IBM® SecureWay® Policy Director

Programming Guide and Reference

Version 3.0.1

IBM® SecureWay® Policy Director

Programming Guide and Reference

Version 3.0.1

Note

Before using this information and the product it supports, read the general information under "Appendix. Notices" on page 97.

First Edition (January 2000)

This edition applies to Version 3, release 0, modification 1 of IBM SecureWay Policy Director product and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this book

This book contains programming guide and reference information about IBM SecureWay Policy Director. This book documents these Policy Director functions:

- Authorization application programming interface (API)
- External authorization service
- Credentials acquisition service

Who should read this book

Developers who are designing and developing applications for IBM SecureWay Policy Director should read this book.

Developers should have some knowledge of IBM Distributed Computing Environment (DCE) and the IBM SecureWay Directory's lightweight directory access protocol (LDAP). DCE and LDAP are co-requisite products of Policy Director. Developers should have basic working knowledge about writing and configuring DCE and LDAP servers.

This *Policy Director Programming Guide and Reference* book assumes basic working knowledge about writing and configuring DCE servers.

How this book is organized

This book contains the following chapters:

- "Chapter 1. IBM SecureWay" on page 1 introduces you to the IBM SecureWay FirstSecure and IBM SecureWay Policy Director products.
- "Chapter 2. Authorization API" on page 3 guides the application designer or developer on the use of the Policy Director Authorization API.
- "Chapter 3. External authorization service" on page 33 describes the remote procedure call (RPC) interface as well as the interface details. This chapter discusses how to implement and customize a custom external authorization service. External authorization service-related API reference information is also provided.
- "Chapter 4. Credentials Acquisition Service" on page 41 describes the Policy Director Credentials Acquisition Service (Policy Director CAS) remote procedure call interface and the interface details. This chapter discusses how to implement and deploy a custom credentials acquisition service. CAS-related API reference information is also provided.
- "Chapter 5. Authorization API manual pages" on page 51 provides reference information about the Policy Director Authorization API. The manual pages for these API are documented in this section.

What is new in this release

This is Version 3, Release 0, Modification 1, of the Policy Directory Authorization API. This version contains changes from Version 3, Release 0. The changes reflect the Open Group Authorization API, Version 1.1, published in January 2000. The changes are described in the following sections.

New API function

The new function azn_error_minor_get_string() returns a string describing minor errors specific to the Policy Director Authorization API.

Renamed API function

The function azn_attrlist_entry_get_num() has been renamed azn_attrlist_name_get_num().

Changes to API initialization

The initialization function azn_initialize() must now be called before calling azn_util_client_authenticate() or azn_util_server_authenticate(). In the previous version, azn_util_client_authenticate() or azn_util_server_authenticate() was called before azn_initialize().

Changes to API error codes

The following changes have been made to Policy Director Authorization API error codes listed in ogauthzn.h:

Error code	**Change description**
AZN_S_INVALID_PREPEND_CREDS_HDL AZN_S_INVALID_SECURITY_CONTEXT AZN_S_APP_CONTEXT_HDL	Deleted
AZN_S_INVALID_APP_CONTEXT	Renamed AZN_S_INVALID_APP_CONTEXT_HDL
AZN_S_INVALID_ENTITLEMENTS_SVC AZN_S_INVALID_STRING_VALUE AZN_S_API_UNINITIALIZED AZN_S_API_ALREADY_INITIALIZED	Added

The following changes have been made to Policy Director Authorization API error codes listed in aznutils.h:

Error code	Change description
AZN_S_U_FAILURE AZN_S_U_CERTIFICATION_FAILED AZN_S_U_PASSWORD_INVALID AZN_S_U_NO_MEMORY AZN_S_U_INVALID_BUFFER AZN_S_U_INVALID_ELEMENT	Deleted
AZN_S_U_INVALID_MECH_ID	Renamed AZN_S_U_INVALID_MECH_ID_REF

Changes to API function parameters

The following table summarizes changes to Authorization API function parameters.

Function	Change description

azn_initialize()
azn_decision_access_allowed_ext()
	Output parameter is now a pointer to a handle to an attribute list. The output parameter of type azn_attrlist_h_t is automatically allocated by the Authorization API. Previously it was necessary to call azn_attrlist_create() to allocate the new attribute list.

azn_creds_get_pac()
azn_attrlist_get_entry_buffer_value()
azn_util_password_authenticat()
	Output parameter is now a pointer to a buffer_t. The Authorization API automatically allocates storage for the buffer structure referred to by the buffer_t pointer. Previously the application had to allocate the storage manually.

azn_creds_get_attrlist_for_subject()	The contents of the credential are now added to the returned attribute list. The string constant attribute names for the contents are defined in the header file ogauthzn.h. A string value is returned for each constant that is supplied. For example, the constant AZN_C_VERSION will contain the Authorization API version number.

azn_release_buffer()
azn_release_string()
azn_release_strings()
azn_creds_delete()
azn_attrlist_delete()
	Input parameter is now a pointer to the data structure to be freed. The function sets the input pointer to NULL when the function returns, in order to ensure that the pointer cannot be used by any other functions.

azn_creds_combine()	The input parameter creds_to_add has been renamed creds. The input parameter creds_to_prepend has been renamed creds_to_add.

What is new between Version 3.0 and Version 2.1

The Policy Director Version 3.0 Authorization API is binary compatible, at the Authorization server remote procedure call (RPC) interface, with applications that are built with the Policy Director Version 2.1 Authorization API. Applications developed with the Policy Director Version 2.1 Authorization API library must be ported before they can be compiled against the Policy Director authorization ADK.

The Policy Director Authorization Service fully supports applications that are built using the Policy Director Version 2.1 Authorization API.

Note that the Policy Director authorization API now requires authentication with the Authorization server (ivacld) before API functions are called.

Policy Director Version 2.1 API applications are required to be members of the remote-acl-servers group before they are permitted to query the Policy Director Authorization Service. Add the application principal to this group to effect this change. You must log the principal out and log in again to create a security context with the new group membership.

Other changes since Policy Director Version 2.1 include:

- Addition of local cache mode.
- Revision of the Authorization API to reflect the standardized Authorization API submission made to The Open Group.
- Addition of initialize and shutdown functions to allow optional reconfiguration of the API.

The following table lists functions and data types that have been deactivated for Policy Director. It lists the new functions and data types that have replaced them.

Version 2.0 Functions and Data Types	**Version 3.0 Functions and Data Types**
ivAuthznInit()	azn_initialize()

ivBuildLocalPrincipal()
ivBuildPrincipalByName()
ivBuildPrincipalFromPAC()
ivBuildUnauthPrincipal()
	azn_id_get_creds()
	azn_pac_get_creds()

ivCheckAuthorization()	azn_decision_access_allowed()
	azn_decision_access_allowed_ext()

ivFreePrincipal()	azn_creds_delete()
ivServerLogin()	azn_util_server_authenticate()
ivauthzn_init_params_t	attribute lists
ivauthzn_service_mode_t	attribute lists

In addition, the *Policy Director Up and Running* book provides information about what is new for IBM SecureWay Policy Director Version 3.0.

Year 2000 readiness

This product is Year 2000 ready. When used in accordance with its associated documentation, it is capable of correctly processing, providing, and/or receiving date data within and between the twentieth and twenty-first centuries, provided that all products (for example, hardware, software, and firmware) used with the products properly exchange accurate date data with it.

Service and support

Contact IBM for service and support for all the products included in the IBM SecureWay FirstSecure offering. Some of these products might refer to non-IBM support. If you obtain these products as part of the FirstSecure offering, contact IBM for service and support.

Conventions

This book uses the following typographical conventions:

Convention	**Meaning**
bold	User interface elements such as check boxes, buttons, and items inside list boxes.
`monospace`	Syntax, sample code, and any text that the user must type.
Italic	Emphasis and first use of special terms that are relevant to Policy Director.
>	Shows a series of selections from a menu. For example, click **File > Run** means click **File**, and then click **Run**.

