (unused)	
	`0040F878`	parameter 2 (unused)	
	`0040F870`	parameter `e` passed to `Bottom`	
`Bottom`'s stack frame	`0040F868`	return address of `Bottom`'s caller	During execution of `Bottom`, ← `rsp = 0040F830`
	`0040F860`	padding for alignment	
	`0040F858`	`bottomlocal1`	
	`0040F850`	`bottomlocal2`	
	`0040F848`	parameter 4	
	`0040F840`	parameter 3	
	`0040F838`	parameter 2	
	`0040F830`	parameter 1	

Of course, once the optimizer kicks in, there will also be saved registers in the stack frame, the unused space will start getting used as scratch variables, and the parameters will almost certainly not be spilled into their home locations. **End digression**.

Consider this crash where we started executing random instructions (data in the code segment) and finally trapped.

```
0:000> r
rax=0000000000000000 rbx=0000000000000005 rcx=0000000000000042
rdx=0000000000000010 rsi=00000000000615d4 rdi=00000000043f48e0
rip=0000000000000000 rsp=00000000001ebf68 rbp=00000000043f32d0
 r8=00000000001ebfd0  r9=0000000000000000 r10=000000007fff3cae
r11=0000000000000000 r12=0000000000000002 r13=0000000000517050
r14=0000000000000000 r15=00000000043f55c0
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
ABC!RandomFunction+0x1234:
00000000`ff6ebaad test    byte ptr [rax+rdx*4],ah ds:00000000`00000040=??
0:000> k
Child-SP          RetAddr           Call Site
00000000`001ebf70 00000000`00000004 ABC!RandomFunction+0x1234
00000000`001ebf78 00000000`0000000e 0x4
00000000`001ebf80 00000000`00000000 0xe
```

Not very helpful. Let's try to reconstruct the call stack. Here's what we have right now:

```
001ebf70  00000000`00000004
001ebf78  00000000`0000000e
001ebf80  00000000`00000000
001ebf88  00000000`77ba21bc ntdll!RtlAllocateHeap+0x16c
001ebf90  00000000`00000000
001ebf98  00000000`ff6e1fa1 ABC!operator new[]+0x20
001ebfa0  00000000`00000000
001ebfa8  00000000`ff6e28ae ABC!DoesUserPreferMetricUnits+0x2a
001ebfb0  00000000`000615d4
001ebfb8  00000000`043f48e0
001ebfc0  00000000`00000002
001ebfc8  00000000`00517050
001ebfd0  00000000`00000010
001ebfd8  00000000`00000000
001ebfe0  00000000`00000005
001ebfe8  00000000`ff6e2b9b ABC!CUIController::UpdateTwoLineDisplay+0x156
001ebff0  00000000`00000002
001ebff8  00000000`005170f0
001ec000  00000000`043f55c0
001ec008  00000000`00510000
001ec010  00000000`00000000
001ec018  00000000`00000000
001ec020  00000000`00000002
001ec028  00000000`005170f0
001ec030  00000000`00000000
001ec038  00000000`00000000
001ec040  00000000`00000002
001ec048  00000000`005170f0
001ec050  00000000`005170f8
001ec058  00000000`ff6e2a94 ABC!CUIController::displayEvent+0xea
001ec060  00000000`00750ed0
001ec070  00000000`00517118
001ec078  00000000`043f5aa0
001ec080  00000000`005170f8
001ec088  00000000`ff6e2f70 ABC!CEventRegistry::fire+0x34
001ec090  00000000`00518090
001ec098  00000000`00517118
001ec0a0  00000000`043f5aa0
001ec0a8  00000000`0000000e
001ec0b0  00000000`00000000
001ec0b8  00000000`00000000
001ec0c0  00000000`043f2f00
001ec0c8  00000000`ff6e2eef ABC!CCalculatorState::storeAndFire+0x126
001ec0d0  00000000`043f5aa0
001ec0d8  00000000`00000000
001ec0e0  00000000`001ec180
001ec0e8  00000000`00000000
```

(Note that this dump shows addresses increasing *downward*, whereas the previous diagram had them increasing *upward*. Being able to read stack dumps comfortably in both directions is one of those skills you develop as you gain experience.)

There is no frame pointer chain here to help you see if what you found is a call frame. You just have to use your intuition based on the function names. For example, it sounds perfectly reasonable for `operator new[]` to call `RtlAllocateHeap` (to allocate memory), but `DoesUserPreferMetricUnits` is probably not going to call `operator new[]`.

