

Windows Authentication - Credential Providers - Part 1

 dennisbabkin.com/blog



Intro

If you ever tried to write a credential provider in Windows you will understand my plight. The only relevant [official sample](#) provided by Microsoft dates back to the days of Windows 8.1, and the documentation for writing one is quite scarce, to say the least.

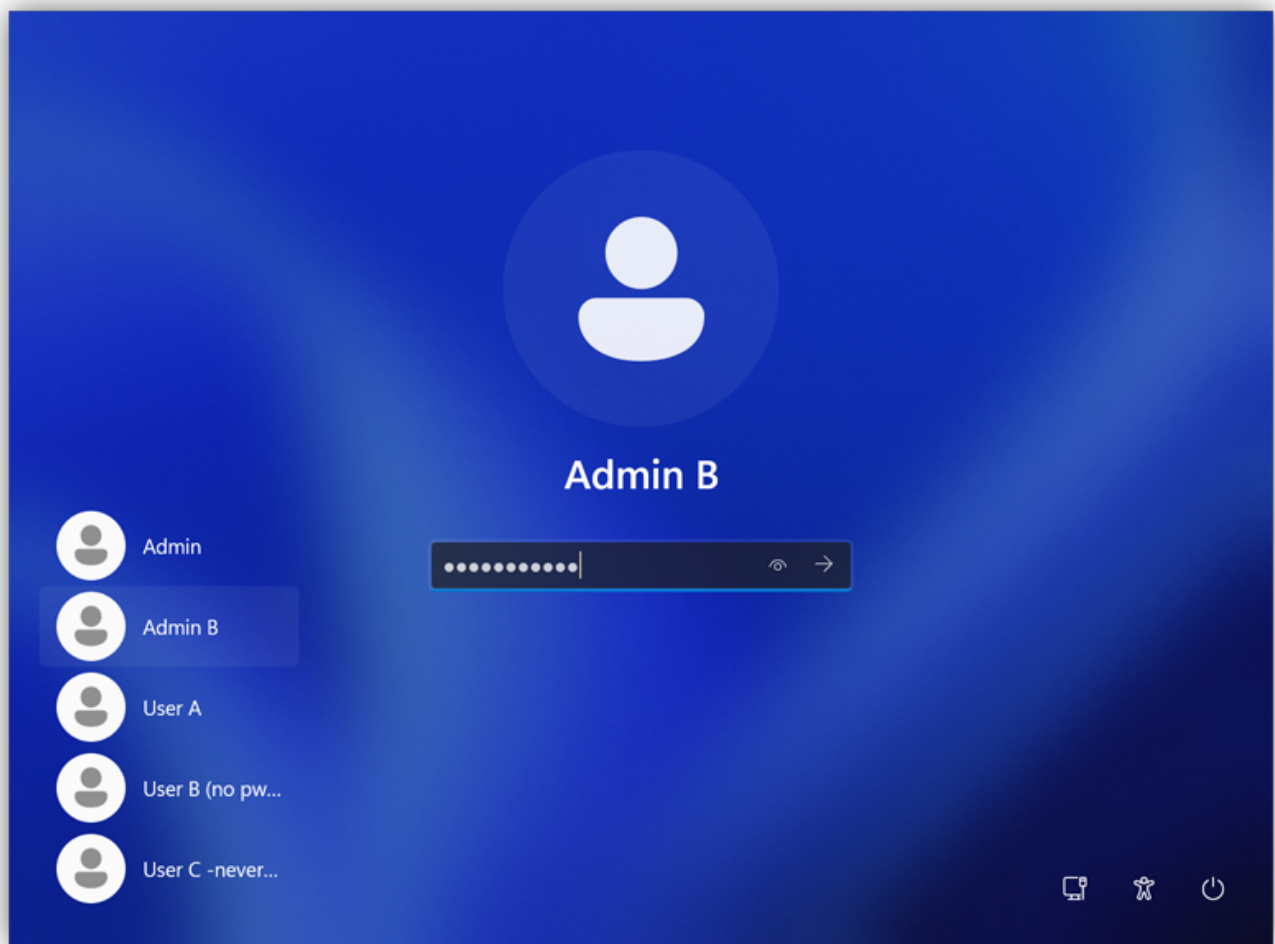
Note that credential providers have undergone quite a fast-paced evolution since the earlier days of Windows. The technique that was previously used during the Windows XP days, known as [GINA](#), or a "Graphical Identification and Authentication dynamic-link library", is now largely obsolete and irrelevant.

In this series of blog posts I will try to dispel some myths about credential providers in Windows and hopefully give the reader some insight on how they operate.

Some Theory

Before we dive into it, let's review what is a credential provider in Windows?