Web information

Information about last-minute updates to Policy Director is available at the following Web address:

`http://www.ibm.com/software/security/policy/library`

Information about updates to other IBM SecureWay FirstSecure products is available by starting at the following Web address:

`http://www.ibm.com/software/security/firstsecure/library`

Chapter 1. IBM SecureWay

IBM SecureWay Policy Director (Policy Director) is available either as a component of IBM SecureWay FirstSecure or as a standalone product.

What is IBM SecureWay FirstSecure?

IBM SecureWay FirstSecure (FirstSecure) is part of the IBM integrated security solution. FirstSecure is a comprehensive set of integrated products that help your company:

- Establish a secure e-business environment.
- Reduce the total cost of security ownership by simplifying security planning.
- Implement security policy.
- Create an effective e-business environment.

The IBM SecureWay products include:

Policy Director
IBM SecureWay Policy Director (Policy Director) provides authentication, authorization, data security, and Web resource management.

Boundary Server
IBM SecureWay Boundary Server (Boundary Server) provides:

- The critical firewall functions of filtering, proxy, and circuit level gateway
- A virtual private network (VPN) connection to the IBM Firewall
- The components for Internet security
- A mobile code security solution

A configuration graphical user interface (GUI) ties together the Policy Director's proxy user function with the Boundary Server's Firewall product.

Intrusion Immunity
Intrusion Immunity provides intrusion detection and antivirus protection.

Trust Authority
IBM SecureWay Trust Authority (Trust Authority) supports public key infrastructure (PKI) standards for cryptography and interoperability. Trust Authority provides support for issuance, renewal, and revocation of digital certificates. These certificates provide a means to authenticate users and to ensure trusted communications.

Toolbox
The IBM SecureWay Toolbox (Toolbox) is a set of application programming interfaces (API) with which application programmers can incorporate security into their software. You can obtain the Toolbox as part of FirstSecure. Both Policy Director and the Toolbox include the Policy Director API library and documentation. The Toolbox README file contains installation instructions for the Policy Director ADK.

Because each IBM SecureWay FirstSecure product can be installed independently, you can plan a controlled move toward a secure environment. This capability reduces the complexity and cost of securing your environment and speeds deployment of Web applications and resources.

See the FirstSecure *Planning and Integration* documentation for more information about the FirstSecure components and for a list of all the IBM SecureWay products' documentation.

What is IBM SecureWay Policy Director?

Policy Director is a standalone authorization and security management solution. Policy Director provides end-to-end security of resources over geographically dispersed intranets and *extranet*s. An *extranet* is a virtual private network (VPN) that uses access control and security features to restrict the use of one or more intranets attached to the Internet to selected subscribers.

Policy Director provides authentication, authorization, data security, and resource-management services. You can use Policy Director in conjunction with standard Internet-based applications to build secure and well-managed intranets and extranets.

Policy Director runs on the Windows NT, AIX, and Solaris operating systems.

Chapter 2. Authorization API

This chapter includes:

Introducing the Authorization API

Using the Policy Director Authorization Application Programming Interface (API), you can code Policy Director applications and third-party applications to query the Policy Director Authorization Service for authorization decisions.

The Policy Director Authorization API is the interface between the server-based resource manager and the authorization service and provides a standard model for coding authorization requests and decisions. The Authorization API lets you make standardized calls to the centrally managed authorization service from any legacy or newly developed application.

The Authorization API supports two implementation modes:

- **Remote cache mode**

  In remote cache mode, you use the Authorization API to call the Policy Director Authorization Server, which performs authorization decisions on behalf of the application. The Authorization Server maintains its own cache of the replica authorization policy database.

- **Local cache mode**

  In local cache mode, you use the Authorization API to download a local replica of the authorization policy database. In this mode, the application can perform all authorization decisions locally.

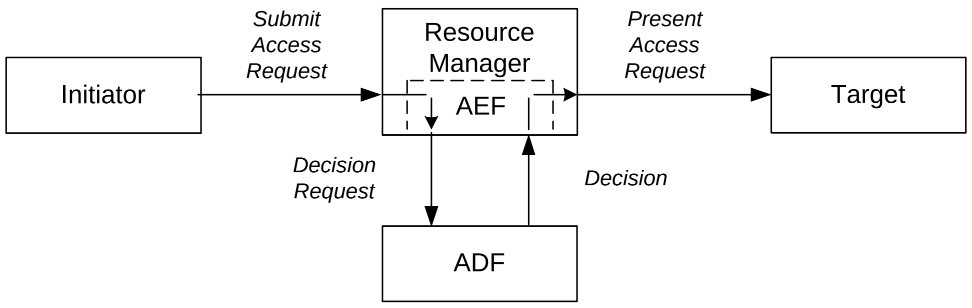
The Authorization API shields you from the complexities of the authorization service mechanism. Issues of management, storage, caching, replication, credentials format, and authentication methods are all hidden behind the Authorization API.

The Authorization API works independently from the underlying security infrastructure, the credential format, and the evaluating mechanism. The Authorization API makes it possible to request an authorization check and get a simple "yes" or "no" recommendation in return.

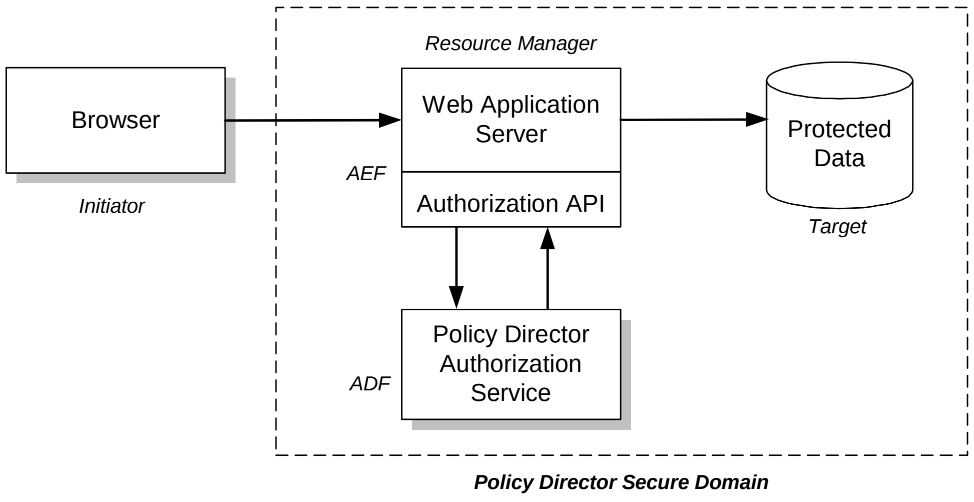
The Authorization API is a component of the Policy Director Application Development Kit (ADK).

The Open Group Authorization API standard

The Policy Director Authorization API implements The Open Group Authorization API (Generic Application Interface for Authorization Frameworks) standard. This interface is based on the International Organization for Standardization (ISO) 10181-3 model for authorization. In this model, an initiator requests access to a target resource. The initiator submits the request to a resource manager, which incorporates an access enforcement function (AEF). The AEF submits the request, along with information about the initiator, to an access decision function (ADF). The ADF returns a decision to the AEF, and the AEF enforces the decision.



Policy Director implements the ADF component of this model and provides the Authorization API as an interface to this function.



In the figure above, a browser (initiator) requests access to a file or other resource on a protected system (target). The browser submits the request to a Web application server (the resource manager incorporating the access enforcement function). The Web application server uses the Authorization API to submit the request to the Policy Director Authorization Service (the access decision function).

The Policy Director Authorization Service returns an access decision, through the Authorization API, to the Web application server. The Web application server processes the request as appropriate.

