New information has been added to this article since publication.
Refer to the Editor's Update below.

Desktop Security

Create Custom Login Experiences With Credential Providers For Windows Vista

Dan Griffin

This article discusses:

- The new Credential Provider architecture
- Why GINA-based authentication was dropped
- Multi-factor authentication
- Developing and debugging a credential provider

This article uses the following technologies:
Windows Vista, C++

Code download available at: CredentialProviders2007_01.exe (241KB)

Contents

W
indows Vista offers developers many new opportunities for integrating with the platform. The new Credential Provider model represents one of the most dramatic changes, making it much easier to implement new user authentication scenarios that are supported by the OS. This has replaced the GINA (Graphical Identification and Authentication) model-a model that, put bluntly, is known for being difficult for developers to understand and implement as well as being expensive for Microsoft to support.

So why is a change to the Windows® logon plug-in interface so exciting? The logon screen is the first thing users see when they turn on the computer. Now that the experience is driven by credential providers, it's much easier to customize the logon experience and integrate the authentication methods that best meet an organization's needs. Simply put, credential providers offer an easier way to develop and implement better, more robust security.

Comparing the Old and the New

I don't want to go into too much detail about the GINA-based logon architecture. However, it is worth taking a little time to compare the two architectures to help you better understand the new architecture and the changes it introduces.

In a pre-Windows Vista™ environment, every session has an instance of winlogon, which is responsible for driving the interactive logon sequence for that session. (Figure 1 illustrates the old logon architecture under Windows XP and Windows Server® 2003.) On a freshly booted system, an interactive logon at the console is always performed in session zero. Session zero hosts system services and other critical processes, including the Local Security Authority process. (In other words, there are lots of processes running in session zero that Figure 1 doesn't show.)

The registered GINA on the machine is loaded into the winlogon process space. (A configuration known as "GINA chaining" is also possible, but such a complex configuration is difficult to test and support.) Finally, GINA makes calls to LogonUser and related authentication APIs.

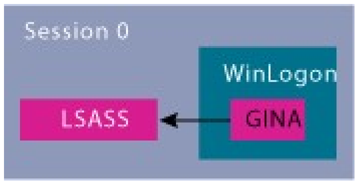

Figure 1 GINA Logon Architecture

In Windows Vista, session zero is never used for interactive logon (see Figure 2). This is good for security- there is now a session boundary that separates all per-machine processes from per-user processes. Additionally, the kernel Global namespace is now more tightly controlled, since objects created by user applications are kept out of it by default.

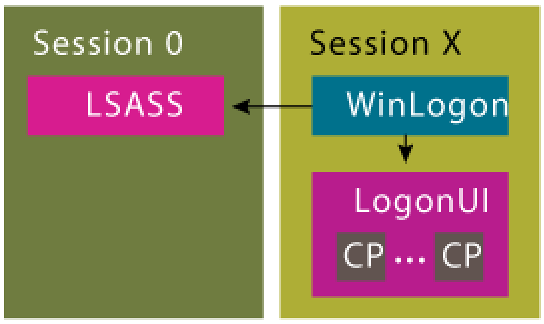

Figure 2 New Logon Architecture

There's still an instance of winlogon in every session other than session zero. The figure illustrates that several credential providers have been registered on the system and loaded by the new LogonUI process.

There has also been an important change regarding which component renders the graphical aspect of the logon. Previously, this was handled by GINA and thus a third-party component could have been doing the rendering. In the new architecture, LogonUI, which is a built-in piece of the operating system, is responsible for this.

So how is per-provider user prompting behavior achieved in the new model? The Credential Provider architecture requires each provider to enumerate its UI elements. For example, in a given scenario, a provider might indicate to LogonUI that it requires two edit boxes, two captions, a checkbox, and a bitmap. In turn, LogonUI renders those controls on behalf of the credential provider. This goes a long way toward achieving the goal discussed previously-a consistent look and approach that supports a broad set of evolving authentication scenarios.

The Microsoft development team responsible for Credential Provider thought that external developers would be more comfortable with a COM-based plug-in model. However, early in the Windows Vista development cycle, the first internal design for the new interface was based (like GINA) purely on LoadLibrary and function pointers. The lessons learned from that first attempt were rolled into the subsequent COM-based redesign, and the resulting interface is cleaner and easier to use. Now let's turn to the sample code to help guide us as we drill into the credprov interface.