From a user's perspective, as well as the simplest way to describe it to pretty much anyone, is to say that a credential provider is the window that you deal with when you need to log in to your user account in Windows. One of these things:



Example of a credential provider on Windows 11.

From a technical standpoint, this is not exactly true. What you see in that screenshot is the logon process, that I will call `LogonUI.exe`.

That thing might have its own internal name, but since Microsoft provides developers with just the bare minimum of crumbs as credential providers are concerned and we have to dig around and research for ourselves, I will take a liberty to give some things my own names. I hope that you don't mind?

A credential provider only tells the `LogonUI.exe` (in that case) which user interface (UI) controls to display.

So the `LogonUI.exe` renders all the buttons, the password field, a nicely translucent background, a user picture and so forth for you. As a credential provider developer you have a very limited set of controls in that regard. (Which, knowing how Windows OS is being abused by some unscrupulous developers, may be a good thing.)

As the author of a credential provider you can pick from a short list of available UI elements that you would want to display for your authentication method. After all, the job of a credential provider is to collect login credentials from a user and to "provide" them (or pass them) to the system. And that is it.

Following Microsoft mantra, a credential provider should not attempt to authenticate the user. It should only collect credentials and pass it to the system.

To do its job a credential provider resides in a dynamic-link library (DLL) that is structured to implement a bunch of COM interfaces that will be called (or invoked) by `LogonUI.exe`, or by some other system process, when the need arises.

Thus, to serve its purpose a credential provider DLL has to be loaded into a host process. I've already shown one: `LogonUI.exe`. That is the process that is responsible for logging in users from a secure desktop. Most people are familiar with it. But that is not the only host process that can load your credential provider. It may be also loaded into the Windows Explorer process, or into `CredentialUIBroker.exe`, or `consent.exe` at a certain condition during the UAC prompt or to retrieve a user account password, or if the user picks "*Run as different user*" option. Alternatively, a credential provider may be loaded into your own process if you invoke the `CredUIPromptForWindowsCredentials` API.

Although you can write a credential provider in a managed language like C#, I personally prefer to write these early boot components in a lower-level language like C++ or even C. Doing so mitigates the possible unpredictability of a missing component in a managed framework.

A true million dollar question with credential providers though is knowing which interface is used where, and when will it be invoked? And this is what I will try to share in this and the following posts.

Lifespan of a Credential Provider

The rest of this blog post will be dedicated to the description of COM interfaces and their events, or callbacks, that happen when a credential provider is loaded, used, and when it is later unloaded by the system.

In most cases you will not be loading a credential provider manually. All of the work of invoking it is done by the OS components.

An important aspect for a developer is to realize that a credential provider can be loaded and used during different stages of the operating system life cycle: from an early boot stage when no user is present at the system and no desktop is available, to some random time during a remote desktop session.

Also, as I pointed out earlier, a credential provider is often loaded into a secure desktop, which limits the number and type of system functions that can be called from it.

For instance, you cannot assume that a user desktop is available, and thus some of the `user32.dll` APIs will not be able to function as you would expect. For instance, calls like `SendMessage` to windows in the user desktop will not go through.

It is also prudent to understand that in most cases a credential provider should not be displaying its own UI. With one exception to that rule that I will show-case later.

Finally, debugging a credential provider poses its own challenges. We will have to leave this subject for some later blog post. For now, the developer of a credential provider should rely on logging instead.

Credential Provider Components

Before we go any further, we need to understand what are the components of a credential provider. Microsoft has their own definition, which, to be honest, was very confusing for me. Thus, I will try to explain it as I think is easier to understand. Or, at least, it was for me.

Factory

Factory is a COM class that is used by the outside callers to request various other COM components from your credential provider. Which ones? We'll look at it [later](#). For now, view it as a base for your credential provider's logic that issues instances of its other interfaces.

As most COM components go, you would generally start writing a credential provider from its factory class. It must be derived and implement the [IClassFactory](#) interface.

A factory class has to implement methods of the following interfaces:

C++[\[Copy\]](#)

```
class MyFactory : public IClassFactory
{
    // IUnknown
    virtual ULONG STDMETHODCALLTYPE AddRef();
    virtual ULONG STDMETHODCALLTYPE Release();
    IFACEMETHODIMP QueryInterface(__in REFIID riid, __deref_out void** ppv);

    // IClassFactory
    IFACEMETHODIMP CreateInstance(__in IUnknown* pUnkOuter, __in REFIID riid, __deref_out void**
ppv);
    IFACEMETHODIMP LockServer(__in BOOL bLock);
};
```

Later on, all of the interfaces will have a similar backbone.

