Novell Developer Kit

www.novell.com

SECRETSTORE DEVELOPER KIT FOR C

October 8, 2003

Novell®

Legal Notices

U.S. Patent Nos 5,553,139; 5,553,143; 5,677,851; 5,758,069; 5,784,560; 5,818,936; 5,864,865; 5,903,650; 5,905,860; 5,910,803 and other Patents Pending.

Novell, Inc.
1800 South Novell Place
Provo, UT 84606
U.S.A.

www.novell.com

NDK SecretStore Developer Kit for C

October 8, 2003

Online Documentation: To access the online documentation for this and other Novell developer products, and to get updates, see developer.novell.com/ndk. To access online documentation for Novell products, see www.novell.com/documentation.

Novell Trademarks

AlarmPro is a registered trademark of Novell, Inc. in the United States and other countries.
AppNotes is a registered trademark of Novell, Inc.
AppTester is a registered trademark of Novell, Inc. in the United States.
ASM is a trademark of Novell, Inc.
BorderManager is a registered trademark of Novell, Inc.
BrainShare is a registered service mark of Novell, Inc. in the United States and other countries.
C3PO is a trademark of Novell, Inc.
Client32 is a trademark of Novell, Inc.
ConsoleOne is a registered trademark of Novell, Inc.
Controlled Access Printer is a trademark of Novell, Inc.
Custom 3rd-Party Object is a trademark of Novell, Inc.
DeveloperNet is a registered trademark of Novell, Inc. in the United States and other countries.
digitalme is a registered trademark of Novell, Inc.
DirXML is a registered trademark of Novell, Inc.
eDirectory is a trademark of Novell, Inc.
exteNd is a trademark of Novell, Inc.
exteNd Composer is a trademark of Novell, Inc.
exteNd Director is a trademark of Novell, Inc.
exteNd Workbench is a trademark of Novell, Inc.
FAN-OUT FAILOVER is a trademark of Novell, Inc.
Full Service Directory is a trademark of Novell, Inc.
GroupWise is a registered trademark of Novell, Inc. in the United States and other countries.
Hardware Specific Module is a trademark of Novell, Inc.
Hot Fix is a trademark of Novell, Inc.
iChain is a registered trademark of Novell, Inc.
instantme is a registered trademark of Novell, Inc.
Internetwork Packet Exchange is a trademark of Novell, Inc.
IPX is a trademark of Novell, Inc.
IPX/SPX is a trademark of Novell, Inc.
jBroker is a trademark of Novell, Inc.
JNOS is a trademark of Novell, Inc.
Link Support Layer is a trademark of Novell, Inc.
LSL is a trademark of Novell, Inc.
ManageWise is a registered trademark of Novell, Inc., in the United States and other countries.
Media Support Module is a trademark of Novell, Inc.
Mirrored Server Link is a trademark of Novell, Inc.
MP/M is a trademark of Novell, Inc.
MSL is a trademark of Novell, Inc.
MSM is a trademark of Novell, Inc.
NCP is a trademark of Novell, Inc.
NDPS is a registered trademark of Novell, Inc.
NDS is a registered trademark of Novell, Inc. in the United States and other countries.
NDS Manager is a trademark of Novell, Inc.
NE2000 is a trademark of Novell, Inc.
NetMail is a trademark of Novell, Inc.
NetWare is a registered trademark of Novell, Inc. in the United States and other countries.
NetWare/IP is a trademark of Novell, Inc.
NetWare Core Protocol is a trademark of Novell, Inc.
NetWare Loadable Module is a trademark of Novell, Inc.
NetWare Management Portal is a trademark of Novell, Inc.
NetWare MHS is a registered trademark of Novell, Inc.
NetWare Name Service is a trademark of Novell, Inc.
NetWare Peripheral Architecture is a trademark of Novell, Inc.
NetWare Print Server is a trademark of Novell, Inc.

Third-Party Trademarks

Contents

Novell SecretStore Developer Kit for C

As a component in the eDirectory™ infrastructure, the Novell® SecretStore™ service is designed to securely store sensitive data such as user IDs, passwords, biometrics, and other login credentials—all commonly called secrets.

Once secrets are safely stored in eDirectory, single sign-on services (such as Novell SecureLogin™, Novell iChain™, and Novell Portal Services™, and third-jm nparty 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 Developer Kit for C (SSOCOMP) provides C functions that enable network applications to securely access user secrets. This document describes the C programming library that augments the Novell SecretStore for Java API set. You also may want to refer to development documentation for Cryptography Libraries for C (Novell International Cryptographic Infrastructure [NICI]™) to understand how security services are implemented by SSOCOMP.

Novell SecretStore Documentation History

Novell SecretStore version 3.2 has undergone a major upgrade, with numerous modifications to its client APIs, client and server platforms, and supported transport protocols. Among other things, it 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.

Originally, Novell Single Sign-on™ (NSSO) version 1.0 provided single sign-on access for a limited number of key applications that were used primarily in intranet environments. Version 2.x, bundled with Passlogix v-GO™ expanded the functionality to most Web sites and Windows-based applications, with limited support for terminal emulators.

In June 2001, Novell released Novell Secure Log-in™ 2.5 (NSL), an interim single sign-on solution that provided enhanced features of NSSO 2.x but lacked integration with several key Novell technologies (SecretStore, NMAS, NICI, etc.). Consequently, Novell introduced Novell Secure Log-in 3.0 during Fall 2001, which combined features of both NSSO and NSL and full integration with Novell security technologies. NSL 3.0 was fully integrated with the SecretStore API described in this document and supersedes earlier versions.

With the February 2002 NDK release, the name of the API was changed from Novell Single Sign-on for C to Novell SecretStore Developer Kit for C, stressing the SecretStore functionality that facilitates the single sign on process for various applications. Novell is now labelling and branding SecretStore components separately from the products that consume them.

While Novell Single Sign-on was the first Novell product that used SecretStore technology, a number of Novell products now consume the SecretStore methods described in this document (for example, Novell Securelogin, Novell iChain, Novell Portal Services (NPS), Novell DirXML, virtual CDs (VCD), etc.).

1

Concepts

The concepts described in this chapter include:

- Understanding SecretStore Functions (page 11)
- Understanding Shared Secret Functions (page 18)
- Shared Secret Functions (page 22)
- Implementation Dependencies (page 24)
- SecretStore API Enhancements (page 24)
- SecretStore Use Scenarios (page 26)
- eDirectory and SecretStore (page 27)
- NICI and SecretStore (page 27)
- Naming Conventions (page 29)
- Enhanced Protection (page 29)
- SecretStore Service Discovery (page 30)
- Using Extension Structures (page 31)

For additional background about enabling your applications to use SecretStore, see:

- 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)

Understanding SecretStore Functions

This section discusses the following specifications for SecretStore functions:

- SecretStore Implementation (page 12)
- SecretStore Architecture (page 12)
- SecretStore API General Information (page 13)
- Single Sign-on Methods (page 13)
- Connector Interfaces (page 15)

- SecretStore Management Utilities (page 17)

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 encrypted on the server using Novell International Cryptographic Infrastructure (NICI) services.

- 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).

NOTE: For more developer implementation information about NICI, see Cryptography Libraries for C.

Hidden attributes, by definition, are not accessible from client operating systems through eDirectory common 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 data against attacks directed to the server, SecretStore provides an optional Enhanced Protection (EP) mode to defend against administrative attacks by locking the user's SecretStore. EP also enables secure unlocking, either through user control or through an action involving two separate administrators.

In this two-administrator scheme, the eDirectory password or authentication credential are 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 Architecture

The overall architecture of the SecretStore API includes:

- Cross platform functions that work on any platform supported by eDirectory (NetWare, Windows, Solaris, AIX, Linux, and HPUX).
- Cross-transport functions that support either NCP (only on Windows clients) or LDAP (on all eDirectory platforms).
- Cross-language functions that can be accessed through any Java* or C++ compatible programming language.
- Two categories of operational functions for writing connectors are
    - Raw Novell SecretStore functions
    - Shared Secret functions
- A set of raw management functions for writing utilities that:
    - Are available across different platforms and can transmit data to and from SecretStore over encrypted transports.
    - Can start and end "stateless" SecretStore sessions for enabled applications (see "Improved API Performance" on page 25))
    - Can start and end "stateful" SecretStore sessions that expand over multiple operations, if necessary (see "Improved API Performance" on page 25).

- Can find services for the users on administratively-designated servers across the eDirectory tree using the SecretStore client's service location feature.
- Provide Enhanced Protection (EP) against possible administrative attacks (see "Enhanced Protection" on page 29).
- Provide two-administrator recovery from EP locking of the SecretStore (see "SecretStore Implementation" on page 12).

SecretStore API General Information

- All of the Novell SecretStore functions 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 in each stateless API session.
- Each of the SecretStore functions 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 (page 79).
- All of the Novell SecretStore functions 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 functions is capable of unbinding or destroying the context upon request.

Single Sign-on Methods

The SecretStore service uses two basic single sign-on methods:

- Application Connectors (page 13)
- Universal Connectors (page 14)
- SecretStore Vault Service (page 14)
- SecretStore Encrypted Attribute Service (page 15)

NOTE: These methods require implementation of the Shared Secret functions described in this document to enable applications to share secrets stored by different services.

Application Connectors

Using application connectors (or "regular" connectors) was the original method that enabled individual applications to access SecretStore. Specific application connectors were created for a number of e-mail systems, database systems, enterprise in-house applications, etc.

One popular example of an application connector is the "Connector for Lotus Notes," which was part of the now-obsolete Novell Single Sign-on product line. While SecretStore remains as the cornerstone enabling technology, application connectors have been a specialized way of enabling applications to perform single sign-on operations.

Connectors are the method of choice in network environments where home grown applications are used and the source of the client component of a network application is available for modification. In this category, there are some very successful custom application connector implementations. For example, Novell GroupWise and the Novell Client for Windows have built-in connectors that provide single sign-on when SecretStore is available.

This set of solutions provides cross-platform support for a wide range of applications.

Universal Connectors

Universal connectors are currently the most efficient, cost effective, and preferred method to enable single sign-on to target services. However, their use is limited to applications operating on Windows and using the Windows GUI. Users interact with the GUI to provide secrets and IDs, which are then stored in SecretStore.

These stored credentials are then passed to the application on subsequent invocations. The passed credentials authenticate the user without active interaction. This also allows for populating SecretStore for known applications beforehand. The greatest benefit of universal connectors is that they can often provide single sign-on to an application without requiring modifications to the application code itself.

Novell SecureLogin (NSL) is an advanced example of a universal connector. NSL relies on the architecture and methods of operation in the Windows operating system, browsers, and terminal emulators to recognize the prompts for authentication. The user or the administrator uses the built-in wizards or scripting language capability of the product to enable NSL to recognize the target applications, browsers, and service authentication dialogs.

Proxy Portals

Proxy portals are another category of universal connectors. Using these, a trusted, secure service can do the following:

- Act as a connector to authenticate the user to the applications and services
- Allow for populating users' SecretStores ahead of time so that the users can subsequently access applications

SecretStore Vault Service

SecretStore can be used as a secure vault to store user or application secrets of any kind. SecretStore by default is installed on an eDirectory user object by extending schema extensions. However, by using the management utilities supplied with SecretStore, you can extend schema on other object types, which allows applications to use SecretStore as a vault for storing secrets.

When SecretStore is enabled on a user object, the service gurantees that the read operation is only available when the requestor owns the SecretStore. This means that the DN of the requestor on the connection is the same as the target object containing the SecretStore. Consequently, if the administrator is the requestor, the service allows all of the operations to be performed for management and administration of the target store except reading of the user's secrets.

For non-user object based stores, since the store is used as a vault for applications, requestors with administrative rights also are allowed to read the secrets and perform necessary operations on behalf of the service owning the vault.

SecretStore Encrypted Attribute Service

SecretStore also provides the Encrypted Attribute Service, which allows requestors to store data in an encrypted secure form in eDirectory. As described in the previous items, access control on the stored data is a function of the type of object used for the target SecretStore.

Connector Interfaces

The interfaces for the administrative utilities and SecretStore connectors are based on SecretStore NCP-92 and/or LDAP extensions on Windows workstations. Client applications running on NetWare and UNIX platforms (Linux, Solaris, and AIX) are only LDAP-based extensions. NCP-92 is SecretStore's encrypted transport for secure transmission of data between a client and a server.

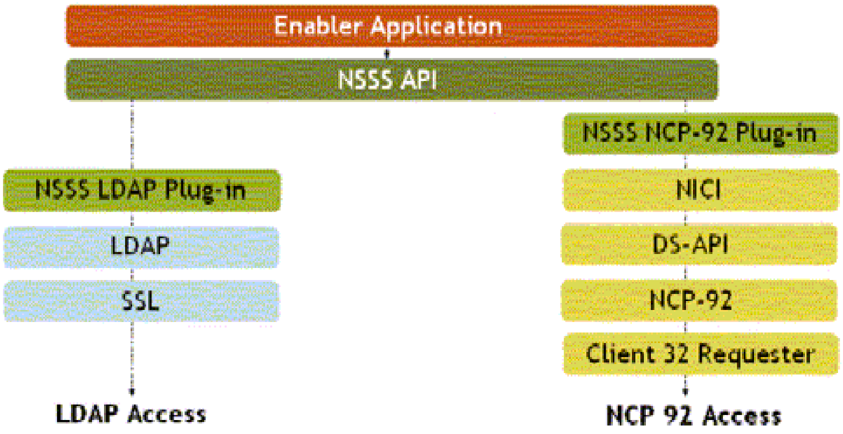
The NCP-92 transport utilizes NICI for encrypting inbound and outbound data to and from SecretStore. For more information, see "NICI and SecretStore" on page 27.

This strong encryption is based on the common algorithm and strongest session key that is negotiated between NICI running on the client and server. The key and algorithm are negotiated at the end of the user's successful Client32 authentication to eDirectory. At the end of each session the keys are destroyed. Each session with a different server has its own keys.

An LDAP extensions-based connection can be established through an SSL LDAP bind with the server. The bind can be done by the application prior to calling SecretStore, or by SecretStore if the application provides the appropriate credentials when calling the SecretStore interface. If a connection over LDAP is not SSL based, SecretStore rejects it. The strength of encryption on the wire is dependent on the SSL-negotiated algorithm for the session.

Figure 1 illustrates the client NCP and LDAP protocol stacks on a client workstation.

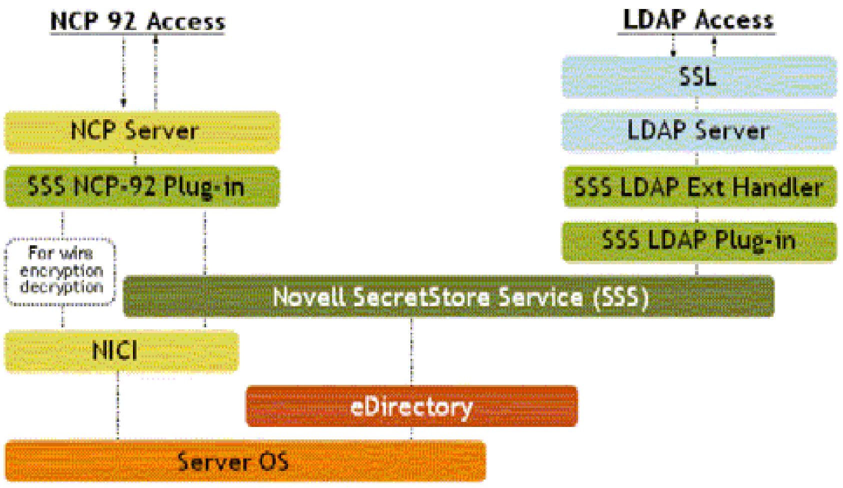
Figure 1 **Novell SecretStore C Server Architecture**



NOTE: The NCP-92 stack is available only on clients running Windows operating systems, when Novell Client 32 is installed.

Figure 2 illustrates the server NCP and LDAP protocol stacks on a server platform.

Figure 2 Novell SecretStore C Server Architecture



NOTE: The NCP-92 stack is available only on NetWare server operating systems.

Operational Functions for Connectors

This section describes the operational functions used by connectors:

Function	Description
NSSSGetServiceInformation (page 79)	Enables binding and 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 used on that SecretStore ◆ Cryptographic strength of the server ◆ Cryptographic strength of the clienbt ◆ 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 (page 101)	Enables populating SecretStore with secrets identified by uniquely-supplied secret IDs. It also enables the creation of 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 causes 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.