A Hybrid Credential Provider

The timing of this new plug-in model couldn't be better (well, OK, perhaps it's overdue). Now, developers can more easily meet the demand for multi-factor authentication scenarios while providing an experience consistent with what Microsoft provides out of the box.

That said, the new interface is rather abstract. An equally abstract description of it would be really boring! A more interesting way to get to know it is by walking through the design, development, and testing of a new credential provider. Also, this will better complement the documentation that Microsoft has already provided-see the "Additional Resources" sidebar for pointers.

I've created a sample, the "hybrid credential provider," which demonstrates some neat new features. The hybrid credential provider allows a user name, password, and domain name to be stored on a smart card. Upon insertion of a card, the user is automatically logged on. (The sample code can be downloaded from the *MSDN®Magazine* Web site.) I didn't write the code from scratch. Instead, I combined code from three sources:

- The sample password-based Credential Provider available in the Microsoft® Windows SDK.
- The old PropCert sample, also from the SDK. At its core is a Win32® thread that reads certificate-based smart card credentials.
- The sample code that is included with my article in the November 2006 issue of *MSDN Magazine*. The article discussed interfacing with the Windows smart card subsystem via managed code.

Further clarification is needed regarding the sample code provided with my November 2006 article. The credential provider architecture and its host support only native code. While my first article focused on managed code, it included a native helper DLL that conveniently exposes the new Smart Card Module interface. The hybrid credential provider is built on that helper DLL. If you want the full source code for that DLL, it's also available via the download accompanying the November 2006 article.

In summary, a high percentage of the hybrid credential provider code base is not new. The net result is minimal time spent testing and debugging. In fact, the core debugging phase took less than a day, which speaks to the ease-of-use of the new interface.

Let's discuss in more detail what I set out to accomplish with the sample credential provider.

The Requirements

When planning the hybrid credential provider, I had these requirements in mind:

- Make it smart card-based
- Maximize code reuse
- Minimize additional config and infrastructure requirements

This led me to my hybrid approach, in other words password (for security) plus smart card (for convenience). Since the hybrid provider concept is user name and password-based, I started with the stripped-down sample password provider from the Platform SDK. Then I added the PropCert sample from the SDK; this includes the logic to enumerate smart card readers, cards, and digital certificates. I figured all I had to do was replace the certificate-based logic in PropCert with some new code for reading my own credential data, and then I'd simply connect the two samples to each other!

Since we'll be reading password logon information from a smart card, this implies another requirement: a tool to initialize a smart card with that credential. I'll save the discussion of the initialization tool for the end.

With those requirements in mind, let's look at the design of the credential provider architecture and how it drove the design of my sample code.

The Design

Let's first discuss the design of the credential provider architecture, from the perspective of a credential provider at run time.

Although I have yet to discuss the hybrid sample in detail, I'm using it as the basis for analyzing the new credential provider architecture in action. To facilitate this discussion, my sample code includes debug tracing. The tracing consists of a call to OutputDebugString from each implemented credential provider routine. In those trace calls, I use two abbreviations. Calls to the new ICredentialProvider interface (excerpted in Figure 3) are prefaced with "Provider::". Calls to the ICredentialProviderCredential interface (see Figure 4) are prefaced with "Credential::". Note that all credential provider-related interfaces are defined in the new public header, credentialprovider.h.

With that in mind, see Figure 5 for the list of the debugging events that occur during a sample scenario (I'll describe most of the events in detail). The scenario for generating the call sequence is simple. Start with a Windows Vista workstation joined to a domain. Configure a smart card with your user name, password, and domain name. Insert the smart card in a reader attached to the machine. Then reboot the system.

First, the console session LogonUI process is started by winlogon. Upon creation, LogonUI enumerates all of the credential providers registered under HKLM\Software\Microsoft\Windows\CurrentVersion\Authentication\Credential Providers. Each provider DLL is loaded and receives a Provider::CreateInstance call. For the hybrid credential provider, this results in the creation of a CHybridProvider. (See steps 1 through 4 in Figure 5.)