To implement this model, developers of AEF applications add Authorization API function calls to their application code.

Note: Developers should refer to the Open Group Authorization API document for additional information on the standard authorization model.

Background and references for using Policy Director authorization

The first step in adding authorization to an application is to define the security policy requirements for your application. Defining a security policy means that you must determine the business requirements that apply to the application's users, operations, and data. These requirements include:

- Objects to be secured
- Operations permitted on each object
- Users that are permitted to perform the operations

After your security requirements have been defined, you can use the Authorization API to integrate your security policy with the Policy Director security model.

Complete the following steps in order to deploy an application into an Policy Director secure domain:

1. Configure the Policy Director secure domain to recognize and support the objects, actions, and users that are relevant to your application.
   - For an introduction to the Policy Director authorization model, see "Chapter 3, Understanding authorization" in the *Policy Director Administration Guide.*
   - For complete information on access control, see "Chapter 7, Understanding Access Control" in the *Policy Director Administration Guide.*
2. Use the Authorization API within your application to obtain the needed authorization decisions.
   - For an introduction to the Authorization API, including information on remote cache mode and local cache mode, see "Chapter 3, Understanding authorization" in the *Policy Director Administration Guide.*
3. Develop your application logic to enforce the security policy.

Locating the Authorization API components

The Authorization API is included as an optional installation package in the Policy Director distribution. The Authorization API files are installed in the authzn_adk directory, directly under the Policy Director installation directory.

If you are installing the Authorization API portion of the Policy Director ADK from the Policy Director CD, the ADK is installed in the subdirectories in the following table. If you are installing the Authorization API port of the ADK from the IBM SecureWay Toolbox, refer to the Toolbox README file for installation instructions.

Directory	Contents
include	C header files
lib	A library that implements the API functions. • On Solaris systems, the library is libivauthzn.so • On AIX systems, the library is libivauthzn.a • On Microsoft Windows systems, the library to include at run time is ivauthzn.dll • On Windows, the library to link is ivauthzn.lib
authzn_demo	An example program that demonstrates usage of the Authorization API. Source files and a MAKEFILE are provided.

For Policy Director installation instructions, including the Policy Director ADK, refer to the *Policy Director Up and Running Guide.*

Header files

The header files are found in the include directory, located directly under the Policy Director Authorization ADK package installation directory.

File	Contents
ogauthzn.h	The Authorization API standard functions
aznutils.h	Utility functions (extensions to The Authorization API)

Error codes

The Authorization API error codes are defined in the following files, located in the include directory:

File	Contents
ogauthzn.h	Major error codes for the standard Authorization API functions.
aznutils.h	Major error codes for the Authorization API utility functions.
dceaclmsg.h	Minor error codes for utility functions and the Policy Director Authorization Service.

Building applications with the Authorization API

The following sections provide information on building an application with the Authorization API:

- "Software requirements" on page 7
- "Linking required libraries" on page 7

Software requirements

To develop applications that use the Policy Director Authorization API, you must install and configure a Policy Director secure domain.

If you do not have a Policy Director secure domain installed, install one before beginning application development. The minimum installation consists of a single system with the following Policy Director components installed:

- Policy Director Base (IVBase)
- Policy Director Management server (IVMgr)
- Policy Director Authorization server (IVAcld)
- Policy Director Application Development Kit (IVAuthADK)
- Policy Director Management Console (IVConsole)

If the Policy Director secure domain uses an LDAP user registry, the application development system must have an LDAP client installed.

For Policy Director installation instructions refer to the *Policy Director Up and Running* guide.

If you already have an Policy Director secure domain installed, and want to add a development system to the domain, the minimum Policy Director installation consists of the following components:

- Policy Director Base (IVBase)
- Policy Director Authorization server (IVAcld)
- Policy Director Application Development Kit (IVAuthADK)

Note: The development environment must include a DCE runtime. The DCE runtime is installed as a prerequisite to the Policy Director installations described above.

Linking required libraries

In order to compile applications that use the Authorization API, you must install the Policy Director ADK on the build machine.

When compiling your application, make sure you add the include directory for the Policy Director ADK to the compiler command line. When linking your application, specify the directory containing the authorization shared library if it is not in the default location.

On Solaris systems, you also need to link to the following libraries:

Platform	**DCE**	**Libraries**
Solaris	Transarc 2.0	libdce.so, libgssdce.so, libC.so

On AIX and Windows NT systems, you do not need to link against the DCE libraries.

On all platforms, the DCE libraries are needed at application runtime. See "Deploying applications with the Authorization API" on page 31.

Understanding the Authorization API functions and data types

The Authorization API provides a set of functions and data types. This section lists the name of each Authorization API construct and the task it accomplishes.

The following functions, structured data types, functions, and constants are defined as part of the Authorization API:

- "API functions" on page 8
- "Character strings" on page 10
- "Buffers" on page 10
- "Attribute lists" on page 11
- "Credential handles" on page 12
- "Status codes and error handling" on page 12

API functions

The following tables list the Authorization API functions and provide a reference to the section in this document that describes each function's task.

Attribute lists

Function	**Task**
"azn_attrlist_add_entry()" on page 52 "azn_attrlist_add_entry_buffer()" on page 53 "azn_attrlist_create()" on page 54 "azn_attrlist_delete()" on page 55 "azn_attrlist_get_entry_buffer_value()" on page 56 "azn_attrlist_get_entry_string_value()" on page 58 "azn_attrlist_get_names()" on page 60 "azn_attrlist_name_get_num()" on page 61	"Attribute lists" on page 11

Credentials

Function	**Task**
"azn_creds_combine()" on page 64	"Creating a chain of credentials" on page 29
"azn_creds_create()" on page 66	"Obtaining user authorization credentials" on page 22
"azn_creds_delete()" on page 67	"Releasing allocated memory" on page 28

Function	Task
"azn_creds_for_subject()" on page 68	"Obtaining a credential from a chain of credentials" on page 30
"azn_creds_get_attrlist_for_subject()" on page 70	"Obtaining an attribute list from a credential" on page 31
"azn_creds_get_pac()" on page 72	"Converting credentials to a transportable format" on page 29
"azn_creds_modify()" on page 74	"Modifying the contents of a credential" on page 30
"azn_creds_num_of_subjects()" on page 76	"Determining the number of credentials in a credentials chain" on page 30
"azn_id_get_creds()" on page 84	"Obtaining user authorization credentials" on page 22
"azn_pac_get_creds()" on page 87	"Converting credentials to the native format" on page 29

Authorization decisions

Function	Task
"azn_decision_access_allowed()" on page 77	"Obtaining an authorization decision" on page 25
"azn_decision_access_allowed_ext()" on page 79	

Initialization, shutdown, and error handling

Function	Task
"azn_error_major()" on page 81	"Status codes and error handling" on page 12
"azn_error_minor()" on page 82	
"azn_error_minor_get_string()" on page 83	
"azn_initialize()" on page 86	"Initializing the authorization service" on page 14
"azn_release_buffer()" on page 89	"Releasing allocated memory" on page 28
"azn_release_string()" on page 90	
"azn_release_strings()" on page 91	
"azn_shutdown()" on page 92	"Shutting down the Authorization API" on page 28

API extensions

Function or Data Type	Task
"azn_util_client_authenticate()" on page 94	"Logging in using a password" on page 20
"azn_util_password_authenticate()" on page 95	"Obtaining an identity for a user" on page 21
"azn_util_server_authenticate()" on page 96	"Logging in using a DCE keytab file" on page 20
"azn_authdce_t" on page 62	"Obtaining user authorization credentials" on page 22
"azn_authldap_t" on page 63	
"azn_unauth_t" on page 93	

Character strings

Many Authorization API functions take character strings as arguments or return character strings as values. Use the azn_string_t data type to pass character string data between your application and the Authorization API:

```
typedef char *azn_string_t;
```

Use azn_release_string() and azn_release_strings() to release memory that has been allocated to strings of type azn_string_t.

