The case of the feature flag that didn’t stay on long enough, part 1

devblogs.microsoft.com/oldnewthing/20250417-00/?p=111079

April 17, 2025

There was a crash in a unit test run under the Test Authoring and Execution Framework (TAEF, pronounced like *tafe*). The crashing stack looked like this:

```
kernelbase!RaiseFailFastException
unittests!wil::details::WilDynamicLoadRaiseFailFastException
unittests!wil::details::WilRaiseFailFastException
unittests!wil::details::WilFailFast
unittests!wil::details::ReportFailure_NoReturn<3>
unittests!wil::details::ReportFailure_Base<3,0>
unittests!wil::details::ReportFailure_Hr<3>
unittests!wil::details::in1diag3::_FailFast_Unexpected
unittests!WidgetRouter::GetInstance
unittests!winrt::Component::implementation::Widget::Close
unittests!winrt::Component::implementation::Widget::~Widget
unittests!winrt::impl::heap_implements<⟦...⟧>::`scalar deleting destructor'
unittests!winrt::implements<⟦...⟧>::Release
unittests!std::vector<winrt::Windows::Foundation::IClosable>::~vector
unittests!std::vector<winrt::Windows::Foundation::IClosable>::clear
unittests!WidgetRouter::~WidgetRouter
unittests!WidgetRouter::GetInstance'::`2'::`dynamic atexit destructor for
'singleton''
ucrtbase!<lambda_⟦...⟧>::operator
ucrtbase!__crt_seh_guarded_call<int>::operator()<<lambda_⟦...⟧>>
ucrtbase!_execute_onexit_table
ucrtbase!__crt_state_management::wrapped_invoke
unittests!dllmain_crt_process_detach
unittests!dllmain_dispatch
ntdll!LdrpCallInitRoutine
ntdll!LdrpProcessDetachNode
ntdll!LdrpUnloadNode
ntdll!LdrpDecrementModuleLoadCountEx
ntdll!LdrUnloadDll
kernelbase!FreeLibrary
wex_common!TAEF::Common::Private::ExecutionPeFileData::`scalar deleting
destructor'
wex_common!TAEF::Common::PeFile::~PeFile
te_loaders!WEX::TestExecution::NativeTestFileInstance::`scalar deleting
destructor'
te_host!<lambda_⟦...⟧>::Execute
te_common!WEX::TestExecution::CommandThread::ExecuteCommandThread
kernel32!BaseThreadInitThunk
ntdll!RtlUserThreadStart
```

The team concluded that the destructor of the `Widget` was running at the conclusion of this function:

```
void WidgetTests::BasicTests()
{
  // Force the feature flag on for the duration of this test
  auto override = std::make_unique<FeatureOverride>(NewFeatureId, true);

  auto widget = winrt::Component::Widget();
  ⟦ test the widget in various ways ⟧

  // widget naturally destructs here
}
```

Their conclusion was that the override was being released, and then the widget was destructing, resulting in an assertion failure in `WidgetRouter::GetInstance`:

```
/* static method */
std::shared_ptr<WidgetRouter>
    WidgetRouter::GetInstance()
{
    assert(FeatureFlags::IsEnabled(NewFeatureId));

    static std::shared_ptr<WidgetRouter> singleton =
        std::make_shared<WidgetRouter>();

    return singleton;
}
```

The team theorized that perhaps there was a race condition between the release of the widget and the lifting of the override, and their proposed fix was to release the `widget` explicitly while the override is still in scope.

```
void WidgetTests::BasicTests()
{
  // Force the feature flag on for the duration of this test
  auto override = std::make_unique<FeatureOverride>(NewFeatureId, true);

  auto widget = winrt::Component::Widget();
  ⟦ test the widget in various ways ⟧

  widget = nullptr;
}
```

I commented on the pull request (which had already been completed) that this change has no effect. The rules for C++ say that local variables are destructed in reverse order of construction, so the widget will naturally be released as part of its destruction, and only after that is finished will the override be released when it destructs.

The team replied that they did observe that the problem disappeared after they made their fix, but then it came back.

Exercise: Explain why the problem went away and then came back.

Answer to exercise: I suspected that they tested their fix in their team's test environment, where the feature is already enabled. The fix works not because it fixed anything but because it was never crashing in their test environment in the first place. The defect tracker, however, doesn't know that. It correctly reported that the bug was not being observed in any branches that had received the fix, which was initially only their own test branch. As the fixed merged into other branches, the bug was still not observed, until it finally merged into a branch where their feature was disabled. At that point, the override was actually doing something (changing a feature from disabled to enabled), and that's when the crashes started coming in.