Function	Description
NSSSReadSecret (page 84)	Enables reading a secret identified by the supplied secret ID. It also enables the repair and synchronization of the SecretStore upon request by using a special flag.The first read operation after a write initiates a background synchronization and repair of the SecretStore. This function 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 enables the Last Access Time Stamp option (which is optional due to the performance penalty involved), that time stamp is returned on the read. If Enhanced Protection is turned on, every read on an EP-flagged secret causes a check to see if there has been an administrative eDirectory login credential 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 (page 87)	Enables removing a secret identified by the supplied secret ID. Removing the last secret in the user's SecretStore results in complete removal of the SecretStore from the object (removal of the Key and Secrets attribute pair). This functions 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 enables 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.

Sample Code

SecretStore sample code is available as part of the SecretStore component of the NDK. Here are the samples that demonstrate the use of the operational functions described above:

Source Code	Sample Program	Description
sstst.c	SSTST.EXE	Demonstrates the use of the functions for connectors over the NCP transport.
lstst.c	LSTST.EXE	Demonstrates the use of the functions over the LDAP transport.
shtst.c	SHTST.EXE	Demonstrates the use of the Shared Secret functions over NCP.
lshtst.c	LSHTST.EXE	Demonstrates the use of the Shared Secret functions over LDAP.
nbstst.c	NBSTST.EXE	Demonstrates the use of the functions 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.

SecretStore Management Utilities

The API also includes the following utilities to configure and manage a 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 (not supplied in the SDK).
- 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 (not supplied in the SDK).
- SSSINIT.EXE enables administrators to extend the schema on a tree without performing a full installation of SecretStore on the server in that tree. The same utility also is used to remove schema extensions from the tree if the extensions are not used by any objects.
- LDAPINIT.EXE enables administrators to add SecretStore extensions to a target server in the tree, which enables SecretStore LDAP operations against the SecretStore on that server. The same utility is used to remove the extensions from the server.

Management Functions for Utilities

SecretStore uses the following management functions for writing utilities:

Function	Description
NSSSEnumerateSecretIDs (page 74)	Enables you to get 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 their Novell SecretStore.
NSSSSetEPMasterPassword (page 96)	(NSSSSetMasterPassword) Enables the owner to set a Master Password and related hint on their 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 before locking occurs. Only the owner is allowed to perform this operation on their SecretStore.
NSSSUnlockSecrets (page 98)	Enables the removal of the lock from a user's locked store by using one of the following methods: ◆ Deleting all of the locked secrets ◆ Providing the last eDirectory password set by the user (only for eDirectory password changes, not other kinds of credentials) ◆ Using the Master Password (if one was set prior to locking that can be used to unlock in any form of administrative credential change) 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 their SecretStore.
NSSSRemoveSecretStore (page 89)	Enables the complete removal of the Novell SecretStore from the target object. Both the user and administrator are allowed to perform this operation on SecretStore.

Understanding Shared Secret Functions

The Shared Secret functions leverage the functions described in "Operational Functions for Connectors" on page 16 and "Management Functions for Utilities" on page 18. Shared Secret functions are used for storing secret data in a "shared secret" format. These functions 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.

To implement these functions, you should understand the following concepts:

- Shared Secret Terminology (page 19)
- Shared Secret Format (page 19)
- Shared Secret Types (page 20)

Shared Secret Terminology

The following terms used with the Shared Secret functions:

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 (see "NICI and SecretStore" on page 27). Secrets 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 enable 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 reserved characters that must be escaped when used outside of their context. See the listing of "Reserved Characters" on page 20.

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, be careful 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 Shared Secret reserved characters:

- 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: Linefeed and null characters cannot be used within a character string.

Shared Secret Types

There are currently two types of shared secrets:

- Application Shared Secrets (page 21)—contain information associating and linking applications with credentials, as well as application-specific data.
- Credential Set Shared Secrets (page 21)—contain login credential information 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 (***which example?), different connectors might, as allowed by the target application, require different credentials for authenticating the users (password, smart card, PIN, etc.). This results in 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 previously defined format with a type value of SS_App. The application ID should be able to uniquely identify the application. As a rule, it should use the appropriate format as shown in the following table:

Application Data Type	Application Format
Windows	program_name.exe
Web	unique URL
Mainframe/host	host application name

Application data contains application-specific information that is necessary to authenticate the user to the application but 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.The key must be SS_CredSet and that the colon character serves as the delimiter for credential set pointers.

NOTE: An exception to the data format for these pointers is that duplicates are permitted when the key is "SS_CredSet" to enable applications to 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 help 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 which keys to use is left up to connector. However, connectors must agree with each other on this 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 Functions

The Shared Secret functions are built on top of the raw SecretStore functions, so they inherently comply with the SecretStore specifications (see "Shared Secret Format" on page 19). Connectors use these functions to create Shared Secret (SHS) compliant secret IDs and secrets.

This section discusses the following specifications for Shared Secret functions:

- Operational Functions (page 22)
- Processing Functions (page 23)
- Sequence of Shared Application or Credential Set Secret Operations (page 23)
- Sample Code (page 23)

Operational Functions

The Shared Secret operational functions operate on SecretStore and require that you set up the context to the SecretStore using regular SecretStore functions before using these function. These calls use SecretIDs that comply with SHS format (see "Shared Secret Format" on page 19):

Function	Description
NSSSReadSharedSecret (page 82)	Enables 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 (page 104)	Enables a secret in the SHS format that is previously assigned to a handle to be written to the SecretStore.
NSSSRemoveSharedSecret (page 91)	Enables 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 Functions

Here are the processing functions that operate on the secret buffers returned by the Shared Secret Operational functions:

Function	Description
NSSSCreateSHSHandle (page 72)	Enables the creation of a handle for an SHS buffer for the first time to populate and process an SHS format compliant secret that is formed as a list of components.
NSSSDestroySHSHandle (page 73)	Enables the destruction of an SHS secret buffer signified by a handle in memory after the completion of the target operations.
NSSSGetNextSHSEntry (page 77)	Enables moving through the SHS buffer components (key/value pairs) of the Shared Secret signified by the handle.
NSSSAddSHSEntry (page 70)	Enables inserting a component (key/value pair) into the Shared Secret buffer signified by the handle at the current position of the Shared Secret.
NSSSRemoveSHSEntry (page 94)	Enables 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 explained in "Shared Secret Format" on page 19, Shared Secret components are on key/value paired structures formed as a list that are used by the processing functions. Operational functions 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 implement the Shared Secret functions, you should understand the sequence of events when reading, writing, and removing a shared Application or Credential Set secret. Follow the procedures outlined in the following sections:

- "Writing Shared Application or Credential Secrets" on page 38
- "Reading Shared Application or Credential Secrets" on page 39
- "Modifying Shared Application or Credential Secrets" on page 40
- "Removing Shared Application or Credential Secrets" on page 42

Keep in mind the following points about the connector:

- For each thread in a connector operating on shared secrets, a call to NSSSCreateSHSHandle (page 72) is needed to return a handle that 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 (page 79).
- 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.

Sample Code

As with the SecretStore raw APIs, a complete collection of Shared Secret sample code is available on Novell's Developer Web site as a component of the Novell NDK. The following sample code 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.

Implementation Dependencies

You must install SSOCOMP included in the Novell Developer Kit (http://developer.novell.com/ndk/ssocomp.htm) download to enable development and access to SecretStore in your applications. No other install dependencies are required to use this API.

This API allows applications to choose either NCP or LDAP access to eDirectory without requiring transport-specific functions. Like earlier versions of this service, NCP protocol is only available on Windows clients and requires installation of Novell Client32 on the workstation. However, the SecretStore LDAP client can be installed on the workstation or server without requiring Novell Client32, but LDAP SDK components should be installed in the target environment (LDAPSDK, LDAPSSL, and LDAPX).

To test and use an application that you have enabled with SecretStore, you need the software installed on the client and server listed below.

Server Requirements

- Novell LDAP SDK for LDAP option.
- NetWare® 6.x or 5.0 with Support Pack 1 or later versions (for eDirectory, whether running NetWare or Windows NT/2000 [eDirectory 8.7x]).
- Novell Directory Services (version installed with NetWare 5 or Windows NT). NetWare 5.1 with eDiectory 8.5 is required to use or install LDAP.

  NOTE: If the product NDS 8 for NetWare is listed, verify that it is 8.12 or later. If using NDS 8, make sure you use NetWare 5.0 Support Pack 1 or later.
- Use Novell International Cryptographic Infrastructure (NICI) 1.5.7 or later on NetWare or NICI 2.4.2 or later (latest version is preferrable) on the Windows client and server.

Client Requirements

- Novell SecretStore service client software must be installed on a workstation running Novell Client for Windows NT Version 4.5 or later, or Novell Client for Windows 95/98 Version 3.0 or later. Use the latest versions of the client for NCP access.
- ConsoleOne 1.3 or later—ConsoleOne 1.3 is installed with NetWare 6. To manage the network, a SecretStore service ConsoleOne Snapin is provided in the Novell SecureLogin service product.

SecretStore API Enhancements

The SecretStore version 3.2 API includes these modifications:

- Improved Transport and OS Platform Access (page 25)
- Improved API Performance (page 25)
- New Shared Secret Format (page 25)

Improved Transport and OS Platform Access

SSOCOMP now allows applications to choose NCP or LDAP access to eDirectory without requiring transport-specific APIs. As before, the NCP protocol is only available on Windows clients and requires that the Novell Client 32 to 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

SecretStore interfaces now provide faster access to the secrets in a user's SecretStore. Older versions of the SecretStore client APIs were designed to be stateless; that is, each API had to setup 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 connectors to the universal connectors, supporting stateful sessions became necessary. Consequently, SecretStore now allows initialization at the beginning of a session by calling NSSSGetServiceInformation (page 79), then reusing the initialized state data in the context across other function calls.

All function calls are now enabled to tear down the connection and end a session upon request after the work is done. Nevertheless, the new API still supports stateless operations if no session is established ahead of time by the connector via the context. These changes to the SecretStore API architecture provide for improved performance by allowing the use of the initialized context across the calls.

New Shared Secret Format

In previous implementations of the API, both regular and universal connectors created overlapping secrets for the same applications in the user's SecretStore, which led to synchronization problems between the applications. For example, when a connector modified a user's credential in the target application, other connectors lost their ability to perform single sign-on for the user, and subsequently became out of sync. Consequently, connectors required user intervention to synchronize the changes made to credentials in the target application.

In addition, creating multiple secrets for common applications in SecretStore results in an increased number of secrets in every store. This can affect application performance and access efficiency of access, in addition to unnecessary storage requirements.

To solve these problems, the SecretStore functions are now implemented as a shim layer to enable applications to process and interpret secret data that conform to the Shared Secret specification originally used by Novell SecureLogin. Connectors conforming to this set of specifications in the Shared Secret API can create secrets in the user's SecretStore that can be shared between regular and universal connectors.

SecretStore Use Scenarios

Users first authenticate to eDirectory where their SecretStore-enabled applications get access to the SecretStore service. Following authentication, the application user is granted access to the enabled application services and resources without seeing a password dialog or other authentication screens when the application starts up. It appears to the user as if access were granted automatically.

NOTE: The APIs are designed for client enablement for SSO and other methods of SecretStore utilization. SecretStore service will be shipped as a component of all eDirectory platforms after the version 8.7.1 release.

When the user first launches an enabled application, the application's client queries eDirectory as to whether or not the user is authenticated to the directory. If the user is not authenticated, then the application's client displays the password entry dialog. After the user enters a password, the application authenticates the user and grants access.

If the user is authenticated to eDirectory, then the client asks for the secret from SecretStore. This can be done in one of the following scenarios.

Establishing the Secret ID

Making a call to NSSSEnumerateSecretIDs (page 74) or directly to NSSSReadSecret (page 84) will indicate whether or not this is the first time SecretStore has been accessed for the application. If the Secret ID is returned by the call to NSSSEnumerateSecretIDs (page 74), the application can then proceed by calling NSSSReadSecret (page 84) and using the returned secret to log in the user to the application. A successful direct call to NSSSReadSecret returns the secret for the application.

If these calls denote failure or a non-existent Secret ID, the application prompts the user for authentication. If the user successfully authenticates, the application uses the authentication information in calls to NSSSWriteSecret (page 101) to create and populate the user's SecretStore for the next time (when the user launches the application).

The NSSSEnumerateSecretIDs (page 74) function provides the method for applications with multiple log in IDs to select the proper authentication method. Applications can then have the necessary configuration information to define a default login in case of multiple IDs.

Enabling the Check for Secret ID Collision

As an alternative for populating the SecretStore for the first time, NSSSWriteSecret (page 101) can be used with the proper flag to check for ID collision or to bypass the check and overwrite the secret if it already exists in the SecretStore. Use the NSSS_CHK_SID_FOR_COLLISION_F flag to enable NSSSWriteSecret for collistion checking.

Enabling the First-Time User

The first time a user authenticates to an application, no secret is available through SecretStore for that application, and the user is prompted to enter a password. Upon successful authentication to the application, the user is granted access and the application authentication information and secret is stored in an encrypted format in the SecretStore for subsequent use.

Password Changes

In the case of password administration where users are required to change their passwords after a certain number of days, a user doesn't have to remember a password that hasn't been recently used. This is done by enabling the password update code section in the application. SecretStore can retrieve and supply the old password to the application. The user then enters only the new password.

After the user changes the password, the new secret is captured and encrypted. The application then calls NSSSWriteSecret (page 101) to overwrite the existing secret, which updates SecretStore with the new credentials. This ensures seamless operation the next time the user launches that application. If the shared secret calls are used here, the secret will be updated in a format that allows all applications to use the secret and retain synchronization.

A user can use Novell's ConsoleOne management utility Version 1.3 or newer, in addition to SSManager.exe, to recall forgotten passwords or view secrets stored in the SecretStore.

eDirectory and SecretStore

SecretStore is a service that leverages the distribution operations built into eDirectory—user profile information is securely stored in eDirectory, not on the client machine. As a result, user access is guaranteed throughout the network and is not workstation based. And, since the user is the only person authorized to access his or her secrets, access is tightly controlled.

With SecretStore, passwords and credentials are never stored or transmitted in the clear; they are first encrypted using the Novell International Cryptographic Infrastructure (NICI). When the client application retrieves the secret from SecretStore, the secret is decrypted at the client side and, upon successful completion of the application's authentication process, is promptly destroyed and removed from memory (similar to how the eDirectory private key is handled in the eDirectory authentication process). See "NICI and SecretStore" on page 27.

To give users access to network services, eDirectory uses an authentication service based on the RSA public-key/private-key encryption/decryption algorithms. This authentication mechanism uses a private key attribute and a digital signature to verify a user's identity. eDirectory authentication is session-oriented, and the client's signature is valid only for the duration of the current session.

With SecretStore, however, once authenticated to eDirectory, users don't have to be reauthenticated every time they ask for additional services or applications, since reauthentication takes place automatically in the background. Therefore, the integrity of SecretStore-enabled applications is protected and secure, and the user can access resources globally without having to continuously reauthenticate.

In LDAP mode of SecretStore access, the client establishes a SSL-based connection with the server and the data is securely transmitted on the wire over SSL. However, as mentioned above, the data in the SecretStore on the server is encrypted through NICI.

NICI and SecretStore

All cryptographic services used for SecretStore are based on Novell International Cryptographic Infrastructure (NICI). NICI is used both in NCP-based transmission of secrets and for storage of secrets in eDirectory.

NOTE: Download and install NICI on your workstation independently if you are using Client 32 and NCP-based access.

The encryption performed on the server by the SecretStore service is divided into two separate processes:

- Storage encryption—stores user secrets in SecretStore within eDirectory.
- Transport encryption—securely transmits secrets across the network between clients and servers.

Storage Encryption

Data stored in SecretStore is first encrypted with the Security Domain Infrastructure™ (SDI) NICI-wrapped key. (SDI is initialized the first time SecretStore is installed into the tree.) This means that the authenticated owner that has rights to SecretStore can use the SecretStore API to decrypt the application's secrets.

The SecretStore service uses NICI to encrypt data that is stored in or retrieved from eDirectory. When the data stored in SecretStore is first created, a symmetric key is generated and stored for the user. This key is wrapped by NICI SDI in a special storage format.

Subsequent reads and writes to the SecretStore cause the key to be unwrapped and used to encrypt and decrypt the stored data. Reading and writing negotiate a supported algorithm for decrypting and encrypting the data. SecretStore always picks the highest strength algorithm available through NICI policies for encrypting the data. Currently this algorithm is 3DES for worldwide usage.