Buffers

Some Authorization API functions take byte string arguments and return byte strings as values. Use the data type azn_buffer_t to pass byte string data between your application and the Authorization API.

The azn_buffer_t data type is a pointer to a buffer descriptor consisting of a length field and a value field. The length field contains the total number of bytes in the data. The value field contains a pointer to the data.

```
typedef struct azn_buffer_desc_struct {
    size_t  length;
    void    *value;
} azn_buffer_desc,  *azn_buffer_t;
```

You must allocate and release the storage necessary for all azn_buffer_desc objects.

Objects of type azn_buffer_t appear as output parameters to the azn_attrlist_get_entry_buffer_value() and azn_creds_get_pac() calls. For these functions, storage for the buffer array referred to by the *value* member of an azn_buffer_desc object is allocated by the Authorization API.

Use "azn_release_buffer()" on page 89 to release storage allocated for use by azn_buffer_desc objects.

Parameters of type azn_buffer_t can be assigned and compared with the following constant values:

Name	Value	Definition
AZN_C_EMPTY_BUFFER	NULL	Empty data value-buffer.
AZN_C_NO_BUFFER	NULL	No value-buffer is supplied or returned.

Attribute lists

Several Authorization API functions take attribute list handles as input parameters or return attribute list handles as output parameters. Use the azn_attrlist_h_t data type to pass attribute list handles between the Authorization API and the calling application.

Variables of type azn_attrlist_h_t are opaque handles to lists of name and value pairs. Use Authorization API functions to add or retrieve name and value pairs from attribute lists.

Many Authorization API functions uses attribute lists to store and retrieve values. Attribute lists are lists of name and value pairs. The values can be stored as either strings or buffers. A name can have more than one value.

Some names are defined by the Authorization API. You can also define additional names as needed by your application.

The Authorization API provides functions to create attribute lists, set or get list entries, and delete attribute lists. The following table summarizes the functions that operate on attribute lists:

Task	Description
Create an attribute list	Use "azn_attrlist_create()" on page 54 to complete the following tasks: • Allocate a new, empty attribute list. • Associate a handle with the attribute list. • Return the handle.
Set an entry in an attribute list	Use "azn_attrlist_add_entry()" on page 52 to add a string name-value pair of type azn_string_t. Use "azn_attrlist_add_entry_buffer()" on page 53 to add a buffer name-value pair of type azn_buffer_t.
Get attribute names from an attribute list	Use "azn_attrlist_get_names()" on page 60 to get all the names in an attribute list, contained in an array of strings of type azn_string_t.
Get the number of values for a specified attribute name	Use "azn_attrlist_get_entry_buffer_value()" on page 56 to get the number, as an integer, of the value attributes for a specified name in the attribute list.

Task	Description
Get a value	Use "azn_attrlist_get_entry_string_value()" on page 58 to get the value attribute of a string (azn_string_t) for a specified name in an attribute list. Use "azn_attrlist_get_entry_buffer_value()" on page 56 to get the value attribute of a buffer (azn_buffer_t) for a specified name in an attribute list. The specified name can have multiple values. You specify the needed value by supplying an index (integer) into the list of values.
Delete an attribute list	Use "azn_attrlist_delete()" on page 55 to delete the attribute list associated with a specified attribute list handle.

Credential handles

A credential handle refers to a credentials chain consisting of the credentials of the initiator and a series of (zero or more) intermediaries through which the initiator's request has passed.

Several Authorization API functions take credentials handles as input parameters or return pointers to credential handles as output parameters. Use the azn_creds_h_t data type to pass credential handles between the Authorization API and the calling application.

Variables of type azn_creds_h_t are opaque handles to credential structures that are internal the Policy Director security framework.

Use the function "azn_creds_create()" on page 66 to complete the following tasks:

- Allocate a new, empty credential structure.
- Associate a handle with the credential structure.
- Return a pointer to the handle.

Call the function "azn_creds_delete()" on page 67 on the handle to release the memory allocated for the credential structure.

Status codes and error handling

Authorization API functions return a status code of type azn_status_t. The values in azn_status_t are integers. The return value for successful completion of the function is AZN_S_COMPLETE, which is defined to be 0.

The returned status code includes both major and minor error codes. A major error code of AZN_S_FAILURE indicates that a minor error code contains the error status.

Use "azn_error_major()" on page 81 to extract major error codes from the returned status. Major error codes are defined according to the The Open Group Authorization API Standard.

Use "azn_error_minor()" on page 82 to extract minor error codes from the returned status. The minor codes contain error messages from the utility function extensions to the API, and contain error messages from the Policy Director authorization server.

Use "azn_error_minor_get_string()" on page 83 to obtain string values for the minor error codes returned by azn_error_minor().

See the following files for a complete list of error codes:

File	Contents
ogauthzn.h	Major error codes for the standard Authorization API functions.
aznutils.h	Major error codes for the Authorization API utility functions.
dceaclmsg.h	Minor error codes for utility functions and the Policy Director Authorization Service.

Summarizing Authorization API tasks

The primary task of the Authorization API is to obtain an authorization decision from the Policy Director Authorization Service.

Use the Authorization API to present information about the user, operation, and requested resource to the Policy Director Authorization Service. Then use the Authorization API to receive the authorization decision. Your application is responsible for enforcing the decision, as appropriate.

Required tasks

To obtain an authorization decision, you must accomplish certain tasks. The following sections in this document provide a step-by-step guide to completing each of these required tasks:

- "Initializing the authorization service" on page 14
- "Authenticating an API application" on page 19
- "Obtaining an identity for a user" on page 21
- "Obtaining user authorization credentials" on page 22
- "Obtaining an authorization decision" on page 25
- "Cleaning up and shutting down" on page 28

Optional tasks

The Authorization API also provides functions for performing optional tasks on user credentials. The following section describes the supported optional tasks:

- "Handling credentials (optional tasks)" on page 28

Runtime environment

To determine whether your network environment is configured correctly to support your application, review the following section:

- "Deploying applications with the Authorization API" on page 31

Initializing the authorization service

To use the Policy Director Authorization API, an application must initialize the API. Initialization consists of specifying initialization data and calling an initialization function.

The Authorization API initialization function azn_initialize() takes as an input parameter an attribute list named init_data. To specify initialization data, you must add the necessary attributes to init_data.

Complete the instructions in the following sections:

- "Specifying the type of cache mode" on page 14
- "Adding attributes for remote cache mode" on page 15
- "Adding attributes for local cache mode" on page 15
- "Adding attributes for LDAP access" on page 18
- "Starting the authorization service" on page 19

Specifying the type of cache mode

The cache mode determines if the Authorization API talks to a Policy Director Authorization server running in the same process space (local cache mode) or in a different process space (remote cache mode) in the secure domain.

Local cache mode can increase application performance because authorization checks can be performed on the same system as the application. Local cache mode, however, requires additional configuration and maintenance of a replicated authorization database.

- For more information on remote cache mode, see “Remote cache mode” in Chapter 3 of the *Policy Director Administration Guide.*
- For more information on local cache mode, see “Local cache mode” in Chapter 3 of the *Policy Director Administration Guide*.

To specify the type of cache mode, complete the following steps:

1. Call "azn_attrlist_create()" on page 54 to create a new attribute list called init_data. This function returns a pointer to an attribute list handle.
2. Use "azn_attrlist_add_entry()" on page 52 to add the attribute azn_init_mode and assign it a value:

Attribute	**Value**	**Description**
azn_init_mode	local	The Policy Director Authorization Service runs in the same server process as the application using the Authorization API.
	remote	The Policy Director Authorization Service runs as a different server process from the application using the Authorization API.

