Understanding the Novell SecretStore 3.2 APIs

Feature Article

NOVELL APPNOTES

Cameron Mashayekhi
Senior Software Engineer
Novell, Inc.
cameron@novell.com

Novell SecretStore has undergone a major upgrade in version 3.2, with numerous modifications to its client APIs, supported transport protocols, and client and server platforms. This AppNote presents an overview of the improvements and modifications made to these APIs. It is a follow-up to "A Technical Overview of Novell SecretStore 3.2" in the May 2003 issue of AppNotes (available online at *http://developer.novell.com/research/appnotes/2003/may/03/*).

Contents:

- Introduction
- Major Improvements to SecretStore APIs
- Understanding the SecretStore APIs
- Operational APIs for Connectors
- Management APIs for Utilities
- Understanding the Shared Secret APIs
- Shared Secret APIs
- Shared Secret APIs

Topics	Secure Identity Management (SIM), SecretStore, single sign-on, authentication, network security
Products	Novell SecretStore 3.2
Audience	network application developers
Level	intermediate
Prerequisite Skills	familiarity with user authentication mechanisms
Operating System	NetWare 5 and above
Tools	NDK
Sample Code	no

Developer Notes

Introduction

As an infrastructural component of the Secure Identity Management (SIM) provisioning architecture, Novell's SecretStore service is designed to securely store sensitive data such as user IDs, passwords, biometrics, and other login credentials. This type of data is commonly called *secrets*. Once the data is safely stored in eDirectory, single sign-on services such as Novell SecureLogin, Novell iChain, and Novell Portal Services, as well as third-party applications, can access and use these credentials on behalf of the authenticated user. SecretStore also protects the methods of storing, accessing, and retrieving these secrets.

Novell SecretStore has undergone a major upgrade in version 3.2, with numerous modifications to its client APIs, client and server platforms, and supported transport protocols. Among other things, SecretStore 3.2 has been enhanced to support LDAP cross-platform access based on Secure Sockets Layer (SSL) to make the service available on all eDirectory-supported client and server platforms.

This AppNote presents an overview of the new SecretStore programming interfaces, or APIs, in version 3.2.

Major Improvements to SecretStore APIs

This section presents an overview of the major improvements made to the SecretStore APIs in version 3.2.

Improved Transport and OS Platform Access

The SecretStore NDK APIs now allow the applications to choose NCP or LDAP access to eDirectory without the need for transport-specific APIs. As before, the NCP protocol is only available on Windows clients and requires that Novell's Client 32 be present on the workstation. However, the SecretStore LDAP client can be installed on either the server or the workstation without requiring Novell Client 32.

Improved API Performance

The performance of the new SecretStore interfaces has been improved to provide faster access to the data in the user's SecretStore. Older versions of the SecretStore client APIs were designed to be stateless; each API had to set up and tear down the connection to the SecretStore server. This approach supported regular connectors and was based on connectors' simple "per API" access to SecretStore.

As the paradigm of accessing SecretStore shifted from regular connectors to universal connectors, it became necessary to support stateful sessions. The new SecretStore API architecture supports the establishment of a session by allowing for initialization of the session through a call to *NSSSGetServiceInformation*, and reusing the initialized state data in the context across other API calls. In the new APIs, every call is able to tear down the connection and end a session upon request after the work is done. (The new APIs still support stateless operations if no session is established ahead of time by the connector via the context.) The use of the initialized context across calls provides for better overall performance when using the new SecretStore APIs.

New Shared Secret Format

In previous implementations, both regular and universal connectors created overlapping secrets for the same applications over a period of time. The presence of multiple secrets for the common applications in the user's SecretStore has led to the problem of synchronization between the applications. This problem can be summarized in the situation when one of these connectors modifies the user's credential in the target application and, as a result, the other connectors lose their ability to perform single sign-on for the user when they are invoked, and subsequently go out of sync. This leads to the complicated circumstance in which the other connectors require user intervention to synchronize with the changes made to credentials in the target application.

