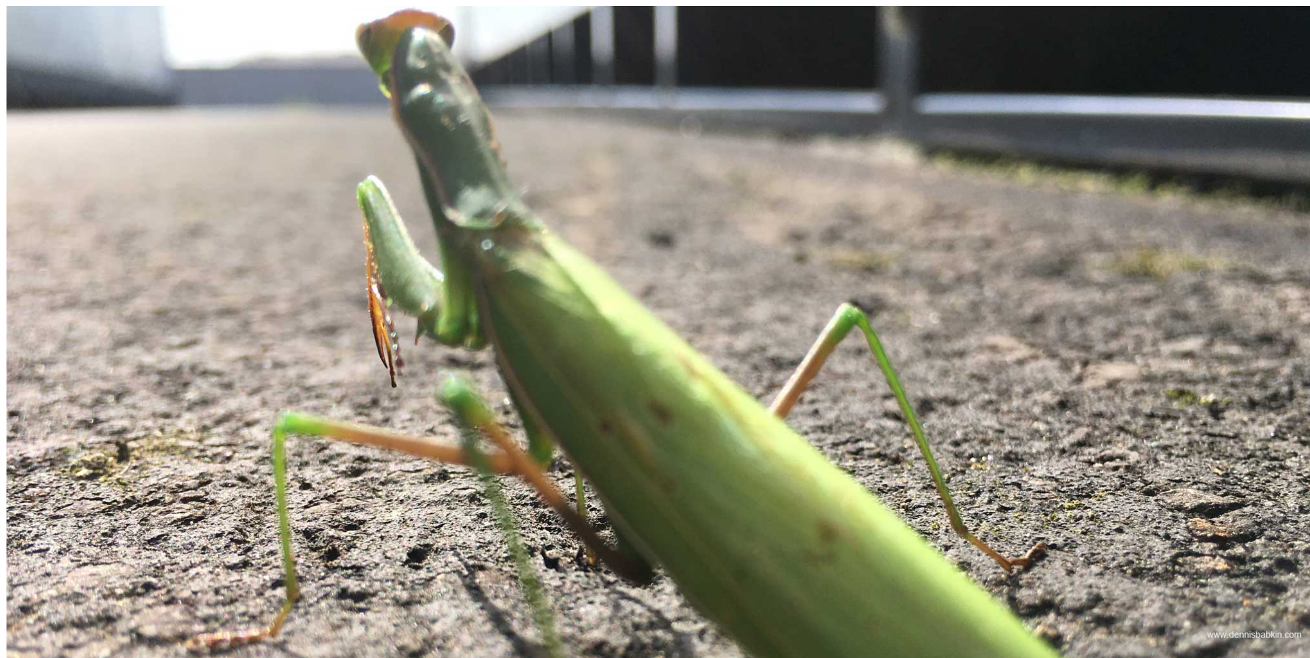
Windows Authentication - Credential Providers - Part 2

[]



Intro

In the first part of this series I've shown some basic concepts of creating a credential provider in Windows, things like what credential provider is, what components make it up, and which COM interfaces it relies on. All of that stuff will be required to follow on in this blog post. So please be sure to read that first.

To recap it very briefly, a credential provider's purpose is to collect credentials needed to log in users to their Windows accounts. And, under the hood, a credential provider is just a collection of COM objects that are invoked by a host caller (in many cases a `LogonUI` process) that perform a certain task.

Thus, what helped me to grasp how to code my own credential provider was to understand its life cycle. Or, in other words, what callbacks come when and what they do. And this is what I will dedicate this part of the series for.

So without further adieu, let's start from the very beginning of the credential provider's life.

Table Of Contents

For an easier navigation here's the table of contents:

Nomenclature
Registration
Debugging & Testing
Uninstalling
- De-registration
- De-registration "The Hard Way"

DLL Exports
Initialization
- Filter::UpdateRemoteCredential
- Filter::Filter
- Provider::SetUsageScenario
- Provider::SetSerialization
- Provider::SetUserArray
- Filter::Filter (PLAP)
- Provider::Advise

User Interface Initialization
- Provider::GetCredentialCount
- Provider::GetCredentialAt
- Provider::GetFieldDescriptorCount
- Provider::GetFieldDescriptorAt

User Interface Callbacks
- Credential::GetStringValue
- Credential::GetFieldState
- Credential::GetFieldOptions
- Credential::GetBitmapBufferValue
- Credential::GetBitmapValue
- Credential::GetCheckboxValue

- Credential::GetComboBoxValueCount
- Credential::GetComboBoxValueAt
- Credential::GetSubmitButtonValue

Post Initialization
- Credential::GetUserSid
- Credential::Advise

User Interaction & State Changes
- Credential::SetSelected
- Credential::SetDeselected
- Provider::SetDisplayState
- Credential::SetStringValue
- Credential::SetCheckboxValue
- Credential::SetComboBoxSelectedValue
- Credential::CommandLinkClicked

Submit Button Sequence
- Credential::Connect
- Credential::Disconnect
- Credential::GetSerialization
- Credential::ReportResult

Uninitialization
- Credential::UnAdvise
- Provider::UnAdvise

Wrapping An Existing Credential Provider

Specifics Of A Remote Desktop Connection

Conclusion

Nomenclature

To simplify the contents of this post let me come up with the following naming convention:

- `MyFactory` will be the name for my factory class.
- `MyFilter` will be my filter class.
- `MyProvider` will be my provider class.
- `MyCredential` will be my credential class.

I already described what each of those classes does in part 1 of this series. So click the links above to read about each class.

Additionally, I showed what "tiles" and "UI fields" are in a credential provider.

Registration

As most of the COM things, a credential provider needs to register itself in the system registry. It has a few other places to do so than just your run-of-the-mill COM component.

The credential provider needs to register its filter and provider classes in the following keys, respectively:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credentia
l Provider Filters
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credentia
l Providers
```

For that create two unique CLSIDs. You can do it using Microsoft's "Create GUID" tool in Visual Studio.

> For the purpose of this demo I will be using the following CLSIDs. But don't borrow mine. If you do that you will create a naming conflict:
> - `{855E2A67-A476-4664-8581-01BEC33975BF}` for `MyFilter` class.
> - `{855E2A67-A476-4664-8581-01BEC33975BC}` for `MyProvider` class.
>
> I separated them just by the last hex digit for readability.

The format to register `MyFilter` class is as follows:

Regedit[Copy]

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credenti
al Provider Filters\{855E2A67-A476-4664-8581-01BEC33975BF}]
@="MyCredentialProvider Filter"

[HKEY_CLASSES_ROOT\CLSID\{855E2A67-A476-4664-8581-01BEC33975BF}]
@="MyCredentialProvider Filter"

[HKEY_CLASSES_ROOT\CLSID\{855E2A67-A476-4664-8581-01BEC33975BF}\InprocServer32]
@="path\\to\\MyCredentialProvider.dll"
"ThreadingModel"="Apartment"
```

The default string value for each key does not matter, except for the `InprocServer32` key. It must contain the path to your credential provider DLL.

> Even though it is possible to specify only a DLL file name in the `InprocServer32` key, for security purposes, I would strongly advise to use the absolute file path to prevent any ambiguity.

> IMPORTANT: Make sure to place your credential provider DLL somewhere in a write-protected folder, where only administrators have the write access to it! Otherwise you will be creating a security loophole.
> A good location would be the `%ProgramW6432%` system directory under your company name. For instance:
>
> ```
> "C:\Program Files\My Company\Credential Provider\MyCredentialProvider.dll"
> ```

> Because you may be testing your credential provider in a virtual machine (VM), with the debugger build configuration, make sure that its DLLs can be loaded in that OS. You can use my WinAPI Search tool to determine if all of its dependent modules are present in that OS. Otherwise your credential provider won't load and you won't see much when you attempt to test it.

Then the format to register `MyProvider` class is as follows:

Regedit[Copy]

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credenti
al Providers\{855E2A67-A476-4664-8581-01BEC33975BC}]
@="MyProvider Password MFA"

[HKEY_CLASSES_ROOT\CLSID\{855E2A67-A476-4664-8581-01BEC33975BC}]
@="MyProvider Password MFA"

[HKEY_CLASSES_ROOT\CLSID\{855E2A67-A476-4664-8581-01BEC33975BC}\InprocServer32]
@="path\\to\\MyCredentialProvider.dll"
"ThreadingModel"="Apartment"
```

> Although you can technically separate your filter and provider classes into different modules (DLLs), it is highly impractical, and most credential providers don't do that.

The names for the keys should describe what your provider does. In my case I'm writing a generic password provider with a secondary authentication, or MFA.

After you set those registry keys, the credential provider will be good to go. No reboot is necessary. All you need to test it, among other things, is to lock the workstation, or to log out the user. If everything goes well, you should see your credential provider in action.

> Keep in mind that a registered credential provider could be immediately used not only during a login process, but also from the Windows Explorer for some UAC prompts, and from other apps that may require Windows user account verification.

Having said that, **I need to warn you** that it is very easy to mess up your operating system with a faulty credential provider!

Debugging & Testing

It should probably go without saying that you should **not install** and test your credential provider on the same system that you are developing it on, or on any production or important PC that you don't want to lose access to.

> By its nature a credential provider loads before you can log in to your system, and thus if you mess something up, you are risking to bork that computer!
> Because of that, always test your credential provider in a virtual machine!

Although for the most hardcore developers, it is possible to debug a credential provider on the same system, installing it in a VM greatly simplifies things. So don't be a masochist.

Debugging a credential provider is an entire subject of its own, thus I will leave it for a future blog post.

> Let me briefly mention that the best way to see what is happening in a credential provider is to use diagnostic logging (into a text file). And as debugging itself is concerned, you can use Visual Studio Remote Debugger to attach to the host process, or `LogonUI.exe` for the login screen.

For now though let me review how you can uninstall your credential provider, an easy way, and a hard way, if you happened to have skipped this chapter before you installed it.

Uninstalling

Let's first review an easy uninstallation, if you have access to any of the user accounts on the PC where the credential provider is installed.

De-registration

Simply remove the registry keys for the filter class:

> I'm using my own CLSIDs for the filter and provider classes. You obviously need to substitute those GUIDs with your own.

Regedit[Copy]

```
[-
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credentia
l Provider Filters\{855E2A67-A476-4664-8581-01BEC33975BF}]

[-HKEY_CLASSES_ROOT\CLSID\{855E2A67-A476-4664-8581-01BEC33975BF}]
```

And for the provider class:

Regedit[Copy]

```
[-
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credentia
l Providers\{855E2A67-A476-4664-8581-01BEC33975BC}]

[-HKEY_CLASSES_ROOT\CLSID\{855E2A67-A476-4664-8581-01BEC33975BC}]
```

And that is pretty much it. Your credential provider will not be loaded anymore. No reboot is necessary.

De-registration "The Hard Way"

But, if you don't have access to a user account on the PC where you installed your credential provider, you will need to uninstall it the hard way.

> The easy-hard-way is to create a snapshot in a VM before you install it. And then simply revert the snapshot if something goes wrong. You will lose all your diagnostic logs and other data that way, but at least it is fast.

But if you didn't make a snapshot to revert to, or if you're testing it on a physical PC, follow these steps to remove your credential provider:

1. Press and hold the power button to initiate a hard reset. In a VM, there should be a menu option for that.

2. While the OS is rebooting, initiate a hard reset once again. This should put it into a recovery mode the next time it boots.

   > If the recovery mode doesn't come up, you may want to repeat the hard reset for a few times. Or, alternatively, you can press and hold the F9 (or, on some systems the F12) keyboard key during the initial boot sequence to initiate it manually.

3. When you enter the recovery mode in Windows, select "Troubleshoot", then "Advanced Options" and "Command Prompt".

   > You may need to provide your administrator password at that stage.

4. When the command prompt opens up, use the following commands to switch to the directory where you installed the credential provider and then remove its DLL:
   Admininstrator Command Prompt[Copy]

   ```
   C:

   cd "path\to"

   del /f MyCredentialProvider.dll
   ```

   In the example above, I'm assuming that you installed your credential provider on drive `C:`, where the `path\to` is the directory where you placed it in, and `MyCredentialProvider.dll` is the file name for the credential provider DLL.

5. Reboot the system.
6. You should not see your credential provider anymore and should be able to log in.

Be careful next time and please heed my advice that I gave above!

DLL Exports

Your credential provider DLL must have the following exported functions to be able to load as a COM component:

C++[Copy]