Reading and Writing Encryption

SecretStore secrets are encrypted and transmitted over the network with a NICI algorithm and the key for this algorithm is encrypted in a NICI session key when using NCP. This key is established between the NICI on the client and NICI on the server at the end of the Client 32 user authentication session. This happens independently from the SecretStore in the end of the eDirectory authentication.

The SecretStore NCP client’s wire encryption requests a NICI session key for the server with which it wants to communicate. Then an encryption key for encrypting the data with the common algorithm is negotiated between SecretStore client and SecretStore server. The data is encrypted in the encryption key, and then the encryption key is wrapped in the session key. Both the encrypted data and the wrapped encryption key are sent over the wire to the server where the encryption key is unwrapped with the session key and the data is decrypted with SecretStore wire encryption key.

NOTE: The encryption algorithm is negotiated between two ends beforehand and highest-strength common algorithm between the client and server is picked for the operation. The algorithm ID is transmitted with the encrypted data,

Remember, in the LDAP mode of SecretStore operations, the client establishes a SSL-based connection with the server and the data is securely transmitted on the wire over SSL.

Session-Oriented Security

SecretStore uses session-oriented security, which means that each internal call from the enabled applications is considered a session from beginning to end. After one session is completed, the keys are destroyed and a new set of keys are established for the next client request.

Naming Conventions

To prevent name collisions in SecretStore, we recommend that non-shared secrets be constructed using the following scheme to assure that names are unique. Notice that our naming guidelines do not prevent you from using an alternate scheme, which may result in secret names that are not unique. Alternative naming schemes are not able to prevent a name collision between non-registered applications.

IMPORTANT: To manage and coordinate SecretStore secret ID changes worldwide, it is important to register your names with Novell Developer Support. Please see the Virtual Resources site (http://developer.novell.com/devres/ss/resource.htm) for further information.

The length of the secret name must be 255 or fewer characters long. Secret IDs should follow this structure:

`//<Company DNS Name>/<AppName>:OptionalComponents`

`<Company DNS Name>`

The registered domain name of the company that has developed the SecretStore-enabled application, such as novell.com or ibm.com.

`<AppName>`

Name of the SecretStore-enabled application.

`<OptionalNamecomponents>`

This can be any sequence of name components that an application chooses to define and use.

Naming Examples

Here are three examples of SecretStore secret IDs:

- //Novell.com/LotusNotes:c:/Lotus/jsmith.id
- //Novell.com/ EntrustEntelligence:c:/Program Files/Entrust/entrust.epf
- //Novell.com/GroupWise:ver.0.jdoe

Enhanced Protection

SecretStore includes an optional enhanced protection setting that a user can apply to each secret. Enhanced protection means that a user's secret is not only protected from the view of others, but it becomes locked when an administrator changes the user's eDirectory password or other login credentials.

For example, suppose a network administrator attempted to view another user's secrets by changing the user's eDirectory password and then logging in as that user using the new password. The administrator would be unable to view the secrets that are flagged with enhanced protection because the SecretStore becomes locked. If the administrator attempted to unlock the secrets, he or she would be prompted to enter the user's previous eDirectory password, rather than the new one that the administrator created.

Secrets can be unlocked with the last valid eDirectory password that the user entered using a user-initiated Change Password request. As a result, even if the user or administrator changes the eDirectory password several times after the secrets have been locked, the status of the locked secrets is not affected.

The other method of unlocking the SecretStore is to supply the Master Password for SecretStore. Master Password is designed to prevent the loss of secrets when administrator password change has been the result of the user forgetting the eDirectory password. Master password should be set by the user before SecretStore gets locked. Master Password also is the method to unlock the store if non-password login credentials were changed by the administrator (such as issuing new Smart Card™ or Proximity Card™, etc.).

In addition, SecretStore supports an optional password to be validated before access to a particular application secret is granted. These optional passwords are set by applications that are used in conjunction with the enhanced protection feature as an additional access control on the secret. A password can be used at secret creation time and should be supplied at retrieval time. These passwords can be a maximum of 64 characters long. This additional access control method allows an application to prevent other applications of the same user to read its secret.

To read those secrets that have the enhanced protection feature turned on and that use the optional enhanced protection password feature, the application must provide the password for reading the secret out of the SecretStore. Failure to provide the correct password that was used at the time of creation of the secrets results in NSSS_E_EP_ACCESS_DENIED error. Consequently, the applications or rouge programs can't read enhanced protection enabled secrets with enhanced protection passwords on them.

Another feature for Enhanced Protection is the ability to create hidden secrets. These secrets are not shown when NSSSEnumerateSecretIDs (page 74) is called and NSSSGetServiceInformation (page 79) returns their count. An application that creates a hidden secret can only read that secret by supplying its SecretID that is known only to that application and was used to create the secret. Without knowing the secret ID the only way to remove a hidden secret is to remove the SecretStore.

SecretStore Service Discovery

The SecretStore client library must be able to locate a SecretStore server that can service its requests before any operations can be performed. A server is only able to service a request from the client if it meets these criteria:

- The server is running the SecretStore service.
- The server contains a writable replica of the partition that contains the user whose secrets are being operated on.

The service discovery process used in the SecretStore API avoids the shortcomings of methods available in earlier versions of the service. In addition, it provides additional flexibility by allowing the administrator to specify a particular set of service configuration parameters that should be used when accessing the SecretStore service.

Distinguished Name Attribute

A new attribute, *sssSingleSignonConfigDN*, has been designated for specifying the distinguished name of the SecretStore object that contains the configuration parameters the user should use. This attribute can be added by the administrator to user objects and containers. During service discovery, the client uses this attribute in its attempts to find a server that supports the specified configuration settings, as described below.

When the SecretStore service is loaded on a server, it builds a cache of the available configurations for different containers it can support (defined by the administrator). This cache can be built using a command line parameter telling it the distinguished name of a SecretStore override object that should be used for obtaining configuration parameters (see NSSSGetServiceInformation (page 79) and NSSSReadSecret (page 84) for a description of these parameters).

While loading, the service adds a special service information attribute to the root of its partition. This attribute contains the distinguished name of the server, its IP address, and the distinguished name of the specified configuration object, if any. When the service is unloaded, it removes this attribute from the root of the partition. Using this technique, clients have a well-known location where they can find information about available SecretStore servers that can possibly support their configuration.

Locating an Acceptable SecretStore Service

To locate an acceptable SecretStore service, the client first determines if the specified user should use a particular configuration by looking for the *sssServerPolicyOverrideDN* attribute on the user object and its parent containers. When this attribute is found, the distinguished name of the specified configuration object is read into memory.

Next, the client reads the service information attribute (added by SecretStore services while loading) from the root of the partition. It then looks at each value of that attribute for a server name that has a matching configuration object's distinguished name (if one was found on the user object or one of its parent containers). If no configuration object distinguished name is required, any server in the list will match.

As soon as a matching entry is found, the client pings the server to see if it is indeed available. If not, it continues looking at attribute value entries and pinging servers until a server is found that responds, or until it has exhausted all attribute value entries.

Using this method, the client avoids the need to traverse the Directory tree looking for servers that contain a writable replica of the partition. Also, the administrator can configure objects that use SecretStore in such a way as to prevent unnecessary traffic over expensive WAN links.

Using Extension Structures

Each function in the NSSS API can accept an optional parameter of type SS_EXT_T. This parameter has been included to provide access to extended functionality in the API for future expansions without having to change the interface.

```
// optional extension structure

typedefstruct_ss_extension

{

unsigned long clientVersion;//* IN

void *extParms;//* IN-pointer to optional data

}SS_EXT_T;
```

Extension Structure Levels

There are three levels of extension structures:

1. The first level is simply the SS_EXT_T structure. It provides a generic way to pass to each API function the address of a structure that extends the functionality of that particular function.

2. The second level of extension structures consists of all the function-specific structures, whose address is placed in the extParms field of the function specific structure prior to calling the API functions. Currently the version 3.2 of SecretStore only have extensions defined for the NSSSGetServiceInformat and NSSSReadSecret API set to optionally return statistical information.

NSSO Function	Extension Structure
NSSSGetServiceInformation (page 79)	SS_GSINFOEXT_T
NSSSReadSecret (page 84)	SS_READEXT_T
NSSSWriteSecret (page 101)	SS_WRITEEXT_T
NSSSRemoveSecret (page 87)	SS_REMEXT_T
NSSSUnlockSecrets (page 98)	SS_UNLOCKEXT_T
NSSSRemoveSecretStore (page 89)	SS_REMSTOREEXT_T
NSSSEnumerateSecretIDs (page 74)	SS_ENUMEXT_T
NSSSSetEPMasterPassword (page 96)	SS_SETMPEXT_T

NOTE: Please refer to the descriptions in "Functions" on page 65 for a discussion of the fields in each of the extension structures mentioned above.

Using Extension Structures

An example of how to use these extension structures is provided in the file *SSTEST.C*. Typically, the steps for using the three levels of extension structures are:

1. Allocate storage for the extension structures. This can be done on the stack or as global variables. Typically it makes sense to have only one SS_SERVER_INFO_T structure used by all the function-specific extension structures since it was designed to preserve information between function calls. Note that the structures are initialized as they are allocated in the following code fragment:

```
/* Structure passed to all API functions. */

SS_EXT_T ext = {0};

/* Example of structures for individual API functions */

SSS_READEXT_T readInfo = {0};

SSS_GSINFOEXT_T gsInfo = {0};
```

2. Initialize the extension structures. Note that all function-specific extension structures point to the same server information structure:

```
readInfo.ssServerInfo = &serverInfo;

gsInfo.ssServerInfo = &serverInfo;
```

3. Prepare to call an API function by placing the address of the function-specific extension structure into the extParms field of the generic extension structure:

```
ext.clientVersion = NSSS_VERSION_NUMBER;

ext.extParms = &gsInfo;
```

4 Call the desired API function.

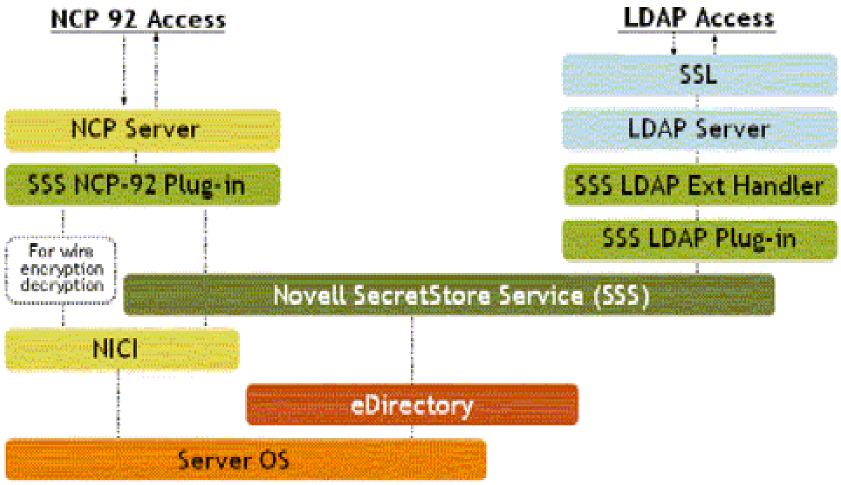
NOTE: If all of the state data (such as changes to context or server information, etc. that can be obtained through an initializing call to NSSSGetServiceInformation (page 79) and used on subsequent calls) are not provided, each API call will initialize the information automatically on entry and destroy it on return. So, when enabling an application that only requires a read from SecretStore, the overhead of initialization is neither required nor recommended through calling NSSSGetServiceInformation.

2 Tasks

Novell SecretStore leverages Novell eDirectory and Novell International Cryptographic Infrastructure (NICI) to securely store and retrieve user authentication information. The SecretStore client application makes read and write calls to SecretStore services on the server, which processes and executes the requests. User secrets (such as the username and password) are encrypted by SecretStore using NICI encryption and stored as eDirectory hidden attributes.

All requests between client and server take advantage of the authenticated credentials established between the client and server after login. SecretStore secure NCP uses NICI ephemeral session keys to guarantee confidentiality and integrity of the user data.

Figure 3 NCP92 Single Sign-on Using SecretStore



The diagram above shows the basic steps used in a NCP92 single sign-on session with SecretStore:

1. The SecretStore-enabled application client requests authentication secrets from the server.
2. The request is sent to SecretStore on the server over the encrypted channel.
3. SecretStore receives the request and retrieves the data from eDirectory.
4. SecretStore decrypts the secret data and sends back the data to the SecretStore client over the same secure connection.

NOTE: LDAP-based access to SecretStore estblishes a SSL-based connection to the target server after a successful bind and does not use NICI for wire encryption.

Displaying a Splash Screen

It is a good idea for SecretStore-enabled applications to have a splash screen appear during automatic login to notify the user of the events that have transpired under cover (that is, when the enabled application executes the NSSSReadSecret (page 84) function). The screen signifies to the user that SecretStore has authenticated them automatically to the application.

Enabling and Maintaining SecretStore

SecretStore can be installed in two different ways, both of which require that the user be logged in and authenticated to eDirectory as admin:

- Use the product installation CD to run the client and server installs to upgrade the server and client to support SecretStore.
- Manual installation of SecretStore components through the Novell Development Kit (NDK) complying with SecretStore requirements. For more information, see "Implementation Dependencies" on page 24.
- Server installation will not be required for eDirectory version 8.7.2 and newer.

Server Installation for NetWare and Windows

1. Login as the admin to the server.
2. Copy sss, ssncp, ssldp, and lsss components to SYS:SYSTEM directory on NetWare or Novell\NDS directory on Windows.
3. While logged in as admin, make the target server your primary tree and connection through Network Neighborhood connection table tab.
4. Run sssinit.exe supplied by the NDK toward the target server to extend the schema.
5. Run ldap.exe supplied by the NDK toward the target server to add LDAP extensions to the target server. /h or /H or /? Command line options will give you the usage information on this program
6. Modify autoexec.ncf to add the following lines on NetWare in the order listed below to load the SecretStore NCP plugin:

   ```
   load ssncp.nlm
   ```

   The NLDAP server automatically loads the LSSS extension manager, which loads ssldp plugin, then sss server.
7. On Windows Servers, use the eDirectory Console to make these components autoload.

   NOTE: The install program extends the schema on the user object to add SecretStore by default.
8. Once SecretStore is installed and the schema has been extended, SecretStore can be tested by executing sstst.exe for NCP or lstst.exe for LDAP access. The source code for these test programs is provided in the NDK for developers use.
9. SecretStore operations can also be tested through ssmanager.exe that is supplied in the SDK.

Development Dependencies

- Server Prerequisites (page 37)

- Workstation Prerequisites (page 37)

Server Prerequisites

- Novell International Cryptographic Infrastructure (NICI) software Version 2.4.x or newer. We recommend that the target servers be upgraded to the latest version of NICI available through Novell's cryptography support web page (http://www.novell.com/products/cryptography).

  A version of NICI is supplied with the SecretStore installation as an option.

  To verify that your server meets this requirement before continuing, type the following at the server console:

  ```
  MODULES XENG* <Enter>
  ```
- Supervisor rights to the [Root] object of the eDirectory tree to install SecretStore.
- Novell Support Pack 1 or later for NetWare 5.0.

Workstation Prerequisites

- Novell Client32 for Windows NT/2000/XP or a Novell Client for Windows 95/98/ME workstation where the NCP client is installed.
- Novell LDAP SDK components where the SecretStore LDAP client is installed.
  - For NCP installation, copy the nsss and nssncp components to the Windows SYSTEM32 directory.
  - For LDAP installation, copy nsss and nssldp components where the shared libraries are supposed to go on the target client operating system.
- In the Windows-based client, supply the necessary registry keys to turn on and off the logger capability on the client DLLs. When turned on and off through SSManager.exe under the test tab, this diagnostic tool creates a NSSS.LOG file that writes a diagnostic trace.

Enabling SSOCOMP in Applications

Follow these steps to enable SecretStore within your applications (also see Chapter 1, “Concepts,” on page 11):

1. Get Service Information (page 37)
2. Read Available Secrets (page 38)
3. Verify Connection to Proper Tree (page 38)

Get Service Information

Enabling an application to use SecretStore requires making a call to NSSSGetServiceInformation (page 79). A tree name also could be included with the context structure with the call.

For use, refer to sststst.c or lstst.c in “Using Extension Structures” on page 32.

NSSSGetServiceInformation informs you whether SecretStore has been enabled, who you are logged in as, and provides information about the SecretStore you work with.

Read Available Secrets

Once you obtain service information, you need to read the available secrets by calling NSSSEnumerateSecretIDs (page 74). If you're looking for a particular application, you know what your Secret ID is.