Filter

A *filter* is another COM class in the credential provider's implementation. It is used primarily to filter out other credential providers in the system that you do not want to support.










A filter class must implement the [ICredentialProviderFilter](#) interface.

The way it works is that the host process sends your filter class two things:

1. The type of a login, or what Microsoft calls "[usage scenario](#)". I view it as "*where was the credential provider called from*" parameter. It can be one of: login screen, unlocking workstation, changing user password, [CredUI-type](#) authentication, and so forth.

2. And the list of credential providers that are available in the system.

As an example, on the Windows 10 system, my installed filter class received the following list of available credential providers. Note that each provider is represented by its own unique **CLSID**, or **GUID**:

CLSID	Provide Name	Module	Bitmaps
{2135F72A-90B5-4ED3-A7F1-8BB705AC276A}	PicturePasswordLogonProvider	credprovslegacy.dll	 
{3DD6BEC0-8193-4FFE-AE25-E08E39EA4063}	NPPProvider	credprovs.dll	
{60B78E88-EAD8-445C-9CFD-0B87F74EA6CD}	PasswordCredentialProvider	credprovs.dll	
{8AF662BF-65A0-4D0A-A540-A338A999D36F}	FaceCredentialProvider	FaceCredentialProvider.dll	
{8FD7E19C-3BF7-489B-A72C-846AB3678C96}	SmartcardCredentialProvider	SmartcardCredentialProvider.dll	 
{BEC09223-B018-416D-A0AC-523971B639F5}	WinBioCredentialProvider	BioCredProv.dll	
{C5D7540A-CD51-453B-B22B-05305BA03F07}	CloudExperience_CredentialProvider	cxcredprov.dll	
{C885AA15-1764-4293-B82A-0586ADD46B35}	IrisCredentialProvider	FaceCredentialProvider.dll	
{D6886603-9D2F-4EB2-B667-1971041FA96B}	NgcPinProvider	ngccredprov.dll	 
{F8A0B131-5F68-486C-8040-7E8FC3C85BB6}	WLIDCredentialProvider	wlidcredprov.dll	
{5537E283-B1E7-4EF8-9C6E-7AB0AFE5056D}	CRasProvider	rasplap.dll	
{855E2A67-A476-4664-8581-01BEC33975BC}	MyProvider		

Hover the mouse over each bitmap to get its resource ID in its specific module.

My own credential provider, or "MyProvider", is also in the list. I gave it an arbitrarily random CLSID of {855E2A67-A476-4664-8581-01BEC33975BC} that I generated with the "Create GUID" tool in Visual Studio.

The job of your filter class is to say whether or not you want to deny this or that credential provider from being shown to the user in this or that usage scenario. Or, in other words, it is the way for your credential provider to say what it can or cannot do, in terms of which credentials it can collect from the user.

A filter class can only deny the existing credential providers that are given to it.

For instance, a credential provider may deny a PIN provider during a user account password-change usage scenario.

And if the filter class allows a certain provider, it is later the job of a *provider class* to tell the hosting process which UI elements it will need to collect the login credentials from the user for that specific login method.

So when a filter is invoked in your credential provider you can decide whether to block (or allow) any other credential providers in the system (btw, including your own.) Your filter will receive a list of providers, like the one that I showed above. Note though, that if you allow this or that credential provider, your own credential provider must be capable of collecting required credentials from the user for that specific login method.

A filter class is also used to process incoming serialized data during a remote desktop connection.

An astute reader may have noticed that the filter class seems to have control over whether or not to show all credential providers in the system. Or, it can disallow other credential providers. This is indeed true. Two or more custom credential providers may not do well in one system.

As a workaround, an installer for your particular credential provider may check for the presence of other filters in the following system registry key and either warn the user, or to refuse installation if any other filters are detected:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credential Provider Filters
```

Note that there's always the `GenericFilter` present there with the {DDC0EED2-ADBE-40b6-A217-EDE16A79A0DE} GUID.

A filter class has to implement methods of the following interfaces:

C++[Copy]

```

class MyFilter : public ICredentialProviderFilter
{
    // IUnknown
    virtual ULONG STDMETHODCALLTYPE AddRef();
    virtual ULONG STDMETHODCALLTYPE Release();
    virtual HRESULT STDMETHODCALLTYPE QueryInterface(__in REFIID riid, __in void** ppv);