```
extern "C" __declspec(dllexport) HRESULT STDAPICALLTYPE DllGetClassObject(REFCLSID rclsid, REFIID riid, void** ppv);

extern "C" __declspec(dllexport) HRESULT STDAPICALLTYPE DllCanUnloadNow();
```

The `DllCanUnloadNow` function must implement module-wide reference counting, and return `S_OK` when the count reaches 0, and `S_FALSE` otherwise. This is your DLL's way of saying that no internal class is active in it.

And the `DllGetClassObject` function is where your credential provider begins its life cycle.

Initialization

Once registered your credential provider will go through the following stages when a host process tries to load it:

1. From within the `DllGetClassObject` call you need to check the `rclsid` parameter, that was passed into it, to be one of your CLSIDs that you registered `MyFilter` class and `MyProvider` class with. And if so, create an instance of `MyFactory` class for each of those CLSIDs, and invoke its `QueryInterface` method with the `riid` and `ppv` parameters that were passed in the `DllGetClassObject` call.
2. The `IClassFactory` will then invoke the `CreateInstance` method of your `MyFactory` class with the `riid` set to `ICredentialProvider`. In turn your `MyFactory` class should create an instance of your `MyProvider` class, and later call its `IUknown`'s `QueryInterface` with the `riid` and `ppv` that were passed into the `CreateInstance` call.

3. Your `MyProvider` class will then begin receiving its own calls to its `QueryInterface` method. The most obvious are requests for the `ICredentialProvider` interface `riid`. That is where it needs to return the pointer to its inherited `ICredentialProvider` interface.

   > To simplify dealing with pointers to inherited interfaces use the `QITABENT` and `QITABENTMULTI` macros in the `QITAB` struct and the `QISearch` function. (Check the description for the `QISearch` function for details.)

   Additionally, the `QueryInterface` method for `MyProvider` class may receive requests for the `IAutoLogonProvider` interface at this early stage. It is an undocumented interface, that is declared as such:

   C++[Copy]

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

   The only method that is available in it deals with `IAutoLogonManager`, that seems to be declared as such (it is also undocumented):

   C++[Copy]

```
struct IAutoLogonManager : public IUnknown
{
        virtual HRESULT STDMETHODCALLTYPE IgnoreAutoLogonMode() = 0;
        virtual BOOL STDMETHODCALLTYPE IsAutoLogonMode() = 0;
        virtual BOOL STDMETHODCALLTYPE IsSingleUserNoPasswordAutoLogonMode() =
0;
        virtual BOOL STDMETHODCALLTYPE IsAutoLogonNotSingleUserNoPasswordMode()
= 0;
        virtual BOOL STDMETHODCALLTYPE IsSystemAutoLogon() = 0;
        virtual HRESULT STDMETHODCALLTYPE UpdateAutoLogonDefaultSID() = 0;
        virtual HRESULT STDMETHODCALLTYPE ResetAutoLogonSetting() = 0;
        virtual HRESULT STDMETHODCALLTYPE ClearSystemAutoLogonSetting() = 0;
        virtual HRESULT STDMETHODCALLTYPE GetAutoLogonCredential(_Out_ PWSTR*,
_Out_ PWSTR*, _Out_ PWSTR*) = 0;
        virtual BOOL STDMETHODCALLTYPE IsSingleUser() = 0;
        virtual HRESULT STDMETHODCALLTYPE
GetSerializedAutoLogonCredential(ICredProviderCredentialSerialization * *) = 0;
};
```

   > Without having done too much reverse engineering of the `IAutoLogonProvider` interface, I would surmise to say that it deals with automatic logins that credential providers support.

4. After that your `MyFactory` class will receive another call to its `CreateInstance` method with `riid` for the `ICredentialProviderFilter` interface. As before, it needs to create an instance of your `MyFilter` class and invoke the `QueryInterface` on the `IUnknow` that it inherited from, with the `riid` and `ppv` that it received in the `CreateInstance` call.

Filter::UpdateRemoteCredential

5. This callback will take place if the system receives a remote desktop connection (RDP) request. Otherwise you won't see this callback.

   The purpose of this callback is to select the correct provider to redirect the serialized login data to. In case of the RDP connection such data comes from a remote computer.

   Note that this request arrives to the credential provider after the RDP client on the remote machine supplies the correct user credentials. Or, in other words, the received serialized data should contain valid credentials for one of the local users in the system where the credential provider is running.

   > I will give an overview of an RDP connection later.

   The job of this callback is to substitute the provider (defined by its CLSID in the `clsidCredentialProvider` parameter) that it received in the `pcpcsIn` parameter on the input, with the provider CLSID that the received serialized data must be redirected to. This is done in the `pcpcsOut` parameter.

   Later on, the provider class that was specified in the `pcpcsOut` parameter, will receive the same serialized data in its `SetSerialization` callback for the actual processing.

   > Make sure to treat provided serialized binary login data in the `pcpcsIn` parameter as if it came from an untrusted source!

   This method should return `S_OK` if it could substitute the provider, to proceed with the login data received. Otherwise it should return an error code.

   If this callback returns a failure error code, the remote user will be presented with a remote credential provider on their screen. After that, they will have to re-login using the remote credential provider.

   > To determine if computer is currently receiving an RDP connection use the `SM_REMOTESESSION` flag, with some additional checks, that are explained here.

Filter::Filter

6. Your `MyFilter` class will receive a call to its `Filter` method with the details of the usage scenario for your credential provider, and with the list of all available providers in the system.

   This is usually where you need to decide if you want to support such scenario in your credential provider. And if so, your job is to deny any providers that you do not want to support by setting the `BOOL` values in the `rgbAllow` array to FALSE. Otherwise, just leave them unchanged.

   > I already showed the list of providers that my credential provider's filter received on Windows 10. This list may be obviously different on some other OS.

   You usually make your decision based on where your provider was called from. That will be supplied to you in the `cpus` parameter.

   For instance, if my MFA provider does not support `CPUS_CHANGE_PASSWORD`, `CPUS_CREDUI` or `CPUS_PLAP` usage scenarios, I should return `E_NOTIMPL` for them from my `Filter` callback. This will begin the process of unloading of my credential provider from the host process.

   Otherwise, when you're done with your filtering logic, return `S_OK` from the `Filter` method if you want to continue loading your credential provider for that usage scenario.

   > Note that if you do not return `S_OK` from the `Filter` method, your credential provider will receive only calls to the following methods before it is unloaded: `Provider::SetUsageScenario`, `Provider::UnAdvise`. Also note that in this case it will not receive a previously matching call to `Provider::Advise`.

Provider::SetUsageScenario

7. Your `MyProvider` class will receive this callback with details of the usage scenario, similar to what your `Filter::Filter` had received earlier. And thus your response to this callback should be coordinated with the `Filter::Filter` callback.

   This callback basically tells `MyProvider` class how your credential provider is loaded: whether it's a login screen, an unlocking of the workstation, a change-of-password screen, a call from the `CredUIPromptForWindowsCredentials` function, etc.

   Return `S_OK` for success, or `E_NOTIMPL` if you don't want to support such usage scenario.

Provider::SetSerialization

8. This callback lets your `MyProvider` class process the serialized input that was received from outside. Such input is usually received from a remote computer via the remote desktop connection (RDP) request, or as a result of a call to the `CredUIPromptForWindowsCredentials` function by some other process (such as the Windows Explorer, when displaying some forms of the UAC prompt, etc.)

   > Make sure to treat provided serialized binary login data in the `pcpcs` parameter as if it came from an untrusted source!

   > Your `MyProvider` class may not receive this callback in other cases. Otherwise, it will follow the `UpdateRemoteCredential` callback in your `MyFilter` class.

   The job of this method is to process serialized data that it received in the `pcpcs` parameter, and to fill in the UI fields in the appropriate credential. It may also determine if the supplied login info is enough to warrant an automatic login. If so, it may later proceed with the submit sequence at the later callbacks.

   > The implementation in the default password credential provider in Windows performs an in-place decoding of received input parameters using an internal `KerbInteractiveUnlockLogonUnpackInPlace` function. You can find it in the official Microsoft sample.
   > After that, if it succeeds at decoding the input credentials, it later initiates an *autologon* in the `GetCredentialCount` callback.

   But, if provided serialized data is not enough, or if there's an error during de-serialization, this callback may return `E_UNEXPECTED`. In that case, the (remote) user requesting the login will be asked to re-enter their credentials.

   The format of the serialized binary data received in the input parameter depends on the provider. For system providers you can use the `CredUnPackAuthenticationBufferW` API to decode it.

   Thus, you can apply the following logic to try to de-serialize the data received:

   C++[Copy]

```
#include <tchar.h>
#include <wincred.h>
#pragma comment(lib, "Credui.lib")

BOOL DeserializeRemoteProviderLoginData(
        __in const CREDENTIAL_PROVIDER_CREDENTIAL_SERIALIZATION* pcpcs)
{
        BOOL bRes = FALSE;
        int nOSError = 0;

        if(pcpcs &&
                pcpcs->rgbSerialization)
        {
                DWORD dwchSzUser = 0;
                DWORD dwchSzPwd = 0;

                DWORD dwFlags = CRED_PACK_PROTECTED_CREDENTIALS;

                BOOL bB = CredUnPackAuthenticationBufferW(dwFlags,
                        pcpcs->rgbSerialization,
                        pcpcs->cbSerialization,
                        NULL, &dwchSzUser,
                        NULL, NULL,
                        NULL, &dwchSzPwd);

                nOSError = GetLastError();
                if(!bB &&
                        nOSError == ERROR_INVALID_PARAMETER)
                {
                        //Try an old method w/o password encryption
                        dwFlags = 0;

                        bB = CredUnPackAuthenticationBufferW(dwFlags,
                                pcpcs->rgbSerialization,
                                pcpcs->cbSerialization,
                                NULL, &dwchSzUser,
                                NULL, NULL,
                                NULL, &dwchSzPwd);

                        nOSError = GetLastError();
                }

                if(!bB &&
                        nOSError == ERROR_INSUFFICIENT_BUFFER)
                {
                        WCHAR* pUsr = new (std::nothrow) WCHAR[dwchSzUser];
                        if(pUsr)
                        {
                                WCHAR* pPwd = new (std::nothrow)
WCHAR[dwchSzPwd];
                                if(pPwd)
                                {
```

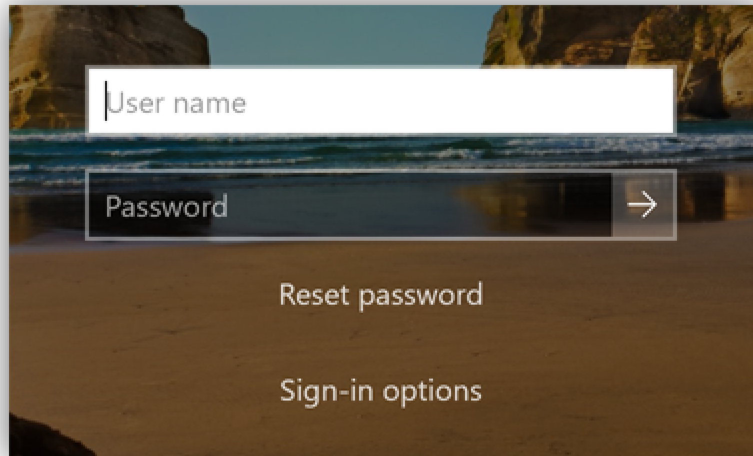
```
if(CredUnPackAuthenticationBufferW(dwFlags,
                                        pcpcs->rgbSerialization,
                                        pcpcs->cbSerialization,
                                        pUsr, &dwchSzUser,
                                        NULL, NULL,
                                        pPwd, &dwchSzPwd))
                                {
                                        bRes = TRUE;
                                        nOSError = 0;

