The case of the mysterious "out of bounds" error from CreateUri and memmove

devblogs.microsoft.com/oldnewthing/20230220-00

February 20, 2023

Raymond Chen

A customer was trying to understand why their program was crashing with an `E_BOUNDS` error in what appears to be a call to `CreateUri`.

```
combase!RoOriginateErrorW+0x50
wincorlib!Platform::Details::ReCreateFromException+0x40
contoso!`__abi_translateCurrentException'::`1'::catch$0+0x10
contoso!memmove+0x217f4
contoso!Windows::Foundation::IUriRuntimeClassFactory::CreateUri+0x44
contoso!Contoso::DashboardView::DashboardView_obj1_Bindings::Update_ViewModel_Layout_C

contoso!Contoso::DashboardView::DashboardView_obj1_Bindings::Update_ViewModel_Layout+0

contoso!Contoso::DashboardView::DashboardView_obj1_Bindings::PropertyChanged+0x1134
contoso!XamlBindingInfo::XamlBindingTrackingBase::PropertyChanged+0x30
```

From the stack, it looks like `memmove` threw a `E_BOUNDS` C++/CX exception, which doesn't make sense. Even more mysteriously, the `memmove` was called from `CreateUri`, but their DashboardView doesn't manipulate URIs in any obvious way. It's just a stack trace of nonsense.

Let's try to unwind the nonsense.

As for the mysterious `memmove`, notice that the offset is `0x217f4`. It's unlikely that the `memmove` function is over 100KB in size. Let's see what's really going on there. This is just some code that has probably been shunted into a rarely-used code page far, far away from the rest of the code, and the nearest symbol to it happens to be `memmove`.

```
    xor     ecx,ecx
    call    contoso!__abi_translateCurrentException
    int     3       ; memmove+0x217f4
```

Yup, this is an exception rethrow. Since exceptions are rare, profile-guided optimization puts all the exception-handling nonsense into faraway pages so that they don't consume valuable space in the hot code pages.

So why is CreateUri throwing an “out of bounds” exception?

Well, are you sure it’s really `CreateUri`?

I looked a frame higher on the stack. “Why is data binding calling `CreateUri`?”

The data binding code is autogenerated by the XAML compiler; it’s not checked into the source tree. Instead of trying to figure out how to build their project (so I can extract the autogenerated file), maybe I can infer what’s going on from the source.

One basic assumption that you make about code in general is that people who write code are doing the best they can, rather than being sadists. This means that function names will generally be descriptive of what they do, variable names will generally be descriptive of what they represent, and so on. So when I see a class called `DashboardView_obj1_Bindings`, I’m going to assume that this class is for dealing with the bindings of some object in DashboardView, and since it has a method called `Update_ViewModel_Layout_Groups`, it probably has something to do with updating the binding of something whose names involve the words `ViewModel`, `Layout`, and `Groups`.

I looked at `DashboardView.xaml` and searched for the word `ViewModel` in elements that appeared to be involved with binding.

```
<ContentControl
    Grid.Row="0"
    x:Name="TogglesGroup"
    IsTabStop="False"
    Width="360"
    Content="{x:Bind ViewModel.Layout.Groups[0], Mode=OneWay}"
    ContentTemplateSelector="{StaticResource DashboardGroupTemplateSelector}"/>
```

Now, this wasn’t the first use of `x:Bind` in the XAML markup, so that doesn’t line up with `obj1`, but the other parts do line up (the `Layout` and `Groups`), so I chalked this up to “Maybe the XAML compiler generates bindings in some order other than the order they appear in the markup.”

How could this binding raise an “out of bounds” exception? Well, there’s a subscript operation, so maybe the `Groups` collection is empty.

I looked at the `Update_ViewModel_Layout_Groups` method to see if that theory lined up.

```
Update_ViewModel_Layout_Groups:
    test    rdx,rdx
    je      ...
    mov     qword ptr [rsp+8],rbx
    mov     qword ptr [rsp+18h],rbp

    push    rsi
    push    rdi
    push    r14
    sub     rsp,20h
    mov     rbp,rdx
    mov     rsi,rcx
    test    r8d,0C0000001h
    je      ...

    xor     edx,edx
    mov     rcx,rbp
    call    contoso!Windows::Foundation::IUriRuntimeClassFactory::CreateUri
```

The function starts with a shrink-wrapped early-out if the first parameter is zero. (This is a C++ method, so `rcx` contains `this` and `rdx` contains the first formal parameter.) I don't know how binding works, but presumably this is just a binding thing.

If the parameter is nonzero, then we build a proper stack frame, test some bits in the third parameter, and if they're set, we call, um, `CreateUri` with `nullptr`? That makes no sense. The XAML isn't asking for a URI, and why is this code trying to create a URI from an empty string?

But then you realize that you've been faked out by COMDAT folding. The `this` parameter for the call to `CreateUri` is supposed to be the `IUriRuntimeClassFactory`, but that's not what we're passing; we're passing the first formal parameter.

Really, this is a call to `IVector::GetAt`, and the parameter is zero, indicating that we want the object at index zero. The functions `IVector::GetAt` and `CreateUri` were folded because they happen to be byte-for-byte identical. They are both "Call the method at index 6 in the vtable with one parameter." For `IUriRuntimeClassFactory`, that method is `CreateUri` and the parameter is a string. For `IVector` that method is `GetAt` and the parameter is an index.

With this explanation, the customer realized that they did have an outstanding bug that said, "If our settings file is corrupted, we end up with no groups," and this bug is likely an alternate manifestation of that bug.