The user now sees the logon screen. Assuming the user presses Ctrl+Alt+Delete, and each provider receives Provider::SetUsageScenario CPUS_LOGON notification. This indicates to the provider that the user wants to perform an interactive logon. Now, the hybrid credential provider attempts to read a credential from any inserted smart card. If it finds one, a CHybridCredential is instantiated and associated with the current CHybridProvider. There will then be a call to Credential::Initialize. (See steps 5 through 7 in Figure 5.)

LogonUI then calls Provider::Advise for each loaded provider. The purpose of Advise is to give the providers a mechanism for notifying LogonUI asynchronously of any desired change to the visible UI elements (of which there are none yet). The built-in smart card provider gives a good example of how this is used. Any time after initialization, card insertion can increase the number of available credentials and card removal can decrease it. When that happens, LogonUI is notified via this mechanism:

```
ICredentialProviderEvents : public IUnknown
{
    HRESULT STDMETHODCALLTYPE CredentialsChanged(
       /* [in] */ UINT_PTR upAdviseContext);
};
```

For the sake of simplicity, the hybrid credential provider doesn't dynamically handle card insertion and removal. Therefore, it doesn't keep track of the ICredentialProviderEvents interface passed to it via Advise.

The next interface call by LogonUI is to Provider::GetCredentialCount, which is step 9 in Figure 5. In the case that a hybrid credential was created (due to an inserted smart card), the hybrid credential provider will take several actions. It will first set the GetCredentialCount *pdwCount output parameter to one. This refers to the number of credential tiles that the provider wants to enumerate. (The hybrid credential provider can only handle one.) When you first install Windows Vista and join a domain, you can infer what pdwCount value the Microsoft password credential provider returned to LogonUI based on the number of tiles rendered.

The hybrid then sets the GetCredentialCount *pdwDefault output parameter to zero. This value refers to a zero-based index into an array of credentials that each provider is assumed to maintain. The actual implementation of how a provider tracks its credentials is up to the implementer, as long as the indices are maintained for the lifetime of a given set of credential objects.

It's entirely possible that multiple providers will enumerate a default credential. For example, in the current scenario, you can expect the built-in password credential provider to enumerate a default credential of its own. How does LogonUI prompt the user to select from multiple default and non-default credentials without causing confusion? In general, the user is shown a tile for each credential, with focus set to the tile that represents the default credential. In the presence of multiple defaults, the true default is selected through a series of precedence rules as each of the default credentials is enumerated. For each credential, if there's already a default without auto-logon, and if this credential will do an auto-logon, this credential becomes the default. If this credential is from the last-logged-on (LLO) provider, and if there isn't already a default with auto-logon, this credential becomes the default. And finally, if there's no default yet, this credential becomes the default. All that said, the auto-logon semantics of my hybrid credential provider make this discussion moot. As long as the enumerated hybrid credential contains valid logon information, the user never sees any tiles. I'll explain this in a bit.

I mentioned the last-logged-on provider with regards to precedence rules, but it should be pointed out that the meaning of LLO changes based on whether the user is logging in or whether it's a post-logon scenario, such as a desktop lock or a password change. At logon, the LLO provider is the last provider that was used for the last logon to the console. Post-logon, the LLO provider is the one that was used for logon to that session only. The idea is that if you always log on with your smart card, your smart card credential provider default tile will win across reboots. But if you lose your smart card and log on with your password, the password credential provider's tile will win for that session when you unlock.

The hybrid credential provider always sets the *pbAutoLogonWithDefault output parameter to TRUE. This serves notice to LogonUI that it should immediately query this provider's default credential for logon information and that there's no need to prompt the user first. Note that the built-in password credential provider has the same capability via the optional password auto-logon information that can be stored in the registry. In fact, this is the default behavior if Windows Vista detects that there's only one user on the machine with no password yet. If multiple credential providers set *pbAutoLogonWithDefault to TRUE, the behavior of LogonUI is undefined.

After GetCredentialCount, LogonUI calls Provider::GetCredentialAt. For the hybrid credential provider, this routine is called at most once, reflecting the maximum credential count for this provider. In response, the provider returns an ICredentialProviderCredential pointer for the credential instance that corresponds to the requested index.

Next, LogonUI calls Provider::GetFieldDescriptorCount, via which the provider returns the maximum number of UI elements that may be found in its credentials. For example, the sample password credential provider has five fields: a bitmap, a username input field, a password input field, a submit button, and a domain name input field. You can see that these same elements are preserved in the hybrid credential provider, even though they're never actually rendered. This completes step 11 in Figure 5.