If the specified ID exists in the SecretStore, call NSSSReadSecret (page 84) with the specified ID to obtain the secret from SecretStore. The returned Secret can then be supplied to your SecretStore-enabled application to complete authentication.

Verify Connection to Proper Tree

When a call is made to SecretStore, verification is made to ensure that you are connected to the tree you have named. If you have not designated a tree, it is assumed the primary connection is being used.

Once an authenticated connection is verified, a search is made for a SecretStore server in that tree that can handle your request. After discovering the name of a suitable SecretStore server, the server is pinged to ensure it is on line and able to process the request. The call can then be made and the requested information returned.

This procedure is repeated for each API. Each API sets up a separate secure session with the server. The process underlying each call made to the API includes the following:

- Establishing a secure connection to the server
- Passing the data over the secure connection to the server.
- Tearing down the connection or continuing with other operations, then tearing down the connection.

None of these processes should impact performance significantly.

Through the SecretStore technology, secrets (that is, passwords and other authentication credentials) are securely encrypted at all times, whether in transmission or in storage, and accessible only by the owner of the data. Since the data type is irrelevant, passwords and tokens can be safely stored and retrieved to enable a true single login experience for all applications. See "NICI and SecretStore" on page 27.

Writing Shared Application or Credential Secrets

1. Call NSSSCreateSHSHandle (page 72) for each application thread that is involved in the sharing of secrets. A void pointer is returned that provides a handle for passing subsequent calls.
2. Call NSSSReadSharedSecret (page 82) to pass in the handle created in Step 1 and a user-populated SS_SH_SECRET_ID_T structure that contains the shared secret type, name, and length.

   NOTE: You also pass in the populated SSS_CONTEXT_T and SS_OBJECT_DN_T structures; ssFlags; and the SS_PWORD_T, SSS_READEXT_T, and SS_EXT_T structures that are typically provided when calling NSSSReadSecret (page 84). Consequently, NSSRReadSharedSecret enables you to make calls to trees and user DNs outside of the primary connection.
3. Internally, NSSSReadSharedSecret (page 82) calls NSSSReadSecret (page 84) to retrieve secret data stored on SecretStore.

   3a The secret data is parsed according to the shared secret format using the parsing library.

3b Sequential internal calls are made to enter key and value data into a linked list.

4 Call NSSSAddSHSEntry (page 70) sequentially to enter key or value data into the linked list (that is, the list associated with the handle).

NOTE: This function contains pointers to user-allocated key and value buffers and the unsigned long context flag member of the SSS_CONTEXT_T structure populated from calling NSSSGetServiceInformation (page 79).

5 Call NSSSWriteSharedSecret (page 104) to pass a handle to write a shared secret, as well as a user-populated SS_SH_SECRET_ID_T structure containing the share secret type, name, and length. This creates a secret ID according to the secret ID format using either the prefix SS_App or SS_CredSet.

NOTE: You also pass in the populated SSS_CONTEXT_T and SS_OBJECT_DN_T structures, ssFlags, and the SS_PWORD and SS_EXT_T structures that are typically passed when calling NSSSWriteSecret (page 101). Consequently, NSSRWriteSharedSecret enables you to make calls to trees and user DNs outside of the primary connection.

6 NSSSWriteSharedSecret (page 104) makes sequential internal calls to populate an internal buffer with data retrieved from the linked list.

6a The internal buffer is parsed according to the shared secret format using the parsing library.

6b Parsed data is passed into the secret buffer, then passed to NSSSWriteSecret (page 101) in the SecretStore client.

7 Before exiting the application, call NSSSDestroySHSHandle (page 73) to free memory associated with the handle of each shared secret thread.

NOTE: A complete set of operations is demonstrated in sshtst.exe and lshtst.exe and related source files for these executables are available in the SDK. In addition to these executable programs, SSManager.exe can be used to create, test, and view shared and raw (non-shared) secrets.

Reading Shared Application or Credential Secrets

1 For each user-defined application thread involved in secrets sharing, call NSSSCreateSHSHandle (page 72) to obtain a *void* pointer as a handle to parse subsequent calls.

2 Call NSSSReadSharedSecret (page 82) to pass the handle and a user-populated SS_SH_SECRET_ID_T structure containing the shared secret type, name, and length.

NOTE: You also pass the populated SSS_CONTEXT_T and SS_OBJECT_DN_T structures; ssFlags; and the SS_PWORD_T, SSS_READEXT_T, and SS_EXT_T structures that are normally passed into the call to NSSSReadSecret (page 84). Consequently, NSSReadSharedSecret enables you to call trees and user DNs outside of the primary connection.

3 Internally, NSSSReadSharedSecret (page 82) calls NSSSReadSecret (page 84) to retrieve secret data stored on SecretStore. This function contains the handle and pointers to internally allocated key and value buffers.

3a The internal secret data is parsed according to the shared secret format using the parsing library.

3b Sequential internal calls are made to enter key and value data into a linked list.

4 Call NSSSGetNextSHSEntry (page 77) sequentially. This function returns the handle, the unsigned long context flag from the user-populated SSS_CONTEXT_T structure passed into NSSSReadSharedSecret (page 82), and pointers to user-allocated key value buffer and length parameters.

5 Use the data returned for application-specific tasks.

6 Before exiting the application, call NSSSDestroySHSHandle (page 73) to free memory associated with the handle of each shared secret thread.

Modifying Shared Application or Credential Secrets

To modify shared secrets in applications or credentials, you must be able to add and remove the keys used to secure secrets stored in a user's SecretStore as explained in the following sections.

- Adding A Shared Secret Key
- Removing A Shared Secret Key

Adding A Shared Secret Key

1 For each user-defined application thread involved in secrets sharing, call NSSSCreateSHSHandle (page 72) to obtain a *void* pointer as a handle to parse subsequent calls.

2 Call NSSSReadSharedSecret (page 82) to pass in the handle created in Step 1 and a user-populated SS_SH_SECRET_ID_T structure that contains the shared secret type, name, and length.

NOTE: You also pass in the populated SSS_CONTEXT_T and SS_OBJECT_DN_T structures; ssFlags; and the SS_PWORD_T, SSS_READEXT_T, and SS_EXT_T structures that are typically provided when calling NSSSReadSecret (page 84). Consequently, NSSRReadSharedSecret enables you to make calls to trees and user DNs outside of the primary connection.

3 Internally, NSSSReadSharedSecret (page 82) calls NSSSReadSecret (page 84) to retrieve secret data stored on SecretStore. This function contains the handle and pointers to internally allocated key and value buffers.

3a The secret data is parsed according to the shared secret format using the parsing library.

3b Sequential internal calls are made to enter key and value data into a linked list.

4 Call NSSSAddSHSEntry (page 70) sequentially to enter key or value data into the linked list.

NOTE: This function contains pointers to user-allocated key and value buffers and the unsigned long context flag member of the SSS_CONTEXT_T structure populated from calling NSSSGetServiceInformation (page 79).

5 Call NSSSWriteSharedSecret (page 104) to pass a handle to write a shared secret, as well as a user-populated SS_SH_SECRET_ID_T structure containing the share secret type, name, and length. This creates a secret ID according to the secret ID format using either the prefix SS_App or SS_CredSet.

NOTE: You also pass in the populated SSS_CONTEXT_T and SS_OBJECT_DN_T structures, ssFlags, and the SS_PWORD_T and SS_EXT_T structures that are typically passed when calling NSSSWriteSecret (page 101). Consequently, NSSRWriteSharedSecret enables you to make calls to trees and user DNs outside of the primary connection.

6 NSSSWriteSharedSecret (page 104) makes sequential internal calls to retrieve data from the link list and populate an internal buffer.

6a The internal buffer is parsed according to the shared secret format using the parsing library. The resultant data is passed into the Secret buffer for passage to NSSSWriteSecret (page 101) in the SecretStore client.

6b The function makes an internal call to store the Secret buffer as a shared secret in SecretStore using NSSSWriteSecret.

7 Before exiting the application, call NSSSDestroySHSHandle (page 73) to free memory associated with the handle of each shared secret thread.

Removing A Shared Secret Key

1 For each user-defined application thread involved in secrets sharing, call NSSSCreateSHSHandle (page 72) to obtain a *void* pointer as a handle to parse subsequent calls.

2 Call NSSSReadSharedSecret (page 82) to pass in the handle created in Step 1 and a user-populated SS_SH_SECRET_ID_T structure that contains the shared secret type, name, and length.

NOTE: You also pass in the populated SSS_CONTEXT_T and SS_OBJECT_DN_T structures; ssFlags; and the SS_PWORD_T, SSS_READEXT_T, and SS_EXT_T structures that are typically provided when calling NSSSReadSecret (page 84). Consequently, NSSRReadSharedSecret enables you to make calls to trees and user DNs outside of the primary connection.

3 Internally, NSSSReadSharedSecret (page 82) calls NSSSReadSecret (page 84) to retrieve secret data stored on SecretStore. This function contains the handle and pointers to internally allocated key and value buffers.

3a The secret data is parsed according to the shared secret format using the parsing library.

3b Sequential internal calls are made to obtain key and value data passed into a linked list.

4 For each key you want to remove, call NSSSRemoveSHSEntry (page 94) to remove the key-value pair. This call contains the handle, pointers to user-allocated key and value buffers, and the unsigned long context flag member of the SSS_CONTEXT_T structure populated from calling NSSSGetServiceInformation (page 79).

5 Call NSSSWriteSharedSecret (page 104) to pass a handle to write a shared secret, as well as a user-populated SS_SH_SECRET_ID_T structure containing the share secret type, name, and length. This creates a secret ID according to the secret ID format using either the prefix SS_App or SS_CredSet.

NOTE: You also pass in the populated SSS_CONTEXT_T and SS_OBJECT_DN_T structures, ssFlags, and the SS_PWORD_T and SS_EXT_T structures that are typically passed when calling NSSSWriteSecret (page 101). Consequently, NSSRWriteSharedSecret enables you to make calls to trees and user DNs outside of the primary connection.

6 NSSSWriteSharedSecret (page 104) makes sequential internal calls to retrieve data from the linked list and populate an internal buffer.

6a The internal buffer is parsed according to the shared secret format using the parsing library. The resultant data is passed into the Secret buffer for passage to NSSSWriteSecret (page 101) in the SecretStore client.

6b The function makes an internal call to store the Secret buffer as a shared secret in SecretStore using NSSSWriteSecret.

7 Before exiting the application, call NSSSDestroySHSHandle (page 73) to free memory associated with the handle of each shared secret thread.

Removing Shared Application or Credential Secrets

1. Call NSSSRemoveSharedSecret (page 91) to pass in a user-populated SS_SH_SECRET_ID_T structure that contains the shared secret type, name, and length.

   NOTE: You also pass in the populated SSS_CONTEXT_T and SS_OBJECT_DN_T structures; ssFlags; and the SS_EXT_T structures that are typically provided when calling NSSSRemoveSharedSecret. Consequently, NSSSRemoveSharedSecret enables you to make calls to trees and user DNs outside of the primary connection.

2. Internally, the Secret Identifier is parsed according to the shared secret format designated for handling delimited characters.

   2a NSSSRemoveSharedSecret (page 91) calls NSSSRemoveSecret to remove shared secrets from SecretStore.

SecretStore Sample Code

See the Single Sign-on sample code (../../../samplecode/ssocomp_sample/index.htm) to view and download examples of the SecretStore API used in client applications.

3 Secret Store Structures

Novell SecretStore interfaces are based on the standard C-language types . SecretStore prototypes and data types are defined in nssscl.h. The topics you should understand that relate to the interface include the following:

- Structured Definitions (page 43)
- Service Location Information (page 50)
- Shared Secret Structures (page 54)

Structured Definitions

All of the character arrays in the following structures have the maximum length in bytes, but applications should pass in strings with the number of characters that are half the length of these character arrays. These arrays are double the size in length for unicode strings. Reference the length in characters in "API Function Flags" on page 66.

The structure definitions include the following:

- SSS_CONTEXT_T (page 44)
- SS_EXT_T (page 46)
- SS_HINT_T (page 47)
- SS_PWORD_T (page 48)
- SS_SECRET_ID_T (page 49)
- SS_SECRET_T (page 50)
- SS_SERVER_INFO_T (page 51)
- SS_SH_SECRET_ID_T (page 55)
- SS_ADV_BIND_INFO_T (page 58)
- SS_ADDR_T (page 57)
- SS_ADV_CRED_T and SS_ADV_CERT_T (page 56)
- SS_OBJECT_DN_T (page 59)

SSS_CONTEXT_T

Contains the optional Directory Services context. Passing a NULL pointer on the APIs causes the system to internally get the proper user context. Passing an initialized version of this structure to the SecretStore functions enables SecretStore to retain the state between calls and establishes an ongoing session between SecretStore's client and server.

Syntax

```
typedef struct SSS_CONTEXT_T
{
    unsigned long   flags;
    unsigned long   dsCtx;
    unsigned long   version;
    SS_SERVER_INFO_TssServerInfo;
    SS_OBJECT_DN_T  callerDN;
    SSS_HANDLES_T  *handles;
    void           *bindInfo;
} SSS_CONTEXT_T;
```

Fields

flags (IN)

Specifiess what type of context was passed in:

Flag	Description	Action
NSSS_NCP_CTX_F	NCP context type (dsCtx)	
NSSS_LDAP_CTX_F	LDAP context type (dsCtx)	
NSSS_CONTEXT_INITIALIZED_F	The context already is initialized.	Context could be initialized outside the SecretStore client or inside through a previous call to NSSSGetServiceInformation (page 79)
NSSS_INIT_LDAP_SS_HANDLE_F	Initializes the handle's structure without a bind.	In the case of context having been initialized outside SecretStore the caller must pass this flag to allocate and initialize the internal handles structure with necessary state data
NSSS_DEINIT_LDAP_SS_HANDLE_F	Deinitializes the handle's structure without unbind.	In the case of context having been initialized outside SecretStore the caller must pass this flag to free the internal handles structure after the state data is no longer needed
NSSS_NSSS_REINIT_TARGET_DN_F	Reinitializes the target DN saved in the context.	In the case of client switching between different eDirectory trees this flag should be used to update the copy of Target DN saved in the context
NSSS_LDAP_CONTEXT_LESS_DN_F	Resolves LDAP DN as contextless.	Indicates that a contextless DN is passed on the request for bind and prompts the SecretStore to search the tree to find the full DN of the user. In case of multiple DNs containing the common

Flag	Description	Action
NSSS_ADV_BIND_INFO_T	Use advanced bind structure and run service location.	Indicates that the advanced bind structure is passed in and SecretStore is prompted to perform a service location to find the proper target server for the user in the eDirectory tree.

dsCtx

Specifies the Directory Service NCP/LDAP context.

version

Points to the LDAP Context (reserved for future).

epPword (optional IN)

Encodes the actual clear password supplied.

Description

This is an optional parameter that allows the SecretStore client to keep a stateful session with the SecretStore server based on the session-related information stored in this context.

SS_EXT_T

This is an optional structure for a passing extended information to the functions.

Syntax

```
typedef struct
{
    unsigned long   clientVersion;
    void           *extParms;
} SS_EXT_T;
```

Fields

clientVersion

(IN) Specifies the version of the SSS client in NSSS_VERSION_NUMBER.

extParms

(IN) Depending on the call and version passed, points to an API specific extension structure. It is the caller's responsibility to initialize this pointer if the SS_EXT_T structure is used to pass extensions that an API calls.

Description

This is an optional parameter that can be defined by the service and be used in the future to extend each API. The contents of the structure that extParms points to can change based on the version of the service. Caller can pass a NULL in place of this parameter.

SS_HINT_T

Structure used to encode the user's Master Password hint.

Syntax

```
typedef struct SS_HINT_T
{
    unsigned long   hintLen;
    char            hint[NSSS_MAX_MP_PWORD_HINT_LEN];
} SS_HINT_T;
```

Fields

hintLen

Specifies the length of enhanced protection and password to set in bytes.

hint[NSSS_MAX_MP_PWORD_HINT_LEN]

Specifies the password characters string that should be passed in.

Description

This identifies the structure that can be used for encoding of the hint on related functions.

SS_PWORD_T

Structure used to encode the user's Password for SecretStore use.

Syntax

```
typedef struct SS_PWORD_T
{
    unsigned long   pwordLen;
    char            pword[NSSS_MAX_EP_PWORD_LEN];
} SS_PWORD_T;
```

Fields

pwordLen

Specifies the length of enhanced protection and password to set in bytes.

pword[NSSS_MAX_EP_PWORD_LEN]

Specifies the number of password characters should be passed in.

