The case of the buffer overflow vulnerability that was neither a buffer overflow nor a vulnerability

devblogs.microsoft.com/oldnewthing/20181207-00

December 7, 2018

Raymond Chen

A security vulnerability report claimed to have found a buffer overrun. Their instructions were to perform a specific sequence of operations in Explorer while watching the output in the debugger:

```
pcshell\shell\explorer\something.cpp(80)\explorer.exe!00007FF813D017E2: (caller:
00007FF813CB690C) ReturnHr(7) tid(b2c) 80070002 The system cannot find the file
specified
pcshell\shell\explorer\something.cpp(80)\explorer.exe!00007FF813D017E2: (caller:
00007FF813CB690C) ReturnHr(8) tid(b2c) 80070002 The system cannot find the file
specified
pcshell\shell\explorer\something.cpp(80)\explorer.exe!00007FF813D017E2: (caller:
00007FF813CB690C) ReturnHr(9) tid(b2c) 80070002 The system cannot find the file
specified
```

Oh, I forgot to mention that the finder didn't copy the text out of the debugger. Nor did they attach a screen shot of the debugger. Nope, they attached a digital photo of their CRT monitor, complete with Moiré pattern.

I'll remove the Moiré pattern for you from this and all the other screen shots. I'll also do the organic OCR[1] and convert it to text. Because I'm a nice guy that wants his writing to be readable. (Unlike the finder, apparently.)

```
pcshell\shell\explorer\something.cpp(80)\explorer.exe!00007FF813D017E2: (caller:
00007FF813CB690C) ReturnHr(7) tid(b2c) 80070002 The system cannot find the file
specified
pcshell\shell\explorer\something.cpp(80)\explorer.exe!00007FF813D017E2: (caller:
00007FF813CB690C) ReturnHr(8) tid(b2c) 80070002 The system cannot find the file
specified
pcshell\shell\explorer\something.cpp(80)\explorer.exe!00007FF813D017E2: (caller:
00007FF813CB690C) ReturnHr(9) tid(b2c) 80070002 The system cannot find the file
specified
```

Okay, this is some debug spew that's logging a *File not found* error.

The finder, however, didn't characterize it as debug spew, but rather described it as "an exception". Their next step was to set a breakpoint on the reported address `00007FF813D017E2` and repeat the scenario, and then when the breakpoint hits, take a stack trace.

```
0:002> k
 # Child-SP
00 0000009c`aa2ffae0 00007ff8`13cb690c explorer!
<lambda_bab32d760b0e6e31>::operator()+0x704c2
01 0000009c`aa2ffb90 00007ff8`13cb5eed
explorer!Windows::Internal::ComTaskPool::CThread::_ThreadProc+0x228
02 0000009c`aa2ffc80 00007ff8`13cb5e39
explorer!Windows::Internal::ComTaskPool::CThread::s_ExecuteThreadPr
03 0000009c`aa2ffcd0 00007ff8`91adc774
explorer!Windows::Internal::ComTaskPool::CThread::s_ThreadProc+0x9
04 0000009c`aa2ffd00 00007ff8`91defd51 kernel32!BaseThreadInitThunk+0x14
05 0000009c`aa2ffd30 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

A little disassembling around the first return address reveals

```
00007ff8`13cb6906 ff1557351100 call    qword ptr [explorer!_guard_dispatch_icall_ptr
(00007ff8`3cbfe08)]
00007ff8`13cb690c 488b4b50     mov     rcx, qword ptr [rbx+50h]
```

The finder explained that "the function is `_guard_ dispatch_ icall_ fptr`, not the `lambda_ bab32d...`. Decompiling the code at `_guard_ dispatch_ icall_ fptr`, you'll see the buffer overflow." Here's the code in question:

```
0:002> u 00007ff8`13cbfe08
explorer!_guard_dispatch_icall_fptr:
00007ff8`13cbfe08 206388     and     byte ptr [rbx-78h],ah
00007ff8`13cb6e0b 9a         ??? ⇐
```

In the screen shot, they highlighted the second line. And that was the end of their report.

It seems that the finder saw a debug message, and then started fumbling around. They saw that the disassembler printed `???` which they interpreted as a clear sign of a security vulnerability.

But that's not what they found.

First of all, the messages that appear on the debugger are some error logging trace messages. They aren't exceptions.

Next, the instruction prior to the return address is an indirect call, but they somehow thought this was a label, because they said that the function was misreported by the debugger as `lambda_ bab32d...` when it was actually `_guard_ dispatch_ icall_ fptr`. Nope,

the function really is `lambda_ bab32d...`. What it's doing is *calling through* the function pointer `_guard_ dispatch_ icall_ fptr`.

Next, the finder misread the disassembly. In Intel notation, square brackets around an effective address mean "the memory stored at this address". The `call qword ptr [_guard_ dispatch_ icall_ fptr]` doesn't mean "call the function `_guard_ dispatch_ icall_ fptr`." It means "read eight bytes from `_guard_ dispatch_ icall_ fptr`, treat those eight bytes as a 64-bit value, interpret that value as an address, and call to that address." The bytes you see at `_guard_ dispatch_ icall_ fptr` are not code; they are data. Disassembling them as code is meaningless.

So +1 on the finder for resourcefulness and curiosity. But −1 for not really understanding what you are doing, −1 for not checking your work with a colleague before sending it in, and −1 for poor presentation.

[1] In other words, I used my eyeballs.

Raymond Chen

Follow