LogonUI then calls Provider::GetFieldDescriptorAt one time for each UI element in order to retrieve its type. For example, in response to the call corresponding to the index of the bitmap, the sample returns the CREDENTIAL_PROVIDER_FIELD_TYPE CPFT_TILE_IMAGE. One feature not used in the hybrid credential provider is writeable versus read-only text fields. If the hybrid credential provider were modified to prompt the user for a smart card PIN, that would be accomplished with CPFT_PASSWORD_TEXT. The user name read from the smart card can be displayed to provide some context for that prompt. But, technically, the user name should be considered read-only since it's bound to the password also stored on the card. Therefore, the CPFT_LARGE_TEXT field type (as opposed to CPFT_EDIT_TEXT) might be used. (For the full list of options, see credentialprovider.h.)

Following the enumeration of field descriptors, LogonUI makes a sequence of calls into the credential provider based on the type of each credential field. For the CPFT_TILE_IMAGE field type, for example, LogonUI follows up with a call to Credential::GetBitmapValue. For text values such as the CPFT_LARGE_TEXT used for the user name edit box, there are subsequent calls to Credential::GetStringValue and Credential::GetFieldState.

Since all of the required logon information (user name, password, and domain name) for my hybrid credential provider has already been read from the smart card, the strings corresponding to each text field are available at this time; and are returned via the ppwz output parameter of GetStringValue. Other providers are likely to return a NULL string value in response to GetStringValue at this point, since the user hasn't had a chance to type in anything yet. Note this one potentially confusing point: the name of the text field is retrieved via GetFieldDescriptorAt while the current text value in the field is retrieved via GetStringValue. (The name or label of the field will be displayed as the cue text in an empty edit control.)

After the various UI elements have been fully described, LogonUI calls Credential::Advise. (See step 26 in Figure 5.) This serves a similar purpose to the Provider::Advise interface that was called earlier; each credential can asynchronously notify LogonUI of relevant changes affecting the state of its UI elements. As an example, the sample password credential provider uses this mechanism when one of its credential tiles is deselected. In that case, ICredentialProviderCredentialEvents SetFieldString (see Figure 6) is used by the credential object to clear out the password field. This is analogous to what happens when you type in only part of your password at the logon screen in Windows XP and then pause. Eventually, the logon dialog times out and the text is cleared.

In terms of completing a user authentication, the next call is the most interesting. As a result of the *pbAutoLogonWithDefault parameter of GetCredentialCount having been set to TRUE, LogonUI knows that the default credential should already contain sufficient data for authenticating the user (even though no UI elements have yet been rendered and, in turn, no user input gathered). In this case, the Credential::GetSerialization routine is called to retrieve the user name, password, and optional domain name. The credential provider prepares the return value for that routine by marshaling the three items into the format expected by Kerberos. Once the serialized credential has been prepared, the credential provider informs LogonUI via the CREDENTIAL_PROVIDER_GET_SERIALIZATION_RESPONSE type output parameter that a complete credential is being returned. The value CPGSR_RETURN_CREDENTIAL_FINISHED makes this distinction. Again, see credentialprovider.h, as well as the implementation of GetSerialization in the sample code. This completes step 27 in Figure 5.

After GetSerialization, LogonUI passes the marshaled credential to winlogon, which in turn passes it to the Local Security Authority (LSA) by calling LogonUser. Prior to this, LogonUI calls Credential::UnAdvise and Provider::UnAdvise to notify both entities that notifications on their respective Events interfaces are not being accepted. UI changes would be pointless with a logon attempt pending (ideally, the next thing the user sees is his desktop).Additional Resources

- Credential Provider Samples
- Smart Storage: Protect Your Data via Managed Code and the Windows Vista Smart Card APIs
- Download Windows Symbol Packages
- Windows Smart Card API Documentation
- Card Module API Documentation

After winlogon gets the result of LogonUser, the result is passed back to LogonUI and then the credential instance (the one that still has focus from GetSerialization) is notified. But before the credential receives the status code returned from LogonUser, it is again given a callback interface for making UI element changes. (See step 31 in Figure 5.)

The result of the authentication attempt is returned to the credential via the Credential::ReportResult routine. Why does a provider (or its credential objects) care about the result of the authentication attempt and why would it start making UI changes at this point?