Description

This structure can be used for encoding of the password on related functions.

SS_SECRET_ID_T

Structure used to encode the Secret ID for SecretStore use.

Syntax

```
typedef struct SS_SECRET_ID_T
{
    long   len;
    char   id[NSSS_MAX_SECRET_ID_LEN];
} SS_SECRET_ID_T;
```

Fields

len

Specifies the maximum length of the ID in bytes.

id

Specifies the actual ID, including the terminating char. This is the length in bytes when the id is passed in local code page. If the string is already in Unicode then the id can accommodate the unisze(id) <= NSSS_MAX_SECRET_ID_LEN.

Description

This structure can be used for encoding of Secret ID on related APIs.

SS_SECRET_T

Structure used to encode the Secret for SecretStore use.

Syntax

```
typedef struct SS_SECRET_T
{
long    len;
char   *data;
} SS_SECRET_T;
```

Fields

len

Specifies the length of the data in bytes.

data

Points to the actual data of data.

Description

This structure can be used for encoding or retrieval of secrets on related functions.

Service Location Information

This SSS_CONTEXT_T contains the SS_SERVER_INFO_T structure for service location information:

SS_SERVER_INFO_T

Holds information about the server which services the SecretStore requests, and helps to accelerate access to SecretStore and provide multi-tree capabilities.

Syntax

```
typedef struct _ss_server_info
{
 char   treeName[ NSSS_MAX_TREE_NAME_LEN ];
 char   ssServerDN[ NSSS_MAX_DN_LEN ];
 char   ssServerIPAddr[ NSSS_MAX_IP_ADDR_LEN ];
 char   sssConfigDN[ NSSS_MAX_DN_LEN ];
} SS_SERVER_INFO_T;
```

Fields

treeName

(IN/OUT) Specifies the field in which the name of the tree is designated where server is located.

ssServerDN

(IN/OUT) Specifies the designated name of the server where SecretStore is located.

ssServerIPAddr

(OUT) Specifies the IP address of the SecretStore server.

ssoConfigDN

(IN/OUT) Specifies the distinguished name of the SecretStore configuration or override configuration objects.

Description

When the address of this structure is provided and contains valid data, the API uses the data to bypass the service discovery process and directly contact the server that serviced the last API function call. Also, rather than rely on the primary connection to identify the tree in which the target user resides, the tree name that is stored in this structure and can be changed by the programmer to allow calls to be made to any tree to which the user is authenticated.

SSS_GSINFOEXT_T

Gets service info extended optional data modified for version 0x00000205.

Syntax

```
typedef struct _ss_get_service_information_extension
{
unsigned long      statFlags;
unsigned long      secretCount;
unsigned long      lockCount;
unsigned long      enumBufLen;
unsigned long      hidSecCount;
unsigned long      clientVersion;
unsigned long      serverVersion;
unsigned long      serverCryptoStrength;
unsigned long      clientCryptoStrength;
unsigned long      unlockTStamp;
unsigned long      admnDNLen;
char               admnDN[NSSS_MAX_DN_LEN];
unsigned long      hintLen;
char               hint[NSSS_MAX_MP_PWORD_HINT_LEN]];

} SSS_GSINFOEXT_T;
```

Fields

statFlags

(OUT) Specifies the return flags on the SecretStore.

secretCount

(OUT) Specifies the number of secrets in the SecretStore.

lockCount

(OUT) Specifies the number of locked secrets.

enumBufLen]

(OUT) Specifies the Secret ID enumeration buffer length in bytes.

hidSecCount

(OUT) Specifies the number of hidden secrets

`clientVersion`

(OUT) Specifies the version of the SecretStore client.

`serverVersion`

(IN) Specifies the version of the SecretStore server:

`serverCryptoStrength`

(IN) Specifies the cryptographic strength of the server:

- NSSS_NICI_DOMESTIC_ENGINE—3DES or stronger
- NSSS_NICI_EXPORT_ENGINE—DES or stronger

clientCryptoStrength

(OUT) Specifies the cryptographic strength of the client:.

unlockTStamp

(OUT) Specifies the time stamp of the last administrative unlocking of the SecretStore on a two admin unlocking scheme by a SecretStore Administrator for audit purposes.

admnDNLen

(OUT)) Specifies the length of the admin DN of the last SecretStore administrator that has unlocked the SecretStore on a two admin unlocking for audit purposes.

admnDN

(OUT) Specifies the admin DN of the last SecretStore administrator that has unlocked the SecretStore on a two admin unlocking for audit purposes.

hintLen

(OUT) Specifies the length of the master password hint.

hint`[NSSS_MAX_MP_PWORD_HINT_LEN]`*]*

(OUT) Specifies the actual password hint.

Description

This structure returns the status information on the target user's SecretStore.

SS_READEXT_T

Reads extended optional data.

Syntax

```
typedef struct _ss_read_extension
{
unsigned long       statFlags;
unsigned long       crtStamp;
unsigned long       latStamp;
unsigned long       lmtStamp;

} SS_READEXT_T;
```

Fields

statFlags

(OUT) Specifies the return flags on the secret.

crtStanp

(OUT) Specifies the secret creation time stamp.

latStamp

(OUT) Specifies the last accessed time stamp (optional).

lmtStamp

(OUT) Specifies the last modified time stamp.

Description

This is the optional extension that can be passed on the NSSSReadSecret (page 84) API to obtain information on a secret.

Shared Secret Structures

The following shared secret structure is defined:

SS_SH_SECRET_ID_T (page 55)

SS_SH_SECRET_ID_T

Defines the structure for encoding a Shared Secret ID.

Syntax

```
typedef struct struct _ss_sh_secret_id
{
int    type;
char   pName[NSSS_MAX_SECRET_ID_LEN];
int    length;

} SS_SH_SECRET_ID_T;
```

Fields

type

Specifies the type of secret to be shared (that is, SS_App or SS_CredSet).

pName[NSSS_MAX_SECRET_ID_LEN]

Specifies the name of the shared secret.

length

Specifies the length in characters of the shared secret's name, including the NULL terminator. This is the same value as used by the identifier.

Description

This structure is used by the SharedSecret functions for encoding the a Shared Secret ID.

SS_ADV_CRED_T and SS_ADV_CERT_T

This is the generic structure used by SS_ADV_BIND_INFO_T to provide generic storage for different authentication credentials. The same structure also can be used to encode a certificate required for authentication.

Syntax

```
typedef struct
{
unsigned long   len;
void           *data];
} SS_ADV_CRED_T;
```

Fields

len

(IN) Specifies the length of the data buffer in bytes.

data

(IN) The actual data buffer to hold the credential.

Description

This structure provides a generic storage for passing credential in an advanced bind to eDirectory. Flags in the SS_ADV_BIND_INFO_T define the type of credential used.

SS_ADDR_T

This is the structure used by SS_ADV_BIND_INFO_T to provide storage for the server's IP address.

Syntax

```
typedef struct
{
unsigned long    len;
char             addr[NSSS_MAX_ADDR_LEN];
} SS_ADDR_T;
```

Fields

len

(IN) Specifies the length of the address in bytes.

addr

(IN) The actual address buffer to hold the IP address.

Description

This structure provides a generic storage for passing the target server's IP address in an advanced bind to eDirectory.

SS_ADV_BIND_INFO_T

Used to bind over supported protocols by eDirectory.

Syntax

```
typedef struct
{
unsigned long    version;
unsigned long    flags;
unsigned long    portNum;
SS_ADDR_T        hName;
SS_CERT_T        cert;
SS_ADV_CRED_T   *cred;
} SS_ADV_BIND_INFO_T;
```

Fields

version

(IN) Specifies the bind structure's version. NSSS_CUR_ADV_BIND_INFO_VER signifies the version this structure.

flags

(IN) Specifies the flags that can be used to determine which type of credential and authentication mechanism is used. Current supported flags include the following:

Flag	Description
NSSS_PWORD_CRED_F	Indicates that a password is being used for credential.
NSSS__SET_ANON_PORT_F	Indicates an anonymous port other than the default is being used.

portNum

(IN) Specifies the port to bind on (optional).

hName

(IN) Designates the DNS name or the IP address of the target server.

cert

(IN) Encodes or stores the certificate needed for authentication.

cred

(IN) This generic structure can be used to encode or store the credential for authentication. (When LDAP is used to login, password can be stored here.)

Description

This structure provides the advance bind information to bind over different protocols.

SS_OBJECT_DN_T

Used by SecretStore functions to provide storage for the object DN.

Syntax

```
typedef struct
{
unsigned long      len;
char               addr[NSSS_MAX_ADDR_LEN];
} SS_ADDR_T;
```

Fields

len

(IN) Specifies the length of the DN in bytes.

addr

(IN) The actual buffer to hold the DN.

Description

This structure provides a storage for passing the target object DN on the functions.

4

Return Values

This table describes the values commonly returned by the SecretStore service.

Value	Return Code	Description
-800	NSSS_E_OBJECT_NOT_FOUND	Target object could not be found.
-801	NSSS_E_NICI_FAILURE	NICI operations have failed.
-802	NSSS_E_INVALID_SECRET_ID	The Secret ID is not in the user secret store.
-803	NSSS_E_SYSTEM_FAILURE	Some internal operating system services have not been available.
-804	NSSS_E_ACCESS_DENIED	Access to the target Secret Store has been denied.
-805	NSSS_E_NDS_INTERNAL_FAILURE	NDS internal NDS services have not been available.
-806	NSSS_E_SECRET_UNINITIALIZED	Secret has not been initialized with a write.
-807	NSSS_E_BUFFER_LEN	Size of the buffer is not in a nominal range between minimum and maximum.
-808	NSSS_E_INCOMPATIBLE_VERSION	Client and server components are not of the compatible versions.
-809	NSSS_E_CORRUPTED_STORE	Secret Store data on the server has been corrupted.
-810	NSSS_E_SECRET_ID_EXISTS	Secret ID already exists in the SecretStore.
-811	NSSS_E_NDS_PWORD_CHANGED	User NDS password has been changed by the administrator.
-812	NSSS_E_INVALID_TARGET_OBJECT	Target NDS user object not found.
-813	NSSS_E_STORE_NOT_FOUND	Target NDS user object does not have a Secret Store.
-814	NSSS_E_SERVICE_NOT_FOUND	Secret Store is not on the network.
-815	NSSS_E_SECRET_ID_TOO_LONG	Length of the Secret ID buffer exceeds the limit.

Value	Return Code	Description
-816	NSSS_E_ENUM_BUFF_TOO_SHORT	Length of the enumeration buffer is too short.
-817	NSSS_E_NOT_AUTHENTICATED	User not authenticated.
-818	NSSS_E_NOT_SUPPORTED	Not supported operations.
-819	NSSS_E_NDS_PWORD_INVALID	Typed in NDS password not valid.
-820	NSSS_E_NICI_OUTOF_SYNC	Session keys of the client and server NICI are out of sync.
-821	NSSS_E_SERVICE_NOT_SUPPORTED	Requested service not yet supported.
-822	NSSS_E_TOKEN_NOT_SUPPORTED	NDS authentication type not supported.
-823	NSSS_E_UNICODE_OP_FAILURE	Unicode text conversion operation failed.
-824	NSSS_E_TRANSPORT_FAILURE	Connection to server is lost.
-825	NSSS_E_CRYPTO_OP_FAILURE	Cryptographic operation failed.
-826	NSSS_E_SERVER_CONN_FAILURE	Opening a connection to the server failed.
-827	NSSS_E_CONN_ACCESS_FAILURE	Access to server connection failed.
-828	NSSS_E_ENUM_BUFF_TOO_LONG	Size of the enumeration buffer exceeds the limit.
-829	NSSS_E_SECRET_BUFF_TOO_LONG	Size of the Secret buffer exceeds the limit.
-830	NSSS_E_SECRET_ID_TOO_SHORT	Length of the Secret ID should be greater than zero.
-831	NSSS_E_CORRUPTED_PACKET_DATA	Protocol data corrupted on the wire.
-832	NSSS_E_EP_ACCESS_DENIED	Enhanced protection's password validation failed. Access to the secret denied.
-833	NSSS_E_SCHEMA_NOT_EXTENDED	Schema is not extended to support SecretStore on the target tree.
-834	NSSS_E_ATTR_NOT_FOUND	One of the optional service attributes is not instantiated.
-835	NSSS_E_MIGRATION_NEEDED	Server has been upgraded and the user's SecretStore should be updated.
-836	NSSS_E_MP_PWORD_INVALID	Master password could not be verified to read or unlock the secrets.
-837	NSSS_E_MP_PWORD_NOT_SET	Master password has not been set on the SecretStore.

Value	Return Code	Description
-838	NSSS_E_MP_PWORD_NOT_ALLOWED	Ability to use master password has been disabled.
-839	NSSS_E_WRONG_REPLICA_TYPE	Not a writeable replica of NDS.
-840	NSSS_E_ATTR_VAL_NOT_FOUND	The API was unable to find a value for an attribute in the Directory.
-841	NSSS_E_INVALID_PARAM	A parameter passed to the API has not been properly initialized.
-842	NSSS_E_NEED_SECURE_CHANNEL	The connection to SecretStore requires SSL to be secure (returned when access is via LDAP).
-843	NSSS_E_CONFIG_NOT_SUPPORTED	The client could not locate a server that supports the policy override required by the caller. In previous versions of the client, if a server supporting the requested configuration (now known as policy override) could not be found, any server would be returned. In version 3.0, if we don't find a server supporting the required policy override, it is considered an error and we don't return anything.
-844	NSSS_E_STORE_NOT_LOCKED	An attempt to unlock SecretStore failed because the store is not locked.
-845	NSSS_E_TIME_OUT_OF_SYNC	The eDirectory replica on the server that holds SecretStore is out of sync with the replica ring.
-846	NSSS_E_VERSION_MISMATCH	Versions of the client dlls don't match.
-847	NSSS_E_SECRET_BUFF_TOO_SHORT	The buffer supplied for the secret is too short (minimum NSSS_MIN_IDLIST_BUF_LEN).
-848	NSSS_E_SH_SECRET_FAILURE	The shared secret processing operations failed.
-849	NSSS_E_PARSER_FAILURE	The shared secret parser operations failed.
-850	NSSS_E_UTF8_OP_FAILURE	The UTF8 string operations failed.
-851	NSSS_E_CTX_LESS_CN_NOT_UNIQUE	The contextless name for LDAP bind does not resolve to a unique DN.
-852	NSSS_E_UNSUPPORTED_BIND_CRED	The login credential for advanced bind is not supported.
-854	NSSS_E_CERTIFICATE_NOT_FOUND	The certificate required for the bind is not found.
-888	NSSS_E_NOT_IMPLMENTED	Feature not yet implemented.

Value	Return Code	Description
-899	NSSS_E_BETA_EXPIRED	The product’s BETA life has expired! Official release copy should be purchased.

5

Functions

The Novell SecretStore API includes functions to enable and maintain transparent authentication capabilities, providing simplification of user authentication within your client/server applications. SecretStore functions can be identified as either of the following categories:

- Enabling Functions
- Administrative Functions
- Shared Secret Functions

Enabling Functions

The following functions prepare applications to use Novell Single Sign-on and include:

Functions	Description
NSSSGetServiceInformation (page 79)	Returns service and SecretStore related information.
NSSSReadSecret (page 84)	Reads the application secrets from the SecretStore service for a logged in and authenticated eDirectory user of a SecretStore enabled application.
NSSSRemoveSecret (page 87)	Removes a specified secret from a user's SecretStore on an eDirectory object for an application.
NSSSWriteSecret (page 101)	Writes new secrets or overwrites the old secrets of an application in the SecretStore service for a logged in and authenticated eDirectory user of a SecretStore-enabled application.

Administrative Functions

These functions create the user's Single Sign-on capability and generate encryption keys for an enabled application:

Functions	Description
NSSSEnumerateSecretIDs (page 74)	Lists the application secret identifiers in a target secret.
NSSSRemoveSecretStore (page 89)	Removes SecretStore from a target object.
NSSSSetEPMasterPassword (page 96)	Allows the owner of SecretStore to set the SecretStore master password.
NSSSUnlockSecrets (page 98)	Unlocks a client's SecretStore after it was locked in an enhanced protection scenario by removing the lock, or by using a previous eDirectory password on master password.