                                        wprintf(L"User: %s\n", pUsr);

                                        //DO NOT output, log, or store
the password!!!
                                        wprintf(L"Password: %s\n",
                                                pPwd && pPwd[0] ?
L"provided" : L"empty");

                                }
                                else
                                        nOSError = GetLastError();

                                //Securely erase the password
                                SecureZeroMemory(pPwd, dwchSzPwd);
                                delete[] pPwd;
                                pPwd = NULL;
                        }
                        else
                                nOSError = ERROR_OUTOFMEMORY;

                        delete[] pUsr;
                        pUsr = NULL;
                }
                else
                        nOSError = ERROR_OUTOFMEMORY;
        }
}
else
        nOSError = ERROR_EMPTY;

SetLastError(nOSError);
return bRes;
}
```

Provider::SetUserArray

9. This callback is invoked with the list of currently available users for the login. A usual implementation is to cache provided user data for future reference.

   > This callback will not be available if you're using an older (version 1) provider class.

   Keep in mind that in some situations a host process may request to display a so-called "*Other User*" tile. Unlike regular tiles, such a tile gives user an option to enter their user name and password:

   "Other User" tile on Windows 10.

   In this case, the user array provided in the `users` parameter will not include any information about the "*Other User*" tile. You can deduce its presence by calling the `GetAccountOptions` method and by examining its `credentialProviderAccountOptions` parameter as such:

   C++[Copy]

```
HRESULT STDMETHODCALLTYPE MyProvider::SetUserArray(
        _In_ ICredentialProviderUserArray* users)
{
        //Determine if we need to display the "Other user" tile
        enum class OTHER_USER_TILE
        {
                NotNeeded,
                Needed,
                NeededMicrosoftAccount,
        };

        OTHER_USER_TILE other_user = OTHER_USER_TILE::NotNeeded;

        CREDENTIAL_PROVIDER_ACCOUNT_OPTIONS cpao;
        if(SUCCEEDED(users->GetAccountOptions(&cpao)))
        {
                if(cpao & CPAO_EMPTY_LOCAL)
                {
                        if(cpao & CPAO_EMPTY_CONNECTED)
                        {
                                //"Microsoft account" is when you sign in using
your email address or phone number.
                                other_user =
OTHER_USER_TILE::NeededMicrosoftAccount;
                        }
                        else
                        {
                                //Regular "Other user" tile
                                other_user = OTHER_USER_TILE::Needed;
                        }
                }
        }

        //Do other work ...
}
```

You can obtain the following information for the available users: user name (eg: `Admin`), user's display name (or the name, that could be added after a user account was created, eg: `Admin Account`), "qualified user name" (eg: `DESKTOP-14CH5ES\Admin`), "logon status string" (eg: `Locked`), user's primary SID (eg: `S-1-5-21-2104516720-2747548040-1419514401-1001`) and "provider ID" (or special GUID for online Microsoft user accounts) using the `GetStringValue` function.

Filter::Filter (PLAP)

10. This optional callback in your MyFilter class may be invoked if the system needs to support pre-logon authentication for a credential provider, or PLAP.

    > An example of such scenario would be the need for the 802.1x authentication to establish a network connection.

    In this case the `cpus` parameter will be set to `CPUS_PLAP`.

    The processing of this filter method should be similar to what I already described, with the exception that if you return `E_NOTIMPL` from this method, your credential provider will not be unloaded. It will tell the host process that it does not support the PLAP features.

Provider::Advise

11. This callback is invoked to ask your MyProvider class if it wants to use certain events (through the `ICredentialProviderEvents` interface). For instance, if you need to refresh (or redraw) the tiles in your credential provider, you can do so by calling the `CredentialsChanged` function. But for that you need to retain a copy of the `ICredentialProviderEvents` interface in your `Advise` callback.

    > To retain a copy of the interface simply increment the reference count by using its `AddRef` method. In that case though, remember to dereference it in the corresponding `UnAdvise` callback by calling its `Release` method.

    If you do not need this callback, simply return `E_NOTIMPL` from it.

User Interface Initialization

The following sequence of callbacks will be sent to your credential provider to initialize the user interface (UI) components:

> From a perspective of a UI developer, the way credential provider's UI is structured may appear odd. It works by the host process asking for UI elements and by your `MyProvider` (or `MyCredential`) classes responding. There's not always a freedom for the latter classes to simply draw their UI elements at some arbitrary location or of an arbitrary size, nor to resize, move or to recreate themselves.

Provider::GetCredentialCount

1. This method is called to let your provider specify how many tiles it wants to show and whether or not it wants to use the *autologon* feature.

   > Note that the number of tiles may not always match the number of users, since some providers may not support all user accounts.

   Set the number of tiles that you want to display in the `pdwCount` parameter. This number must also include the "*Other User*" tile, if the host process requested it in the `SetUserArray` callback.

   > Note that once you specify the number of tiles in this callback you will not be able to add or remove them.

   In case your provider wants to initiate the *autologon*, it needs to set the `pdwDefault` parameter to the index of the tile that it wants to use for it, and the `pbAutoLogonWithDefault` parameter to point to a TRUE value. Additionally, it needs to fill out the UI fields for the default tile in your `MyCredential` class.

   > The *autologon* is a feature of the credential provider, that instructs it to try to perform user login automatically, or without waiting for a user to type in their login credentials.
   > The *autologon* may be used during an RDP login, or for an automatic logon that could be set up via the `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon` system registry key.
   >
   > Note that the *autologon* is not guaranteed to succeed, and it may be aborted if the automatically provided user credentials are not validated. In that case the credential provider should behave as if the *autologon* was not initiated.

   If *autologon* is not needed, set the credential index pointed by `pdwDefault` to `CREDENTIAL_PROVIDER_NO_DEFAULT`, or -1.

Provider::GetCredentialAt

2. This method is called to retrieve a pointer to an instance of your `MyCredential` class for each tile by its index, supplied in the `dwIndex` parameter. Tile indexes go sequentially from 0 to the number of tiles (minus one) specified in the `GetCredentialCount` callback.

   > Remember that a "tile" is a visual representation of a "credential" class.

   The usual implementation of this callback is to create an instance of your `MyCredential` class for each invocation of this callback and to pass it back to the host process in the `ppcpc` parameter.

Provider::GetFieldDescriptorCount

3. Is called to retrieve the number of UI fields that will be ever needed for the usage scenario for all tiles in your `MyProvider` class. Set the `pdwCount` parameter to the overall number of UI fields.

   > Remember that UI fields are controls on the screen that a user interacts with to provide the login credentials. These could be text input fields, buttons, checkboxes, etc.

   > Once UI fields are created you will not be able to add or remove them. There are some limited forms of modifications that UI fields allow, like changing their text, or hiding/showing them.

   The usual implementation of this callback is to create a large array of UI fields and set them as hidden in the following `GetFieldState` callbacks. After that the logic in your credential provider can show or hide UI fields in response to other callbacks, such as when tiles become selected or deselected.

Provider::GetFieldDescriptorAt

4. This callback is invoked for each UI field to request its order on the screen, unique ID, text and its type, i.e. whether it's a text, input/password box, checkbox, etc. The number of UI fields is defined by a previous call to the `GetFieldDescriptorCount` method.

   Every invocation of this callback has a zero-based index of the UI field, provided in the `dwIndex` parameter. The job of this method is to fill out the `CREDENTIAL_PROVIDER_FIELD_DESCRIPTOR` struct and to return it in the `ppcpfd` parameter.

   You will need to allocate the memory for the `CREDENTIAL_PROVIDER_FIELD_DESCRIPTOR` struct using the `CoTaskMemAlloc` function, as well as to allocate strings in it using the `SHStrDupW` API, similar to the following code snippet:

   C++[Copy]

```
HRESULT STDMETHODCALLTYPE MyProvider::GetFieldDescriptorAt(__in DWORD dwIndex,
        __deref_out CREDENTIAL_PROVIDER_FIELD_DESCRIPTOR** ppcpfd)
{
        HRESULT hr;
        CREDENTIAL_PROVIDER_FIELD_DESCRIPTOR *pD =
(CREDENTIAL_PROVIDER_FIELD_DESCRIPTOR*)
                CoTaskMemAlloc(sizeof(CREDENTIAL_PROVIDER_FIELD_DESCRIPTOR));
        if(pD)
        {
                hr = SHStrDupW(source_cred_fields[dwIndex].label, &pD-
>pszLabel);
                if(SUCCEEDED(hr))
                {
                        pD->dwFieldID = source_cred_fields[dwIndex].dwID;
                        pD->cpft = source_cred_fields[dwIndex].type;
                        pD->guidFieldType = source_cred_fields[dwIndex].guid;

                        //Free old memory (if it's there, in case of a
"wrapper")
                        if(*ppcpfd)
                        {
                                CoTaskMemFree(*ppcpfd);
                        }

                        *ppcpfd = pD;
                }
                else
                {
                        //Failed to alloc
                        CoTaskMemFree(pD);
                }
        }
        else
                hr = E_OUTOFMEMORY;

        //Do other work ...
}
```

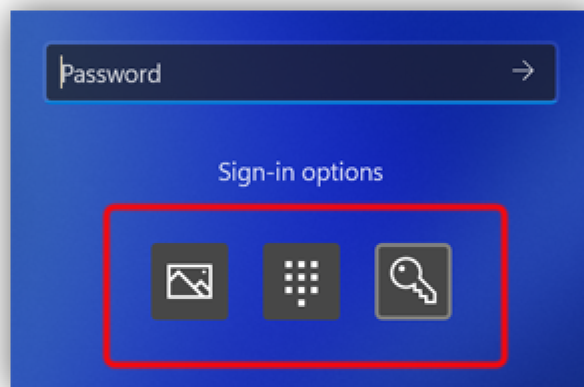
Where:

C++[Copy]