Many of the interesting ReportResult scenarios stem from authentication failure. One of the canonical examples is user password expiration. If the user's password is near expiration, the authentication sub-status, returned via the ReportResult ntsSubstatus parameter, indicates this. In response, the built-in password credential provider prompts the user to (optionally) initiate a password change. This prompt, as well as the password change dialog itself, requires different UI elements. Thus, the password credential provider utilizes the ICredentialProviderCredentialEvents interface pointer to drive the required changes to its prompting fields.

There are interesting actions a credential provider can take in response to authentication success. For example, the built-in smart card credential provider uses this success notification as a cue to start monitoring for removal of the card that was used for authentication. It does this in order to enforce the optional session lock-on-removal policy. Once the credential-handling sequence is complete, LogonUI notifies the credential provider via Credential::UnAdvise to decrement its reference to the ICredentialProviderCredentialEvents interface pointer.

The Hybrid Credential Provider

Now that I've discussed the new credential provider architecture and how it's used, let's look at the design of the sample hybrid credential provider in more detail. Recall the high-level layout of the Windows Vista interactive logon architecture shown in Figure 2. Figure 7 augments that diagram to include the Windows Smart Card API stack and to focus on the new credential provider.

The main point to take from Figure 7 is that the hybrid credential provider interfaces with the Windows Smart Card API both directly and indirectly. The direct interface is via public routines, such as SCardEstablishContext and SCardListReaders, which allow the detection of a smart card. The indirect interface is via the Card Module API, which allows the credential provider to read a user credential file from the card in a convenient way, without having to resort to low-level card-specific commands. For the sample, almost all of the smart card-related logic is abstracted by a helper library called ScHelp.lib. (I discuss this later in the Implementation section.) The hybrid credential provider's auto-logon behavior offers an interesting view into the capabilities and subtleties of the Credential Provider architecture as a whole.

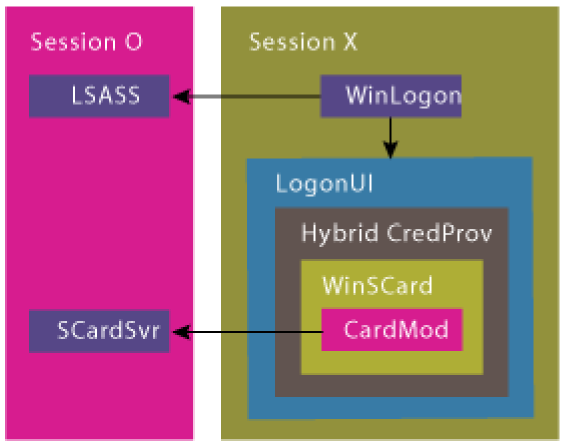

Figure 7 Hybrid Credential Provider

First, what do I mean by auto-logon? This means that if a hybrid credential is available then the user isn't prompted at all, as we've already seen. Instead, in that case, an authentication attempt is made automatically as soon as the user presses Ctrl+Alt+Delete.

The auto-logon behavior as implemented might be confusing to some users. For example, it's different from the built-in password provider, which, by default, always renders tiles, even in the non-domain-joined scenario in which the user isn't prompted for a password.

Changing the hybrid credential provider to require the user to first click a tile in all scenarios would be easy. Please note that the current implementation of GetCredentialCount is to set *pbAutoLogonWithDefault = TRUE. Instead, you should set it to FALSE. Now, the provider is guaranteed the opportunity to show at least one tile (unless another provider overrides it with a no-tile auto-logon).

When the user clicks the tile, LogonUI calls the provider's ICredentialProviderCredential::SetSelected method. In response, the credential class will set *pbAutoLogon = TRUE, triggering a call to ICredentialProviderCredential::GetSerialization by LogonUI, and a subsequent authentication attempt, without first rendering any UI element changes. In other words, if the authentication is successful, the next thing the user will see is his desktop.

The Hybrid Implementation

The changes required to transform the password credential provider into my hybrid implementation were minimal. Use a graphical comparison tool (such as windiff.exe) to compare the SDK CSampleProvider.cpp to my CHybridProvider.cpp; likewise, compare CSampleCredential.cpp to CHybridProvider.cpp. Most of the modified lines of code are from globally replacing "Sample" with "Hybrid"!