In addition to the need for solving the synchronization problem, creating multiple secrets for common applications in SecretStore causes an increase in the number of secrets in every store. This can affect performance and efficiency of access, in addition to causing an unnecessary increase in storage capacity requirements.

To solve these problems, a set of APIs based on the SecretStore APIs has been implemented as a shim layer to provide the ability for applications to process and interpret secret data that conforms to the SharedSecret specification originally used by Novell SecureLogin. Connectors conforming to this set of specifications in the SharedSecret APIs will be able to create secrets in the user's SecretStore that can be shared between regular and universal connectors.

Understanding the SecretStore APIs

This section discusses the specifications of the new SecretStore APIs. To help developers use these tools more effectively, Novell provides sample code in the Novell Developers Kit (NDK). The SecretStore NDK, sample code, and documentation is available on the Novell Developers Web site at *http://developer.novell.com/ndk*.

SecretStore Implementation

Novell SecretStore is a collection of hidden attributes on an object (User object by default) in eDirectory. These attributes are implemented through eDirectory schema extensions, and they are encrypted on the server via Novell International Cryptographic Infrastructure (NICI) servies.

- The *single-valued Key attribute* holds the cryptographic data and is the repository for stores-related statistical information.
- The *multi-valued Secrets attribute* holds encrypted secrets identified by secret IDs and is the repository for the secrets' statistical information (time stamps and so on).

Hidden attributes, by definition, are not accessible from client operating systems through eDirectory interfaces and utilities. These attributes can only be accessed by SecretStore interfaces that apply strict access control and security measures to their access, transport, and disclosure.

In addition to these hidden and encrypted attributes, which protect the data against attacks directed to the physical server, SecretStore provides an optional *Enhanced Protection* (EP) mode to defend against administrative attacks which cause the user's SecretStore to be locked. Enhanced Protection also provides for secure unlocking of the EP locks either through owner control or via an action involving two separate administrators. In this two-administrator scheme, the eDirectory password is reset by a network administrator, but SecretStore can only be unlocked by a separate SecretStore Administrator. To provide full control over this high-security feature, audit information is recorded after each administrative unlocking of Novell SecretStore in EP mode.

SecretStore Management Utilities

The following utilities are provided for configuring and managing users' SecretStores.

- *SSManager* is a Windows-based utility that allows users to manage their Novell SecretStores.
- *SSStatus* is a Windows-based "light" version of SSManager that allows users to turn on Enhanced Protection, set the Master Password, and unlock their SecretStore when necessary.
- *ConsoleOne*, with the appropriate snap-ins installed, is the all-purpose administrative utility for managing users' SecretStores, in addition to configuring and deploying the service in an eDirectory environment.

SecretStore API Architecture

Here is a summary of the overall architecture of the new SecretStore APIs.

- They are *cross-platform* APIs that work on any platform supported by eDirectory (NetWare, Windows, Solaris, AIX, Linux).

- They are *cross-transport* APIs that support either NCP (only on Windows clients) or LDAP (all eDirectory platforms).
- They are *cross-language* APIs that can be accessed through any Java or C++ compatible programming language.
- There are two categories of operational APIs for writing connectors:
  - Raw Novell SecretStore APIs
  - SharedSecret APIs

  In addition, there is a set of Raw Management APIs for writing utilities.

Here are some points to remember about the raw operational APIs:

- They are available across different platforms and can transmit data to and from SecretStore over encrypted transports.
- They can start and end "stateless" SecretStore sessions that are based on a single operation.
- They can start and end "stateful" SecretStore sessions that expand over multiple operations, if necessary.
- They can find services for the users on administratively-designated servers across the eDirectory tree using the SecretStore client's service location feature.
- They provide Enhanced Protection (EP) against possible administrative attacks.
- They provide two-administrator recovery from EP locking of the SecretStore.