Continue to the appropriate section:

- "Adding attributes for remote cache mode" on page 15.
- "Adding attributes for local cache mode" on page 15.

Adding attributes for remote cache mode

If you specified remote cache mode, use "azn_attrlist_add_entry()" on page 52 to add the attribute azn_init_qop and assign it a value:

Attribute	Value	Description
azn_init_qop	none	No protection.
	integrity	Data stream integrity. The data can be seen but not modified or replayed by a third party.
	privacy	Data stream privacy. The data cannot be seen, modified, or replayed by a third party.

For example, the following code shows the creation of a new attribute list. It also shows the assigning of name-value pairs for cache mode (azn_init_mode) and quality of protection (azn_init_qop):

```
azn_attrlist_create(&init_data);

/*** Don't use a local replica, use the authorization server ***/
status = azn_attrlist_add_entry(init_data,
                                azn_init_mode,
                                "remote");
if (status != AZN_S_COMPLETE)
return (status);

/* Set quality of protection for communications with ivacld to be
 * privacy.
 */

status = azn_attrlist_add_entry(init_data,
                                azn_init_qop,
                                “privacy");
if (status != AZN_S_COMPLETE)
return (status);
```

Initialization of remote cache mode is now complete.

- If your secure domain uses an LDAP user registry, refer to"Adding attributes for LDAP access" on page 18.
- If your secure domain uses a DCE user registry, refer to"Starting the authorization service" on page 19.

Adding attributes for local cache mode

When you specify local cache mode, you must decide how the local copy of the authorization database will be updated.

Choose one of the following methods to implement updating:

- Set the Authorization API to poll the master authorization service database.
- Register the local (replicated) database with the master database, and enable a listener process on the local database’s system. This process listens for update notifications.
- Configure the Authorization API to both poll and listen.
- Configure the Authorization API to neither poll nor listen. This could be useful, for example, when the local system is not connected to a network.

The above methods are configured by adding attributes to the init_data attribute list.

Complete all the steps in this section in order to implement your chosen method:

1. Use azn_attrlist_add_entry() to specify pathnames for files used by the authorization service.

Attribute	**Value**	**Description**
azn_init_db_file	*filename*	Path name to the persistent authorization policy database replica.
azn_init_audit_file	*filename*	Path and file name for the file that collects Authorization API audit events.

2. Use azn_attrlist_add_entry() to configure the Authorization API to poll the master authorization database.

Attribute	**Value**	**Description**
azn_init_cache_refresh_interval		
	disable	Refreshing of the local authorization policy database disabled.
	default	600 seconds.
	number of seconds	Number of seconds between refreshes of the local authorization policy database. Set appropriate values to ensure that the replicated database is updated in a timely manner to reflect changes made to the master database.

3. Use azn_attrlist_add_entry() to configure the notification listener.

Attribute	**Value**	**Description**
azn_init_listen_flags	disable	Disable the notification listener.
	enable	Enable the notification listener.
	When you select enable, you can also specify any combination of the following values. The values are logically OR'd together.	
	use_tcp_port	Enable the listener to use Transmission Control Protocol (TCP).
	use_udp_port	Enable the listener to use User Datagram Protocol (UDP).
	dynamic_port_selection	Instruct the listener to use randomly assigned ports.

4. If you enable the notification listener, you must use the **ivadmin** command to inform the Policy Director Management server (**ivmgrd**) of your location in order to receive notification of updates. Use the **ivadmin server register dbreplica** command to inform the Policy Director Authorization Service (specifically, the Management server) of the existence and location of applications using the Authorization API in local cache mode.

The following syntax applies:

`ivadmin>server register dbreplica` *`server-name ns-location server-host`*

Where:

Option	Description
server-name	A name (or label) for this application. This is the name that appears in the display of the object space on the Management Console and in the ivadmin server list command.
ns-location	The RPC entry in the CDS namespace where the application exports its RPC bindings.
server-principal	The name of the DCE principal representing this application process.
server-host	The Domain Name System (DNS) name or IP address of the machine where this application process resides.

5. If you enabled the notification listener, use azn_attrlist_add_entry() to add the following attributes:

   Note: If you disabled the notification listener, skip this step.

Attribute	Value	Description
azn_init_tcp_port	*port number*	If you specified use_tcp_port and did not specify dynamic_port_selection for the attribute azn_init_listen_flags, use this value to specify a TCP port.
azn_init_udp_port	*port number*	If you specified use_udp_port and did not specify dynamic_port_selection for the attribute azn_init_listen_flags, use this value to specify a UDP port.
azn_init_namespace_location	*CDS location*	Specify the CDS namespace location for exporting the RPC endpoints for local policy cache updates.

For example, the following code shows the creation of a new attribute list init_data, and also shows the addition of entries to specify configuration settings for local cache mode:

```
azn_attrlist_create(&init_data);
status = azn_attrlist_add_entry(initdata,
                                azn_init_mode,
                                "local");
if (status != AZN_S_COMPLETE)
return (status);

/*** The file name of the replica policy database ***/

status = azn_attrlist_add_entry(initdata,
                                azn_init_db_file,
                                "./auth_demo.db");
if (status != AZN_S_COMPLETE)
return (status);
```

```
/*** The file name of the audit file ***/

status = azn_attrlist_add_entry(initdata,
                              azn_init_audit_file,
                              "./auth_demo.audit");
if (status != AZN_S_COMPLETE)
return (status);

/*** Enable polled updates at the default interval ***/

status = azn_attrlist_add_entry(initdata,
                              azn_init_cache_refresh_interval,
                              "default");
if (status != AZN_S_COMPLETE)
return (status);

/*** Enable the update notification listener ***/

status = azn_attrlist_add_entry(initdata,
                              azn_init_listen_flags,
                              "enable");
if (status != AZN_S_COMPLETE)
return (status);

status = azn_attrlist_add_entry(initdata,
                              azn_init_listen_flags,
                              "use_tcp_port");
if (status != AZN_S_COMPLETE)
return (status);

status = azn_attrlist_add_entry(initdata,
                              azn_init_tcp_port,
                              "6056");
if (status != AZN_S_COMPLETE)
return (status);

status = azn_attrlist_add_entry(initdata,
                              azn_init_namespace_location,
                              CDSloc);
if (status != AZN_S_COMPLETE)
return (status);
```

Adding attributes for LDAP access

When your application runs in a Policy Director secure domain that uses an LDAP user registry, you must provide the LDAP configuration settings to the Authorization API. The required LDAP configuration settings match the settings that were entered when Policy Director was installed on the local system.

Note: When your application runs in a Policy Director secure domain that uses a DCE user registry, skip this step and go to"Starting the authorization service" on page 19.

1. Use azn_attrlist_add_entry() to add the following attributes to the init_data attribute list:

Attribute	**Value**	**Description**
azn_init_ldap_host	*host name*	Host name of LDAP server.
azn_init_ldap_port	*port number*	Port number for communicating with the LDAP server.
azn_init_ldap_admin_dn	*LDAP DN*	Distinguished Name of the LDAP administrator.

Attribute	Value	Description
azn_init_ldap_admin_pwd	*password*	Password for the LDAP administrator.

2. If the communication between the Policy Director Authorization server and the LDAP server is over Secure Sockets Layer (SSL), use azn_attrlist_add_entry() to add the following attributes to the init_data attribute list:

Attribute	Value	Description
azn_init_ldap_ssl_keyfile	*filename*	Name of the SSL key file.
azn_init_ldap_ssl_keyfile_dn	*KeyLabel*	Key label to identify the client certificate that is presented to the LDAP server.
azn_init_ldap_ssl_keyfile_pwd	*password*	Password to access the SSL key file.

Starting the authorization service

Complete the following steps:

1. Ensure that the attribute list init_data has been created and filled in, as described in the preceding sections.
2. Call azn_initialize() to bind to and initialize the authorization service.