    // ICredentialProviderFilter
    virtual HRESULT STDMETHODCALLTYPE Filter(
        /* [in] */ CREDENTIAL_PROVIDER_USAGE_SCENARIO cpus,
        /* [in] */ DWORD dwFlags,
        /* [annotation][size_is][in] */
        _In_reads_(cProviders) GUID* rgclsidProviders,
        /* [annotation][size_is][out][in] */
        _Inout_updates_(cProviders) BOOL* rgbAllow,
        /* [in] */ DWORD cProviders);

    virtual HRESULT STDMETHODCALLTYPE UpdateRemoteCredential(
        /* [annotation][in] */
        _In_ const CREDENTIAL_PROVIDER_CREDENTIAL_SERIALIZATION* pcpcsIn,
        /* [annotation][out] */
        _Out_ CREDENTIAL_PROVIDER_CREDENTIAL_SERIALIZATION* pcpcsOut);
};

```

The [UpdateRemoteCredential](#) callback is used to de-serialize incoming credentials during a remote desktop connection.

Provider

In the parlance of a credential provider project, a somewhat confusingly named "*provider*" class, is a COM interface that implements the logic for deciding which user interface elements will be required for a specific login method.

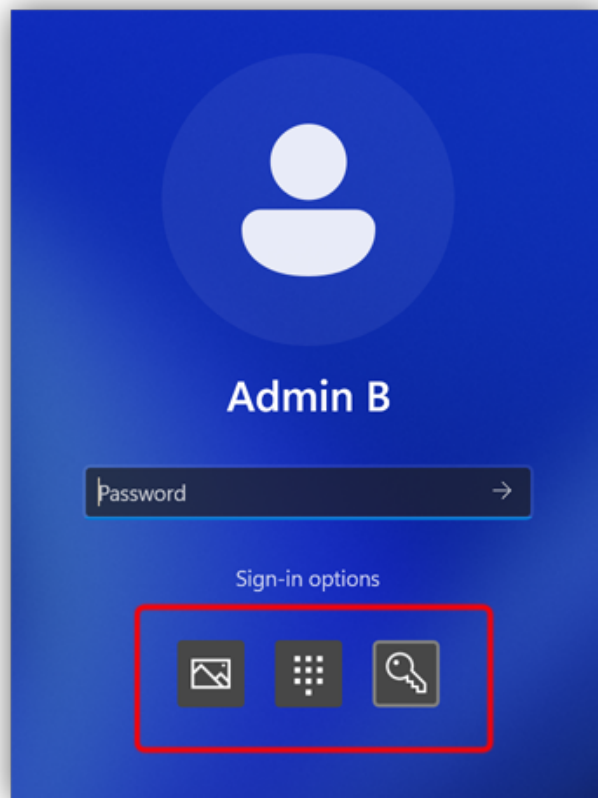
There could be multiple providers, each for its own login method. (I showed the list of possible login methods when I was describing the [filter class](#).) For instance, it could be a provider for a password login, or for a PIN login, and so forth.

A version 2 (or V2) of the provider class has to be derived from the [ICredentialProvider](#) and from the [ICredentialProviderSetUserArray](#) interfaces. Additionally, the [credential class](#) must be derived from [ICredentialProviderCredential2](#), or later interface.

While an older provider, version 1 (or V1, [V1PasswordCredentialProvider](#)) has to be derived only from [ICredentialProvider](#).

You can pretty much safely ignore V1 provider if you want to develop a credential provider for Windows 10 and later OS.

From a user's perspective they can switch between different login methods (and thus between provider classes that are used in your credential provider) using the "*Sign-in options*" control in the [LogonUI](#):



"Sign-in options" in Windows 11.

In the screenshot above, you can see the "picture password", "PIN" and a "regular password" login methods, available to the user. Each of those come with their own provider classes.

Although there's no documented way to know which provider class is currently active, or is interacting with the user in the credential provider, you can use a *hack* to determine that. A credential class (that I will describe below) has two methods: `SetSelected` and `SetDeselected` that are invoked when a certain credential is activated. Thus, if you associate each credential class with a provider class that created it, you will know which provider class is currently active by responding to the two callbacks that I named above.

A provider class can be also used to request methods of communication with the host process, such as `LogonUI` and others, and to request other callbacks using the `ICredentialProviderEvents` interface in its `Advise` method. A provider class is also used to obtain basic information needed for the credential provider, such as the list of available user accounts via the `ICredentialProviderSetUserArray` interface; or to receive various other notifications via the undocumented `ICredentialProviderWithDisplayState` interface. (We'll get back to it later.)

A provider class (version 2) has to implement methods of the following interfaces:

C++[Copy]


```

class MyProvider : public ICredentialProvider,
                  ICredentialProviderSetUserArray
{
    ////////////////////////////////////////////////////
    // IUnknown
    virtual ULONG STDMETHODCALLTYPE AddRef();
    virtual ULONG STDMETHODCALLTYPE Release();
    virtual HRESULT STDMETHODCALLTYPE QueryInterface(__in REFIID riid, __in void** ppv);