Here is some general information about the SecretStore APIs:

- All of the Novell SecretStore APIs are capable of obtaining a context internally in NCP mode and operate by setting up a session and tearing it down at the end of that call.
- Each of the SecretStore APIs is capable of accepting a context (LDAP or NCP) from the caller that has been created outside the SecretStore client or inside it with a prior call to *NSSSGetServiceInformation*.
- All of the Novell SecretStore APIs default to set the target object's SecretStore Distinguished Name (DN) to be the same as the caller DN, unless specified otherwise by the calling application.
- For a caller DN to be different than the SecretStore's target object DN, it is required for the caller DN to have administrative rights over the SecretStore's target object DN or to be defined as SecretStore administrator through configuration.
- In LDAP mode, the caller DN and the SecretStore target object DN should comply with LDAP form.
- Each of the SecretStore C APIs is capable of unbinding or destroying the context upon request.

Operational APIs for Connectors

The following is a list of sample code files that can be downloaded from the Novell Developer Web site referenced above as a demonstration and guidelines on the use of the SecretStore APIs:

- *sstst.c* is the source code for the SSTST.EXE sample program that demonstrates the use these APIs over the NCP transport.
- *lstst.c* is the source code for the LSTST.EXE sample program that demonstrates the use of these APIs over the LDAP transport.
- *nbstst.c* is the source code for the NBSTST.EXE sample program that demonstrates the use of these APIs over LDAP when the bind is made by the application outside SecretStore and the context passed by the application is used to access the store.

A complete collection of SecretStore-related sample code is available as part of the SecretStore component of the NDK. All of the API prototypes, flags, structures, and error codes are defined in the *nsscl.h* file.

Here is a brief explanation of each of the SecretStore operational APIs.

NSSSGetServiceInformation

This API allows for binding/unbinding over LDAP, or creating and destroying NDS contexts over NCP, to a target SecretStore server. It returns the following statistical information regarding a user's SecretStore:

- Number of Secrets
- Cryptographic Algorithm ID
- Cryptographic Strength of the server
- Target server
- Caller's DN
- Target SecretStore's DN
- Master Password hint
- DN of the last SecretStore administrator that unlocked the SecretStore

In addition to the user (owner), the administrator is allowed to perform this operation on the user's SecretStore (except for the return of the Master Password hint).

NSSSWriteSecret

This API allows for populating SecretStore with secrets identified by uniquely-supplied secret IDs. It also allows for creating new secrets (with the optional ability to overwrite existing secrets), checking for secret ID collision, and creating the secret for the first time by using special flags.

Creating the first secret in a user's Novell SecretStore will cause the creation of the Key and Secret hidden attribute pair for that user.

In addition to the user (owner), the administrator is allowed to perform this operation on the user's Novell SecretStore.

NSSSReadSecret

This API allows for reading a secret identified by the supplied secret ID. It also allows for repair and synchronization of the SecretStore upon request by using a special flag. The first read operation after a write will initiate a background synchronization and repair of the SecretStore.

This API returns secret-related statistical information such as creation time stamp, last modified time stamp, and last accessed time stamp, in addition to the secret value. If the SecretStore configuration allows for the Last Access Time Stamp option (which is optional due to the performance penalty involved), then that time stamp is returned on the read.

If Enhanced Protection is turned on, every read on an EP-flagged secret will cause a check to see if there has been an administrative eDirectory password change. If so, the user's Novell SecretStore is locked.

Only the owner is allowed to perform this operation on his or her SecretStore.

NSSSRemoveSecret

This API allows for removing a secret identified by the supplied secret ID. Removing the last secret in the user's SecretStore will result in complete removal of the SecretStore from the object (removal of the Key and Secrets attribute pair).

