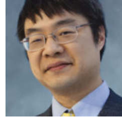
Debugging a GDI resource leak: Case study

devblogs.microsoft.com/oldnewthing/20170519-00

May 19, 2017



Raymond Chen

I was asked to help debug a problem. A program was leaking GDI bitmaps like crazy, and after a while, the GDI resource handle count reached 9,999, at which point GDI said, "That's it, I'm cutting you off."

The problem isn't discovered until after the limit has been reached.

To debug this, I'm gong to use different poor man's way of identifying memory leaks. We begin the story with a GDI handle that has been identified as leaked: `0x13054e2f`. My first step is to get some basic information about this GDI object by simulating a call to `Get-Object`.

```
0:061> .dvalloc 1
Allocated 1000 bytes starting at 00000000`03610000
```

First, I allocated some memory that I can use to hold the `BITMAP` structure.

```
ntdll!DbgBreakPoint:
00007ffb`65ef7570 int     3
0:061> t
ntdll!DbgBreakPoint:
00007ffb`65ef7571 ret
0:061> r rip=gdi32!GetObjectW
0:061> r r8=0x00000000`03610000
0:061> r rdx=0x68
0:061> r rcx=0x13054e2f
0:061> r
GDI32!GetObjectW:
00007ff9`9658e2f0 mov     qword ptr [rsp+8],rbx
```

Next, I simulate a call to the `GetObjectW` function[1] by setting up the inbound parameter registers: `rcx` is the GDI object we are geting information about, `rdx` is the size of the output buffer, `r8` is the output buffer itself (which we just allocated).

I'm pulling a super-sneaky trick here. Normally, we reserve shadow space for the outbound call, and then simulate the `call` instruction by pushing the return address on the stack. But we didn't do any of that here. We just moved `rip` directly to the function we wanted to call.

But I can skip those steps because I stepped to the `ret` instruction. This means that the stack is already set up the way it would be immediately upon entry to the function. We are reusing the stack frame of the `DbgBreakPoint` function! We are reusing the shadow space provided by the caller, and we are taking advantage of the proper stack alignment established by the caller.

```
0:061> gu
ntdll!DbgUiRemoteBreakin+0x34:
00007ff9`9730f4f4 jmp     ntdll!DbgUiRemoteBreakin+0x36 (00007ff9`9730f4f6)
0:061> r
rax=0000000000000020 rbx=00007ff99730f4c0 rcx=00007ff9965b877a
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000000000000
rip=00007ff99730f4f4 rsp=0000000007a7f7e0 rbp=0000000000000000
 r8=0000000007a7f7a8  r9=0000000000000000 r10=0000000000000000
r11=0000000000000206 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
ntdll!DbgUiRemoteBreakin+0x34:
00007ff9`9730f4f4 jmp     ntdll!DbgUiRemoteBreakin+0x36 (00007ff9`9730f4f6)
0:061> dd 0x00000000`03610000 L8
00000000`03610000  00000000 00000010 00000010 00000002
00000000`03610010  00010001 fffff901 00000000 00000000
0:061> dt contoso!BITMAP 0x00000000`03610000
   +0x000 bmType           : 0n0
   +0x004 bmWidth          : 0n16
   +0x008 bmHeight         : 0n16
   +0x00c bmWidthBytes     : 0n2
   +0x010 bmPlanes         : 1
   +0x012 bmBitsPixel      : 1
   +0x018 bmBits           : (null)
```

I run the `GetObjectW` function with the `gu` command, which means "go until the current function returns". When it returns, I verify that the call succeeded (by checking the return value in `rax`), and the dump the `BITMAP` structure.

This tells me that I have a 16×16 monochrome bitmap. Monochrome bitmaps are rarely-used in Windows nowadays. One place you'll see them is in icons, since an icon consists of two bitmaps: A monochrome mask and a color image.

So let's assume that we're leaking the mask of an icon. These things come out of two functions: `GetIconInfo` and `GetIconInfoEx`, so I set breakpoints on both. (Actually three functions since `GetIconInfoEx` has both Unicode and ANSI variations.)

```
0:061> bp user32!geticoninfo
0:061> bp user32!geticoninfoexw
0:061> bp user32!geticoninfoexa
0:061> g
```

The breakpoint hits pretty quickly.

```
Breakpoint 0 hit
USER32!GetIconInfo:
00007ff9`96cf1dd0 sub     rsp,38h
0:005> kc3
Call Site
USER32!GetIconInfo
contoso!CNotificationIcon::IsIconCorrectSize+0x42
contoso!CNotificationIcon::Modify::__l6::
    <lambda_5a5c6bde8a112e005a026c6f34886eee>::operator()+0x1f
```

Going back to the source code for `IsIconCorrectSize` confirms that this function calls `GetIconInfo` (presumably to get the size of the icon) and forgets to delete the two bitmaps.

Root cause identified. The fix is to delete those bitmaps.

Forgetting to delete the bitmaps that come out of the `GetIconInfo` family of functions is a common mistake.

[1] The `ntsd` debugger doesn't have a C compiler built-in, so we have to build these things manually.

Raymond Chen

Follow