The most substantial change to CHybridProvider.cpp is in its handling of SetUsageScenario. In response to this call, the provider attempts to read a credential from a smart card. This is done via the ScHelpInit routine in the ScHelp library, which abstracts most of the smart card logic. ScHelpInit connects to the smart card subsystem, finds the first inserted card, parses the credential if it finds one, and returns the strings contained therein.

The primary change to CHybridCredential.cpp is to handle the optional domain name string to be read as part of the credential file on the inserted smart card. In GetSerialization, if a domain name is read from the card, it is used in the serialized authentication data to be passed to Kerberos. Otherwise, the result of calling the public GetComputerName is used.

The changes made to convert the PropCert sample into the ScHelp library were more extensive. A few aspects of this helper library are worth noting.

First, the main PropCert thread routine is now being called synchronously; a separate thread is not used. However, card-related operations should be performed asynchronously and that would be a relatively simple change to make. In fact, this change would be critical if the hybrid credential provider were extended to allow the user to select from multiple credential tiles. In that case, you would want the provider to immediately enumerate a tile while reading smart card data in the background, since I/O operations on some older cards can be quite slow. To round out the implementation of this change, note that you would need to establish some sort of notification mechanism to allow the smart card thread to notify the provider of credential availability changes. The provider could then notify LogonUI of the same via CredentialsChanged.

The remaining logic in ScHelp.lib includes the routines _ReadCreds, _Connect, _UnpackCred (see Figure 8), and ScHelpPackCred. The latter two deserialize and serialize the password credential file stored on a smart card. The first two implement the logic I've described briefly: enumerate smart card readers and cards, obtain a read lock on the first enumerated card, bind to the card module corresponding to that card, and read the credential file (if one exists) from the card.

The Death of GINA

Why did GINA have to go? This question is more complex than it may seem. Microsoft has a respectable track record for supporting third-party developers, and the decision to drop support for a public interface is rarely taken lightly. Nonetheless, there were strong arguments for dropping GINA and ultimately Microsoft determined that that was the right decision.

First, multi-factor authentication is in much higher demand now than it was in the early days of Windows NT®. Smart cards, biometrics, and one-time password solutions are being widely deployed in enterprises. Each evolution in authentication technology places a greater burden on the abstraction layer between the core Windows credentialing engine and the GUI that must prompt the user in a certain way. (For example, the pictorial cues and screen real estate required to prompt the user to type in a password are different than those required to prompt the user to place a finger on a fingerprint reader.) At the same time, Microsoft needs to provide as consistent an experience as possible so that users don't get confused.

In addition, the Windows logon process (winlogon.exe) has been completely rearchitected in Windows Vista. One core requirement of that effort was to move plug-ins out of the winlogon process space to the fullest extent possible. That requirement was born of reliability concerns. If, for example, a poorly written GINA is loaded into the winlogon.exe instance running in session zero on a server, a software fault could kill that critical process and in turn the machine itself. And even if GINA could have been adapted to run out of process, there would still be the issue that it wasn't designed to provide a consistent, controlled experience across arbitrarily complex, interactive credential-gathering scenarios.

Opportunities for Improvement

While I believe that my design for the hybrid credential provider, in tandem with the significant level of code reuse I achieved in the implementation, meets the requirements I laid out earlier, there are aspects of the implementation that prevent the provider from being deployment-ready in its current state. I've already discussed one such shortcoming-data should be read from the smart card asynchronously. Now I'll discuss the remaining opportunities I see for improvement in decreasing order of severity.

First, the user credential is not being stored securely on the smart card. Ideally, reading the card-based file that stores the user password should only be possible after supplying the correct PIN. However, a limitation in the current card module interface makes that difficult to implement. Namely, the set of predefined card file access conditions does not include such a "user-only read" option. (I suppose it is poetic justice that, having been involved in the card module design decisions that led to this limitation, I'm now inconvenienced by it!) I hope the product team will extend the card module interface in a subsequent version.

In the meantime, the smart card password file ought to be encrypted in such a way that, if the card is stolen, the PIN is required in order to decrypt the password. This can be achieved easily via Crypto API, as well as via the new Windows Vista CNG ("Next Generation" Crypto API). An RSA key pair, created and stored on the card, would suffice, but rather than using the RSA public key to encrypt the password file directly, cryptographic best-practice advises you to use a symmetric key and algorithm, such as Advanced Encryption Standard (AES). The RSA key would instead encrypt the symmetric key.