```
const struct
{
    DWORD dwID;
    CREDENTIAL_PROVIDER_FIELD_TYPE type;
    LPCWSTR label;
    GUID guid;
}
source_cred_fields[] =
{
    { 0,  CPFT_LARGE_TEXT,    L"Large text",     {},                    },
    { 1,  CPFT_EDIT_TEXT,     L"User name",      CPFG_LOGON_USERNAME, },
    { 2,  CPFT_PASSWORD_TEXT, L"Password field", CPFG_LOGON_PASSWORD, },
        //...
};
```

Most UI field types are self-explanatory, but some require clarification:

- First off, the `guidFieldType` member of the `CREDENTIAL_PROVIDER_FIELD_DESCRIPTOR` struct is an extension of the UI field type. It allows you to specify an extra meaning for that field. You can use your own unique GUID for that, or use Microsoft-defined ones, like `CPFG_LOGON_USERNAME` or `CPFG_LOGON_PASSWORD` in the code example above. In case of the latter ones, they add special properties to the field, that could be recognized by the host process.
- `CPFT_COMMAND_LINK` is a clickable text (or a link) that you can respond to in your `CommandLinkClicked` callback. It can also be transformed into a button.
- `CPFT_TILE_IMAGE` field type with the `CPFG_CREDENTIAL_PROVIDER_LOGO` extension is a special field that displays your provider's logo. You can see UI images of this type when you click the "*Sign in options*" link:



"Sign in options" with three UI images on Windows 11.

- `CPFT_SMALL_TEXT` field type with the `CPFG_CREDENTIAL_PROVIDER_LABEL` extension is another special field that contains an accessibility user prompt for the provider logo that I showed above. You can see such prompt if you hover the mouse pointer over one of the UI images for a provider in the "*Sign in options*" section.

To let you better visualize what UI fields are available for a credential, let's assume that you specified the following fields in your `GetFieldDescriptorAt` callback in this particular order:

ID	Type	Text	guidFieldType
1	`CPFT_SMALL_TEXT`	"Small text"	
2	`CPFT_LARGE_TEXT`	"Large Text"	
3	`CPFT_EDIT_TEXT`	"Edit field - User name"	`CPFG_LOGON_USERNAME`
4	`CPFT_EDIT_TEXT`	"Edit field"	
5	`CPFT_PASSWORD_TEXT`	"Password Field 1"	`CPFG_LOGON_PASSWORD`
6	`CPFT_PASSWORD_TEXT`	"Password Field 2"	
7	`CPFT_COMBOBOX`	"Combo field"	
8	`CPFT_CHECKBOX`	"Checkbox 1"	
9	`CPFT_COMMAND_LINK`	"Command link"	
10	`CPFT_TILE_IMAGE`	"Tile image 1"	
11	`CPFT_TILE_IMAGE`	"Provider logo"	`CPFG_CREDENTIAL_PROVIDER_LOGO`
12	`CPFT_SMALL_TEXT`	"Provider label"	`CPFG_CREDENTIAL_PROVIDER_LABEL`
13	`CPFT_SUBMIT_BUTTON`	"Submit Btn"	

These UI fields can be rendered as follows, marked with their IDs, in the login credential provider:

Admin

1 Small text

2 Large Text

3 Edit field - User name

4 Edit field

5 ●●●●●●●●●●●●●●●● 13

6 ●●●●●●●●●●●●●●●●

7 Item 1

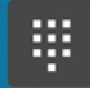
8 Checkbox 1

9 Command link

10 Tile image 1

12 Provider label

Sign-in options



UI fields in the login screen on Windows 11.

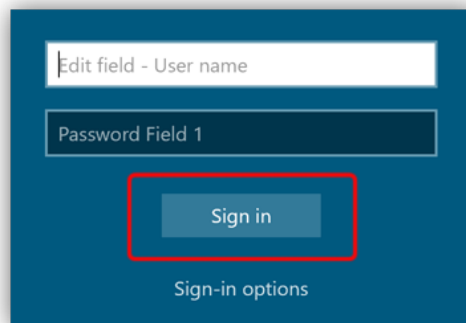
There are multiple restrictions that apply to UI fields:

- The order at which you specify UI fields matter. The first fields will appear on top.
- Note that hidden UI fields do not take any space on the screen.
- You cannot specify the position of a UI field on the screen. You can only specify its order.
- All UI fields will be spaced vertically by their order. With the exception of the "submit" button, UI fields cannot be placed horizontally next to each other.
- You cannot specify the size, or style of a UI field.
- On Windows 10 and later OS, the login screen (or `LogonUI.exe` host process) will always display a user account image on top, followed by the user account name in the selected tile. You cannot disable, alter, or style it from your credential provider.
- `CPFT_LARGE_TEXT` field type seems to be ignored by `LogonUI.exe` host process on Windows 10 and later OS. It is always rendered as the `CPFT_SMALL_TEXT` type.
- `CPFG_LOGON_USERNAME` and `CPFG_LOGON_PASSWORD` are special extension GUIDs for the `CPFT_EDIT_TEXT` field type that hint to the host process which fields contain login credentials for the "submit" button.
- `CPFT_TILE_IMAGE` field type may contain any arbitrary bitmap and a label under it. Your `MyCredential` class can specify it in the `GetBitmapValue` callback.

  > Note that the size of such image is defined by the host process and cannot be changed from the credential provider.

- `CPFT_TILE_IMAGE` field type was used to display a user account image if you used an older V1 credential provider. The credential provider derived from the V1 provider class could specify its own bitmap for it. The newer provider class does not support this functionality and the user account bitmap is rendered by the host process (i.e. `LogonUI.exe`).
- `CPFT_TILE_IMAGE` field type with the `CPFG_CREDENTIAL_PROVIDER_LOGO` extension GUID allows to specify an arbitrary bitmap to be used for your `MyProvider` class. It is usually created with the `CPFS_DISPLAY_IN_DESELECTED_TILE` attribute for a credential. With that setup, it will be displayed by default in the "*Sign in options*" section, as I showed above.

- `CPFT_SMALL_TEXT` field type with the `CPFG_CREDENTIAL_PROVIDER_LABEL` extension GUID allows to specify an accessibility label for the bitmap image for your `MyProvider` class. It should be also created with the `CPFS_DISPLAY_IN_DESELECTED_TILE` attribute.
- `CPFT_SUBMIT_BUTTON` is a special field type for the "submit" button. It has a few unique properties. There can be only one submit button. Although there could be no submit buttons, like for the `NgcPinProvider`. Additionally, Microsoft recommends *attaching* your submit button to another control, usually to a password field. You can do so by responding to the `GetSubmitButtonValue` callback in your `MyCredential` class. After that, the "submit" button will appear on the side of that UI field, as I showed in the screenshot above (item with ID=13).

> In case you do not want to attach your submit button to any fields, create it with the `CPFG_STANDALONE_SUBMIT_BUTTON` extended guidFieldType:
> C++[Copy]
>
> ```
> // {0b7b0ad8-cc36-4d59-802b-82f714fa7022}
> DEFINE_GUID(CPFG_STANDALONE_SUBMIT_BUTTON, 0x0b7b0ad8, 0xcc36,
> 0x4d59, 0x80, 0x2b, 0x82, 0xf7, 0x14, 0xfa, 0x70, 0x22);
> ```
>
> And return its own field index in the `GetSubmitButtonValue` callback. In that case the credential provider will display it as such:
>
> 
>
>
> Standalone "submit" button in the login screen on Windows 10.

- To create a button, that is not a "submit" one, use the `CPFT_COMMAND_LINK` field type with the `CPFG_STYLE_LINK_AS_BUTTON` extended GUID:
  C++[Copy]

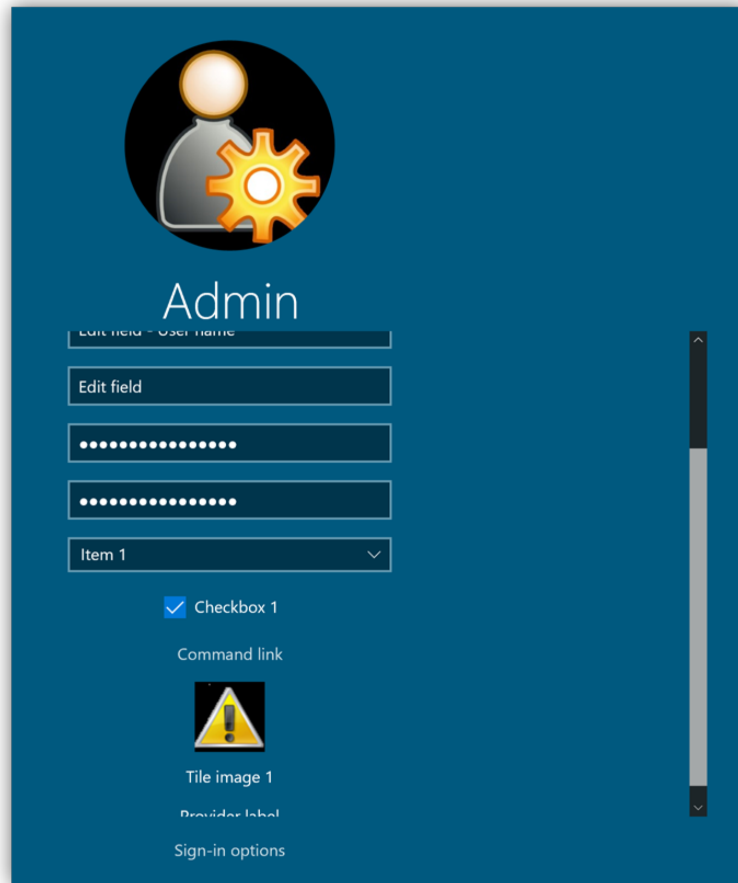
```
// {088fa508-94a6-4430-a4cb-6fc6e3c0b9e2}
DEFINE_GUID(CPFG_STYLE_LINK_AS_BUTTON, 0x088fa508, 0x94a6, 0x4430, 0xa4,
0xcb, 0x6f, 0xc6, 0xe3, 0xc0, 0xb9, 0xe2);
```

  This will turn the command link from my screenshot above (with the ID=9) into a button:

  Command link button in the login screen on Windows 10.

  You can then interact with it in the same way as you would with the command link, namely, you would generally process `CommandLinkClicked` callbacks when it is clicked.

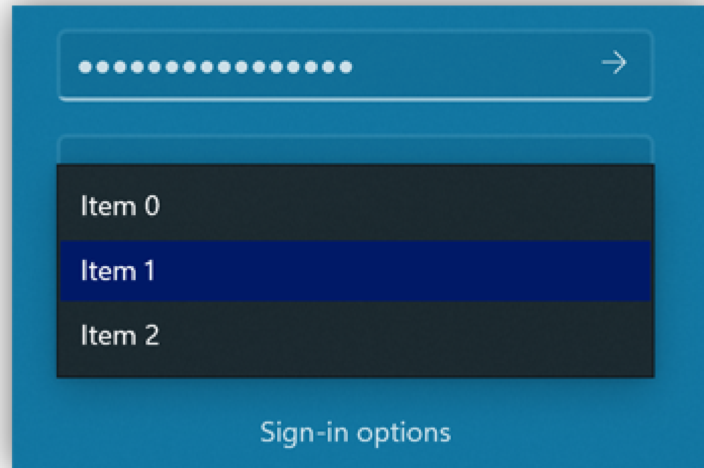
- Note that the amount of vertical space available for your UI fields is quite limited, and if you go over it, the `LogonUI.exe` will hide your lower fields and display an ugly vertical scrollbar to reach them:



Vertical scrollbar in the login screen on Windows 10.

There's no documented way to know the amount of available vertical space, and whether or not the scrollbar is shown.

- A `CPFT_COMBOBOX` field type displays a dropdown box where a user can select from the list of options. You will need to specify the number of items in that list in the `GetComboBoxValueCount` callback, and then provide text for each of those options in the `GetComboBoxValueAt` callback for your `MyCredential` class:



Open combo box in the login screen on Windows 11.

> Note that unlike desktop UI combo boxes, the one available for credential providers does not allow edited input, or specifying a list containing anything other than text.

User Interface Callbacks

The following callbacks may be dispatched to your `MyCredential` class, based on the UI fields that you specified for it:

Credential::GetStringValue

1. Is called to retrieve text for a UI field by its `dwFieldID`. The UI field ID comes from what you specified in the `GetFieldDescriptorAt` callback.

   Keep in mind that you need to return a requested string using the `SHStrDupW` function, as such:

   C++[Copy]

```
HRESULT STDMETHODCALLTYPE MyCredential::GetStringValue(
        __in DWORD dwFieldID,
        __deref_out PWSTR* ppwsz)
{
        HRESULT hr;

        if(dwFieldID == FIELD_ID_PWD_PROMPT)
        {
                const WCHAR* pstrTxt = L"Please enter password";

                hr = SHStrDupW(pstrTxt, ppwsz);
        }
        else
        {
                hr = E_NOTIMPL;
        }

