The case of the vector with an impossibly large size

devblogs.microsoft.com/oldnewthing/20240105-00

January 5, 2024

Raymond Chen

A customer had a program that crashed with this stack:

```
contoso!Widget::GetCost
contoso!StandardWidgets::get_TotalCost+0x12f
rpcrt4!Invoke+0x73
rpcrt4!Ndr64StubWorker+0xb9b
rpcrt4!NdrStubCall3+0xd7
combase!CStdStubBuffer_Invoke+0xdb
combase!ObjectMethodExceptionHandlingAction<<lambda_...> >+0x47
combase!DefaultStubInvoke+0x376
combase!ServerCall::ContextInvoke+0x6f3
combase!ComInvokeWithLockAndIPID+0xacb
combase!ThreadInvoke+0x103
rpcrt4!DispatchToStubInCNoAvrf+0x18
rpcrt4!RPC_INTERFACE::DispatchToStubWorker+0x1a9
rpcrt4!RPC_INTERFACE::DispatchToStubWithObject+0x1a7
rpcrt4!LRPC_SCALL::DispatchRequest+0x308
rpcrt4!LRPC_SCALL::HandleRequest+0xdcb
rpcrt4!LRPC_SASSOCIATION::HandleRequest+0x2c3
rpcrt4!LRPC_ADDRESS::HandleRequest+0x183
rpcrt4!LRPC_ADDRESS::ProcessIO+0x939
rpcrt4!LrpcIoComplete+0xff
ntdll!TppAlpcpExecuteCallback+0x14d
ntdll!TppWorkerThread+0x4b4
kernel32!BaseThreadInitThunk+0x18
ntdll!RtlUserThreadStart+0x21
```

They wondered if some recent change to Windows was the source of the problem, since it didn’t happen as much in earlier versions of Windows.

The stack trace pointed to `Widget::IsEnabled`, which was crashing on the first instruction because it was given an invalid `this` pointer.

```
00007fff`73a8a59f mov edx,dword ptr [rcx+40h]
                                    ds:00000000`00000040=????????
```

The `Widget` pointer came from a `std::vector` that is a member of the `StandardWidgets` class.

```
using namespace Microsoft::WRL;

class StandardWidgets : RuntimeClass<IStandardWidgets, FtmBase>
{
    IFACEMETHODIMP get_TotalCost(INT32* result);
    ⟦ other methods not relevant here ⟧

private:
    HRESULT LazyInitializeWidgetList();

    static constexpr PCWSTR standardWidgetNames[] = {
        L"Bob", L"Carol", L"Ted", L"Alice" };
    static constexpr int standardWidgetCount =
        ARRAYSIZE(standardWidgetNames);

    std::vector<ComPtr<Widget>> m_widgets;
};
```

The code crashed at this call to `Widget::GetCost`:

```
IFACEMETHODIMP StandardWidgets::get_TotalCost(INT32* result)
{
    *result = 0;

    RETURN_IF_FAILED(LazyInitializeWidgetList());

    INT32 totalCost = 0;
    for (int i = 0; i < standardWidgetCount; i++) {
        totalCost += m_widgets[i]->GetCost(); // here
    }
    *result = totalCost;
    return S_OK;
}
```

The customer's debugging showed that at the point of the crash, not only was the `widget` garbage, but the `m_widgets` vector had an impossibly large number of elements. The `m_widgets` is expected to have only four widgets, but it somehow found itself with ten, and sometimes as many as a hundred widgets. Of course, they were nearly all corrupted.

Here's the code that lazy-initializes the widget list:

```
HRESULT StandardWidgets::LazyInitializeWidgetList()
{
    // Early-out if already initialized.
    if (!m_widgets.empty()) {
        return S_OK;
    }

    // Lazy-create the vector of standard widgets
    try {
        m_widgets.reserve(standardWidgetCount);
        for (auto name : standardWidgetNames) {
            ComPtr<Widget> widget;
            RETURN_IF_FAILED(
                MakeAndInitialize<Widget>(&widget, name));
            m_widgets.push_back(widget);
        }
    } catch (std::bad_alloc const&) {
        return E_OUTOFMEMORY;
    }
    return S_OK;
}
```

The customer noted that the `reserve` method is always called with the value 4, and the code never pushes more than four items into the vector. They admitted that if there is a problem creating all four of the standard widgets, the vector could end up with fewer than four widgets, but it should never have *more* than four.

You already have multiple clues that point toward what the customer's problem is. I'll give you some time to think about it.

In the meantime, let's look at other issues with how the code lazy-initializes the widget list.

As the customer noted, if there is a problem creating any of the four standard widgets, the failure is propagated to the caller of `LazyInitializeWidgetList`, and `get_TotalCost` in turn propagates the error to its own caller, and it never gets to the point where it walks through the vector adding up all the costs.

If there is a memory allocation failure at the `reserve()`, or if there is a problem with the first standard widget, then the vector remains empty, and a second call to `LazyInitializeWidget-List` will make a new attempt at initialization.