This API returns secret-related statistical information such as creation time stamp, last modified time stamp, and last accessed time stamp, in addition to the secret value. If the SecretStore configuration allows for the Last Access Time Stamp option (which is optional due to the performance penalty involved), then that time stamp is returned on the read.

In addition to the user (owner), the administrator is allowed to perform this operation on the user's SecretStore.

Management APIs for Utilities

Here is a brief explanation of each of the SecretStore management APIs used for writing utilities.

NSSSEnumerateSecretIDs

This API allows for getting a list of the secret IDs for secrets stored in the user's SecretStore.

In addition to the administrator, the user is allowed to perform this operation on his or her Novell SecretStore.

NSSSSetMasterPassword

This API allows the owner to set a Master Password and related hint on his or her SecretStore for the first time, if setting of Master Passwords is allowed through configuration on the server. The Master Password is used for unlocking the SecretStore and should be set prior to Novell SecretStore getting locked.

Only the owner is allowed to perform this operation on his or her SecretStore.

NSSSUnlockSecrets

This API allows for the removal of the lock from a user's locked Novell SecretStore using one of the following three methods:

- By deleting all of the locked secrets
- By providing the last eDirectory password set by the user
- By using the Master Password is one was set prior to locking

In addition to the user (owner), a designated and configured SecretStore Administrator can unlock the user's SecretStore. If the two-administrator password reset scheme is not being used, the owner must use one of the methods listed above to unlock his or her SecretStore.

NSSSRemoveSecretStore

This API allows for the complete removal of the Novell SecretStore from the target object.

In addition to the administrator, the user is allowed to perform this operation on his or her Novell SecretStore.

Understanding the Shared Secret APIs

The Shared Secret APIs leverage the SecretStore APIs described earlier. They are used for storing secret data in a "shared secret" format. The intent of these APIs is to provide access to the SecretStore so that the single sign-on information in a user's SecretStore can be shared between different connectors. To make this possible, the connectors (regular or universal) should conform to the Shared Secret specification, which defines the Shared Secret format as the means by which connectors may share login credentials.

Terminology

The following is a list of terms and their definitions with respect to the Shared Secret specification.

- *Secret:* An addressable data member of a user's SecretStore that contains authentication data. Referenced by a SecretID, a secret may contain up to 60 KB of data, though in practice most are much smaller. Secrets are securely encrypted using NICI and the tree and user keys. They are accessible only to the authenticated NDS/eDirectory user via secure calls to the SecretStore server.
- *SecretID:* An identifier or name by which a data member of SecretStore is referenced. These names are used when reading and writing secrets. They are limited in length to 255 displayable characters encoded in UNICODE.
- *Application:* A Windows program, Web application, or mainframe host application for which login information is being stored.
- *Credential Set:* Data used to authenticate a user to a desktop, Web, or host application. Typically comprised of a username and password, a credential set can also include a PIN, e-mail address, domain or database name, certificate, or other information as required by the application. A credential set is also referred to as Login Credentials or Details.
- *Login Details:* The user-friendly name exposed in the user interface and documentation to represent one or more credential sets associated with a given application login.

Shared Secret Format

A shared secret is a regular secret that follows a set of rules that allow it to be read by all connectors that conform to this specification. A shared secret is made up of a *type*, a *name*, and a set of *key/value* pairs. The type and name of a shared secret combine to form the shared secret's identifier (secretID); the set of key/value pairs make up the data.

Note: The Shared Secret specification identifies certain characters as being reserved. These characters must be escaped when used outside of their context. The reserved characters are listed later in this AppNote.

SecretID Format. The following format should be used for the SecretIDs to comply with the Shared Secret format:

<*type*>:<*name*>

where <*type*> identifies the type of shared secret (either "SS_App" for Application secrets or "SS_CredSet" for Credential Set secrets), and <*name*> identifies the name of the shared secret.

The colon (:) serves as a delimiter and should not be escaped. All reserved characters used in either the type or name must be escaped. When combined, the type, colon, and name string cannot exceed 255 displayable characters in length.