        //Do other work ...
}
```

Credential::GetFieldState

2. Is called to retrieve UI field state by its `dwFieldID`, such as focused, disabled, hidden, etc. The UI field ID comes from what you specified in the `GetFieldDescriptorAt` callback.

   The UI field state should be returned in the `pcpfs` parameter, while additional state flags, in the `pcpfis` parameter.

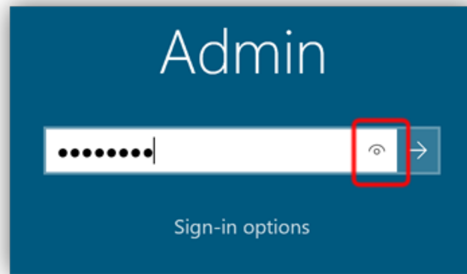
   This callback is where you would display or hide certain UI fields, based on some specific criteria.

Credential::GetFieldOptions

3. Is called to ask for options for the UI field by its `fieldID`. The UI field ID comes from what you specified in the `GetFieldDescriptorAt` callback.

   > You will receive this callback if you derived your `MyCredential` class from `ICredentialProviderCredentialWithFieldOptions` interface.

   This callback lets you specify some additional options for the UI field, such as the button to briefly reveal the password, or to limit input to an email or to numbers only, or to specify which on-screen keyboard to show:

   "Reveal password" button in the login screen on Windows 10.

   These are handy UI enhancements for your credential provider. So don't ignore them.

Credential::GetBitmapBufferValue

4. Allows to specify a bitmap image for the UI fields of the `CPFT_TILE_IMAGE` type using raw pixels. The UI field is identified by its ID, that was specified in the `GetFieldDescriptorAt` callback.

   > This callback in not officially documented.

   > Your `MyCredential` class must be derived from `ICredentialProviderCredential3` or later to receive this callback.

   The pixel data must be supplied in the `BITMAPINFO` struct and allocated using the `CoTaskMemAlloc` function.

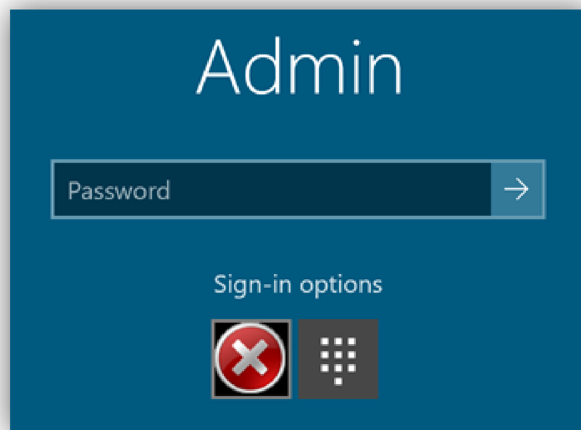
   Here's an example how you can specify an icon for your provider in the "Sign in options" area:

   C++[Copy]

```
HRESULT STDMETHODCALLTYPE MyCredential::GetBitmapBufferValue(
        __in DWORD fieldID,
    __out DWORD* pImageBufferSize, __out BYTE** ppImageBuffer)
{
        HRESULT hr = E_NOTIMPL;

    if(fieldID == FIELD_ID_PROVIDER_IMAGE)
    {
        //Let's use the system error icon for our test
        HICON hIcon = LoadIcon(NULL, IDI_ERROR);
        if(hIcon)
        {
            ICONINFOEX ix = {};
            ix.cbSize = sizeof(ix);
            if(GetIconInfoExW(hIcon, &ix))
            {
                HBITMAP hBmp = ix.hbmColor;

                BITMAP bm = {};
                if(GetObjectW(hBmp, sizeof(BITMAP), &bm))
                {
                    HDC hDC = GetDC(NULL);
                    if(hDC)
                    {
                        WORD wBitCount = 32;
                        DWORD dwBmpSize = ((bm.bmWidth * wBitCount + 31) / 32) *
4 * bm.bmHeight;
                        DWORD dwcbFullSize = sizeof(BITMAPINFOHEADER) +
dwBmpSize;

                        BYTE* lpbitmap = (BYTE*)CoTaskMemAlloc(dwcbFullSize);
                        if(lpbitmap)
                        {
                            memset(lpbitmap, 0, dwcbFullSize);

                            BITMAPINFO* pBMI = (BITMAPINFO*)lpbitmap;

                            pBMI->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
                            pBMI->bmiHeader.biWidth = bm.bmWidth;
                            pBMI->bmiHeader.biHeight = bm.bmHeight;
                            pBMI->bmiHeader.biPlanes = 1;
                            pBMI->bmiHeader.biBitCount = wBitCount;
                            pBMI->bmiHeader.biCompression = BI_RGB;

                            //Retrieve bitmap bits
                            if(GetDIBits(hDC, hBmp, 0,
                                (UINT)bm.bmHeight,
                                pBMI->bmiColors,
                                pBMI, DIB_RGB_COLORS))
                            {
                                //Pass our memory array to the host process
                                *pImageBufferSize = dwcbFullSize;
```

```
                                        *ppImageBuffer = lpbitmap;

                                        lpbitmap = NULL;

                                        hr = S_OK;
                                    }
                                    else
                                    {
                                        //Failed - free memory
                                        CoTaskMemFree(lpbitmap);
                                        lpbitmap = NULL;
                                    }
                                }

                                                        //Remember to free the DC
                                ReleaseDC(NULL, hDC);
                            }
                        }

                        //Free icon resources
                        DeleteObject(ix.hbmColor);
                        DeleteObject(ix.hbmMask);
                    }
                }
            }

        //Do other work ...
}
```

This should result in the following image:



Custom icon in the "Sign in options" area in the login screen on Windows 10.

> Note that alternatively you can return a failure error code from the `GetBitmapBufferValue` callback to use an older, but documented `GetBitmapValue` callback instead.

Credential::GetBitmapValue

5. Is called to retrieve a bitmap for the UI field of the `CPFT_TILE_IMAGE` type. The UI field is identified by its ID in the `dwFieldID` parameter, that was specified in the `GetFieldDescriptorAt` callback.

   > Even though this callback is invoked in all cases for the UI fields of the `CPFT_TILE_IMAGE` type, the provided bitmap will be used only if the previous `GetBitmapBufferValue` callback returned a failure code, or was not implemented.

   The bitmap must be returned in the `phbmp` parameter. Here's an example:

   C++[Copy]

```
HRESULT STDMETHODCALLTYPE MyCredential::GetBitmapValue(
        __in DWORD dwFieldID,
        __out HBITMAP* phbmp)
{
        HRESULT hr = E_NOTIMPL;

    if(dwFieldID == FIELD_ID_PROVIDER_IMAGE)
    {
        //Let's use the system warning icon for our test
        HICON hIcon = LoadIcon(NULL, IDI_WARNING);
        if(hIcon)
        {
                ICONINFOEX ix = {};
                ix.cbSize = sizeof(ix);
                if(GetIconInfoExW(hIcon, &ix))
                {
                                //Return the color bitmap
                        *phbmp = ix.hbmColor;

                                //Remember to free the mask bitmap
                                DeleteObject(ix.hbmMask);

                hr = S_OK;
                }
        }
    }

        //Do other work ...
}
```

   This will result in a warning icon being displayed as your provider image, similar to what I showed above.

Unfortunately there's no documented way of knowing which background color to use for the image.
For instance, for the provider icons in the "Sign in options" area, Microsoft themselves aren't using any specific system color. They use a prepared bitmap that already comes with a background color. In other words, they don't use transparency.

You can find this bitmap as the `RT_BITMAP` resource in the system credential providers. For instance, for the `PasswordCredentialProvider` provider, the bitmap is in the `credprovs.dll` module in the System32 folder. On my Windows 10 that DLL has only one bitmap resource of the size 72x72 pixels with the ID of `13500`, as I showed here.

Thus, if you want to know the background color, you can load the bitmap that Microsoft used with the `LoadBitmapW` function from the appropriate module. (Say, for `credprovs.dll` there's only one bitmap resource in it.) And then extract the pixel color from it at each corner. This can be a *hack* to give you the background color that Microsoft uses:

C++[Copy]

```
BOOL GetProviderBitmapBkgndColor(__out COLORREF* pclrBkgnd)
{
        BOOL bRes = FALSE;

        COLORREF clrBkgnd = {};

        HMODULE hMod = LoadLibraryExW(L"credprovs.dll",
                NULL,
                LOAD_LIBRARY_AS_IMAGE_RESOURCE |
LOAD_LIBRARY_SEARCH_SYSTEM32);

        if(hMod)
        {
                HBITMAP hBmp = LoadBitmap(hMod, MAKEINTRESOURCE(13500));
                if(hBmp)
                {
                        //Get bitmap size
                        BITMAP bm = {};
                        if(GetObject(hBmp, sizeof(bm), &bm))
                        {
                                HDC hDC = CreateCompatibleDC(NULL);
                                if(hDC)
                                {
                                        HGDIOBJ hOldBmp = SelectObject(hDC,
hBmp);

                                        #define PX_OFFSET 3
                                        if(bm.bmWidth > (PX_OFFSET * 2) &&
                                                bm.bmHeight > (PX_OFFSET *
2))
                                        {
                                                //Sanity check: Get color
from all 4 corners at a certain offset,
                                                //              in case
there's a border
                                                COLORREF clrTL =
GetPixel(hDC, PX_OFFSET, PX_OFFSET);
                                                COLORREF clrTR =
GetPixel(hDC, bm.bmWidth - PX_OFFSET - 1, PX_OFFSET);
                                                COLORREF clrBL =
GetPixel(hDC, PX_OFFSET, bm.bmHeight - PX_OFFSET - 1);
                                                COLORREF clrBR =
GetPixel(hDC, bm.bmWidth - PX_OFFSET - 1, bm.bmHeight - PX_OFFSET - 1);

                                                //All 4 colors must be the
same to use them
                                                if(clrTL == clrTR &&
                                                        clrTR == clrBL &&
                                                        clrBL == clrBR)
                                                {
                                                        //Can use it
                                                        clrBkgnd = clrTL;
```

```
                                                bRes = TRUE;
                                        }
                                }

                                SelectObject(hDC, hOldBmp);
                                DeleteDC(hDC);
                        }
                }

                DeleteObject(hBmp);
        }

        FreeLibrary(hMod);
    }

    if(pclrBkgnd)
        *pclrBkgnd = clrBkgnd;