    ////////////////////////////////////////////////////
    // ICredentialProvider
    virtual HRESULT STDMETHODCALLTYPE SetUsageScenario(__in CREDENTIAL_PROVIDER_USAGE_SCENARIO
cpus, __in DWORD dwFlags);
    virtual HRESULT STDMETHODCALLTYPE SetSerialization(__in const
CREDENTIAL_PROVIDER_CREDENTIAL_SERIALIZATION* pcpcs);

    virtual HRESULT STDMETHODCALLTYPE Advise(__in ICredentialProviderEvents* pcpe, __in UINT_PTR
upAdviseContext);
    virtual HRESULT STDMETHODCALLTYPE UnAdvise();

    virtual HRESULT STDMETHODCALLTYPE GetFieldDescriptorCount(__out DWORD* pdwCount);
    virtual HRESULT STDMETHODCALLTYPE GetFieldDescriptorAt(__in DWORD dwIndex, __deref_out
CREDENTIAL_PROVIDER_FIELD_DESCRIPTOR** ppcpfd);

    virtual HRESULT STDMETHODCALLTYPE GetCredentialCount(__out DWORD* pdwCount,
__out_range(< , *pdwCount) DWORD* pdwDefault,
__out BOOL* pbAutoLogonWithDefault);

    virtual HRESULT STDMETHODCALLTYPE GetCredentialAt(__in DWORD dwIndex,
__deref_out ICredentialProviderCredential** ppcpc);

    ////////////////////////////////////////////////////
    // ICredentialProviderSetUserArray
    virtual HRESULT STDMETHODCALLTYPE SetUserArray(
/* [annotation][in] */
_In_ ICredentialProviderUserArray* users);
};

```

If you're writing a credential provider for versions of Windows prior to Windows 8, you may not use `ICredentialProviderSetUserArray` and will have to retrieve all available user accounts manually.

Alternatively, if you want additional functionality for your provider class you may also derive it from `ICredentialProviderWithDisplayState` and `IAutoLogonProvider` interfaces, to gain access to the following undocumented methods:

C++[Copy]

```

    ////////////////////////////////////////////////////
    // ICredentialProviderWithDisplayState
    virtual HRESULT STDMETHODCALLTYPE SetDisplayState(__in
CREDENTIAL_PROVIDER_DISPLAY_STATE_FLAGS Flags);

    ////////////////////////////////////////////////////
    // IAutoLogonProvider
    virtual HRESULT STDMETHODCALLTYPE SetAutoLogonManager(IAutoLogonManager* pAutoLogonManager);

```

The declarations for these undocumented interfaces and flags are:

C++[Copy]

```

enum CREDENTIAL_PROVIDER_DISPLAY_STATE_FLAGS
{
    CPDSF_NONE = 0x00000000,           //Default initialization value.
    CPDSF_AWAKE = 0x00000001,         //Is set when display is on (can be used for power-
saving.)
    CPDSF_UNKNOWN_1 = 0x00000002,     //Transitive flag. It is mostly absent. It only
appears before/after CPDSF_AWAKE changes.
    CPDSF_CURTAIN = 0x00000004,       //Is set when the "curtain" is pulled down, or
obscures the tiles.
    CPDSF_CURTAIN_MOVING = 0x00000008, //Is set when the "curtain" is moving. It appears
when the "curtain" is being pulled.
    CPDSF_STATION_LOCKED = 0x00000040, //Is set when the workstation is locked.
    CPDSF_PROVIDER_VISIBLE = 0x00000080, //Is set when provider screen is active/visible.
    CPDSF_NO_LOGGED_USERS = 0x00000400, //Is set when there is no logged in users.
    CPDSF_UNKNOWN_2 = 0x00000800,     //Some unknown flag that is used by the PIN (NGC)
provider.
    CPDSF_SHUTDOWN_STARTED = 0x00001000, //Is set when reboot or shutdown had started.
    CPDSF_ACTIVE = 0x00010000,        //Is set to indicate display activity. Is reset after
some time when UI is idle.
    CPDSF_TBAL_LOGON = 0x00080000     //Is set for the trusted boot auto-logon (TBAL) boot.
};

// A "curtain" is when the LogonUI.exe hides all credential tiles after a period of inactivity,
// or right after it is initially shown. This appears to be a security feature.
//

MIDL_INTERFACE("A09BCC29-D779-4513-BB59-B4DB5D82D2B6")
ICredentialProviderWithDisplayState : public IUnknown
{
    virtual HRESULT STDMETHODCALLTYPE SetDisplayState(__in
CREDENTIAL_PROVIDER_DISPLAY_STATE_FLAGS Flags) = 0;
};