If there is a problem with the second or subsequent standard widget, however, things get weird. The `LazyInitializeWidgetList` function returns a failure, which causes `get_TotalCost` to return failure. But the second time someone calls `get_TotalCost`, `LazyInitializeWidget-List` will see a nonempty vector and assume that everything was initialized. This time, the `get_TotalCost` method will proceed with the summation and perform an out-of-bounds array access when it gets to the widget that failed to be created.

Oops.

This particular problem boils down to leaving a partially-initialized `m_widgets` if the lazy initialization fails. To avoid this problem, we should create the vector in a local variable and transfer it to the member variable only after we are sure all of the widgets were created successfully.

```
HRESULT StandardWidgets::LazyInitializeWidgetList()
{
    // Early-out if already initialized.
    if (!m_widgets.empty()) {
        return S_OK;
    }

    // Lazy-create the vector of standard widgets
    try {
        std::vector<ComPtr<Widget>> widgets;
        widgets.reserve(standardWidgetCount);
        for (auto name : standardWidgetNames) {
            ComPtr<Widget> widget;
            RETURN_IF_FAILED(
                MakeAndInitialize<Widget>(&widget, name));
            widgets.push_back(widget);
        }
        m_widgets.swap(widgets);
    } catch (std::bad_alloc const&) {
        return E_OUTOFMEMORY;
    }
    return S_OK;
}
```

This ensures that the `m_widgets` is either totally empty or totally initialized. It is never in a half-initialized state.

While we're at it, we probably should convert the loop in `get_TotalCost` into a ranged `for` loop. Right now, `get_TotalCost` has a hidden dependency on `LazyInitializeWidgets`: It assumes that `LazyInitializeWidgets` always creates exactly the number of widgets as there are `standardWidgetNames`. Maybe in the future, you might want to suppress some of the standard widgets based on some configuration setting. If you add that configuration setting and forget to update `get_TotalCost` to account for suppressed widgets, you will have an out-of-bounds index. All the logic to decide which widgets are standard should be local to `LazyInitializeWidgets`.

```
IFACEMETHODIMP StandardWidgets::get_TotalCost(INT32* result)
{
    *result = 0;

    RETURN_IF_FAILED(LazyInitializeWidgetList());

    INT32 totalCost = 0;
    for (auto&& widget : m_widgets) {
        totalCost += widget->GetCost();
    }
    *result = totalCost;
    return S_OK;
}
```

Or if you want to get fancy,

```
IFACEMETHODIMP StandardWidgets::get_TotalCost(INT32* result)
{
    *result = 0;

    RETURN_IF_FAILED(LazyInitializeWidgetList());

    *result = std::transform_reduce(
        m_widgets.begin(), m_widgets.end(), 0, std::plus<>(),
        [](auto&& w) { return w->GetCost(); });
    return S_OK;
}
```

Okay, but back to the crash. I think I've added enough filler to give you time to consider what is happening.

When I looked at this crash, I noticed that the class is implemented with the `Microsoft::WRL::RuntimeClass` template class, and the implementation explicitly listed `FtmBase` as a template parameter, marking this class as free-threaded (also known as "agile"), which means that it can be used from multiple threads simultaneously.[1]

The object is eligible for multithreaded use, but there are no mutexes to protect two threads from modifying `m_widgets` at the same time. I suspected a race condition.

You can also observe that the class is being used in a free-threaded manner because the stack trace that leads to the crash says that it's running on a thread pool thread (`TppWorkerThread`), and thread pool threads default to the multi-threaded apartment.[2] The only code on the stack between `TppWorkerThread` and the application code is all COM and RPC, so no application code snuck in and initialized the thread into single-threaded apartment mode.