To round out the design for an encrypted password file, you could modify the existing password file format to include the associated encrypted key. If you go this route, be sure to consider the versioning challenges that come with committing to a crypto algorithm and key size. Assume that the algorithm you choose will get hacked some day. Also, the design as discussed thus far doesn't include a cryptographically secure data integrity check. That might seem like a minor point, since an attacker theoretically needs to know the PIN in order to modify anything on the card. But I would consider such a feature to be a necessary aspect of defense-in-depth.

The second limitation of the current implementation is that it only supports a single credential per card. Take a look at the ScHelp.cpp!_UnpackCred routine from the sample code in Figure 8. This performs a simple deserialization of the password file, which was presumably read from the card. The credential-parsing logic as a whole will only handle a single credential per card. However, some users may require multiple distinct domain credentials to get their work done. Would you extend the provider to support that, or would you issue those users multiple cards? The former increases implementation complexity while the latter increases deployment management complexity.

As an aside, I've attempted to demonstrate secure buffer parsing techniques in _UnpackCred. (Again, refer to Figure 8.) Assume that an attacker can create an evil smart card and insert it into workstations on your network. At the application level, the primary mitigation against this threat is to ensure that no assumptions are made about the validity of data read from the card-don't assume that the embedded character counts are correct and don't assume that the strings are well-formed. Simply check that the element count at the beginning of every string doesn't exceed the length of the unparsed portion of the credential, and check that the true length of any string doesn't exceed the buffer allocated for it.

The last limitation I want to discuss in the current hybrid credential provider is that only one smart card reader per machine is supported. For example, if I boot a system with two smart card readers attached, each with a smart card inserted, and each card initialized with a different credential, which card takes precedence will depend on the order in which the readers are enumerated by the smart card subsystem. The fix entails changing the semantics of the SCHELP_CONTEXT struct, defined in ScHelp.h:

```
typedef struct _SCHELP_CONTEXT
{
    LPWSTR wszUserName;
    LPWSTR wszPassword;
    LPWSTR wszDomainName;
} SCHELP_CONTEXT, *PSCHELP_CONTEXT;
```

The SCHELP_CONTEXT struct defines the data exchange between the credential provider code and the ScHelp code. It's clear that the struct only supports a single credential; a simple array or singly linked list could be introduced to enhance it. If you want this feature, don't forget to modify the handling of the _rgpCredentials member of CSampleProvider, as well, since it is currently hardcoded to support only a single credential per provider instance.

[**Editor's Update - 6/22/2007:** There is a problem with the implementation of the Hybrid Credential Provider sample for this article. As with the credential provider sample included with the RTM release of the Windows SDK for Windows Vista, the Credential::GetSerialization method returns a KERB_INTERACTIVE_LOGON structure. Unfortunately, this doesn't support unlock scenarios. Instead, a KERB_INTERACTIVE_UNLOCK_LOGON structure should be used to support both unlock and logon scenarios. You can see working examples of this in the Windows Vista Credential Provider Samples available for download at http://www.microsoft.com/downloads/details.aspx?FamilyID=b1b3cbd1-2d3a-4fac-982f-289f4f4b9300.]

Testing and Debugging

As I mentioned, testing my credential provider was a relatively painless process. Regarding my testing strategy, I knew that I wanted to be able to attach a user-mode debugger to LogonUI in order to have maximum flexibility, both for live debugging of my sample code and for generating the tracing information I discussed earlier. I also knew that since LogonUI runs as system and that the interactive logon scenario I was targeting is accessible only from the secure desktop, writing a simple self-contained test program to exercise the various credential provider COM interfaces would be a wise use of time. Nevertheless, because of my background with the technology, I decided to skip writing a test program and instead proceeded with live debugging. But I don't recommend this approach.

Since I didn't write a test program, I had placed additional pressure on myself to get a robust debugging environment set up. Unless you frequently do kernel-mode development, getting this kernel debugger-based test environment properly configured can be frustrating. At a high-level, here's the preferred way of doing this.

First, set up two machines in a standard kernel debugging configuration. One should be a reliable development system (the debugger), the other the Windows Vista test system (the debuggee). They should be connected via serial cable.