    return bRes;
}
```

> On my Windows 10 and 11, that background color was `#474747`.

Credential::GetCheckboxValue

6. Is called to retrieve whether the UI field of the `CPFT_CHECKBOX` type is checked or not, as well as to get the label for the checkbox itself. The UI field is identified by its ID in the `dwFieldID` parameter, that was specified in the `GetFieldDescriptorAt` callback.

   Return the checkbox state in the `pbChecked` parameter, and the label text in `ppszLabel`.

   > Keep in mind that the text label must be allocated with the `SHStrDupW` function, similar to what I showed here.

Credential::GetComboBoxValueCount

7. Is called to get the number of items to display in the list for the UI field of the `CPFT_COMBOBOX` type. The UI field is identified by its ID in the `dwFieldID` parameter, that was specified in the `GetFieldDescriptorAt` callback.

   Specify the count of items in the `pcItems` parameter, and the index of the selected item in `pdwSelectedItem`.

Credential::GetComboBoxValueAt

8. Is called to retrieve the text label for each item in the list for the UI field of the `CPFT_COMBOBOX` type. This callback is invoked after `GetComboBoxValueCount`. The UI field is identified by its ID in the `dwFieldID` parameter, that was specified in the `GetFieldDescriptorAt` callback.

   The list item is identified by its index in the `dwItem` parameter. The job of this callback is to return the text for the list item in the `ppszItem` parameter.

   > Make sure to allocate the text label with the `SHStrDupW` function, similar to what I showed here.

Credential::GetSubmitButtonValue

9. Is called to ask which UI field should have the "submit" button attached to.

   > A "submit" button is the button that initiates the user login process.

   The `dwFieldID` parameter will contain the ID of the "submit" button in question.

   Provide the ID of another UI field that you want to attach the "submit" button to. (Field IDs are assigned during the `GetFieldDescriptorAt` callback.)

   > If you do not provide a valid ID, the "submit" button will be attached to the first visible UI field, which will look somewhat ugly. Additionally, Microsoft advises not to attach the "submit" button to static text controls or images.
   > You can create a standalone "submit" button (without attaching it to anything) by following this advice.

Post Initialization

The following callbacks are called among the last ones in the initialization sequence for your credential provider:

Credential::GetUserSid

1. This callback is sent to your `MyCredential` class to retrieve the user SID that is associated with the tile. You can cache the user SID from the earlier `SetUserArray` callback to your `MyProvider` class.

   > You will receive this callback only if you implement the `ICredentialProviderCredential2` interface.

   Provide the requested user SID in the `sid` parameter. Keep in mind that it has to be passed as a string, and allocated with the `SHStrDupW` function, similar to what I showed here.

   > You can convert user SID from the binary form to a string using the `ConvertSidToStringSid` function.

   In case of the "*Other User*" tile, return its user SID as a NULL. Additionally, return S_FALSE from the `GetUserSid` callback.

Credential::Advise

2. Is called to let your `MyCredential` class(-es) to get pointers to the `ICredentialProviderCredentialEvents`, or later interface. It is provided to it in the `pcpce` parameter. Make sure to `AddRef` it and then `Release` it in the matching `UnAdvise` callback.

   The `ICredentialProviderCredentialEvents` interface can be used for multiple purposes, mostly to interact with the host process UI:

   - One of its most often used functions is to request an update for the UI fields in your `MyCredential` class.
     Because unlike traditional UI, you cannot have handles to the UI fields displayed by the host process to update them on-demand. The only way for you to update them is by responding to callbacks. Thus you need a way to request those callbacks from the host process.

- The only viable way to display your own custom UI from a credential provider, is to request the `HWND` handle for the host process window. This can be done via a call to the `OnCreatingWindow` function. After that use returned window handle as a parent for your own window(s).

  > One very important thing to remember, if you decide to show your own custom UI, is that the thread that the `MyCredential::Advise` callback is invoked from, as well as any other callbacks that you may receive in your `MyCredential` and `MyProvider` classes, may not be the UI thread that the `HWND` window returned by the `OnCreatingWindow` call is running in.
  > Thus, if you attempt to create a window from a different thread than the parent window, this may result in some very unexpected behavior and UI artifacts that are hard to diagnose and debug.
  >
  > As a workaround, make sure to subclass the window procedure for the window returned by `OnCreatingWindow`. Then send yourself some custom message to that window (say, using the `WM_NULL` message with a special `wParam` or `lParam`), intercept it in your subclassed `WndProc`, and create your UI from that thread. Also make sure to remove your subclass from the `UnAdvise` callback for your `MyCredential` class.
  >
  > Note that you **cannot** use the `SetWindowSubclass` function for subclassing because you'll be doing it for a window in another thread.
  >
  > Alternatively, you may use the `WH_CALLWNDPROC` window hook to achieve the same effect.

  When designing your custom UI keep in mind that the login screen's credential provider can time-out and be unloaded at any time. If you parent all your UI from the window, returned by `OnCreatingWindow`, and don't display any blocking UI from your own windows, the host process (`LogonUI.exe`) will be able to close your UI gracefully when it decides to unload your provider. Doing it otherwise may lead to unexpected crashes.

You may request pointers to the following newer interfaces, by `QueryInterface`-ing them from `ICredentialProviderCredentialEvents`:

- `ICredentialProviderCredentialEvents2` that can be used to set on-demand some additional UI options like I showed above.
- `ICredentialProviderCredentialEvents3` is an undocumented interface that adds the following method:
  C++[Copy]

  ```
  MIDL_INTERFACE("2D8DEEB8-1322-4973-8DF9-B282F2468290")
  ICredentialProviderCredentialEvents3 : public
  ICredentialProviderCredentialEvents2
  {
          virtual HRESULT STDMETHODCALLTYPE SetFieldBitmapBuffer(__in
  ICredentialProviderCredential * pcpc,
                  __in DWORD fieldID,
                  __in DWORD imageBufferSize,
                  __in BYTE const* pImageBuffer);
  };
  ```

  Without going into too much reverse engineering, you can probably deduce what `SetFieldBitmapBuffer` method does to your `MyCredential` class, remembering the `GetBitmapBufferValue` method.

- `ICredentialProviderCredentialEvents4` is another undocumented interface that adds the following methods:
  C++[Copy]

  ```
  MIDL_INTERFACE("DF50EA86-B7A9-4485-8F04-930A49686E5B")
  ICredentialProviderCredentialEvents4 : public
  ICredentialProviderCredentialEvents3
  {
          virtual HRESULT STDMETHODCALLTYPE RequestSerialization();
          virtual HRESULT STDMETHODCALLTYPE RequestSelection();
  };
  ```

  The `RequestSerialization` method is designed to request serialization, or "submit" button action on demand.

- `ICredentialProviderCredentialEvents5` is one more undocumented interface that adds the following methods:
  C++[Copy]

```
MIDL_INTERFACE("c4a56475-d6f5-43e3-80ae-1aa99833cc05")
ICredentialProviderCredentialEvents5 : public
ICredentialProviderCredentialEvents4
{
        virtual HRESULT STDMETHODCALLTYPE SetTextFieldMaxLength(
                _In_ ICredentialProviderCredential * pcpc, _In_ DWORD
dwFieldID, _In_ DWORD MaxLength) = 0;

virtual HRESULT STDMETHODCALLTYPE SetAccessibilityTextForField(
        _In_ ICredentialProviderCredential *pcpc, _In_ DWORD
dwFieldID, _In_ PCWSTR pcszText) = 0;

virtual HRESULT STDMETHODCALLTYPE SetRawAccessibilityViewForField(
        _In_ ICredentialProviderCredential *pcpc, _In_ DWORD
dwFieldID, _In_ BOOL bRaw) = 0;

virtual HRESULT STDMETHODCALLTYPE RequestWebDialogVisibilityChange(
        _In_ ICredentialProviderCredential *pcpc, _In_ BOOL bVisible)
= 0;
};
```

User Interaction & State Changes

The following callbacks may be invoked as a result of user interactions and state changes in the host process:

Credential::SetSelected

1. This callback is invoked when a tile is selected, either by a user, or automatically when your credential provider is loading.

    > Only a single tile can be selected at a time.

    This callback must set the `pbAutoLogon` parameter to TRUE if your `MyCredential` class wants to initiate an *autologon* for this user. This setting should be coordinated with a previous `GetCredentialCount` callback. The *autologon* will initiate an automatic "submit" button action.

    There's no need to show or hide UI fields in response to this callback. You can set that up in the `GetFieldState` callback using the `CPFS_DISPLAY_IN_SELECTED_TILE` state flag.

    > Note that the fact that a tile has become selected, sadly, does not mean that it is visible. It may be obscured by what is known as a "curtain", or a screen that obscures all tiles and UI fields until the user clicks on it.
    > You can use an undocumented `SetDisplayState` callback to determine when the "curtain" has been lifted.

    Although not specifically documented, your `MyProvider` class (that created the `MyCredential` class, that in turn is associated with the tile) also becomes an active provider when the tile is selected.

Credential::SetDeselected

2. Is called when a tile is deselected by a user. This callback is often followed by the `SetSelected` callback for the tile that was selected.

    > Microsoft advises that a deselected tile should securely clear all sensitive information from memory of your `MyCredential` class, such as passwords, PINs and encryption keys.

    > Microsoft has shown a good example in their official GitHub repo how you can safely scrub from memory all sensitive user data (such as plain-text passwords, PINs, keys, etc.) that could've been collected in your `MyCredential` class while it was selected.

Provider::SetDisplayState

3. Is invoked to notify your `MyProvider` class about some external event.

    > This callback in undocumented. So use it at your own risk.

    > To receive this event, your `MyProvider` class must be derived from the `ICredentialProviderWithDisplayState` interface:
    > C++[Copy]
    >
    > ```
    > MIDL_INTERFACE("A09BCC29-D779-4513-BB59-B4DB5D82D2B6")
    > ICredentialProviderWithDisplayState : public IUnknown
    > {
    >         virtual HRESULT STDMETHODCALLTYPE SetDisplayState(__in
    > CREDENTIAL_PROVIDER_DISPLAY_STATE_FLAGS Flags) = 0;
    > };
    > ```

    > Your provider does not have to be active to receive this notification.

    The `Flags` parameter receives a bitwise combination of flags that denote various types of notifications. I showed their meanings in the comments to the `CREDENTIAL_PROVIDER_DISPLAY_STATE_FLAGS` struct. Some of those flags are not well studied, but some can be explained:

    - `CPDSF_CURTAIN` (or `0x4`) - is set when the "curtain" has been pulled down, or when it hides user tiles and corresponding UI fields.
    - `CPDSF_PROVIDER_VISIBLE` (or `0x80`) - is set when your `MyProvider` class is visible.

        > To detect if a selected tile is visible to the user, check for the `CPDSF_PROVIDER_VISIBLE` flag to be on, and for the `CPDSF_CURTAIN` flag to be off.

    - `CPDSF_SHUTDOWN_STARTED` (or `0x1000`) - is set when the system initiated a reboot or shutdown. As a response to this notification you may hide any custom UI that your credential provider might have shown.

Credential::SetStringValue

4. This callback is invoked every time a UI field that contains text is changed by the user. If it's a field with the `CPFT_EDIT_TEXT` or `CPFT_PASSWORD_TEXT` types, and the user types into it, this callback will be invoked for every single character that the user typed.

   The `dwFieldID` parameter will contain the UI field ID that was changed, and the `psz` parameter will point to the new text as a null-terminated string. The UI field ID comes from what you specified in the `GetFieldDescriptorAt` callback.

   Use some internal variables in your `MyCredential` class to remember what the user types into text and password fields. You will need that later during "submit" button sequence.

   > Be careful not to log what the user types into the credential provider, as it may contain very sensitive information, such as plain-text passwords, PINs, etc.

   Unlike a regular edit box, there's no way to restrict through this callback what the user types into a text-based UI field.

Credential::SetCheckboxValue

5. It is called every time the user toggles the checkmark in the checkbox UI field. Such field has the `CPFT_CHECKBOX` type. The `dwFieldID` parameter will denote the UI field that was changed, and the `bChecked` will contain the new value of the checkmark. The UI field ID comes from what you specified in the `GetFieldDescriptorAt` callback.

   This callback is also invoked if you call `ICredentialProviderCredentialEvents::SetFieldCheckbox` function on the UI field.

   Use some internal variables in your `MyCredential` class to remember user selection. You will need it later during "submit" button sequence.

Credential::SetComboBoxSelectedValue

6. Is called every time the user changes selected item in the combo-box for the UI field with the `CPFT_COMBOBOX` type. The `dwFieldID` parameter will have the ID of the UI field that was changed. The UI field ID comes from what you specified in the `GetFieldDescriptorAt` callback.

   The `dwSelectedItem` parameter will contain the index of the list item in the combo box that was selected.

   This callback is also invoked if you call `ICredentialProviderCredentialEvents::SetFieldComboBoxSelectedItem` function on the UI field.

   Use some internal variables in your `MyCredential` class to remember user selection. You will need it later during "submit" button sequence.

Credential::CommandLinkClicked

7. Is called when a UI field with the `CPFT_COMMAND_LINK` type was clicked by the user. The `dwFieldID` parameter will denote the UI field that was clicked. The UI field ID comes from what you specified in the `GetFieldDescriptorAt` callback.

   Use some internal variables in your `MyCredential` class to remember user selection. You will need it later during "submit" button sequence.

Submit Button Sequence

When the user clicks "submit" button, or if such action is initiated automatically, this will involve the following sequence of callbacks:

Credential::Connect

- Is invoked immediately after "submit" button click, either manually by the user, or as a result of *autologon*.

> This method is invoked only if you implemented the `IConnectableCredentialProviderCredential` interface in your `MyCredential` class.
> The original purpose of this callback is to allow implementation of a slower pre-logon authentication before performing an actual authentication of a Windows user. An example of such requirement could be the 802.1x network that requires authentication before a Windows client computer can connect to a domain controller to authenticate Windows users.
>
> So in a sense, such Pre-Logon-Access Provider (or PLAP), also known as the SSO provider, acts as the first authenticator in the chain of two authentications needed to login a user.
>
> And since the network authentication can be a slow process, the `Connect` method was designed with two features: 1) The "Cancel" button to abort it, and 2) the status message to inform the user of the progress.

The benefit of this callback is that you can reuse its "cancel" button and status text features for your own purposes.

For instance, if you want to code a two-step login process, where you require a regular Windows user authentication, followed by an authentication using that user's smartphone, you can use the `Connect` method to implement it, as such:

- First we check the user credentials in the `Connect` callback and ensure that such user can login. In this case we will break Microsoft mantra and perform user authentication in our credential provider, say, by using the `LsaLogonUser` function.
- If the user login fails, we remember such state and return that error from the `GetSerialization` callback, which will abort the login.

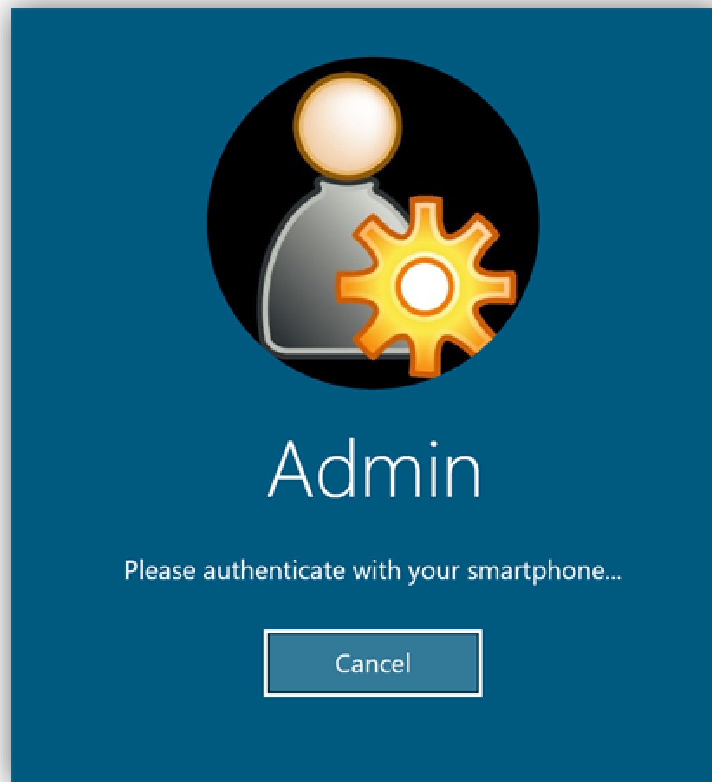
- But if the user login was successful, we serialize it (as we would do in the `GetSerialization` callback, and remember it for that callback at a later iteration.) After that we perform our secondary authentication with the user's smartphone. This can be done using something similar to the following pseudo-code:

  C++[Copy]