And when I looked at the crash dump, I caught the code red-handed: There was another thread also calling into this code.

```
// Crashing thread
0:006> .frame 1
01 contoso!StandardWidgets::get_TotalCost+0x12f
0:006> dv
    this = 0x000001b7`492833b0
    ...

// Another thread running at the time of the crash
0:004> kn
 # Call Site
00 ntdll!ZwDelayExecution+0x14
01 ntdll!RtlDelayExecution+0x4c
02 KERNELBASE!SleepEx+0x84
04 kernel32!WerpReportFault+0xa4
05 KERNELBASE!UnhandledExceptionFilter+0xd3a02
06 ntdll!TppExceptionFilter+0x7a
07 ntdll!TppWorkerpInnerExceptionFilter+0x1a
08 ntdll!TppWorkerThread$filt$3+0x19
09 ntdll!__C_specific_handler+0x96
0a ntdll!__GSHandlerCheck_SEH+0x6a
0b ntdll!RtlpExecuteHandlerForException+0xf
0c ntdll!RtlDispatchException+0x2d4
0d ntdll!KiUserExceptionDispatch+0x2e
0e contoso!StandardWidgets::get_TotalCost+0x12f
0f rpcrt4!Invoke+0x73
10 rpcrt4!Ndr64StubWorker+0xb9b
11 rpcrt4!NdrStubCall3+0xd7
12 combase!CStdStubBuffer_Invoke+0xdb
14 combase!ObjectMethodExceptionHandlingAction<<lambda_...> >+0x47
16 combase!DefaultStubInvoke+0x376
1a combase!ServerCall::ContextInvoke+0x6f3
1f combase!ComInvokeWithLockAndIPID+0xacb
21 combase!ThreadInvoke+0x103
22 rpcrt4!DispatchToStubInCNoAvrf+0x18
23 rpcrt4!RPC_INTERFACE::DispatchToStubWorker+0x1a9
25 rpcrt4!RPC_INTERFACE::DispatchToStubWithObject+0x1a7
27 rpcrt4!LRPC_SCALL::DispatchRequest+0x308
29 rpcrt4!LRPC_SCALL::HandleRequest+0xdcb
2a rpcrt4!LRPC_SASSOCIATION::HandleRequest+0x2c3
2b rpcrt4!LRPC_ADDRESS::HandleRequest+0x183
2c rpcrt4!LRPC_ADDRESS::ProcessIO+0x939
2d rpcrt4!LrpcIoComplete+0xff
2e ntdll!TppAlpcpExecuteCallback+0x14d
2f ntdll!TppWorkerThread+0x4b4
30 kernel32!BaseThreadInitThunk+0x18
31 ntdll!RtlUserThreadStart+0x21
0:004> .frame 0xe
0e contoso!StandardWidgets::get_TotalCost+0x12f
0:004> dv
    this = 0x000001b7`492833b0
    ...
```

Notice that the `this` pointer is the same for both threads, so we have proof that this object is being used from multiple threads simultaneously.

The fix for the multithreading issue is to ensure that only one thread tries to initialize the widget vector at a time. We can do this by adding a mutex, but I'm going to go even further and use the `std::once_flag`, whose purpose in life is to be used in conjunction with `std::call_once` to perform thread-safe one-time initialization, which is exactly what we want.

```
class StandardWidgets : RuntimeClass<IStandardWidgets, FtmBase>
{
    IFACEMETHODIMP get_TotalCost(INT32* result);
    ⟦ other methods not relevant here ⟧

private:
    HRESULT LazyInitializeWidgetList();

    static constexpr PCWSTR standardWidgetNames[] = {
        L"Bob", L"Carol", L"Ted", L"Alice" };
    static constexpr int standardWidgetCount =
        ARRAYSIZE(standardWidgetNames);

    std::once_flag m_initializeFlag;
    std::vector<ComPtr<Widget>> m_widgets;
};

HRESULT StandardWidgets::LazyInitializeWidgetList()
{
    try {
        std::call_once(m_initializeFlag, [&] {
            std::vector<ComPtr<Widget>> widgets;
            widgets.reserve(standardWidgetCount);
            for (auto name : standardWidgetNames) {
                ComPtr<Widget> widget;
                THROW_IF_FAILED(
                    MakeAndInitialize<Widget>(&widget, name));
                widgets.push_back(widget);
            }
            m_widgets = std::move(widgets);
        });
    } catch (std::bad_alloc const&) {
        return E_OUTOFMEMORY;
    }
    return S_OK;
}
```

Update: We had to change the `RETURN_IF_FAILED` to `THROW_IF_FAILED` because (1) without the change, the compiler will complain that not all code paths return a value, because the lambda also falls off the end, and more importantly, (2) `call_once` doesn't care about the lambda return value; it uses exceptions to detect errors. **End Update**.

This version also deals with the edge case where there are no standard widgets at all. The original code would continuously try to reinitialize the vector since it couldn't tell whether an empty vector means "Not yet initialized" or "Successfully initialized (and it's empty)".

Bonus chatter: How did this multithreaded race condition lead to a vector with a ridiculous size? Well, we saw some time ago that the internal structure of a `std::vector` is three pointers, one for the start of the vector data, one for the end of the valid data, and one for the end of the allocated data. If two threads call `reserve()` simultaneously, both will allocate new data, and then they were race to update the three pointers. You might end up with a "start" pointer that points to the data allocated by the first thread, but an "end" pointer that points to the data allocated by the second thread, resulting in a vector of unusual size.

[1] It was actually convenient that the implementation lists `FtmBase` explicitly, since the default behavior varies depending on whether `__WRL_CONFIGURATION_LEGACY__` is set.

Template parameter	Standard mode	Legacy mode
Nothing specified	Free-threaded	Not free-threaded
`FtmBase`	Free-threaded	
`InhibitFtmBase`	Not free-threaded	
`InhibitFtmBase` + `FtmBase`	Not free-threaded	

In pseudocode:

```
bool isFreeThreaded = !InhibitFtmBase && (FtmBase || Standard mode);
```

The explicitly inclusion of `FtmBase` saved me the trouble of looking up what mode the customer's project is using.

[2] Assuming the multi-threaded apartment exists at all.