Don't neglect to configure the debuggee with a safe boot partition with Windows XP loaded. Configuration (specifically, getting the two machines to talk to each other via serial cable) can take some trial and error. What's the best way to test the serial connection? Boot both machines to Windows XP and run HyperTerminal (by selecting All Programs | Accessories | Communications | HyperTerminal). On both machines, point the program to the serial port you're using and select the data rate you'll be passing to the debugger. If the characters you type into one machine's HyperTerminal window echo on the other machine, you're finished. If not, try another serial port, connection speed, or cable.

To describe the second reason for configuring a safe boot, I need to skip forward slightly. If you find that the credential provider under test causes the host LogonUI process to die or deadlock, then you won't be able to log into Windows Vista any more. This happened to me with my initial build configuration for HybridCredProv.dll. I was using the redistributable msvcr80.dll as my runtime library. My first mistake was that I forgot to copy that binary into the system32 directory of the debuggee. However, that only prevented my credential provider from being loaded.

The next reboot back into Windows Vista left me confused. This time, I saw LogonUI start and I saw my credential provider get loaded, but I never saw any UI appear. In the debugger, I found that the msvcr80.dll startup code was deadlocked on the process loader lock with another thread. Rather than drill into this, I modified the build configuration of HybridCredProv.dll to use a statically linked runtime library.

In summary, a safe boot partition provides a useful option for fixing configuration problems during credential provider testing.

Back to the debugging configuration. The next step is to install the latest debugger package from Microsoft (the one that includes ntsd.exe and i386kd.exe) on both systems. Install the public debug symbols for Windows Vista. (A full local copy of the symbols is recommended for the debuggee.) Some people might consider this step unnecessary, but there are few things more annoying than finding yourself in the middle of live debugging with insufficient data due to missing symbols. I offer the following points based on my own experiences:

- I find it's important to always get a clean stack trace.
- Some operating system symbol files (such as ntdll.pdb) may be required in order to get even an approximate stack trace.
- Certain system-level debugging scenarios can inhibit network access in unforeseen ways, hence the need for local symbols.

Using Image File Execution Options in the system registry, configure LogonUI.exe to start within the user-mode debugger (ntsd.exe). That debugger will, in turn, redirect its output to the kernel debugger. That's why you need the latter-otherwise, since the application is only visible from the secure desktop, you have no reliable way to interact with both it and an attached debugger from the console of the test machine.

You should note that you can specify the path for loading symbols to ntsd.exe in two ways: via the -y command-line option or by the _NT_SYMBOL_PATH environment variable. The former is what's recommended when configuring Image File Execution Options. The latter, however, is the way I prefer, since it allows me to set it once, systemwide, on my test machines.

Finally, note that if your credential provider becomes unusable and you need to regain access to your machine, boot into safe mode and Windows Vista will load only the password provider, as well as the smart card credential provider if you boot into safe mode plus network. (There is a policy to turn off this fallback behavior for the security hyper-conscious.) Once booted into safe mode, you can edit the registry as appropriate.

Smart Cards and Initialization

Of course, any significant testing of the hybrid credential provider requires a credential. That is, a smart card must be prepared with the appropriately formatted credential file containing the user name, password, and optional domain name. The sample code download includes a test utility called WriteCred.exe to accomplish this. To initialize a smart card inserted in the default reader with your credential information, use the following command-line options:

```
WriteCred.exe -p <PIN> -u <UserName> -d <DomainName> -w <PassWord>
```

Unlike the behavior of the hybrid credential provider, the current implementation of WriteCred assumes that the domain name parameter is non-optional. Also, note that the password parameter should be the Windows logon password corresponding to the user name, whereas the PIN parameter is required in order to write the credential data file to the smart card.

Finally, regarding testing, the WriteCred tool, as well as the hybrid credential provider itself, only support smart cards compatible with the Microsoft Base Smart Card Crypto Provider. Your best bet for an up-to-date list of compatible smart cards is to do a Web search for "card module smart cards".

Dan Griffin is a software security consultant in Seattle, WA. He previously spent seven years at Microsoft on the Windows Security development team. Dan can be reached via www.jwsecure.com.
He'd like to thank Brian McNeill and Eric Perlin at Microsoft for their feedback on this article.