Data Format. Shared Secret data is made up of key/value pairs in the following format:

<*key*><*delimiter*><*value*><linefeed>...<*key*><*delimiter*><*value*><linefeed><null>

where <*key*> identifies a key, <*delimiter*> is the equals character (=) in most cases (otherwise the colon character), <*value*> identifies a value, <linefeed> is the linefeed character, and <null> is the null terminator that must always appear at the end of the data.

Linefeed characters separate the pairs from each other. All of the pairs combined cannot exceed 60 KB in size. Keys should be treated as case-ignore strings, whereas the values are case-sensitive. A null terminator must follow the last pair to signal the end of the sequence. The null terminator must be present even when no entry exists. Duplicate keys are not allowed; however, duplicate values are permitted and should be discarded by connectors if they are encountered during parsing operations. The data must be in 16-bit Little Endian Unicode, including the null terminator. All reserved characters used in the key or value must be escaped.

In processing, care must be taken to modify only those key/value pairs on which the connector is directly dependent. For example, suppose you have two connectors that are dependent on the same shared secret, but one uses a Password key whereas the other requires a PIN key. In such a case, each application should add its own key without modifying any of the other keys that may exist but are not required for its operations.

Reserved Characters. Here is the list of characters that are reserved in the Shared Secret specification:

- Backslash (\ )
- Colon (:)
- Equals sign (=)
- Linefeed (0x0A(

- Null terminator (0x00)

Reserved characters that are used outside of their context must be escaped by preceding them with a backslash (\ ) character. Note that the linefeed and null characters cannot be used within a character string.

Shared Secret Types

Two types of shared secrets currently exist. They are:

- *Application shared secrets*, which contain information associating and linking applications with credentials, as well as application-specific data
- *Credential Set shared secrets*, which contain login credential informaiton used for one or more applications

When used together, these types of shared secrets can enable several applications to use the same set of credentials, or enable one application to have more than one set of credentials. This effectively means that, depending on the connector's choosing, there can be a many-to-many relationship between the two types. As indicated in the example earlier, different connectors might, as allowed by the target application, require different credentials for authenticating the users (Password, smart card, PIN, and so on). This gives rise to the need for a many-to-many grouping of these secrets.

Application Shared Secrets. Application shared secrets are used to represent Windows applications, Web applications, or mainframe/host applications. Application SecretIDs follow the format outlined previously with a type value of "SS_App". The application ID should be able to identify the application in a unique fashion. As a rule, it should use the appropriate format as shown in the following table:

Windows application	program_name.exe
Web application	unique URL
Mainframe/host application	host application name

Application data contains application-specific information that is necessary to authenticate the user to the application, but that is not considered a credential. Examples include the Control ID for a password field in a Windows application, or the name of a password field in a Web form. Users do not normally need to enter such information to authenticate; it is inherent in the application. This application-specific data is in the key/value pair format with an equals character as the delimiter. The choice of what keys to use is left up to the connector.

Application data also contains pointers to the credential sets with which the application is associated. These pointers follow the key/value pair format as follows:

SS_CredSet:<*credsetname*>

where <*credsetname*> is the name of the credential set that is associated with the application.

Note that the key must be "SS_CredSet" and that the colon character serves as the delimiter for credential set pointers. An exception to the data format for these pointers is that, when the key is "SS_CredSet", duplicates are permitted so that applications may be associated with more than one credential set.

Credential Set Shared Secrets. Credential Set shared secrets are used to contain login credentials needed to authenticate a user to a Windows application, Web site, or mainframe/host application. They may be shared between multiple applications of the same or different type, primarily when the authentication database or mechanism behind such applications is the same. Examples include NDS authentication to the same object in a tree, or Windows and Web interfaces to the same application such as GroupWise. By sharing credential sets, a password saved or changed in one application automatically applies to other applications that share the same authentication database and secret data at the target.