MIDL_INTERFACE("8A4E89FE-C09D-475E-88CB-F8F11E047C50")
IAutoLogonProvider : public IUnknown
{
    virtual HRESULT STDMETHODCALLTYPE SetAutoLogonManager(IAutoLogonManager *) = 0;
};

```

I'll explain what `SetDisplayState` does in a [later post](#). But it's basically used to receive notifications about various states of the OS, and of the credential provider itself.

Credentials, Tiles & UI Fields

The best way to describe *credentials* is to show them visibly in a credential provider:

Five credential tiles in Windows 11.

Credentials are also called "tiles" in the Microsoft documentation. My guess, the term "tile" is used when referring to the visual representation of a credential.

Like before, a *credential* is a COM class that implements a visual representation of a user account, or of some other *login entity*, depending on your credential provider's purpose. But, in most cases, a *credential* class represents a UI object that a user can click to log in with.

A credential can have an arbitrary number of *fields*, or UI elements, that can be displayed when a tile for a credential is selected in the UI of the credential provider. Those UI fields is what a user interacts with by providing their login information.

The screenshot below demonstrates the "User A" tile that is currently selected in a credential provider. That tile respectively has the following UI fields enabled (or visible): 1. "Password field", 2. "Submit button", 3. "Password hint" text and 4. "Reset password link":



Selected "User A" tile with its four visible UI fields in Windows 11.

A credential class receives multiple callbacks from the host process to decide which fields to show (or to hide). It also gets notified when a certain tile is selected or deselected. I'll explain more in [part 2](#).

A credential class must be derived from and implement [ICredentialProviderCredential](#) interface (or, it's later cousins, that have a consecutive number at the end), and optionally [IConnectableCredentialProviderCredential](#), [ICredentialProviderCredentialWithOptions](#), undocumented [ICredentialProviderCredentialTileDataInfo](#) and so on, to support newer features of the credential providers.

All credentials must be created from within the [GetCredentialCount](#) and [GetCredentialAt](#) callbacks of the [provider class](#). The first call must return the number of credentials that your credential provider wants to show. This is basically dictated by the number of user accounts that are supplied for your provider class in the [SetUserArray](#) callback. Then each invocation of the [GetCredentialAt](#) callback for your provider class (for the number of desired credentials) must create and return an instance of your credential class.

The same rule applies to the UI fields that are shown when a tile is selected. You first specify how many fields a provider will need in the [GetFieldDescriptorCount](#) callback. And then describe each field in the following [GetFieldDescriptorAt](#) callbacks. Each time the latter callback is invoked, you need to [specify](#) what each field is: whether it's a text field, a password field, an image, a submit button, and so forth.

Once UI fields and credentials are created they cannot be destroyed or added for the lifetime of the provider class. The way to address this limitation is by letting you show or hide UI fields, as well as to change their state, or text displayed in them. This is accomplished by responding to various callbacks for the credential class, such as: GetFieldState, GetStringValue, GetBitmapValue, GetCheckboxValue, GetSubmitButtonValue, GetComboBoxValueCount, GetComboBoxValueAt; or by calling the following methods: SetStringValue, SetCheckboxValue, SetComboBoxSelectedValue, and so forth.

The UI fields cannot be arbitrarily positioned on the screen, resized, or in any way styled. The host process that calls your credential provider decides all those things. The only control that the author of a credential provider has is the sequence at which UI elements, or fields, will appear in the host process.

Newer undocumented class, like `ICredentialProviderCredentialTileDataInfo` has methods such as `SetTitleVisibility` and `GetTileVisibility` for changing tile visibility. As well as the `ICredTileData` class that has the `IsTileVisible` method, and the `ICredentialProviderCredentialTileDataEvents` class that has `RequestVisibilityChange`. But Microsoft, in their usual fashion, *forgot* to document those.

Finally, the credential class is the one that gets notified when a user clicks "Submit" button to begin the login process via the GetSerialization callback. In response to that callback the credential class must collect all user-provided credentials in the UI, serialize them and feed the resulting binary data back to the host process.

Note that Microsoft admonishes developers that the credential class should not authenticate a user. Its job is only to collect user login credentials and to pass them to the host process. In reality though, this is not always possible. Especially if you're writing a credential provider wrapper to implement a form of an MFA login.

Next, after your credential class passes serialized credentials to the host process, the latter one internally invokes some of the LSASS functions to perform the actual user authentication, usually via a call to the LsaLogonUser API.

The result of the login is reported back to the credential class via the eponymous ReportResult method. Your credential class should respond to it by setting an appropriate user message, if the login fails.

Note that `ReportResult` callback is not called for the CredUI-type scenario.

One important caveat to remember about the `ReportResult` callback is that if the login was successful (or if its ntsStatus parameter is set to `S_OK`), your credential provider will begin unloading immediately after you return from `ReportResult`, regardless of its return status code.

I would definitely consider the latter condition a deficiency in the Microsoft's implementation of the credential provider interface. Lacking the ability to halt a successful login, any implementations of the secondary (or MFA) logins become impossible without some sort of a hack.

Because of that, developers usually resort to invoking the LsaLogonUser function from within their credential provider before passing a serialized user input to the host process in the GetSerialization callback.

Microsoft could've prevented such behavior by providing a mechanism to cancel a successful login from within the `ReportResult` callback.

A credential class has to implement methods of the following interfaces:

C++[Copy]

