Debugging coroutine handles: Looking for the source of a one-byte memory corruption

devblogs.microsoft.com/oldnewthing/20220930-00

September 30, 2022

Raymond Chen

A team was chasing a one-byte use-after-free memory corruption bug. These bugs are really frustrating to chase down because the memory corruption typically doesn't trigger an immediate crash, but rather results in a delayed crash, which means that the culprit has done the damage and run away long before the problem is detected.

We join the debugging session already in progress. We have determined that the corruption is to a memory block that previously contained a coroutine frame at offset `0xc0`.

The state machine of a coroutine exists in the `_ResumeCoro$2` function, so we can start there:

```
contoso!DoStuffLater$_ResumeCoro$2:
    mov     r11,rsp
    ... stack frame nonsense ...

    mov     rsi,rcx                 // rsi = coroutine frame pointer
    mov     [rsp+28h],rcx
    movzx   eax,word ptr [rcx+8]    // eax = coroutine state
    mov     [rsp+20h],ax
    inc     ax                      // artificially add 1
    cmp     ax,8
    ja      contoso!DoStuffLater$_ResumeCoro$2+0x3e5
    ja      00007ffc`e777a5b5       // invalid index, die (jump to int 3)

    movsx   rax,ax
    lea     rdx,[contoso!__ImageBase]
    mov     ecx,[rdx+rax*4+1BA5E0h] // look up jump table RVA
    add     rcx,rdx                 // convert to absolute address
    jmp     rcx                     // jump there

contoso!DoStuffLater$_ResumeCoro$2+0x3e5:
    int     3
```

We see from the disassembly that the jump table starts at relative offset `0x1ba5e0`. We won't dig into the jump table yet; let's see if we can find the corruption point, which is a single-byte corruption at offset `0xc0` from the start of the coroutine frame. Maybe we'll be lucky and the access is directly into the frame.

```
0:026> #c0h contoso!DoStuffLater$_ResumeCoro$2
contoso!DoStuffLater$_ResumeCoro$2+0x136:
    mov     [rsi+0C0h],al
```

Oh my goodness, we found a single-byte write at offset `0xc0` in the coroutine frame! Let's see who is doing it.

```
    mov     eax,6
    mov     [rsi+8],ax

    mov     rdx,rsi
    mov     rcx,rbx
    call
contoso!winrt::impl::notify_awaiter<`winrt::resume_foreground'::`2'::awaitable>::

await_suspend<std::experimental::coroutine_traits<winrt::fire_and_forget>::promise_typ

    mov     [rsi+0C0h],al   // WRITE HAPPENS HERE
```

The first two instructions set the coroutine state to 6, which happens as part of coroutine suspension.

The second group of instructions call the `await_suspend` for a `resume_foreground` awaiter. This is in code that is moving forward to state 6, and we know that the Microsoft compiler records coroutine states as even numbers starting at 2 (for the initial state), and then increases by two for each suspension point. Therefore, moving to state 6 means suspending for the second time.

```
winrt::fire_and_forget DoStuffLater()
{
    co_await winrt::resume_after(100ms);
    co_await winrt::resume_foreground(GetDispatcherQueue());
    DoStuff();
}
```

Okay, good, that second suspension theory lines up with the code: The second suspension is a call to `resume_foreground`, and the code showed that we were calling `resume_foreground`.

And we see the bug: The code is storing the result of `await_suspend` into the coroutine frame. This is something I called out in my C++ coroutines: Getting started with awaitable objects article:

> Therefore, it is important that your awaiter not use its `this` pointer once it has arranged for the handle to be invoked somehow, because the `this` pointer may no longer be valid.

In this case, not only did the awaiter get destructed, the entire coroutine frame was destructed!

The compiler team confirmed that this is a known code-generation bug, fixed in versions 16.11 and 17.0.

If you are stuck on 16.10 or older, you will have to work around the problem. From my investigation, it seems that the code generation problem occurs when you have an `await_suspend` that returns `bool`. In C++/WinRT, there are only four places where this happens:

- `resume_foreground(Windows::System::CoreDispatcher)`
- `resume_foreground(Microsoft::System::CoreDispatcher)`
- `deferrable_event_args.wait_for_deferrals()`
- `final_suspend`

In the first two cases, you can work around the problem by switching to the `wil::resume_foreground` function, which addresses this and other design issues with the original `winrt::resume_foreground` function.

If you'd rather not pull in another library, and you don't want to upgrade your compiler, you can work around the problem by using an explicit continuation-passing model:

```
winrt::fire_and_forget DoStuffLater()
{
    co_await winrt::resume_after(100ms);
    GetDispatcherQueue().EnqueueAsync([=]()
    {
        DoStuffLater();
    });
}
```

In the last case (`final_suspend`), my exploration suggests that the code generation problem does not occur in that case, so we're okay there.

But upgrade your compiler if you can.

Raymond Chen

Follow