Shared Secret Functions

The shared secret functions enable you to share single sign-on data with other single sign-on applications. There are two categories of shared secret functions:

- Shared Secret Data Management Functions
- Shared Secret Support Functions

Shared Secret Data Management Functions

These functions call the Shared Secret Support Functions (page 66) to populate or extract data from a shared secret:

- NSSSReadSharedSecret (page 82)
- NSSSRemoveSharedSecret (page 91)
- NSSSWriteSharedSecret (page 104)

Shared Secret Support Functions

These support functions populate or extract data from a shared secret:

- NSSSAddSHSEntry (page 70)
- NSSSCreateSHSHandle (page 72)
- NSSSDestroySHSHandle (page 73)
- NSSSGetNextSHSEntry (page 77)
- NSSSRemoveSHSEntry (page 94)

API Function Flags

The following flags are defined for the Single Sign-on API functions:

Input Only Flags for Write API

Value	Flag	Description
0x00000001L	NSSS_ENHANCED_PROTECTION_F	Enhanced Protection indicator flag for Read and Write.
0x00000040L	NSSS_EP_PASSWORD_USED_F	(Optional) Enhanced Protection optional password indicator flag for Read and Write.
0x00004000L	NSSS_CHK_SID_FOR_COLLISION_F	Check for existing SID to prevent collision and overwrite.

Input Only Flags for Unlock API

Value	Flag	Description
0x00000020L	NSSS_EP_MASTER_PWORD_USED_F	The master password used to read a secret in place of the Enhanced Protection password or to unlock in place of the old eDirectory password.
0x00000004L	NSSS_REMOVE_LOCK_FROM_STORE_F	Can delete locked secrets from store to remove lock.

Input Only Flags for Read API

Value	Flag	Description
0x00000020L	NSSS_EP_MASTER_PWORD_USED_F	The master password used to read a secret in place of the Enhanced Protection password or to unlock in place of the old eDirectory password.
0x00000008L	NSSS_REPAIR_THE_ STORE_F	Request all possible repairs on damaged store.

Input Only Flags for All APIs

Value	Flag	Description
0x00000010L	NSSS_ALL_STRINGS_UNICODE_F	Informs the service that the strings, such as secretID, DN, searchString, etc., are already converted to unicode and no conversion is necessary. (Results returned in unicode.)
0x00000200L	NSSS_DESTROY_CONTEXT_F	Internally destroys the DS context passed in. This flag can be used on the last call to SecretStore to destroy the context that was used.
0x00000800L	NSSS_UNBINDLDAP_F	Indicates LDAP-based access to directory should be terminated.
0x00000080L	NSSS_SET_TREE_NAME_F	Use the tree name in the context to set the tree.

Input Only Flag for GetServiceInfo API

Value	Flag	Description
0x00000080L	NSSS_SET_TREE_NAME_F	Sets the tree name.
0x00000100L	NSSS_GET_ CONTEXT_F	Returns a DS context for reuse in the subsequent calls.
0x00000800L	NSSS_BINDLDAP_F	Bind over LDAP to eDirectory hosting the SecretStore is requested.

Output Only Flags from Read API

These flags come back on the returned optional extension structures, NSSSGetServiceInformation (page 79) and NSSSReadSecret (page 84) (statFlags on reading a secret and statFlags on the store):

Value	Flag	Description
0x0001000L	NSSS_SECRET_LOCKED_F	Enhanced protection lock on a secret.
0x0002000L	NSSS_SECRET_NOT_INITIALIZED_F	Secret not yet initialized with a Write.
0x0004000L	NSSS_ENHANCED_PROTECT_INFO_F	Secret is marked for enhanced protection.
0x0008000L	NSSS_STORE_NOT_SYNCED_F	Store is not yet synchronized across replicas.
0x0020000L	NSSS_EP_PWORD_PRESENT_F	There is an Enhanced Protection application password on the secret.

Output Only Flag from GetServiceInformation API statFlags

Value	Flag	Description
0x0080000L	NSSS_MP_NOT_ ALLOWED_F	The use of master password has been disabled by the service.
0x0040000L	NSSS_EP_MASTER_PWORD_PRESENT_F	There is a master password on the SecretStore (Admin configurable option on the server).

Context Flags for The Type of Context Passed in to Initialize Context Structure

Value	Flag	Description
0x00000001L	NSSS_NCP_CTX_F	NCP context.
0x00000002L	NSSS_LDAP_CTX_F	LDAP context. (Reserved for the future.)
0x00000008L	NSSS_INIT_LDAP_SS_HANDLE_F	Initialize the client supplied context for SS use.
0x00000010L	NSSS_DEINIT_LDAP_SS_HANDLE_F	Deinitialize the client context for application unbind
0x00000020L	NSSS_REINIT_TARGET_DN_F	Reinitialize the target DN for admin in the context when admin is switching target.
0x00000040L	NSSS_LDAP_CONTEXT_LESS_DN_F	Resolving the context less DN for the bind is requested because the DN that is passed in is contextless.
0x00000080L	NSSS_ADV_BIND_INFO_F	Use the advanced bind structure and preform service location.

Context Flags for Input and Returned from the Context Structure

Value	Flag	Description
0x00000004L	NSSS_CONTEXT_INITIALIZED_F	Connection to server is established and context structure is initialized (returned from SS when context is initialized or can be supplied when the context is preinitialized outside SS and is passed in for SS use).

Function Prototypes

The following are **_stdcall** function prototypes exported APIs in Windows. For definitions of the SecretStore function return types, refer to nssscl.h.

NSSSAddSHSEntry

Enters a key or a value in a key-value pair stored in a Shared Secret.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSAddSHSEntry
(
 void            *handle,
 unsigned char   *key,
 unsigned char   *val,
 unsigned long    ssCtxFlags
);
```

Parameters

handle

(IN) Specifies the handle created by NSSSCreateSHSHandle (page 72).

key

(IN) Adds a key to a key-value pair stored in a Shared Secret.

val

(IN) Adds a value to a key-value pair stored in a Shared Secret.

ssCtxFlags

(IN) Specifies an optional structure that can be initialized by calling NSSSGetServiceInformation (page 79) prior to use here. The flags field of the structure can take on the following values to indicate the type of context used.

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

Return Values

Value	Description
NSSS_SUCCESS	The operation succeeded.
NSSS_E_SH_SECRET_FAILURE	Shared Secret processing and operations failed.
NSSS_E_INVALID_PARAM	API parameter is not initialized.

Remarks

This function contains pointers to user-allocated key and value buffers and the unsigned long context flag member of the SSS_CONTEXT_T structure populated from calling NSSSGetServiceInformation (page 79).

See Also

NSSSCreateSHSHandle (page 72)
NSSSGetNextSHSEntry (page 77)
NSSSRemoveSHSEntry (page 94)
See other Shared Secret buffer calls here.

NSSSCreateSHSHandle

Returns a void pointer that handles the passing of shared secret data to subsequent calls.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL_PTR(void) NSSSCreateSHSHandle (void);
```

Return Values

Returns a void pointer that functions as a handle.

Remarks

This function is called for each thread in a user-defined application that shares secrets.

See Also

NSSSDestroySHSHandle (page 73)
See other Shared Secret buffer calls here.

NSSSDestroySHSHandle

Frees the memory associated with the handle created for each Shared Secret thread of execution.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSDestroySHSHandle
(
 void   *handle
);
```

Parameters

handle

(IN) Specifies the handle created by NSSSCreateSHSHandle (page 72).

Remarks

This call signifies the end of shared secret processing by destroying an internal Shared Secret buffer.

See Also

NSSSCreateSHSHandle (page 72)
See other Shared Secret buffer calls here.

NSSSEnumerateSecretIDs

Enables the administrator or user to list the secret identifiers (secret IDs) for secrets stored in the user's SecretStore.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSEnumerateSecretIDs
(
   SSS_CONTEXT_T    *callerContext,
   SS_OBJECT_DN_T   *targetObject,
   unsigned long     ssFlags,
   char             *searchString,
   unsigned long    *count
   SS_SECRET_T      *secretIDList,
   SS_EXT_T         *ext
);
```

Parameters

callerContext

(IN) This optional structure can be initialized by making a call to NSSSGetServiceInformation (page 79) prior to use here. The flags field of the structure can take on the following values to indicate the type of context used.

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

targetObject

(IN) This is the optional RDN (relative distinguished name or "short name") of the target object that contains the user's secrets. You should have at least READ/WRITE privileges over the target object.

ssFlags

(IN) This is a set of flags for initializing secrets:

Value	Description
NSSS_ALL_STRINGS_UNICODE_F	Indicates that all applicable char strings (such as targetObject, secretID, etc.) are already in Unicode and the API does not need to perform conversion from local code page to Unicode. This flag also instructs the API not to convert the return char strings to local code page. **NOTE:** This is a feature of SSO client Version 1.1 and will not work on the older versions.
NSSS_SET_TREE_NAME_F	Sets the tree name.
NSSS_DESTROY_CONTEXT_F	Internally destroys the DS context passed in. This flag can be used on the last call to SecretStore to destroy the context that was used.

searchString

(IN) Set to *, NULL, or "" if all entries are desired in the search. Use the asterisk "*" as delimiter to search for specific entries with known prefixes, such as "MYAppSecretNumber_*".

count

(OUT) The number of secret identifiers stored for the user.

secretIDList

(OUT) An asterisk "*" separated list of secret identifiers matching the search string.

ext

(OUT) If present, this structure can return a set of applicable future extensions for the secrets.

Return Values

These are common return values for this function; see "Return Values" on page 61 for more information.

Value	**Description**
NSSS_SUCCESS	The operation succeeded.
NSSS_E_SYSTEM_FAILURE	Some internal operation failed due to some failure such as memory allocation.
NSSS_E_CORRUPTED_STORE	This indicates data corruption in SecretStore.
NSSS_E_NICI_FAILURE	NICI failure detected.
NSSS_E_INVALID_SECRET_ID	Secret ID is not in the SecretStore.
NSSS_E_ACCESS_DENIED	eDirectory denied access to an object or attributed related to the SecretStore.
NSSS_E_STORE_NOT_FOUND	Client does not have a SecretStore or the schema is not extended to begin with.
NSSS_E_NDS_INTERNAL_FAILURE	Some eDirectory operation has failed internally.
NSSS_E_INCOMPATIBLE_VERSION	Client SecretStore is not compatible with server SecretStore.
NSSS_E_INVALID_TARGET_OBJECT	Target object is not the type designated for SecretStore attachment.

Remarks

The memory allocated for the secretIDBuffer should be set to NSSS_ENUM_BUFFER_GUESS. This should be enough memory for most applications.

If this call returns NSSS_ERR_MORE_DATA (not a fatal error), call it again with a buffer the size of returned secretIDList->len. If the buffer is too small for all of the data in the SecretStore, the returned buffered from the server is stuffed as much as it has room. search string can be used to change the scope of the search when buffer size is a constraint.

See Also

NSSSReadSecret (page 84)
NSSSWriteSecret (page 101)
NSSSRemoveSecret (page 87)

NSSSGetNextSHSEntry

Specifies sequential calls required to obtain the Shared Secret data returned to NSSSReadSharedSecret (page 82).

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSGetNextSHSEntry
(
 BOOL              bRestart,
 void             *handle,
 in               *keyLen,
 unsigned long    *key,
 int              *valLen,
 unsigned char    *val,
 unsigned long     ssCtxFlags
);
```

Parameters

bRestart

(IN) Specifies location in the buffer to begin search. Set to 1 to begin from the beginning of the buffer, otherwise, set to 0.

handle

(IN) Points to the handle created by NSSSCreateSHSHandle (page 72).

keyLen

(OUT) Points to the length of the key.

key

(OUT) Points to the key used for storing key-value pair data as defined in the SharedSecret format.

valLen

(OUT) Points to the length of the value.

val

(OUT) Points to the value used to for storing key-value pair data as defined in the SharedSecret format.

ssCtxFlags

(IN) Specifies an optional structure that can be initialized by calling NSSSGetServiceInformation (page 79) prior to use here. The flags field of the structure can take on the following values to indicate the type of context used.

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

Return Values

Value	Description
NSSS_SUCCESS	The operation succeeded in returning a key or value data.
NSSS_E_SH_SECRET_FAILURE	Shared Secret processing and operations failed
–1	The operation has completed its search.

Remarks

Sequential calls to this function contain the handle, the unsigned long context flag from the user-populated SSS_CONTEXT_T struct passed into NSSSReadSharedSecret (page 82), and pointers to user-allocated key/value buffer and length parameters which are populated upon return of the call.

See Also

NSSSAddSHSEntry (page 70)
NSSSRemoveSharedSecret (page 91)
See other Shared Secret buffer calls here.

NSSSGetServiceInformation

Returns service information from the SecretStore for authenticated users of a Single Sign-on enabled application.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSGetServiceInfomaion
(
 SSS_CONTEXT_T     *callerContext,
 SS_OBJECT_DN_T    *targetObjDN,
 unsigned long      ssFlags,
 SSS_GSINFOEXT_T   *gsInfo,          //mandatory
 SS_EXT_T          *ext
);
```

Parameters

callerContext

(IN) This handle can be initialized by making calls to eDirectory prior to SecretStore or requesting NSSSGetServiceInformation to initialize it. The flags field can take on these values to indicate the type of context used:

Value	Description
NSSS_NCP_CTX_F	Sets the NCP context indicator flag.
NSSS_LDAP_CTX_F	Sets the LDAP context indicator flag.
NSSS_INIT_LDAP_SS_HANDLE_F	Set when init is passed into the handle without bind. This is done when you do the bind for contexts initialized outside of SecretStore client.
NSSS_DEINIT_LDAP_HANDLE_F	Deinitializes the passed in handle when you want to perform the unbind later (for contexts initialized outside of SecretStore client).
NSSS_REINIT_TARGET_DN_F	Reinitializes the handle to a new target DN. Set when the administrator plans to switch from one target DN to another.

targetObjDN

(IN) This is the optional RDN (relative distinguished name or "short name") of the target object that contains the user's secrets. The caller should have at least READ/WRITE privileges over the target object.

NOTE: For binding to server over LDAP/SSL this parameter is mandatory and it should be in fully qualified LDAP form ("cn=user, ou=users, o=novell").

ssFlags

(IN) This is a set of flags for initializing secrets:

NSSS_ALL_STRINGS_UNICODE_F as defined by "Input Only Flags for All APIs" on page 67.
NSSS_SET_TREE_NAME_F as defined by "Input Only Flags for All APIs" on page 67.
NSSS_GET_CONTEXT_F as defined by "Input Only Flag for GetServiceInfo API" on page 67.

NSSS_DESTROY_CONTEXT_F

(OUT)

NSSS_ENHANCED_PROTECT_INFO_F—Secret is marked for enhanced protection.

NSSS_EP_MASTER_PWORD_PRESENT_F—There is a master password on the SecretStore (Admin configurable option on the server).

NSSS_MP_NOT_ALLOWED_F—The use of master password has been disabled by the service.

ext

(OUT) If present, this structure can return a set of applicable future extensions for the secrets.

Return Values

These are common return values for this function; see "Return Values" on page 61 for more information.

NSSS_SUCCESS	The operation succeeded.
NSSS_E_SYSTEM_FAILURE	Some internal operation failed due to some failure such as memory allocation.
NSSS_E_CORRUPTED_STORE	This indicates data corruption in SecretStore.
NSSS_E_NICI_FAILURE	NICI failure detected.
NSSS_E_INVALID_SECRET_ID	Secret ID is not in the SecretStore.
NSSS_E_ACCESS_DENIED	eDirectory denied access to an object or attributed related to the SecretStore.
NSSS_E_STORE_NOT_FOUND	Client does not have a SecretStore or the schema is not extended to begin with.
NSSS_E_NDS_INTERNAL_FAILURE	Some eDirectory operation has failed internally.
NSSS_E_INCOMPATIBLE_VERSION	Client SecretStore is not compatible with server SecretStore.
NSSS_E_INVALID_TARGET_OBJECT	Target object is not the type designated for SecretStore attachment.

Remarks

This can authenticate and connect the SecretStore client to the target SecretStore server. The initialized context (NCP/LDAP) can be utilized across other calls to have an ongoing session with SecretStore. This allows considerable performance enhancement by reusing credentials across multiple calls and avoiding reinitialization per call. These new SecretStore calls can still perform per-call initialization and operations for connectors.