```
HRESULT STDMETHODCALLTYPE MyCredential::Connect(_In_
IQueryContinueWithStatus* pqcws)
{
        HRESULT hr;

        //Checked user credentials with LsaLogonUser and received a valid
user token.
        //Thus we know that the user login info is valid.
        //...

        //After that, performed needed serialization of the user login info
and remembered it in some class variable...
        //...

        //Now ask the user to perform secondary authentication
    DWORD dwmsTimeout = 15 * 1000;
//15 sec timeout

        //Class variable that stores the result of the secondary
authentication
    _hrConnect = E_FAIL;

        //Display message for the user
    pqcws->SetStatusMessage(L"Please authenticate with your
smartphone...");

        //Go into a waiting loop
    for(DWORD dwmsIniTicks = GetTickCount();;)
    {
        //Ask our web server when the smartphone authentication is received
        HRESULT hrServerResult;
        if(PollRemoteServerAuthForResult(&hrServerResult))
        {
            //Received a result from our web server
            pqcws->SetStatusMessage(L"Smartphone authentication finished");

                        _hrConnect = SUCCEEDED(hrServerResult) ?
hrServerResult : E_UNEXPECTED;
            hr = S_OK;

            break;
        }

        //Simulate a short delay
        Sleep(500);

        //See if user wants to cancel
        hr = pqcws->QueryContinue();
        if(hr == E_ABORT)
        {
            //User aborted
            pqcws->SetStatusMessage(L"User aborted");
```

```
                _hrConnect = hr;

                break;
            }
            else if(hr != S_OK)
            {
                //Some other error
                pqcws->SetStatusMessage(L"Error authenticating");

                _hrConnect = FAILED(hr) ? hr : E_UNEXPECTED;

                break;
            }

            //Check for timeout
            if(GetTickCount() - dwmsIniTicks >= dwmsTimeout)
            {
                //Timed out
                pqcws->SetStatusMessage(L"Timed out");

                hr = _hrConnect = RPC_E_TIMEOUT;

                break;
            }
        }

            //Do other work ...
    }
```

After the user clicks "submit" the code snippet above will display the following message:



Custom "*Please authenticate with your smartphone*" message in the login screen on Windows 10.

- If our secondary authentication fails (for whatever reason) we remember the error code and check it again in the `GetSerialization` callback and return failure there, which will abort the login.

- But if our secondary authentication succeeds, we also pass it to our `GetSerialization` callback, which retrieves our previously saved serialized user login data, and returns it to the host process, which proceeds with the user login. This could be visualized with the following pseudo-code:

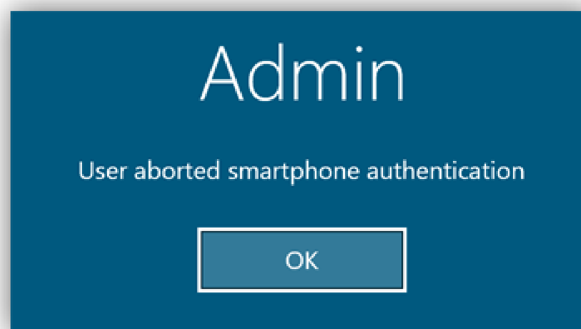
  C++[Copy]

```
HRESULT STDMETHODCALLTYPE MyCredential::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)
{
    HRESULT hr = E_UNEXPECTED;

    //Did we succeed during the "Connect" stage?
    if(SUCCEEDED(_hrConnect))
    {
                //Copy our previously serialized user login credentials in
the Connect() callback
                //into the 'pcpcs' parameter using the CoTaskMemAlloc()
function.
                //...

                //If success so far
                *pcpgsr = CPGSR_RETURN_CREDENTIAL_FINISHED;
                *ppwszOptionalStatusText = NULL;
                *pcpsiOptionalStatusIcon = CPSI_NONE;

                hr = S_OK;
        }
        else
        {
        //Failed during "Connect" stage
        *pcpgsr = CPGSR_NO_CREDENTIAL_FINISHED;
        *pcpcs = {};

                //Pick the error message to display to the user
        std::wstring strError;
        if(_hrConnect == E_ABORT)
        {
            strError = L"User aborted smartphone authentication";
        }
        else if(_hrConnect == RPC_E_TIMEOUT)
        {
            strError = L"Timed out waiting for the user to authenticate
with the smartphone";
        }
        else
        {
            //Some other error
            WCHAR buff[128] = {};
            StringCchPrintf(buff, _countof(buff), L"Error (0x%08X)
authenticating with the user's smartphone", _hrConnect);
            strError = buff;
        }

        SHStrDupW(strError.c_str(), ppwszOptionalStatusText);
        *pcpsiOptionalStatusIcon = _hrConnect != E_ABORT ? CPSI_ERROR :
```

```
CPSI_WARNING;

        //Pass success
        hr = S_OK;
        }

        return hr;
}
```

> Note that in this case we will be technically calling `LsaLogonUser` function twice, but that is a small price to pay for an added feature of the MFA login.

If the user clicks the "cancel" button to abort secondary authentication, the code above will display the following message:

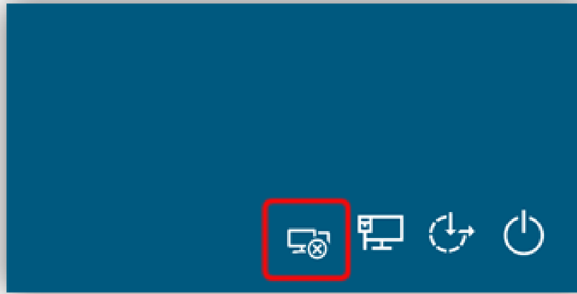


Custom "*User aborted smartphone authentication*" message in the login screen on Windows 10.

Note that the `HRESULT` status code returned from the `Connect` callback tells the host process whether or not to display the "Disconnect" button, that I will describe next.

Credential::Disconnect

- This callback complements the `Connect` method. It is called when the user clicks the "Disconnect" button in the credential provider's UI:



"Disconnect" button in the bottom right corner of the login screen on Windows 10.

Such button will appear after `Connect` method returns a successful result code, such as S_OK. It will not be displayed if `Connect` returns an error.

> You can use the above-mentioned feature of the "Disconnect" button not to display it.

Note that the "Disconnect" button will disappear from the UI after user clicks it.

The intended purpose of this callback is to disconnect any pre-logon network connection that was established in the `Connect` method earlier.

> This method is invoked only if you implemented the `IConnectableCredentialProviderCredential` interface in your `MyCredential` class.

In case you are reusing the `Connect` method for your own purposes, not related to pre-logon network authentication, you will still need to implement this method, but you can safely ignore it by simply returning S_OK.

Credential::GetSerialization

- In case `Connect` method is implemented in your `MyCredential` class, this callback is invoked right after it. Otherwise, `GetSerialization` will be the first callback to be invoked after the user clicks "submit" button, or if such action is initiated after *autologon*.

  The job of this method is to serialize user-provided login credentials from the UI fields in the selected tile, and to pass the result to the host process.

  > *Serialization*, in this sense, is the process of converting user login credentials into a binary byte array. This way such data can be passed between processes.

  > The way credential provider's UI is structured is somewhat different from a traditional UI programming design. In a traditional app, you would ask a UI control to give you its text or some other selection in it. In a credential provider though there's no such mechanism.
  > To receive what the user had specified in the UI fields you will need to save it in the internal variables in your `MyCredential` class while the user interacts with the UI fields. You can do so by intercepting callbacks such as `SetStringValue`, `SetCheckboxValue`, `SetComboBoxSelectedValue`, `CommandLinkClicked`, etc.

  I already showed above how you can incorporate the `Connect` method for the two-step MFA authentication. For now I'll assume only single-step logins.

  There are several ways how you can process this callback. One, you can evaluate user-provided credentials for immediate errors. Say, if the user did not provide their name for the "*Other User*" tile, or if the PIN for your PIN-provider is empty, etc. In that case you can immediately return an error without doing any serialization:

  C++[Copy]

```
HRESULT STDMETHODCALLTYPE MyCredential::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)
{
        HRESULT hr = E_UNEXPECTED;

        //Quickly evaluate user-provided credentials
        //...

        if(bBadLoginCredentials)
        {
                //Fail without doing serialization
                *pcpgsr = CPGSR_NO_CREDENTIAL_FINISHED;
                *pcpcs = {};
                *pcpsiOptionalStatusIcon = CPSI_ERROR;

        SHStrDupW(L"Please provide valid login credentials",
                        ppwszOptionalStatusText);

                return S_OK;
        }

        //Do other work ...
}
```

But for any other case you will need to serialize the login data. Microsoft have shown already an example how to do it for a password-provider in their official code sample. So I won't be repeating it.

> Make sure to provide a copy of the serialized login data in the `pcpcs` parameter using the `CoTaskMemAlloc` function.

When the serialized user credentials are ready, and there was no errors in the process, you can return success as follows:

C++[Copy]

```
HRESULT STDMETHODCALLTYPE MyCredential::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)
{

        //After serialized data was copied into 'pcpcs' with CoTaskMemAlloc
        //...