Credential Set SecretIDs follow the format outlined previously with a type value of "SS_CredSet". The administrator or user normally determines the name of a credential set. The name should be such that it helps to associate shared credentials with the appropriate applications. An example of a shared secret identifier is:

SS_CredSet:Groupwise

Credential data contains information that users provide to authenticate to an application. Examples include their username and their password. This data is in the key/value pair format, with an equals character (=) as the delimiter. The choice of what keys to use is left up to connector. However, connectors must agree with each other on this in order to share single sign-on information. Thus, where possible, the following known keys should be used:

- Username
- Password
- Other

An example of a shared secret value is:

Password=zuma<linefeed>Username=jdoe<null>

Shared Secret APIs

These APIs are built on top of the raw SecretStore APIs so they inherently comply with the SecretStore specifications. Connectors can use these APIs to create Shared Secret (SHS) compliant secret IDs and secrets.

As with the SecretStore raw APIs, a complete collection of Shared Secret sample code files is available on Novell's Developer Web site as a component of the Novell NDK. The following sample code files can be downloaded and used as template that completely demonstrates the use of SecretStore APIs:

- *sshtst.c* is the source code for the SSHTST.EXE program that demonstrates the use of the shared Secret APIs over the NCP transport.
- *lshtst.c* is the source code for the LSHTXT.EXE program that demonstrates the use of the shared Secret APIs over LDAP.

All of the API prototypes, flags, structures, and error codes are defined in the *ssshs.h* file.

Operational APIs

The Shared Secret Operational APIs operate on SecretStore and require that the context to the SecretStore be set up using regular SecretStore APIs prior to the use of these APIs. These calls use SecretIDs that comply with SHS format.

- *NSSSReadSharedSecret* allows for a secret in the SHS format to be read out of the SecretStore and assigned to a handle previously created with a Create Handle call to be used by these calls.
- *NSSSWriteSharedSecret* allows for a secret in the SHS format that is previously assigned to a handle to be written to the SecretStore.
- *NSSSRemoveSharedSecret* allows for a secret in the SHS format to be removed from the SecretStore. The SecretID is assigned to a previously initialized handle.

These operational APIs are created and formed using the Processing APIs listed below.

Processing APIs

Here are the Processing APIs that operate on the secret buffers returned by the Shared Secret Operational APIs.

- *NSSSCreateSHSHandle* allows for the creation of a handle for an SHS buffer for the first time to populate and process an SHS format compliant secret that will be formed as a list of components.
- *NSSSDestroySHSHandle* allows for the destruction of an SHS secret buffer signified by a handle in memory after the completion of the target operations.

- *NSSSGetNextSHSEntry* allows for moving through the SHS buffer components (key/value pairs) of the Shared Secret signified by the handle.
- *NSSSAddSHSEntry* allows for inserting a component (key/value pair) into the Shared Secret buffer signified by the handle at the current position of the Shared Secret.
- *NSSSRemoveSHSEntry* allows for removing a component (key/value pair) from the Shared Secret buffer signified by the handle passed in at the current position of the Shared Secret.

Note: As described earlier, Shared Secret components are on key/value paired structures formed as a list that are used by the Processing APIs. Operational APIs can consume SHS buffers (list of components) signified by a handle and convert them to and from raw secret format for raw read and write operations to and from SecretStore.

Sequence of Shared Application or Credential Set Secret Operations

To help you get an idea of what is involved in using the Shared Secret APIs, here is the sequence of events that occur when reading, writing, and removing a shared Application or Credential Set secret.

Keep in mind the following points about the connector:

- For each thread in a connector operating on shared secrets, a call to *NSSSCreateSHSHandle* is needed that returns a handle which is used for passing to subsequent calls.
- All of these calls require a SecretStore context handle that has previously been initialized through calls to *NSSSGetServiceInformation*.
- All of these calls require the handle as a well as a user-populated SS_SH_SECRET_ID_T structure containing the shared secret type, name, and length to be passed to them.
- All of these calls create a SecretID according to the SecretID format using either “SS_App” or “SS_CredSet” as the prefix.

