I have the handle to a file; how can I get the file name from the debugger?

devblogs.microsoft.com/oldnewthing/20151016-00

October 16, 2015

Raymond Chen

Suppose you're debugging and you find yourself with the handle to a file, and you would like to know what file that handle corresponds to.

Well, the way you would do this from code is to call `GetFinalPathNameByHandle`, but you're not in code; you're in the debugger.

But you can simulate what code did from the debugger.

Now, if `GetFinalPathNameByHandle` were a function you had written, you could just use the `.call` command to ask the debugger to build a call frame and call the function and then clean up. But since you didn't write that function, you'll have to build the call frame manually.

Let's say the handle you are interested is 0x5CC.

```
0:000> !handle 5cc f
Handle 5cc
  Type           File
  Attributes     0
  GrantedAccess 0x120089:
         ReadControl,Synch
         Read/List,ReadEA,ReadAttr
  HandleCount    2
  PointerCount  262145
  No Object Specific Information available
```

Yup, it's a file. Let's see what file it is. First, the x86 way. We start by recording the contents of the volatile registers so that we can restore them afterwards.

```
0:000> r
eax=00000925 ebx=00000003 ecx=267ad2be edx=00000000 esi=00000000 edi=00b300e0
eip=00b36e1d esp=0089f7cc ebp=0089f838 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00200246
contoso!CContoso::OnCommand+0x1a5:
00b36e1d 8bd1            mov     edx,ecx
```

Next, we allocate some memory to hold the result.

```
0:000> .dvalloc 1000
Allocated 1000 bytes starting at 05740000
```

Next, we build the function call. Since this is x86, all parameters go on the stack, and the stdcall calling convention pushes the last parameter first.

```
; simulate "push dwFlags"
0:000> r esp=esp-4;ed esp 0
; simulate "push cchFilePath"
0:000> r esp=esp-4;ed esp 0x800
; simulate "push lpszFilePath"
0:000> r esp=esp-4;ed esp 05740000
; simulate "push hFile"
0:000> r esp=esp-4;ed esp 5cc
; simulate "call kernelbase!GetFinalPathNameByHandleW"
0:000> r esp=esp-4;ed esp eip
0:000> r eip=kernelbase!GetFinalPathNameByHandleW
; execute until the function returns
0:000> g poi esp
eax=00000025 ebx=00000003 ecx=7715fb62 edx=00000046 esi=00000000 edi=00b300e0
eip=00b36e1d esp=0089f7cc ebp=0089f838 iopl=0         nv up ei pl zr na pe cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00200247
contoso!CContoso::OnCommand+0x1a5:
00b36e1d 8bd1            mov     edx,ecx
```

Okay, we've returned. We see that `eax` is nonzero, so the call succeeded. Let's look at the output buffer.

```
0:000> du 05740000
05740000  "\\?\C:\Users\Public\Documents\lo"
05740040  "g.txt"
```

There's our path to the file, which looks like a log file.

Now we have to play Boy Scout and leave things the way we found them.

```
; do not need to restore stack pointer since stdcall is callee-clean
0:000> r eax=925
0:000> r ecx=267ad2be
0:000> r edx=0
0:000> r efl=200246
0:000> r
eax=00000925 ebx=00000003 ecx=267ad2be edx=00000000 esi=00000000 edi=00b300e0
eip=00b36e1d esp=0089f7cc ebp=0089f838 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00200246
contoso!CContoso::OnCommand+0x1a5:
00b36e1d 8bd1            mov     edx,ecx
```

Most of the registers are preserved by the function call, but we need to restore the volatile registers `eax`, `ecx`, `edx`, and flags.

In practice, you may be able to get away with not restoring everything. For example, flags may not be meaningful at the point in the code you broke in. And if you broke in immediately after a function call or immediately on entry to a function, the values in the nonvolatile registers are already trashed or trashable, so you don't need to restore them either.

Next up is x64. This is harder because of the shadow space and many more registers. Again, we start by recording the contents of the volatile registers so that we can restore them afterwards.

```
0:000> r
rax=0000000000000a0c rbx=0000000000000000 rcx=230fe38816f90000
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000080004005
rip=000007f709e471fc rsp=0000002e45a9f880 rbp=0000002e45a9f949
 r8=0000000000000000  r9=000007fbab8aabd0 r10=0000000000000003
r11=0000002e45a9f2a8 r12=0000000000000000 r13=0000000000000001
r14=ffffffffffffffff r15=0000000000000000
iopl=0         nv up ei pl nz na po cy
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000207
contoso!CContoso::OnCommand+0x3b9:
000007f7`09e471fc 33c0            xor     eax,eax
```

Next, we allocate some memory to hold the result.

```
0:000> .dvalloc 1000
Allocated 1000 bytes starting at 0000002e`475e0000
```

Next, we build the function call. On x64, the first four parameters go into registers `rcx`, `rdx`, `r8`, and `r9`, and corresponding shadow space is reserved on the stack.

```
; create shadow space
0:000> r rsp=@rsp-20
; first parameter is hFile
0:000> r rcx=5cc
; second parameter is lpszFilePath
0:000> r rdx=0000002e`475e0000
; third parameter is cchFilePath
0:000> r r8=800
; fourth parameter is dwFlags
0:000> r r9=0
; simulate "call kernelbase!GetFinalPathNameByHandleW"
0:000> r rsp=@rsp-8; ep rsp @rip
0:000> r rip=kernelbase!GetFinalPathNameByHandleW
; execute until the function returns
0:000> g poi @rsp
0:000> r
rax=0000000000000025 rbx=0000000000000000 rcx=00000000ffffffff
rdx=0000000000008000 rsi=0000000000000000 rdi=0000000080004005
rip=000007f709e471fc rsp=0000002e45a9f860 rbp=0000002e45a9f949
 r8=0000002e4a159390  r9=000007fbab91a3a0 r10=0000000000000003
r11=0000002e45a9f3a8 r12=0000000000000000 r13=0000000000000001
r14=ffffffffffffffff r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
contoso!CContoso::OnCommand+0x3b9:
000007f7`09e471fc 33c0            xor     eax,eax
```

Again, we look at the result.

```
0:000> du 0000002e`475e0000
0000002e`475e0000  "\\?\C:\Users\raymondc\Desktop\lo"
0000002e`475e0040  "g.txt"
```

And again, we need to undo everything we did so the application doesn’t freak out.

```
0:000> r rsp=@rsp+20
0:000> r rax=0a0c
0:000> r rcx=230fe38816f90000
0:000> r rdx=0
0:000> r r8=0
0:000> r r9=7fbab8aabd0
0:000> r r10=3
0:000> r r11=2e45a9f2a8
0:000> r efl=207
0:000> r
rax=0000000000000a0c rbx=0000000000000000 rcx=230fe38816f90000
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000080004005
rip=000007f709e471fc rsp=0000002e45a9f880 rbp=0000002e45a9f949
 r8=0000000000000000  r9=000007fbab8aabd0 r10=0000000000000003
r11=0000002e45a9f2a8 r12=0000000000000000 r13=0000000000000001
r14=ffffffffffffffff r15=0000000000000000
iopl=0         nv up ei pl nz na po cy
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000207
contoso!CContoso::OnCommand+0x3b9:
000007f7`09e471fc 33c0            xor     eax,eax
```

Again, most of the registers are preserved by the function call, but we need to restore the volatile registers `eax` , `ecx` , `edx` , `r8` through `r11` , and flags.

Exercise: I could've skipped restoring `rax` and flags. Why?

Raymond Chen

Follow