```

class MyCredential : public ICredentialProviderCredential4
{
    //////////////////////////////////////
    // IUnknown
    virtual ULONG STDMETHODCALLTYPE AddRef();
    virtual ULONG STDMETHODCALLTYPE Release();
        virtual HRESULT STDMETHODCALLTYPE QueryInterface(__in REFIID riid, __in void** ppv);

    //////////////////////////////////////
    // ICredentialProviderCredential
    virtual HRESULT STDMETHODCALLTYPE Advise(__in ICredentialProviderCredentialEvents* pcpcce);
    virtual HRESULT STDMETHODCALLTYPE UnAdvise();

    virtual HRESULT STDMETHODCALLTYPE SetSelected(__out BOOL* pbAutoLogon);
    virtual HRESULT STDMETHODCALLTYPE SetDeselected();

    virtual HRESULT STDMETHODCALLTYPE GetFieldState(__in DWORD dwFieldID,
        __out CREDENTIAL_PROVIDER_FIELD_STATE* pcpfs,
        __out CREDENTIAL_PROVIDER_FIELD_INTERACTIVE_STATE* pcpfis);

    virtual HRESULT STDMETHODCALLTYPE GetStringValue(__in DWORD dwFieldID, __deref_out PWSTR* ppwsz);
    virtual HRESULT STDMETHODCALLTYPE GetBitmapValue(__in DWORD dwFieldID, __out HBITMAP* phbmp);
    virtual HRESULT STDMETHODCALLTYPE GetCheckboxValue(__in DWORD dwFieldID, __out BOOL* pbChecked,
__deref_out PWSTR* ppwszLabel);
    virtual HRESULT STDMETHODCALLTYPE GetComboBoxValueCount(__in DWORD dwFieldID, __out DWORD*
pcItems, __out DWORD* pdwSelectedItem);
    virtual HRESULT STDMETHODCALLTYPE GetComboBoxValueAt(__in DWORD dwFieldID, __in DWORD dwItem,
__deref_out PWSTR* ppwszItem);
    virtual HRESULT STDMETHODCALLTYPE GetSubmitButtonValue(__in DWORD dwFieldID, __out DWORD*
pdwAdjacentTo);

    virtual HRESULT STDMETHODCALLTYPE SetStringValue(__in DWORD dwFieldID, __in PCWSTR pwz);
    virtual HRESULT STDMETHODCALLTYPE SetCheckboxValue(__in DWORD dwFieldID, __in BOOL bChecked);
    virtual HRESULT STDMETHODCALLTYPE SetComboBoxSelectedValue(__in DWORD dwFieldID, __in DWORD
dwSelectedItem);
    virtual HRESULT STDMETHODCALLTYPE CommandLinkClicked(__in DWORD dwFieldID);

    virtual HRESULT STDMETHODCALLTYPE GetSerialization(__out
CREDENTIAL_PROVIDER_GET_SERIALIZATION_RESPONSE* pcpgsr,
        __out CREDENTIAL_PROVIDER_CREDENTIAL_SERIALIZATION* pcpcs,
        __deref_out_opt PWSTR* ppwszOptionalStatusText,
        __out CREDENTIAL_PROVIDER_STATUS_ICON* pcpsiOptionalStatusIcon);

    virtual HRESULT STDMETHODCALLTYPE ReportResult(__in NTSTATUS ntsStatus,
        __in NTSTATUS ntsSubstatus,
        __deref_out_opt PWSTR* ppwszOptionalStatusText,
        __out CREDENTIAL_PROVIDER_STATUS_ICON* pcpsiOptionalStatusIcon);

    //////////////////////////////////////
    // ICredentialProviderCredential2
    virtual HRESULT STDMETHODCALLTYPE GetUserSid(
        /* [annotation][string][out] */
        _Outptr_result_maybenull_ LPWSTR* sid);