See Also

NSSSReadSecret (page 84)
NSSSWriteSecret (page 101)
NSSSRemoveSecret (page 87)

NSSSReadSharedSecret

Reads data from an existing Shared Secret to retrieve secret data from a user's SecretStore located on eDirectory.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSReadSharedSecret
(
 void                 *handle,
 SS_SH_SECRET_ID_T    *pSharedSecret,
 SSS_CONTEXT_T        *context,
 SS_OBJECT_DN_T       *targetObjDN,
 unsigned long         ssFlags,
 SS_PWORD_T           *epPassword,
 SSS_READEXT_T        *readData,
 SS_EXT_T             *ext
);
```

Parameters

handle

(IN) Specifies the handle created by NSSSCreateSHSHandle (page 72).

pSharedSecret

(IN) Points to the user-populated SS_SH_SECRET_ID_T struct containing the shared secret type, name, and length.

context

(IN) Points to an optional structure that can be initialized by making a prior call to NSSSGetServiceInformation (page 79). The flags field of the structure can take on the following values to indicate the type of context used.

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

targetObjDN

(IN) Points to the optional RDN (relative distinguished name or "short name") of the target object that contains the user's secrets. The caller should have at least READ/WRITE privileges over the target object.

NOTE: For binding to server over LDAP/SSL this parameter is mandatory and it should be in fully qualified LDAP form ("cn=user, ou=users, o=novell").

ssFlags

(IN) Specifies the flags used when making the call to NSSSReadSecret (page 84).

epPassword

(IN) Points to an optional field to pass in the Master Password or the Enhanced Protection Password for reading a secret. When neither one of the passwords are present, you can pass in a NULL.

readData

(IN) Points to the extension to be set to read data stored in a user's secrets.

ext

(IN) Points to the extensions used for the secrets.

Return Values

Value	Description
NSSS_SUCCESS	The operation succeeded.
NSSS_E_SYSTEM_FAILURE	Some internal operation failed due to a failure such as improper memory allocation.
NSSS_E_UTF8_OP_FAILURE	The UTF8 string operations failed.
NSSS_E_INVALID_PARAM	The API parameter is not initialized.
NSSS_E_SECRET_ID_TOO_SHORT	The length of the Secret ID should be greater than zero.

Remarks

This function passes the handle, as well as a user-populated SS_SH_SECRET_ID_T structure, containing the shared secret type, name, and length. It also points to internally allocated key and value buffers

See Also

NSSSReadSecret (page 84)
NSSSWriteSharedSecret (page 104)
See other Shared Secret Buffer calls here.

NSSSReadSecret

Reads the secrets from the SecretStore service for an authenticated user of a SecretStore-enabled application.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSReadSecret
(
   SSS_CONTEXT_T      callerContext,
   SS_OBJECT_DN_T    *targetObject,
   unsigned long      ssFlags,
   SS_PWORD_T         epPassword
   SSS_READEXT_T      readInfo,
   SS_SECRET_ID_T    *secretID,
   SS_SECRET_T       *secretValue,
   SS_EXT_T          *ext
);
```

Parameters

callerContext

(IN) Specifies an optional structure that can be initialized by making a call to NSSSGetServiceInformation (page 79) prior to use here. The flags field of the structure can take on the following values to indicate the type of context used.

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

targetObject

(IN) Points to the optional RDN (relative distinguished name or "short name") of the target object that contains the user’s secrets. You should have at least READ/WRITE privileges over the target object.

ssFlags

(IN) Specifies a set of flags for initializing secrets.

Value	Description
NSSS_ALL_STRINGS_UNICODE_F	This flag indicates that all applicable char strings such as targetObject, secretID etc., are already in Unicode and the API does not need to perform conversion from local code page to Unicode. This flag instructs the API not to convert the return char strings to local code page as well. This is a feature of SSO client v1.1 and does not work on the older versions.
NSSS_ENHANCED_PROTECTION_F	Enhanced Protection indicator flag for Read and Write.
NSSS_EP_MASTER_PWORD_USED_F	Enables the user to supply the EP master password to unlock the SecretStore in place of the previous eDirectory password.

Value	Description
NSSS_REPAIR_THE_STORE_F	Request all possible repairs on damaged store.
NSSS_SET_TREE_NAME_F	Sets the tree name.
NSSS_DESTROY_CONTEXT_F	Internally destroys the DS context passed in. This flag can be used on the last call to SecretStore to destroy the context that was used.

epPassword

(IN) Specifies an optional field to pass in the Master Password or the Enhanced Protection Password for reading a secret. When neither one of the passwords are present, you can pass in a NULL.

readInfo

(OUT) Specifies the structure that returns the status information coming back from reading a secret.

secretID

(IN) Points to a unique secret identifier chosen by the application that should be supplied to locate the application secret values in the user's secrets, preferably in the Novell conventional format described earlier in this document.

secretValue

(OUT) Points to a buffer that the client allocates for the returned secret value. A call with "secretValue->len=0" returns the required buffer size in "secretValue->size".

ext

(OUT) If present, points to a set of applicable future extensions for the secrets.

Return Values

These are common return values for this function (see "Return Values" on page 61 for more information):

Value	Description
NSSS_SUCCESS	The operation succeeded.
NSSS_E_SYSTEM_FAILURE	Some internal operation failed due to some failure such as memory allocation.
NSSS_E_CORRUPTED_STORE	This indicates data corruption in SecretStore.
NSSS_E_NICI_FAILURE	NICI failure detected.
NSSS_E_INVALID_SECRET_ID	Secret ID is not in the SecretStore.
NSSS_E_ACCESS_DENIED	eDirectory denied access to an object or attribute related to the SecretStore.
NSSS_E_STORE_NOT_FOUND	Client does not have a SecretStore or the Schema is not extended to begin with.

Value	Description
NSSS_E_NDS_PWORD_CHANGE	Admin has changed the user password and as a result the client’s SecretStore is locked (non-repudiation).
NSSS_E_NDS_INTERNAL_FAILURE	Some eDirectory operation has failed internally.
NSSS_E_INCOMPATIBLE_VERSION	Client SecretStore is not compatible with server SecretStore.
NSSS_E_INVALID_TARGET_OBJECT	Target object is not the type designated for SecretStore attachment.
NSSS_E_EP_ACCESS_DENIED	Enhanced Protection password validation failed.

Remarks

This SecretStore call accesses the service on behalf of a logged in and authenticated user. It returns to the client component of the application a clear copy of the application’s secrets stored in SecretStore.

The unique *secretID* that was chosen for this application when the user's SecretStore was being populated is passed in as input. As a result, the object is located in the tree and the SecretStore is read until the *secretID* is located. When the proper secret is located in the SecretStore, it is decrypted and returned in the *secretValue* buffer allocated for the purpose. Since the actual required size of the secret buffer is returned regardless of the success or failure of this call, the client can make a second call with the proper buffer size if the original request failed due to insufficient buffer size.

If the *targetObject* is of the "User" type in eDirectory, then the *callerContext* and the *targetObject* should match. In other words, only the owner of the SecretStore can read the secrets. If the *targetObject* is not a User, the call that has proper access rights can read SecretStore on a non-User object type in eDirectory.

The *SecretCount* field can return the count of secrets in the SecretStore if the client is talking to a Version 2.0 SecretStore on the server.

NOTE: *sssinit.exe* enables an administrator to extend the schema on a non-user object for SecretStore. This tool and the SecretStore product installation by default extends schema on a non-user object.

See Also

NSSSWriteSecret (page 101)

NSSSRemoveSecret

Removes the specified secret from SecretStore.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NWSSRemoveSecret
(
   SSS_Context_T      callerContext,
   SS_OBJECT_DN_T    *targetObject,
   unisgned long      ssFlags,
   SS_SECRET_ID_T    *secretID,
   SS_EXT_T          *ext
);
```

Parameters

callerContext

(IN) Specifies an optional structure that can be initialized by calling NSSSGetServiceInformation (page 79) prior to use here. The flags field of the structure can take on the following values to indicate the type of context used.

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

targetObject

(IN) Points to an optional RDN (relative distinguished name or "short name") of the target object that contains the user's secrets. You should have at least READ/WRITE privileges over the target object.

ssFlags

(IN) Specifies a set of flags for initializing secrets.

Value	Description
NSSS_ALL_STRINGS_UNICODE_F	Indicates that all applicable char strings such as targetObject, secretID etc., are already in Unicode and the API does not need to perform conversion from local code page to Unicode. This flag instructs the API not to convert the return char strings to local code page as well. This is a feature of SSO client v1.1 and will not work on the older versions.
NSSS_SET_TREE_NAME_F	Sets the tree name.
NSSS_DESTROY_CONTEXT_F	Internally destroys the DS context passed in. This flag can be used on the last call to SecretStore to destroy the context that was used.

secretID

(IN) Points to a unique secret identifier chosen by the application that should be supplied to locate the application secret values in the user's SecretStore to be removed.

ext

(OUT) If present, returns a set of applicable future extensions for the secrets.

Return Values

These are common return values (for more information, see "Return Values" on page 61):

Value	Description
NSSS_SUCCESS	The operation succeeded.
NSSS_E_SYSTEM_FAILURE	Some internal operation failed due to some failure such as memory allocation.
NSSS_E_CORRUPTED_STORE	This indicates data corruption in SecretStore.
NSSS_E_NICI_FAILURE	NICI failure detected.
NSSS_E_INVALID_SECRET_ID	Secret ID is not in the SecretStore.
NSSS_E_ACCESS_DENIED	eDirectory denied access to an object or attribute related to the SecretStore.
NSSS_E_STORE_NOT_FOUND	Client does not have a SecretStore or the Schema is not extended to begin with.
NSSS_E_NDS_INTERNAL_FAILURE	Some eDirectory operation has failed internally.
NSSS_E_INCOMPATIBLE_VERSION	Client SecretStore is not compatible with server SecretStore.
NSSS_E_INVALID_TARGET_OBJECT	Target object is not the type designated for SecretStore attachment.

Remarks

The NSSSRemoveSecret (page 87) call removes an identified secret from the SecretStore for the user. If the secret happens to be the last secret in the user's SecretStore the SecretStore is removed completely.

NOTE: This function formerly was called NSSSRemoveSecretID.

See Also

NSSSReadSecret (page 84)
NSSSWriteSecret (page 101)
NSSSEnumerateSecretIDs (page 74)
NSSSRemoveSecret (page 87)

NSSSRemoveSecretStore

Removes SecretStore from the eDirectory object.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSRemoveSecretStore
(
   SSS_CONTEXT_T      callerContext,
   SS_OBJECT_DN_T    *targetObject,
   unsigned long      ssFlags,
   SS_EXT_T          *ext
);
```

Parameters

callerContext

(IN) Specifies an optional structure that can be initialized by making a call to NSSSGetServiceInformation (page 79) prior to use here. The flags field of the structure can take on the following values to indicate the type of context used.

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

targetObject

(IN) Points to the optional RDN (relative distinguished name or "short name") of the target object that contains the user's secrets. You should have at least READ/WRITE privileges over the target object.

ssFlags

(IN) Specifies a set of flags for initializing secrets.

Value	Description
NSSS_ALL_STRINGS_UNICODE_F	Indicates that all applicable char strings such as targetObject, secretID etc., are already in Unicode and the API does not need to perform conversion from local code page to Unicode. This flag instructs the API not to convert the return char strings to local code page as well. This is a feature of SSO client Version 1.1 and will not work on the older versions.
NSSS_SET_TREE_NAME_F	Sets the tree name.
NSSS_DESTROY_CONTEXT_F	***

ext

(OUT) If present, points to a set of applicable future extension returns for the secrets.

Return Values

These are common return values (for more information, see "Return Values" on page 61):

Value	Description
NSSS_SUCCESS	The operation succeeded.
NSSS_E_SYSTEM_FAILURE	Some internal operation failed due to some failure such as memory allocation.
NSSS_E_CORRUPTED_STORE	This indicates data corruption in SecretStore.
NSSS_E_NICI_FAILURE	NICI failure detected.
NSSS_E_INVALID_SECRET_ID	Secret ID is not in the SecretStore.
NSSS_E_ACCESS_DENIED	eDirectory denied access to an object or attribute related to the SecretStore.
NSSS_E_STORE_NOT_FOUND	Client does not have a SecretStore or the Schema is not extended to begin with.
NSSS_E_NDS_INTERNAL_FAILURE	Some eDirectory operation has failed internally.
NSSS_E_INCOMPATIBLE_VERSION	Client SecretStore is not compatible with server SecretStore.
NSSS_E_INVALID_TARGET_OBJECT	Target object is not the type designated for SecretStore attachment.

See Also

NSSSEnumerateSecretIDs (page 74)
NSSSReadSecret (page 84)
NSSSWriteSecret (page 101)

NSSSRemoveSharedSecret

Removes a Shared Secret from a user's SecretStore on eDirectory.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSRemoveSharedSecret
(
 SS_SH_SECRET_ID_T   *pSharedSecret,
 SSS_CONTEXT_T       *context,
 SS_OBJECT_DN_T      *targetObjDN,
 unsigned long        ssFlags,
 SS_EXT_T            *ext
);
```

Parameters

pSharedSecret

(IN) Points to the user-populated SS_SH_SECRET_ID_T struct containing the shared secret type, name, and length.

context

(IN) Specifies an optional structure that can be initialized by making a call to NSSSGetServiceInformation (page 79) prior to use here. The flags field of the structure can take on the following values to indicate the type of context used.

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

targetObjDN

(IN) Points to the optional relative distinguished name (RDN or “short name”) of the target object that contains the user’s secrets. You should have at least READ/WRITE privileges over the target object.

NOTE: For binding to server over LDAP/SSL this parameter is mandatory and it should be in fully qualified LDAP form ("cn=user, ou=users, o=novell").

ssFlags

(IN) Specifies the flags passed to NSSSRemoveSecret (page 87).

Value	Description
NSSS_ALL_STRINGS_UNICODE_F	Indicates that all applicable char strings such as targetObject, secretID etc., are already in Unicode and the API does not need to perform conversion from local code page to Unicode. This flag instructs the API not to convert the return char strings to local code page as well. This is a feature of SSO client v1.1 and will not work on the older versions.
NSSS_SET_TREE_NAME_F	Sets the tree name.

Value	Description
NSSS_DESTROY_CONTEXT_F	Internally destroys the DS context passed in. This flag can be used on the last call to SecretStore to destroy the context that was used.

ext

(IN) Points to the extensions used for the secrets.

Return Values

Value	Description
NSSS_SUCCESS	The operation succeeded.
NSSS_E_SYSTEM_FAILURE	Some internal operation failed due to some failure such as memory allocation.
NSSS_E_CORRUPTED_STORE	This indicates data corruption in SecretStore.
NSSS_E_NICI_FAILURE	A NICI failure was detected.
NSSS_E_INVALID_SECRET_ID	Secret ID is not in the SecretStore.
NSSS_E_ACCESS_DENIED	eDirectory denied access to an object or attribute related to the SecretStore.
NSSS_E_STORE_NOT_FOUND	The client does not have a SecretStore or the schema is not extended to begin with.
NSSS_E_NDS_INTERNAL_FAILURE	Some eDirectory operation has failed internally.
NSSS_E_INCOMPATIBLE_VERSION	The client SecretStore is not compatible with the server SecretStore.
NSSS_E_INVALID_TARGET_OBJECT	The target object is not the type designated for the SecretStore attachment.
NSSS_E_SECRET_ID_TOO_SHORT	The length of the Secret ID should be greater than zero.
NSSS_E_INVALID_PARAM	The API parameter is not initialized.

Remarks

This function passes a user-populated SS_SH_SECRET_ID_T structure that contains the shared secret type, name, and length. It also passes the populated SSS_CONTEXT_T and SS_OBJECT_DN_T structures, ssFlags, and the SS_EXT_T structures that are normally passed into the call to NSSSRemoveSecret. Consequently, this function provides flexibility in making calls to trees and user DNs other than the primary connection.

See Also

NSSSRemoveSecret (page 87)

See other Shared Secret buffer calls here.

NSSSRemoveSHSEntry

Removes a key or a value in a key-value pair stored in a Shared Secret.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSRemoveSHSEntry
(
 void             *handle,
 unsigned char    *key,
 unsigned char    *val,
 unsigned long     ssCtxFlags
);
```

Parameters

handle

(IN) Specifies the handle created by NSSSCreateSHSHandle (page 72).

key

(IN) Points to the value of the key-value pair desired to be removed.

val

(IN) Points to the value of the key-value pair desired to be removed.

ssCtxFlags

(IN) Specifies an optional structure that can be initialized by calling NSSSGetServiceInformation (page 79) prior to use here. The flags field of the structure can take on the following values to indicate the type of context used.

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

Return Values

Value	Description
NSSS_SUCCESS	The operation succeeded in returning a key or value data.
NSSS_E_SH_SECRET_FAILURE	Shared Secret processing and operations failed.
NSSS_E_INVALID_PARAM	The API parameter is not initialized.
NSSS_E_SYSTEM_FAILURE	Some internal operating system services have not been available.