   For example:

```
/* Start the service */
status = azn_initialize(init_data, &init_info);
if (status != AZN_S_COMPLETE)
   return(status);
```

In the example code above, azn_initialize**()** returns the attribute list init_info. This attribute list is appended with any initialization information attributes that apply. This includes the AZN_C_VERSION attribute, which contains the version number of the API implementation.

Note: To re-initialize the API, use azn_shutdown**()** and then call azn_initialize**()**.

For more information, see "azn_initialize()" on page 86.

Authenticating an API application

The API application must establish its own authenticated identity within the Policy Director secure domain, in order to request authorization decisions from the Policy Director Authorization Service.

Before you run the Authorization API application for the first time, you must create a unique identity for the application in the Policy Director secure domain.

In order for the authenticated identity to perform API checks, the application must be a member of at least one of the following groups:

- **ivacld-servers**

  This group membership is needed for applications using local cache mode.

- **remote-acl-users**

  This group membership is needed for applications using remote cache mode.

When the application wants to contact one of the secure domain services, it must first log in to the secure domain.

The Policy Director Authorization API provides two utility functions the application can use to log in and obtain an authenticated identity. One function performs a login by using username and password information. The other function performs a DCE login by using a keytab file.

Use the appropriate API login functions, as described in the following sections:

- **"Logging in using a DCE keytab file"** on page 20
- **"Logging in using a password"** on page 20

Logging in using a DCE keytab file

Some application servers are executed non-interactively, such as in response to an access request from an application client. These application servers must establish an authenticated identity without manual intervention by an administrator.

To avoid the need for manual intervention, the application developer can create and store a password in a keytab file.

The Authorization API utility function azn_util_server_authenticate() submits the user name and the name of the keytab file to the Policy Director authentication service. The Policy Director authentication service can use the DCE keytab file to establish an authenticated identity.

For example, the following code logs in a server svrPrin using a keytab file svrKeytab:

```
status = azn_util_server_authenticate(svrPrin, svrKeytab);
if ( status != AZN_S_COMPLETE ) {
   fprintf(stderr, "Could not perform keytab login\n");
   exit(1);
}
```

Note: You can use azn_util_server_authenticate() in a Policy Director secure domain that uses an LDAP user registry, but it can only be used for DCE principals (as registered in a DCE user registry).

For more information, see "azn_util_server_authenticate()" on page 96.

Logging in using a password

Some applications might be used by more than one identity in the Policy Director secure domain. These applications can choose their login identity based on application requirements. For example, the application can prompt the user, or examine user information contained in an HTTP header, or simply supply a username and password that denotes a category of user.

The Authorization API provides the utility function azn_util_client_authenticate() to enable the application to log in as a specific identity with a user name and password.

For example, the following code logs in the application as “testuser”:

```
/* Login and start context refresh thread */
status = azn_util_client_authenticate(testuser, testuserpwd);
if ( status != AZN_S_COMPLETE ) {
    fprintf(stderr, "Could not perform client login\n");
    exit(1);
}
```

You can use azn_util_client_authenticate() in a Policy Director secure domain with a DCE user registry.

For more information, see "azn_util_client_authenticate()" on page 94.

Obtaining an identity for a user

The application must determine the identity of the user who has submitted a request. The identity can be expressed as one of the following types of users:

- **Authenticated**

  In this case, the user’s identity in the secure domain is registered in either an LDAP or DCE user registry. The user is authenticated, and information about the user can be obtained. This information includes, for example, the Distinguished Name (LDAP) or principal (DCE).

- **Unauthenticated**

  In this case, the user’s identity in the secure domain is not specifically registered in either an LDAP or DCE user registry. The user is defined to be unauthenticated, and further information about the user’s identity is irrelevant to the authorization process.

Applications can obtain user identities through a variety of methods. These can include the use of a Credentials Acquisition Server, or a call to an application-specific method for querying user registries and establishing a security (login) context.

Optionally, applications can use the Policy Director Authorization API utility function azn_util_password_authenticate() to obtain user identity information from the secure domain.

The function azn_util_password_authenticate() requires the user name and password as input parameters. Typically, an application receives a user name and password from the user who initiated the access request.

The function performs a login using the supplied user name and password. If the login is successful, the function returns the following information:

- The string mechanism_id, which specifies the authentication mechanism (DCE or LDAP) that was used.
- A pointer to the buffer authinfo**,** which contains user identity information.

Note: The function azn_util_password_authenticate() does not obtain a security (login) context for the user.

For more information, see "azn_util_password_authenticate()" on page 95.

After the application has obtained identity information for the user, you can use the Authorization API to obtain authorization credentials for the user.

Obtaining user authorization credentials

In order to submit an authorization request to the Policy Director Authorization Service, an application must obtain authorization credentials for the user making the request. The authorization credentials contain user identity information that is needed to make authorization decisions, such as group memberships and a list of actions or rights that the user can exercise.

To obtain credentials for a user who has submitted an access request, an application must obtain user identity information from the user registry (DCE or LDAP) that is used by the Policy Director secure domain.

The Authorization API function azn_id_get_creds() takes user identity information as input parameters and returns user authorization credentials.

The credentials can then be submitted to the authorization service for an authorization decision.

Note: Identity information can also be obtained from a privilege attribute certificate (PAC). See "Converting credentials to the native format" on page 29.

To obtain a credential, complete the instructions in each of the following sections:

1. "Specifying the authorization authority" on page 22
2. "Specifying authentication user registry type" on page 22
3. "Specifying user authentication identity" on page 23
4. "Specifying additional user information" on page 23
5. "Placing user information into an API buffer" on page 24
6. "Obtaining authorization credentials for the user" on page 24

Specifying the authorization authority

Assign the appropriate value for the authorization authority to a string of type azn_string_t. This string is passed as the parameter authority to azn_id_get_creds(). Set authority to NULL to specify Policy Director authorization.

Specifying authentication user registry type

Applications must know the type of user registry used in the Policy Director secure domain, in order to obtain an authenticated identity for the user. The type of registry used was determined in "Obtaining an identity for a user" on page 21.

If the user was not authenticated in a user registry, then the user registry type is unauthenticated.

Assign a value for the type of user authentication identity to a string of type azn_string_t. This string is passed as the parameter mechanism_id to azn_id_get_creds().

Set mechanism_id to one of the following values:

User Registry	Value
DCE User Registry	IV_DCE
LDAP User Registry	IV_LDAP
Unauthenticated	IV_UNAUTH

Specifying user authentication identity

For each user to be authenticated, information is loaded into the data structure that corresponds to the type of user registry used in the secure domain, or is loaded into a data structure corresponds to a user category of unauthenticated.

If the user is authenticated, you must load the user's identity into the appropriate string in the data structure that corresponds to the user registry type.

User Identity Type	Data Structure	String	Example
DCE User Registry	azn_authdce_t	principal	cell_admin
LDAP User Registry	azn_authldap_t	ldap_dn	cn=root
Unauthenticated User	azn_unauth_t	*none*	*none*

If the user is unauthenticated, you do not have to load an identity into azn_unauth_t.

Specifying additional user information

When the application authenticates the user, the application can optionally obtain additional information about the user. This additional information is for use by the application as needed. The Policy Director Authorization Service does not use this information.

The application can store the additional user information in the data structures that the Authorization API provides for each type of authenticated identity. The data structures are: azn_authdce_t, azn_authldap_t, and azn_unauth_t.

The elements in each data structure are character strings, with the exception of ipaddr, which is an integer.

Element	Description
auth_method	Indicates that the user was authenticated through either the DCE user registry or the LDAP user registry. This value can be any string that is useful to the application. Not available in azn_unauth_t.
authnmech_info	Additional authentication information. This value can be any string that is useful to the application. For example, if the DCE authentication was accomplished using SSL certificates, the certificate's Distinguished Name could be stored here. Not available in azn_unauth_t.
qop	Quality of protection level for requests made by this user. This level is set by the application and is specified as an arbitrary character string.