Some disassembling around of candidate return addresses suggests that the `DoesUserPreferMetricUnits` is the one likely to have jumped into space, because it is calling through a function pointer variable, whereas the other candidate return addresses used a direct call (or a call to an import table entry, which is unlikely to be invalid).

How do we reconstruct the stack based on this assumption? You trick the debugger into thinking that execution stopped inside the `DoesUserPreferMetricUnits` just before or after the fateful jump. It's easier to do "just after", since that's just the return address. We're going to pretend that instead of jumping into space, we jumped to a `ret` instruction.

Since we don't know what the junk code did before it finally crashed, the current value of `rsp` is probably not accurate. We'll have to think backward to a point in time whose stack pointer we can infer, and then replay the code forward.

From our knowledge of stack frames, we see that the `rsp` register had the value `001ebfb0` during the execution of `DoesUserPreferMetricUnits` just before it called the bad function pointer. Let's temporarily set our `rsp` and `rip` to simulate the return from the function.

```
0:000> r rsp=1ebfb0
0:000> r rip=ff6e28ae
0:000> k
Child-SP          RetAddr           Call Site
00000000`001ebfb0 00000000`ff6e2b9b ABC!DoesUserPreferMetricUnits+0x2a
00000000`001ebff0 00000000`ff6e2a94 ABC!CUIController::UpdateDisplay+0x156
00000000`001ec060 00000000`ff6e2f70 ABC!CUIController::displayEvent+0xea
00000000`001ec090 00000000`ff6e2eef ABC!CEventRouter::fire+0x34
00000000`001ec0d0 00000000`ff6e3469 ABC!CEngineState::storeAndFire+0x126
00000000`001ec110 00000000`ff6e4149 ABC!CEngine::SetDisplayText+0x39
00000000`001ec140 00000000`ff6ea48d ABC!CEngine::DisplayResult+0x648
00000000`001ec3c0 00000000`ff6e49c6 ABC!CEngine::ProcessCommandWorker+0xa1a
00000000`001ec530 00000000`ff6e4938 ABC!CEngine::ProcessCommand+0x2a
00000000`001ec560 00000000`ff6e460a ABC!CUIController::ProcessInput+0xaa
00000000`001ec5a0 00000000`ff6e4744 ABC!CContainer::ProcessInputs+0x7a1
00000000`001ec700 00000000`77a6c3c1 ABC!CContainer::WndProc+0xa12
00000000`001ecbe0 00000000`77a6a6d8 USER32!UserCallWinProcCheckWow+0x1ad
00000000`001ecca0 00000000`77a6a85d USER32!SendMessageWorker+0x682
00000000`001ecd30 00000000`ff70c5d8 USER32!SendMessageW+0x5c
00000000`001ecd80 00000000`77a5e53b ABC!CMainDlgFrame::MainDlgProc+0x87
00000000`001ecdc0 00000000`77a5e2f2 USER32!UserCallDlgProcCheckWow+0x1b6
00000000`001ece80 00000000`77a5e222 USER32!DefDlgProcWorker+0xf1
00000000`001ecf00 00000000`77a6c3c1 USER32!DefDlgProcW+0x36
00000000`001ecf40 00000000`77a6a6d8 USER32!UserCallWinProcCheckWow+0x1ad
00000000`001ed000 00000000`77a6a85d USER32!SendMessageWorker+0x682
00000000`001ed090 000007fe`fc890ba3 USER32!SendMessageW+0x5c
00000000`001ed0e0 000007fe`fc8947e2 COMCTL32!Button_ReleaseCapture+0x157
00000000`001ed120 00000000`77a6c3c1 COMCTL32!Button_WndProc+0xcde
00000000`001ed1e0 00000000`77a6c60a USER32!UserCallWinProcCheckWow+0x1ad
00000000`001ed2a0 00000000`ff6e1a76 USER32!DispatchMessageWorker+0x3b5
00000000`001ed320 00000000`ff6fa00f ABC!WinMain+0x1db4
00000000`001efa10 00000000`7794f33d ABC!__mainCRTStartup+0x18e
00000000`001efad0 00000000`77b82ca1 kernel32!BaseThreadInitThunk+0xd
00000000`001efb00 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
0:000> r rsp=001ebf68
0:000> r rip=ff6ebaad
```

After getting what we want, we restore the registers to their original values at the time of the crash so that future investigation won't be misled by our editing.

Raymond Chen

Follow