        //Let the host process commence with the user login
        *pcpgsr = CPGSR_RETURN_CREDENTIAL_FINISHED;
        *ppwszOptionalStatusText = NULL;
        *pcpsiOptionalStatusIcon = CPSI_NONE;

        return S_OK;
}
```

> Note that returning success (or S_OK) alone from the `GetSerialization` callback will not grant an automatic login. For that you need to provide valid user login data in the serialized byte array. The decision to proceed with the user login will be made by the host process, based on the user login data that you give it in the `GetSerialization` callback.

Credential::ReportResult

- This callback is invoked after `GetSerialization` with the results of the user login. The `ntsStatus` and `ntsSubstatus` parameters will contain the status codes after calling the `LsaLogonUser` function on the serialized login data that was collected in the `GetSerialization` callback.

  > For details why you receive two status error codes for the login, refer to the `LsaLogonUser` function documentation.

  To check if the user login succeeded, it's enough to check the `ntsStatus` parameter to be STATUS_SUCCESS, or 0.

  If the login fails, make sure to specify the failure reason to be displayed to the user in the `ppszOptionalStatusText` parameter. You can copy it there using the `SHStrDupW` function, similarly to what I showed here.

  In case of a successful login, there's no need to do anything else. Your credential provider will unload shortly thereafter.

  Microsoft showed how to process this callback in their GitHub sample.

  > Keep in mind that if the `ntsStatus` parameter is set to `STATUS_SUCCESS`, once you return from this callback, your credential provider will begin to unload and the user login will succeed. The `ReportResult` callback serves as a notification only. It does not allow for your credential provider to stop the login.
  > To prevent user login use the `GetSerialization` callback instead.

  Do not show any custom UI from this callback as the desktop may be changing from a secure desktop to the one for the logged in user. Showing any UI at this late stage may prevent the user from properly interacting with it due to such transition.

Uninitialization

The following callbacks are invoked when a credential or provider goes out of scope, or when their respective classes are unloaded:

> Note that your credential provider may be unloaded at any time, and not only after a successful user login. One such reason could be an elapsed timeout.

Credential::UnAdvise

- This callback is invoked before your `MyCredential` class is torn down. This is a good place to release all references to other interfaces that your `MyCredential` class was holding and to free other resources.

  Alternatively return E_NOTIMPL if you do not care about this callback.

  > Make sure that your code does not display any UI or blocks execution during this callback.

Provider::UnAdvise

- This callback is invoked for your `MyProvider` class before it is torn down, but after all instances of your `MyCredential` class for it are deleted.

  > Keep in mind that this callback may be called out-of-order, or without invoking a previous `Advise` method for your `MyProvider` class in case you block loading of the said provider class in the `Filter` callback.

  Alternatively return E_NOTIMPL if you do not care about this callback.

  > Make sure that your code does not display any UI or blocks execution during this callback.

At this stage your credential provider will be invoking destructors for global classes and will be unloaded at any moment. Do not do any heavy processing at this stage.

Wrapping An Existing Credential Provider

The technique of *wrapping* a credential provider, usually the system one, is available because of the structure and operation of said credential providers that I show-cased above.

Wrapping is a handy technique to enhance an existing credential provider, similar to what subclassing does for Win32 windows.

> Note that not all system credential providers can be *wrapped* easily. To be able to be *wrapped* a credential provider must *play* by the documented rules that I described in this post.
> The following system credential providers can be *wrapped*:
> `PasswordCredentialProvider`, `NgcPinProvider`, `SmartcardCredentialProvider`, `WLIDCredentialProvider`, `OnexCredentialProvider` (with additional adjustments.)
>
> An example of the credential provider that *does not* play by the rules is `PicturePasswordLogonProvider`.

Wrapping involves retrieving a pointer to the interface being *wrapped* and by invoking its callbacks from the same methods in your own interface. With that technique you will be able to alter the *wrapped* interface's input or output parameters to your liking, and thus modify the behavior of the credential provider that you are *wrapping*.

Review the following pseudo-code that alters the "submit" button behavior of the *wrapped* credential provider:

C++[Copy]

```
HRESULT STDMETHODCALLTYPE MyCredential::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)
{
        //Let the wrapped provider do all the work
        assert(_pWrappedCredential);
        hr = _pWrappedCredential->GetSerialization(pcpgsr, pcpcs,
ppwszOptionalStatusText, pcpsiOptionalStatusIcon);

        if(SUCCEEDED(hr))
        {
                //Quick-and-dirty way to prevent users with short passwords from
logging in
                WCHAR buffUsr[256];
                WCHAR buffPwd[256];
                DWORD dwchUser = _countof(buffUsr);
                DWORD dwchPwd = _countof(buffPwd);
                if(CredUnPackAuthenticationBufferW(CRED_PACK_PROTECTED_CREDENTIALS,
                        pcpcs->rgbSerialization,
                        pcpcs->cbSerialization,
                        buffUsr, &dwchUser,
                        NULL, NULL,
                        buffPwd, &dwchPwd))
                {
                        //Securely clear the password from memory
                        SecureZeroMemory(buffPwd, sizeof(buffPwd));

                        //Limit is 3 chars
                        if(dwchPwd - 1 <= 3)
                        {
                                //Block this user
                                *pcpgsr = CPGSR_NO_CREDENTIAL_FINISHED;
                                *pcpcs = {};
                                *pcpsiOptionalStatusIcon = CPSI_ERROR;

                                SHStrDupW(L"Your password is too short",
                                        ppwszOptionalStatusText);

                                //Reset user password field
                                _pcpce-
>SetFieldString(static_cast<ICredentialProviderCredential*>(this),
                                        PASSWORD_FIELD_ID,
                                        L"");

                                return S_OK;
                        }
                }
        }
```

```
        return hr;
}
```

In this case the `_pWrappedCredential` variable holds the pointer to the original wrapped credential class.

> Microsoft uses this technique as well. Internally the `OnexCredentialProvider` class is implemented as a *wrapper* for the `PasswordCredentialProvider` class.

> Keep in mind that credential provider *wrapping* carries an inherent risk of breaking things if the underlying *wrapped* interfaces are changed without notice in future versions of the OS.

Credential provider *wrapping* is a whole topic of its own. Thus, with enough interest I may write a separate blog post on that subject.

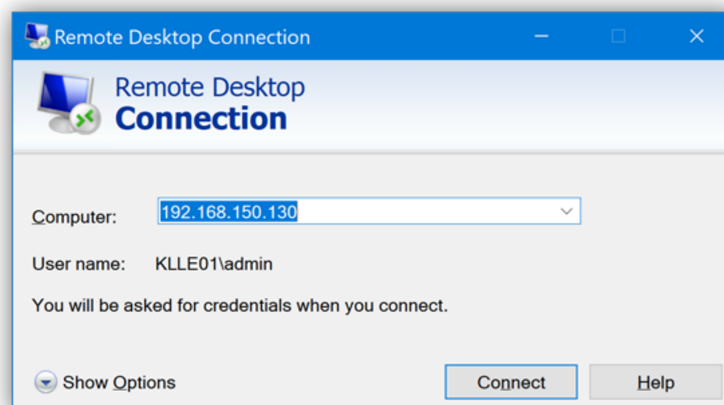
Specifics Of A Remote Desktop Connection

A remote desktop connection (or RDP) is somewhat different from a regular user login. It deserves a blog post of it's own, but until I write it, let's review it briefly here.

When a remote desktop connection is initiated you are technically dealing with two systems: the *client* computer that is making an RDP connection, and the *remote* computer that the user is connecting to. I'll use this nomenclature further down the description.

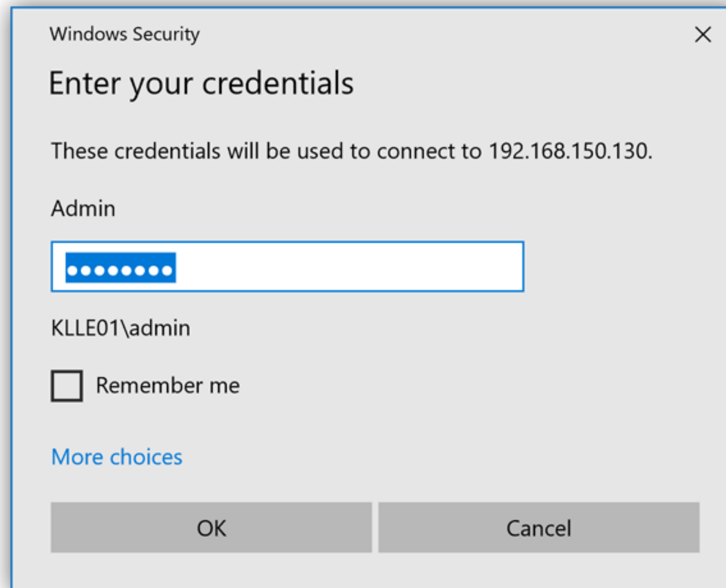
Let's review the sequence of events that take place during an RDP connection:

1. A user initiates an RDP connection on the *client* computer by providing the *remote* computer's IP:



RDP connection from the *client* computer side running on Windows 10.

2. If the *remote* computer is available, the RDP software running on the *client* computer will request the user credentials for the *remote* computer:



   RDP connection user account from the *client* computer side running on Windows 10.

   > Note that only if the user credentials match the ones on the *remote* computer the RDP connection will proceed to the next step. Or, in other words, the credential provider on the *remote* computer will not receive an incorrect set of login credentials from the *client* computer.

3. There could be some negotiations at this stage if the *remote* computer has a different user logged in at the time, than the one requested in the RDP connection. So provided that said negotiations were resolved in favor of a remote user.
4. When the RDP connection is established, the *remote* computer will create two secure desktop sessions:
   - One session for the RDP connection. This is the desktop that will be displayed on the *client* computer that is connecting via RDP connection.
   - Another session for the *remote* computer. This is the desktop that will be displayed on the *remote* computer's physical screen, usually a credential provider itself.

     > The purpose of this secure desktop session is to obscure the user desktop that is now transmitted via RDP to the *client* computer.

   Note that both sessions will have its own instance of the credential provider running in the *remote* computer. Thus, in a sense, there will be two instances of the credential provider running at the same time.

5. When the RDP session is disconnected, its desktop session will be closed, and the *remote* computer will have only the second session with the credential provider running.
6. A user on the *remote* computer can now log in to the user desktop using its local credential provider.

Finally, if you want to retrieve the RDP session details of a connecting *client* computer from the *remote* computer, you can use the following technique:

C++[Copy]

```
//Get additional details about connecting 'client' computer during RDP session
LPTSTR ppBuffer = NULL;
DWORD dwcbBytesReturned = 0;
WTS_CLIENT_ADDRESS* pWTSCA = NULL;

if(WTSQuerySessionInformation(WTS_CURRENT_SERVER_HANDLE,
        WTS_CURRENT_SESSION,
        WTSClientAddress,
        &ppBuffer,
        &dwcbBytesReturned))
{
        //Sanity check
        if(dwcbBytesReturned >= sizeof(WTS_CLIENT_ADDRESS))
        {
                pWTSCA = (WTS_CLIENT_ADDRESS*)ppBuffer;

                // Address family can be only:
                // AF_UNSPEC  = 0 (unspecified)
                // AF_INET    = 2 (internetwork: UDP, TCP, etc.)
                // AF_IPX     = AF_NS = 6 (IPX protocols: IPX, SPX, etc.)
                // AF_NETBIOS = 17 (NetBios-style addresses)
                //
                int nAddrFam = pWTSCA->AddressFamily;

                // The client local IP address is located in bytes 2, 3, 4, and 5.
                // The other bytes are not used.
                // If AddressFamily returns AF_UNSPEC, the first byte in Address
                // is initialized to zero.
                //
                wprintf(L"AddressFamily: %d, RemoteIP: %u.%u.%u.%u\n",
                        nAddrFam,
                        pWTSCA->Address[2],
                        pWTSCA->Address[3],
                        pWTSCA->Address[4],
                        pWTSCA->Address[5]);
        }
}

if(ppBuffer)
{
        WTSFreeMemory(ppBuffer);
        ppBuffer = NULL;
}
```

To determine if the RDP connection is on from the *remote* computer's side, you can use this approach.

Conclusion

This is probably the longest blog post that I've written to date. But I had to keep it all in one place to allow ease of search and reference. This post though is far from being a complete manual on writing a credential provider. But at least it has some useful notes that may come handy to developers.

If you feel like you have something to add to my overview, please leave a comment below.

And, if your organization needs me to code a credential provider for you, or if you are needing help with your existing credential provider project, don't hesitate to send me a private message below. We provide consulting and development services for businesses.