Remarks

This function contains the handle, pointers to user-allocated key and value buffers, and the unsigned long context flag member of the SSS_CONTEXT_T struct populated from calling NSSSGetServiceInformation (page 79).

See Also

NSSSAddSHSEntry (page 70)
NSSSGetNextSHSEntry (page 77)

NSSSSetEPMasterPassword

A special function for use by administrative utilities.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSSetEPMasterPassword
(
 SSS_CONTEXT_T    *callerContext,
 SS_OBJECT_DN_T   *targetObjDN,
 unsigned long     ssFlags,
 SS_PWORD_T       *password,
 SS_HINT_T        *hint,
 SS_EXT_T         *ext
);
```

Parameters

callerContext

(IN) Points to an optional structure that can be initialized by making a call to NSSSGetServiceInformation (page 79) prior to use here. The flags field of the structure can take on the following values to indicate the type of context used.

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

targetObject

(IN) Points to the optional RDN (relative distinguished name or "short name") of the target object that contains the user's secrets. The caller should have at least READ/WRITE privileges over the target object.

ssFlags

(IN) Specifies the set of flags for initializing secrets:

Value	**Description**
NSSS_ALL_STRINGS_UNICODE_F	Indicates that all applicable char strings such as targetObject, secretID, etc., are already in Unicode and the API does not need to perform conversion from local code page to Unicode. This flag instructs the API not to convert the return char strings to local code page as well. This is a feature of SSO client Version 1.1 and will not work on the older versions.
NSSS_REMOVE_LOCK_FROM_STORE_F	Causes SecretStore to delete all of the enhanced protected secrets that are locked and, therefore, remove the lock from SecretStore.
NSSS_EP_MASTER_PWORD_USED_F	Enables the user to supply the EP master password to unlock the SecretStore in place of the previous eDirectory password.

Value	Description
NSSS_SET_TREE_NAME_F	Sets the tree name.
NSSS_DESTROY_CONTEXT_F	***

password

(IN) Points to the master password to be set.

hint

(IN) Points to the hint for the master password to be set by the user.

ext

(OUT) If present, points to a set of applicable future extension returns for the secrets.

Return Values

These are common return values for this function; see "Return Values" on page 61 for more information.

Remarks

This call can set a master password on the user's SecretStore if it is allowed by the service and if the user has enhanced protection set on their SecretStore.

See Also

See other Shared Secret Buffer calls here.

NSSSUnlockSecrets

This call unlocks the client's SecretStore after an administrative change of the client's eDirectory password has caused the user's SecretStore with enhanced protection secrets to become locked.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSUnlockSecrets
(
   SSS_CONTEXT_T      callerContext,
   SS_OBJECT_DN_T    *targetObject,
   unsigned long      ssFlags,
   SS_PWORD_T        *password,
   SS_EXT_T          *ext
);
```

Parameters

callerContext

(IN) Specifies the optional structure that can be initialized by making a call to NSSSGetServiceInformation (page 79) prior to use here. The flags field of the structure can take on the following values to indicate the type of context used:

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

targetObject

(IN) Points to the optional RDN (relative distinguished name or "short name") of the target object that contains the user's secrets. The caller should have at least READ/WRITE privileges over the target object.

ssFlags

(IN) Specifies the set of flags for initializing secrets:

Value	Description
NSSS_ALL_STRINGS_UNICODE_F	Indicates that all applicable char strings such as targetObject, secretID, etc., are already in Unicode and the API does not need to perform conversion from local code page to Unicode. This flag instructs the API not to convert the return char strings to local code page as well. This is a feature of SSO client Version 1.1 and will not work on the older versions.
NSSS_REMOVE_LOCK_FROM_STORE_F	Causes SecretStore to delete all of the enhanced protected secrets that are locked and, therefore, remove the lock from SecretStore.
NSSS_EP_MASTER_PWORD_USED_F	Enables the user to supply the EP master password to unlock the SecretStore in place of the previous eDirectory password.

Value	Description
NSSS_SET_TREE_NAME_F	Sets the tree name.
NSSS_DESTROY_CONTEXT_F	***

password

(IN) This points to the client's clear text password.

ext

(OUT) If present, this structure can return a set of applicable future extensions for the secrets.

Return Values

These are common return values for this function; see "Return Values" on page 61 for more information.

NSSS_SUCCESS	The operation succeeded.
NSSS_E_SYSTEM_FAILURE	Some internal operation failed due to some failure such as memory allocation.
NSSS_E_CORRUPTED_STORE	This indicates data corruption in SecretStore.
NSSS_E_NICI_FAILURE	NICI failure detected.
NSSS_E_INVALID_SECRET_ID	Secret ID is not in the SecretStore.
NSSS_E_ACCESS_DENIED	eDirectory denied access to an object or attribute related to the SecretStore.
NSSS_E_STORE_NOT_FOUND	Client does not have a SecretStore or the schema is not extended to begin with.
NSSS_E_NDS_INTERNAL_FAILURE	Some eDirectory operation has failed internally.
NSSS_E_INCOMPATIBLE_VERSION	Client SecretStore is not compatible with server SecretStore.
NSSS_E_UNLOCKING_FAILED	Verification of the old eDirectory password failed; therefore, unlocking the store failed.
NSSS_E_INVALID_TARGET_OBJECT	Target object is not the type designated for SecretStore attachment.

Remarks

NSSSUnlockSecrets the client's SecretStore after an administrative change of the client's eDirectory password. When administrator changes a eDirectory user's password, the SecretStore service is automatically locked. A prior call to NSSSReadSecret (page 84) will fail with the NSSS_E_NDS_PWORD_CHANGED. Then the client should make a call to NSSSUnlockSecrets (page 98) and supply the client's old eDirectory password to unlock the SecretStore.

If the service allows master password for users and the user has set a master password on their SecretStore prior to locking, then the user can use the master password to unlock the SecretStore. This helps for instances when the user forgets the eDirectory password.

If the password change has been due to a user forgetting the password and there is no master password, then SecretStore is not recoverable. Consequently, the locked SecretStore should be deleted and recreated by the client. eDirectory password changes by the user will not cause the SecretStore to be locked.

The owner of the SecretStore can use this function call with proper flags to remove the locked secrets or unlock the SecretStore with the previous eDirectory password or master password.

See Also

NSSSEnumerateSecretIDs (page 74)
NSSSReadSecret (page 84)
NSSSWriteSecret (page 101)
Chapter 4, “Return Values,” on page 61

NSSSWriteSecret

This call writes a secret to the user's SecretStore for authenticated users of SecretStore-enabled applications.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSWriteSecret
(
   SSS_Context_T      callerContext,
   SS_OBJECT_DN_T    *targetObject,
   unsigned long      ssFlags
   SS_PWORD_T        *epPassword,
   SS_SECRET_ID_T    *secretID,
   SS_SECRET_T       *secretValue,
   SS_EXT_F         *ext
);
```

Parameters

callerContext

(IN) This optional structure can be initialized by making a call to NSSSGetServiceInformation (page 79) prior to use here. The flags field of the structure can take on the following values to indicate the type of context used.

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

targetObject

(IN) This is the optional RDN (relative distinguished name or "short name") of the target object that contains the user's secrets. You should have at least READ/WRITE privileges over the target object.

ssFlags

(IN) This is a set of flags for initializing secrets.

Value	Description
NSSS_ENHANCED_PROTECTION_F	Sets the enhanced protection ON for this secret.
NSSS_EP_PASSWORD_USED_F	If Enhanced Protection is turned on (using the NSSS_ENHANCED_PROTECTION flag), this optional flag specifies the use of the password supplied through the SS_WRITEEXT_T and SS_EXT_T structures.
NSSS_CHECK_SID_FOR_COLISION_F	Forces the NSSSWriteSecret (page 101) to check for the existance fo the secret in the SecretStore to prevent from overwriting a secret by returning the appropriate error.

Value	Description
NSSS_ALL_STRINGS_UNICODE_F	This flag indicates that all applicable char strings (such as targetObject, secretID, etc.) are already in Unicode and the API does not need to perform conversion from local code page to Unicode. This flag instructs the API not to convert the return char strings to local code page as well. This is a feature of SSO client Version 1.1 and will not work on the older versions.
NSSS_SET-TREE_NAME_F	Sets the tree name.
NSSS_DESTROY_CONTEXT_F	Internally destroys the DS context passed in. This flag can be used on the last call to SecretStore to destroy the context that was used.

epPassword

(IN) This is an optional field to pass in the Master Password or the Enhanced Protection Password for writing a secret. When neither one of the passwords are present, you can pass in a NULL.

secretID

(IN) This is a unique secret identifier chosen by the application that should be supplied to locate the application secret values in the user's SecretStore.

secretValue

(IN) This is a buffer that the client allocates for the application secret value and encodes the secret within it.

ext

(OUT) If present, this structure can return a set of applicable future extensions for the secrets.

Return Values

These are common return values for this function; see "Return Values" on page 61 for more information.

NSSS_SUCCESS	The operation succeeded.
NSSS_E_SYSTEM_FAILURE	Some internal operation failed due to some failure such as memory allocation.
NSSS_E_CORRUPTED_STORE	This indicates data corruption in SecretStore.
NSSS_E_NICI_FAILURE	NICI failure detected.
NSSS_E_INVALID_SECRET_ID	Secret ID is not in the SecretStore.
NSSS_E_ACCESS_DENIED	eDirectory denied access to an object or attribute related to the SecretStore.
NSSS_E_STORE_NOT_FOUND	Client does not have a SecretStore or the Schema is not extended to begin with.
NSSS_E_NDS_INTERNAL_FAILURE	Some eDirectory operation has failed internally.

NSSS_E_INCOMPATIBLE_VERSION	Client SecretStore is not compatible with server SecretStore.
NSSS_E_INVALID_TARGET_OBJECT	Target object is not the type designated for SecretStore attachment.

Remarks

The NSSSWriteSecret call begins a session with SecretStore to populate it with a new secret. This function call takes the clear copy of the application's secret from the application, encrypts it, and stores it in the user's SecretStore in eDirectory. The user of the application must be logged in and authenticated to eDirectory in order for this call to succeed.

The unique secret ID that was chosen for this application when installing the user's SecretStore is passed in as input. As a result, the object is located in the tree and the SecretStore is populated by adding the application secret values to it. Once the proper attribute value is located in the SecretStore, it is populated or overwritten with the application secret value in the incoming buffer. The application secret is encrypted and written to the user's secret associated with the target application. This call overwrites the existing value if present. This call by default creates and writes the secret and if the secret identified by the secret ID is found it will overwrite it.

The NSSS_CHECK_SID_FOR_COLLISION_F flag is used to force a check for existing secret identified by the secret ID in the SecretStore to prevent collision. The owner and other persons with proper access rights can use this function.

See Also

NSSSReadSecret (page 84)

NSSSWriteSharedSecret

Creates a SecretID according to the Shared Secret format utilizing either the prefix SS_App or SS_CredSet.

Syntax

```
#include <nssscl.h>

SS_EXTERN_LIBCALL(int) NSSSWriteSharedSecret
(
 void                 *handle,
 SS_SH_SECRET_ID_T    *pSharedSecret,
 SSS_CONTEXT_T        *context,
 SS_OBJECT_DN_T        targetObjDN,
 unsigned long        *ssFlags,
 SS_PWORD_T           *epPassword,
 SS_EXT_T             *ext
);
```

Parameters

handle

Specifies the handle created by NSSSCreateSHSHandle (page 72).

pSharedSecret

(IN) Points to the user-populated SS_SH_SECRET_ID_T struct containing the Shared Secret type, name, and length.

context

(IN) Specifies an optional structure that can be initialized by making a call to NSSSGetServiceInformation (page 79) prior to use here. The flags field of the structure can take on the following values to indicate the type of context used.

NSSS_NCP_CTX_F—Directory Service API context indicator flag

NSSS_LDAP_CTX_F—LDAP context indicator flag <reserved>

targetObjDN

(IN) This is the optional RDN (relative distinguished name or "short name") of the target object that contains the user's secrets. The caller should have at least READ/WRITE privileges over the target object.

NOTE: For binding to server over LDAP/SSL this parameter is mandatory and it should be in fully qualified LDAP form ("cn=user, ou=users, o=novell").

ssFlags

(IN) Specifies the flags used when making the call to NSSSWriteSecret (page 101).

epPassword

(IN) Specifies an optional field to pass in the Master Password or the Enhanced Protection Password for reading a secret. When neither one of the passwords are present, you can pass in a NULL.

ext

(IN) Points to the extensions used for the secrets.

Return Values

Value	Description
NSSS_SUCCESS	The operation succeeded.
NSSS_E_SYSTEM_FAILURE	Some internal operation failed due to some failure such as memory allocation.
NSSS_E_UTF8_OP_FAILURE	Utf8 string operations failed.
NSSS_E_INVALID_PARAM	API parameter is not initialized..
NSSS_E_SECRET_ID_TOO_SHORT	Length of the Secret ID should be greater than zero.
NSSS_E_SH_SECRET_FAILURE	Shared Secret processing and operations failed.
NSSS_E_SECRET_ID_EXISTS	Secret ID already exists in the Secret Store.

Remarks

The internal buffer is parsed according to the shared secret format defined by the parsing library. The resulting data is passed into the secret buffer for passage into NSSSWriteSecret (page 101) and stored as shared secrets in the SecretStore. Consequently, this function provide flexibility in making calls to trees and user DNs other than the primary connection.

See Also

NSSSReadSharedSecret (page 82)
NSSSWriteSecret (page 101)
See other Shared Secret buffer calls here.

6

SecretStore Samples

See the SecretStore samples (../../../samplecode/ssocomp_sample/index.htm) to enable both a client and server application using the SecretStore API.

- The *sstst.c* sample code is the source for sstst.exe, the NCP version of the API operations.
- The *lstst.c* sample code is the source for lstst.exe, the LDAP version of the API operations.
- The *nbstst.c* sample code is the source for nbstst.exe, the LDAP version of the API operations with binding outside of the SecretStore APIs.
- The *shtst.c* sample code is the source for shtst.exe, the NCP version of the Shared Secret functions.
- The *lshtst.c* sample code is the source for lstst.exe, the LDAP version of the Shared Secret functions.

Revision History

October 8, 2003

Revised document overall and added new conceptual topics about implementing shared secrets:

- "Understanding SecretStore Functions" on page 11
- "Understanding Shared Secret Functions" on page 18
- "Shared Secret Functions" on page 22
- "SecretStore API Enhancements" on page 24

Added new task topics to explain how to implement shared secrets:

- "Enabling and Maintaining SecretStore" on page 36
- "Modifying Shared Application or Credential Secrets" on page 40
- "Removing Shared Application or Credential Secrets" on page 42

Revised SecretStore Structure section and added the following new shared secret structures:

- SSS_GSINFOEXT_T (page 52)
- SS_SH_SECRET_ID_T (page 55)
- SS_ADV_CRED_T and SS_ADV_CERT_T (page 56)
- SS_ADDR_T (page 57)
- SS_ADV_BIND_INFO_T (page 58)
- SS_OBJECT_DN_T (page 59)

Added the following new functions to enable shared secret functionality:

- NSSSAddSHSEntry (page 70)
- NSSSCreateSHSHandle (page 72)
- NSSSDestroySHSHandle (page 73)
- NSSSGetNextSHSEntry (page 77)
- NSSSRemoveSharedSecret (page 91)
- NSSSRemoveSHSEntry (page 94)
- NSSSWriteSharedSecret (page 104)
- NSSSReadSharedSecret (page 82)
- Changed name of NSSRemoveSecretId to NSSSRemoveSecret (page 87).
- Deprecated NSSSAddSecretID (obsolete 7/03) and replaced with new shared secret functionality.

February 2002	Changed document name from “Novell Single Sign-on for C” and revised documentation to reflect SecretStore service functionality. Deleted NSSOOpenStreamAttribute function.
June 2001	Added extended optional data structure information for each function; added input and output flag definitions for the functions; revised return values; and made a number of other corrections to documentation. Also added: 1. “SecretStore Service Discovery” on page 30. 2. “Enabling SSOCOMP in Applications” on page 37. 3. NSSSGetServiceInformation (page 79). 4. NSSSSetEPMasterPassword (page 96).
September 2000	Added extended optional data structure information for each function; added input and output flag definitions for the functions; revised return values; and made a number of other corrections to documentation.
May 2000	Added new parameter support for Version 1.1 APIs.