    //////////////////////////////////////
    // ICredentialProviderCredential3
    virtual HRESULT STDMETHODCALLTYPE GetBitmapBufferValue(__in DWORD fieldID,
        __out DWORD* pImageBufferSize, __out BYTE** ppImageBuffer);

    //////////////////////////////////////
    // ICredentialProviderCredential4

```



```
    virtual HRESULT STDMETHODCALLTYPE GetTextFieldMaxLength(__in DWORD dwFieldID, __out DWORD*
pMaxLength);
};
```

If you need to support Windows 10 and later OS, I would use `ICredentialProviderCredential4` instead of the older `ICredentialProviderCredential` interface.

Additionally, you may also derive your credential class from and implement the `IConnectableCredentialProviderCredential`, `ICredentialProviderCredentialWithFieldOptions` and `ICredentialProviderCredentialTileDataInfo` to gain access to the following methods:

C++[Copy]

```
////////////////////////////////////
// IConnectableCredentialProviderCredential
virtual HRESULT STDMETHODCALLTYPE Connect(__In_ IQueryContinueWithStatus* pqcws);
virtual HRESULT STDMETHODCALLTYPE Disconnect();

////////////////////////////////////
// ICredentialProviderCredentialWithFieldOptions
virtual HRESULT STDMETHODCALLTYPE GetFieldOptions(
    /* [in] */ DWORD fieldID,
    /* [annotation][out] */
    _Out_ CREDENTIAL_PROVIDER_CREDENTIAL_FIELD_OPTIONS* options);

////////////////////////////////////
// ICredentialProviderCredentialTileDataInfo
virtual HRESULT STDMETHODCALLTYPE SetTileVisibility(__in BOOL bVisible);
virtual HRESULT STDMETHODCALLTYPE GetTileVisibility(__out PBOOL pbVisible);
```

The declarations for undocumented interfaces are:

C++[Copy]

```

MIDL_INTERFACE("64a5010e-4363-41f8-9738-19045c20dabc")
ICredentialProviderCredential3 : ICredentialProviderCredential2
{
    virtual HRESULT STDMETHODCALLTYPE GetBitmapBufferValue(__in DWORD dwFieldID,
        __out DWORD * pImageBufferSize, __out BYTE * *ppImageBuffer) = 0;
};

MIDL_INTERFACE("4a54a3b6-a8d3-46a8-9080-811ba8ccb07d")
ICredentialProviderCredential4 : ICredentialProviderCredential3
{
    virtual HRESULT STDMETHODCALLTYPE GetTextFieldMaxLength(__in DWORD dwFieldID, __out DWORD *
pMaxLength) = 0;
};

MIDL_INTERFACE("1ecf61d8-745e-4484-bcc7-182cea64c787")
ICredentialProviderCredential5 : ICredentialProviderCredential4
{
    virtual HRESULT STDMETHODCALLTYPE GetAccessibilityTextForField(_In_ DWORD dwFieldID, _Out_
PWSTR* ppszText) = 0;
    virtual HRESULT STDMETHODCALLTYPE GetIsAccessibilityViewRawForField(_In_ DWORD dwFieldID,
_Out_ BOOL* pbRaw) = 0;
};

MIDL_INTERFACE("F6247CF9-061D-46E7-AAA7-0FDE071A5C1A")
ICredentialProviderCredentialTileDataInfo : public IUnknown
{
    virtual HRESULT STDMETHODCALLTYPE SetTileVisibility(__in BOOL bVisible) = 0;
    virtual HRESULT STDMETHODCALLTYPE GetTileVisibility(__out PBOOL pbVisible) = 0;
};

```

Note that the [IConnectableCredentialProviderCredential](#) interface is especially useful if you want to have access to the credential provider's UI during the login process, say if such process may take a longer time to complete. (One example would be the use of a companion device, such as a smartphone, for a form of the [MFA login](#).)

The [Connect](#) method is invoked before [GetSerialization](#) callback, so you can perform all lengthy login and serialization there and pass the result into the [GetSerialization](#) callback to return to the host process. The [IConnectableCredentialProviderCredential](#) interface provides access to the status text message that is displayed to the user, as well as to the "Cancel" button that can be used to cancel a lengthy login process:



Example of the `Connect` method implementation in Windows 11.

The `IConnectableCredentialProviderCredential` interface is used by the Windows itself for some slower pre-logon PLAP providers, such as 802.1x.

Conclusion

In the next part of this mini-series on credential providers lets review a sequence of callbacks that a credential provider receives to better understand what happens under the hood, as well as to learn how one can write a *wrapper* to an existing (system) credential provider to expand its functionality.