Element	Description
user_info	Additional user information for auditing purposes. This string can contain any information that is useful to the application.
browser_info	Information about the type of browser through which the user has submitted the request, if applicable. This string can contain any information that is useful to the application.
ipaddr	The IP address of the user. This is optional information for use by the application.

Placing user information into an API buffer

Place the data structure you filled out in"Specifying user authentication identity" on page 23 and"Specifying additional user information" on page 23 into an Authorization API buffer.

Complete the following steps:

1. Declare a buffer of type azn_buffer_t:

```
typedef struct azn_buffer_desc_struct {
size_t  length;
void    *value;
} azn_buffer_desc,  *azn_buffer_t;
```

2. Determine the length of your data structure and assign that value to length.
3. Set the pointer value to point to the address of your data structure.

This buffer is passed as the parameter mechanism_info to azn_id_get_creds()**.**

Obtaining authorization credentials for the user

To obtain authorization credentials, call azn_id_get_creds() with the following input parameters:

Parameter	Description
authority	The authorization authority, as described in "Specifying the authorization authority" on page 22.
mechanism_id	The authentication mechanism, as described in "Specifying authentication user registry type" on page 22.
mechanism_info	User information, as described in the following sections: • "Specifying user authentication identity" on page 23. • "Specifying additional user information" on page 23 • "Placing user information into an API buffer" on page 24

The azn_id_get_creds() function returns a handle to the authorization credentials for the user. The authorization credentials are contained in an azn_creds_h_t structure.

For example, the following sample code demonstrates the assigning of identity information for a user authenticated in an LDAP user registry, and calls azn_id_get_creds() to obtain authorization credentials:

```
azn_authldap_t ldap_minfo;
azn_string_t mech = NULL;
azn_buffer_desc buf = { 0, 0 };
azn_creds_h_t creds;

azn_creds_create(&creds);

/* Specify authentication registry type */
mech = IV_LDAP;

/* Specify LDAP user name */
ldap_minfo.ldap_dn = “cn=testuser”;

/* Set LDAP user information. Note: these values are just placeholders
*/
ldap_minfo.auth_method = "ldap_auth_method";
ldap_minfo.authnmech_info = "ldap_authnmech_info";
ldap_minfo.qop = "ldap_qop";
ldap_minfo.user_info = "ldap_user_info";
ldap_minfo.browser_info = "ldap_browser_info";
ldap_minfo.ipaddr = 0x0a000002;

/* Set a buffer to point to the LDAP user information */
buf.length = sizeof(ldap_minfo);
buf.value = (unsigned char *)&ldap_minfo;

/* Obtain an authorization credential. Specify the authority as NULL */

status = azn_id_get_creds(NULL, mech, &buf, &creds);
if (status != AZN_S_COMPLETE) {
    fprintf(stderr, "Could not get creds.\n");
    continue;
}
```

For more information, see "azn_id_get_creds()" on page 84. Refer also to the Authorization API demonstration program. See "Example program authzn_demo" on page 32.

The application is now ready to submit the authorization request. See "Obtaining an authorization decision" on page 25.

Obtaining an authorization decision

After the application has obtained authorization credentials for the user, the application passes the requested operation and the requested resource to the Authorization API function azn_decision_access_allowed(). This function returns the authorization decision.

To obtain an authorization decision, complete the instructions in each of the following sections:

- "Mapping the user operation to an Policy Director permission" on page 26
- "Mapping the requested resource to a protected object" on page 26
- "Assigning the user credentials to a credentials handle" on page 26
- "Building an attribute list for additional application information" on page 26
- "Obtaining an authorization decision" on page 27

Mapping the user operation to an Policy Director permission

The operation requested by the user must correspond to one of the operations for which an Policy Director permission has been defined. The operation is a standard action supported in all Policy Director secure domains. Examples operations are azn_operation_read and azn_operation_traverse.

Note: For a complete list of supported operations, see the file aznutils.h.

Alternatively, the operation can be a custom operation defined by an external authorization service.

- Assign the operation to a string named “operation”**.** Pass this string as an input parameter to azn_decision_access_allowed().

Mapping the requested resource to a protected object

The requested resource to query for must correspond to a resource that has been defined as a protected object in the secure domain’s protected object namespace.

The resource can be a standard WebSEAL protected resource, such as a file in the Web space. Alternatively, the resource can be a custom protected object.

Complete the following step:

- Assign the protected object to the string protected_resource. Pass this string as an input parameter to azn_decision_access_allowed().

Assigning the user credentials to a credentials handle

The authorization credentials for a user obtained in "Obtaining user authorization credentials" on page 22 can be accessed through the handle returned by azn_id_get_creds().

These credentials contain the user’s identity information and include information such as the user’s group membership and permitted operations.

Complete the following step:

- Pass the handle returned by azn_id_get_creds() as an input parameter to azn_decision_access_allowed().

Note: Authorization credentials can also be obtained from azn_pac_get_creds(). See "Converting credentials to the native format" on page 29.

Building an attribute list for additional application information

The Policy Director Authorization API provides the extended function azn_decision_access_allowed_ext() for obtaining an access decision. This function extends azn_decision_access_allowed() by providing an additional input parameter and an additional output parameter.

These parameters can be used to supply additional information as needed by the application. The Policy Director Authorization Service does not use these parameters when making the access control decision. However, you can write external authorization servers to use this information.

The parameters consist of an attribute list. You can build an attribute list of any length to hold information specific to the application.

To add additional application-specific context, complete the following steps:

1. Use azn_attrlist_create() to create a new, empty attribute list.
2. Use azn_attrlist_add_entry() or azn_attrlist_add_entry_buffer() to add attributes.
3. When all attributes have been added, assign the input parameter app_context to point to the attribute list.

For more information, see "azn_decision_access_allowed_ext()" on page 79.

Obtaining an authorization decision

To obtain an authorization decision, call one of the following functions:

- azn_decision_access_allowed()
- azn_decision_access_allowed_ext()

If the API is operating in remote cache mode, the authorization request will be forwarded to the Policy Director Authorization server (**ivacld**). The Authorization Server makes the decision and returns the result.

If the API is operating in local cache mode, the API uses the local authorization policy database replica to make the authorization decision.

The result of the access request is returned in the following output parameter:

Type	**Parameter**	**Description**
int	*permission*	The result of the access request. Consists of one of the following constants: AZN_C_PERMITTED AZN_C_NOT_PERMITTED

The extended function azn_decision_access_allowed_ext() also returns the following information:

Type	**Parameter**	**Description**
azn_attrlist_h_t	**permission_info*	Application-specific context information contained in attribute list.

For more information on the above functions, see:

- "azn_decision_access_allowed()" on page 77
- "azn_decision_access_allowed_ext()" on page 79

Cleaning up and shutting down

The Authorization API provides functions to perform the following clean up and shut down functions:

- "Releasing allocated memory" on page 28
- "Shutting down the Authorization API" on page 28

Releasing allocated memory

The Authorization API provides the following functions to perform the releasing of memory functions:

- "azn_attrlist_delete()" on page 55

  Use this function to release memory that is allocated for attribute lists.
- "azn_creds_delete()" on page 67

  Use this function to release memory that is allocated for the azn_creds_h_t structure that is returned by a call to azn_creds_create().
- "azn_release_buffer()" on page 89

  Use this function to release memory that is allocated for buffers of type azn_buffer_t. Buffers of this type are used by some attribute list functions, and also by some of the credentials handling functions.
- "azn_release_string()" on page 90

  Use this function to release memory allocated for any strings of type azn_string_t. Many Authorization API functions use this data type to store values in strings.
- "azn_release_strings()" on page 91

  Use this function to release memory allocated for an array of strings of type azn_string_t.