Read Operation. Here are the steps involved in a read operation:

1. *NSSSReadSharedSecret* converts the shared SecretID to the SecretStore format and makes a call to *NSSSReadSecret* to retrieve secret data from SecretStore.
2. The secret data is parsed according to the Shared Secret format using the parsing library. This is done through internal sequential calls made to add entries of key/value pairs into the list associated with the handle. This call contains the handle as well as pointers to internally-allocated key and value buffers.

Write Operation. Here are the steps involved in a write operation:

1. The connector makes sequential calls to *NSSSAddSHSEntry* to add entries of key or value data into the list associated with the handle. This call contains pointers to the user-allocated key and value buffers.
2. A connector's call to *NSSSWriteSharedSecret* requires a handle as well as a connector-populated SS_SH_SECRET_ID_T structure containing the shared secret type, name, and length. This call converts the shared SecretID to the SecretStore format and makes a call to *NSSSWriteSecret* to write secret data to SecretStore.
3. Internally, *NSSSWriteSharedSecret* makes sequential internal calls to retrieve data from the list associated with the handle and populates an internal buffer.
4. The internal buffer is parsed according to the Shared Secret format using the parsing library, and the resultant data is passed into the Secret buffer for passage to the SecretStore *NSSSWriteSecret* API. Then the API call is made.
5. Before concluding the write operation, the connector makes a call to *NSSSDestroySHSHandle* to free the list associated with the handle for each shared secret thread of execution.

Remove Operation. Here are the steps involved in a remove operation:

1. The connector makes a call to *NSSSRemoveSharedSecret* and passes it a pre-populated SS_SH_SECRET_ID_T structure containing the shared secret type, name, and length.
2. Internally, the Secret Identifier is parsed according to the Shared Secret format for handling of delimited characters.
3. Also internally, a call to *NSSSRemoveSecret* is made and the SecretID formed in SecretStore format is passed into the API to remove the target secret from SecretStore.

Accessing and Modifying a Shared Secret. Here is the sequence involved when a connector accesses and modifies a shared Application buffer or Credential Set secret.

1. The connector makes sequential calls to *NSSSGetNextSHSEntry* to operate on the list associated with the handle.
2. The connector can then make calls to *NSSSAddSHSEntry* to add data to the list associated with the handle.
3. The connector can also make calls to *NSSSRemoveSHSEntry* to remove data from the list associated with the handle.
4. The connector can then utilize the data for application-specific purposes.

Conclusion

Novell SecretStore provides a secure, eDirectory-based infrastructure for applications and services to store and access user authentication secrets. Applications and services are enabled to utilize this service either as regular connectors or through the use of universal connectors. SecretStore is designed to provide a secure solution for customers who want to take advantage its single sign-on capabilities.

For Additional Information

For the latest information on deploying Novell SecretStore on your network, refer to the installation and configuration manuals supplied on the product CD.

As of this writing, Novell SecretStore 3.2 code and documentation are available as an early access release at *http://developer.novell.com/ndk*.

The following AppNotes and Developer Notes may also be of interest:

- "A Technical Overview of Novell SecretStore 3.2"
  http://developer.novell.com/research/appnotes/2003/may/03/
- "Understanding Novell's Single Sign-On"
  http://developer.novell.com/research/appnotes/2000/february/02/
- "SecretStore: Novell Single Sign-on Version 1.1"
  http://developer.novell.com/research/devnotes/2000/april/02/d000402.htm
- "SecretStore Single Sign-on"
  http://developer.novell.com/research/devnotes/1999/november/05/

For other AppNotes on security-related topics, refer to the Novell Research Web site at *http://www.novell.com/appnotes*.