Diagnosing a crash when trying to call ReadFile via language interop

devblogs.microsoft.com/oldnewthing/20230831-00

August 31, 2023

Raymond Chen

A customer was having trouble calling the `ReadFile` function with the interop feature of their language. It crashed with

```
(df4.1ef4): Access violation - code c0000005 (first chance)

rax=0000000001260000 rbx=00000093834eb020 rcx=0000000000000264
rdx=00000093834eb028 rsi=0000000000000264 rdi=0000000000000000
rip=00007ff9eaddaf5f rsp=00000093834eaf50 rbp=00000093834eb029
 r8=000000000000003c  r9=00000093834eb020 r10=00000093834eb028
r11=0000000000000246 r12=fffffff89010f67 r13=0000000000000001
r14=000000d900000000 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=00010206
KERNELBASE!ReadFile+0xaf:
00007ff9`eaddaf5f mov     [r14],103h ds:000000d9`00000000=????????????????
```

Their interop declaration looked something like this.

```
function ReadFile(
    handle: integer,
    buffer: array of byte,
    length: integer,
    actual: ref integer,
    overlapped: integer): integer;

// called as follows

var handle: integer; /* assume we get a handle somehow */
var buffer: array(256) of byte;
var actual: integer;
var result: integer;

result = ReadFile(handle, buffer, 256, actual, 0);
```

From the crash, we see that we are moving the value `0x0103` into an invalid pointer.

Can you solve the mystery?

Hint: The disassembly shows that this code is running in 64-bit mode.

Another hint: The value `0x103` is `STATUS_PENDING`.

Yet another hint: The invalid address happens to be an exact multiple of 4GB.

Okay, let's put the hints together.

The last hint steers you to the answer: In the `ReadFile` function, the `handle` and `overlapped` parameters are pointer-sized, whereas the `length` and `actual` parameters are 32-bit integers, but the interop declaration calls them all the same thing: `integer`. I'm guessing that in this language, `integer` is a 32-bit integer.

Due to the size mismatch, the caller puts a 32-bit zero on the stack, but the function expects a pointer and reads a 64-bit value. As a result, only the lower-order 32 bits end up zero, and the upper 32 bits contain uninitialized stack data.

The other clue that the problem is with the `overlapped` parameter is that the first use of the invalid pointer is to write `STATUS_PENDING` to it, which makes sense because that's what gets written to an `OVERLAPPED` structure to indicate that the I/O is in progress.

I mentioned that the first parameter `handle` is also pointer-sized, yet nothing appears to go wrong with the handle. Why do we get away with an incorrect declaration for `handle`, but not for `overlapped`?

The Windows x86-64 calling convention puts the first four parameters in registers, so the `handle` parameter goes into the `rcx` register. The x86-64 architecture has the policy that if you load a 32-bit value into a 64-bit register, the upper 32 bits are set to zero by default.[1] The value being loaded into the `rcx` register almost certainly came from memory or another register, and the load of a 32-bit value into that register will implicitly zero-extend the value to 64 bits.

[1] There is a separate instruction for loading with sign extension.

Related reading: Which processors prefer sign-extended loads, and which prefer zero-extended loads?