Another example of the Windows Runtime interop pattern: Using the UserConsentVerifier from a Win32 program

devblogs.microsoft.com/oldnewthing/20240925-00

September 25, 2024

The `UserConsentVerifier` Windows Runtime class lets you display one of those authentication prompts to confirm the identity of the user. We discussed last time that this is a consent verifier, not a data protector.

The class was originally designed for UWP apps, but you can also use it from Win32 apps by using the `IUserConsentVerifierInterop` interface. This follows the normal pattern for interop interfaces that correspond to static methods:[1]

- Obtain the interface from the activation factory.
- The methods have the same names as the corresponding names of the static Windows Runtime methods, except that they also say `ForWindow`.
- The methods take the same parameters as the static Windows Runtime methods, except that there is an extra `HWND` parameter, and the output is returned in the form of a `REFIID`/`void**`.

Let's add a consent verifier to our scratch program.

```
#include <winrt/Windows.Foundation.h>              // for IAsyncOperation
#include <winrt/Windows.Security.Credentials.UI.h> // for UserConsentVerifier
#include <wrl/wrappers/corewrappers.h>             // for HStringReference
#include <UserConsentVerifierInterop.h>            // for IUserConsentVerifierInterop

namespace winrt
{
    using namespace winrt::Windows::Foundation;
    using namespace winrt::Windows::Security::Credentials::UI;
}

namespace WRL = Microsoft::WRL;

winrt::fire_and_forget RequestConsent(HWND hwnd)
{
    auto consentResult = co_await wil::capture_interop<
        winrt::IAsyncOperation<
            winrt::UserConsentVerificationResult>,
        winrt::UserConsentVerifier>(
        &::IUserConsentVerifierInterop::
            RequestVerificationForWindowAsync,
            hwnd,
            WRL::Wrappers::HStringReference(
                L"Just checking that it's you.").Get());

    PCWSTR message;
    switch (consentResult)
    {
    case winrt::UserConsentVerificationResult::Verified:
        message = L"User verified.";
        break;
    case winrt::UserConsentVerificationResult::DeviceBusy:
        message = L"Authentication device is busy.";
        break;
    case winrt::UserConsentVerificationResult::DeviceNotPresent:
        message = L"No authentication device found.";
        break;
    case winrt::UserConsentVerificationResult::DisabledByPolicy:
        message = L"Authentication device verification is disabled by policy.";
        break;
    case winrt::UserConsentVerificationResult::NotConfiguredForUser:
        message = L"Please go to Account Settings to set up PIN "
                  L"or other advanced authentication.";
        break;
    case winrt::UserConsentVerificationResult::RetriesExhausted:
        message = L"There have been too many failed attempts. "
                  L"Device authentication canceled.";
        break;
    case winrt::UserConsentVerificationResult::Canceled:
        message = L"Device authentication canceled.";
        break;
    default:
        message = L"Authentication device is currently unavailable.";
        break;
    }

    SetWindowText(hwnd, message);
}

void OnChar(HWND hwnd, TCHAR ch, int cRepeat)
{
    if (ch == VK_SPACE)
    {
        RequestConsent(hwnd);
    }
}

        HANDLE_MSG(hwnd, WM_CHAR, OnChar);
```

When you hit the space bar, we go into action. We take advantage of the `capture_interop` helper function we added to the Windows Implementation Library some time ago as part of our initial exploration of C++/WinRT interop pattern.

Taking that helper apart, the first thing that happens is that we call `get_activation_factory`, asking for the `IUserConsentVerifierInterop` interface of the `UserConsentVerifier` factory.

```
auto interop = winrt::get_activation_factory<
        winrt::UserConsentVerifier>,
        ::IUserConsentVerifierInterop>();
```

At the ABI, this would be

```
::IUserConsentVerifierInterop* interop;

hr = RoGetActivationFactory(
  HStringReference(
    RuntimeClass_Windows_Security_Credentials_UI_UserConsentVerifier)
  .Get(),
  IID_PPV_ARGS(&interop));

THROW_IF_FAILED(hr);
```

Next, we call `IUserConsentVerifierInterop::RequestVerificationForWindowAsync`, which is the `HWND` equivalent of `UserConsentVerifier::RequestVerificationAsync`. We ask for the result in the form of a `winrt::IAsyncOperation<UserConsentVerificationResult>` so we can `co_await` it. The `capture_interop` function does it by calling `winrt::capture`, which is a C++/WinRT helper function that calls a method and returns the produced object.

```
winrt::capture<
    winrt::IAsyncOperation<
        winrt::UserConsentVerificationResult>>(
    interop,
    &::IUserConsentVerifierInterop::
        RequestVerificationForWindowAsync,
    hwnd,
    WRL::Wrappers::HStringReference(
        L"Just checking that it's you.").Get());
```

This is shorthand for

```
winrt::IAsyncOperation<winrt::UserConsentVerificationResult> result;
winrt::check_hresult(
    interop->RequestVerificationForWindowAsync(
        hwnd,
        WRL::Wrappers::HStringReference(
            L"Just checking that it's you.").Get(),
        winrt::guid_of<decltype(result)>(),
        winrt::put_abi(result())));
```

The ABI equivalent is

```
ABI::Windows::Foundation::IAsyncOperation<
    ABI::Windows::Security::Credentials::UI::UserConsentVerificationResult>*
        result;

hr = interop->RequestVerificationForWindowAsync(
    hwnd,
    WRL::Wrappers::HStringReference(
        L"Just checking that it's you.").Get(),
    IID_PPV_ARGS(&result));
```

Once we get our asynchronous operation, we await it and show the result in our title bar.