Shutting down the Authorization API

When an application has obtained an authorization decision and when it does not need further authorization decisions, use "azn_shutdown()" on page 92 to disconnect from and shut down the Authorization API.

Handling credentials (optional tasks)

The Authorization API provides functions to accomplish the following optional tasks:

- "Converting credentials to a transportable format" on page 29
- "Converting credentials to the native format" on page 29
- "Creating a chain of credentials" on page 29
- "Determining the number of credentials in a credentials chain" on page 30
- "Obtaining a credential from a chain of credentials" on page 30
- "Modifying the contents of a credential" on page 30
- "Obtaining an attribute list from a credential" on page 31

Converting credentials to a transportable format

Use the function "azn_creds_get_pac()" on page 72 to place user credentials into a format that can be transported across a network to another application. Use this function when you need to delegate the authorization decision to an application on another system.

Complete the following steps:

1. Set the input string pac_svc_id to NULL.
2. Set the input credentials handle creds to the credentials handle returned by a previous call to azn_id_get_creds() or azn_pac_get_creds().
3. Call azn_creds_get_pac().

The privilege attribute certificate (PAC) is returned in an output buffer named pac. This buffer can be transported to another system, where the function azn_pac_get_creds() can be used to return the credentials to a native format.

Converting credentials to the native format

Use the function "azn_pac_get_creds()" on page 87 when an application receives credentials from another system on the network. Typically, these credentials are placed into a buffer by azn_creds_get_pac().

Complete the following steps:

1. Set the input string pac_svc_id to NULL.
2. Set the input buffer pac to the buffer returned by a previous call to azn_creds_get_pac().
3. Call azn_pac_get_creds().

This function returns a handle to a credentials structure (azn_creds_h_t**)**, for access by other Authorization API functions.

Creating a chain of credentials

Use the function "azn_creds_combine()" on page 64 to combine, or chain, two credentials together. Use this, for example, when the credentials for a server application must be combined with user credentials in order to delegate the authorization decision to another application.

Complete the following steps:

1. Assign the credentials handle creds_to_prepend to point to the credentials of the initiator of the request.
2. Assign the credentials handle creds_to_add to point to the credentials to be added.
3. Call azn_creds_create() to create a new, empty credentials structure.
4. Call azn_creds_combine().

The combined credentials are placed in a credentials structure that can be referenced by the credentials handle combined_creds.

Determining the number of credentials in a credentials chain

Use the function "azn_creds_num_of_subjects()" on page 76 to determine the number of credentials that are contained in a credentials chain. Credentials chains are created by the azn_creds_combine() function.

This functions takes as an input parameter the credentials handle of the credentials chain, and returns an integer containing the number of credentials.

Obtaining a credential from a chain of credentials

Use the function "azn_creds_for_subject()" on page 68 to extract individual credentials from a credentials chain. Credentials chains are created by the azn_creds_combine() function.

Complete the following steps:

1. Assign the credentials handle creds to point to the credentials chain.
2. Assign the integer subject_index the index of the needed credential within the credentials chain.

   The credentials of the user who made the request are always stored at index 0. To retrieve the credentials for the initiator (user), you can pass the constant AZN_C_INITIATOR_INDEX as the value for subject_index.

   Use azn_creds_num_of_subjects(), if necessary, to determine the number of credentials in the chain.
3. Call azn_creds_for_subject().

This function returns the requested credentials in the credentials structure *new_creds*.

Modifying the contents of a credential

Use the function "azn_creds_modify()" on page 74 to modify a credential by placing additional information, contained in an attribute list, into the credentials structure. Use this function when you need to add application-specific information to a user's credentials.

Complete the following steps:

1. Use the attribute list functions to create an attribute list containing the information to be added. Assign the attribute list handle mod_info to the new attribute list.

   For more information on attribute lists, see"Attribute lists" on page 11.
2. Set the credential modification service mod_svc_id to NULL.
3. Assign the credentials handle creds to point to the credentials to be modified.
4. Call azn_creds_create() to create a new, empty credentials structure.
5. Call azn_creds_modify().

The modified credentials are placed in the credentials structure *new_creds*.

Obtaining an attribute list from a credential

Use the function "azn_creds_get_attrlist_for_subject()" on page 70 to obtain information, in the form of an attribute list, from a credential. Attribute lists are added to credentials structures by calls to azn_creds_modify()**.**

You can use this function to obtain the attribute list for a credential that is part of a credentials chain.

Complete the following steps:

1. Assign the credentials handle creds to point to the credentials chain.
2. Assign the integer subject_index to the index of the credential within the credentials chain.

   If the credential is not part of a chain, set subject_index to 0.

   The credentials of the user who made the request are always stored at index 0. To retrieve the credentials for the initiator (user), you can pass the constant AZN_C_INITIATOR_INDEX as the value for subject_index.

   Use azn_creds_num_of_subjects(), if necessary, to determine the number of credentials in the chain.
3. Call azn_attrlist_create() to create a new, empty attribute list.
4. Call azn_creds_get_attrlist_for_subject().

The function returns a pointer to a handle to the attribute list containing the credential’s attribute information. The handle is named creds_attrlist**.**

Deploying applications with the Authorization API

To deploy an application with the Authorization API, verify that your environment contains the necessary supporting software. You can test your environment by building and running the example program that is provided with the Authorization API.

See the following sections:

- "Software requirements" on page 31
- "Example program authzn_demo" on page 32

Software requirements

Applications that have been developed with the Policy Director Authorization API must be run on systems that are configured into an Policy Director secure domain. When the Policy Director secure domain uses an LDAP user registry, the application deployment system must have an LDAP client installed.

The minimum Policy Director installation on a system that will run an application is:

- Policy Director Base (IVBase)
- Policy Director Authorization server (IVAcld)
- Policy Director Application Development Kit (IVAuthADK)

DCE client runtime requirements

The application runtime environment must include a DCE client runtime. The DCE runtime is installed as a prerequisite to the Policy Director installations described above.

Note: On Windows NT, Policy Director NetSEAT client provides the DCE client runtime environment.

Example program authzn_demo

The Policy Director Authorization API is provided with an example program called **authzn_demo** that demonstrates use of the Authorization API. The example directory contains source files and a MAKEFILE. Refer to the README file, located in the same directory, for information regarding the use of this example program.

Chapter 3. External authorization service

This chapter contains:

Introducing the external authorization service

Note: This *Policy Director Programming Guide and Reference* assumes basic working knowledge about writing and configuring DCE servers.

An *external authorization service* is an optional extension of the Policy Director Authorization Service that allows you to impose additional authorization controls and conditions. These additional controls and conditions are dictated by a separate (external) authorization server program.

External authorization capability is automatically built into the Policy Director Authorization Service. If you configure an external authorization server, the Policy Director Authorization Service simply incorporates the new controls and conditions into its evaluation process.

Applications that use the Policy Director Authorization Service — such as WebSEAL, NetSEAL, and any application using the Policy Director Authorization API — benefit from the additional, but seamless, contribution of a configured external authorization server. Any addition to the security policy through the use of an external authorization service is transparent to these applications and requires no change to the applications.

The external authorization service architecture allows the full integration of an organization’s existing security service. The external authorization service preserves a company’s initial investment in security mechanisms by allowing legacy servers to be incorporated into the Policy Director authorization decision-making process.

The following general steps are required to set up an external authorization service:

1. Write a server program that can be referenced during an authorization decision.
2. Configure the server into a DCE environment.
3. Register the external authorization server with Policy Director.

After the service is registered, a new permission that represents this service appears in the Policy Director Management Console. You can now use this permission in any access control list (ACL) entry to force the authorization mechanism to include the external authorization server in the decision-making process.

When the permission is encountered during an authorization check, the external authorization server is referenced for additional authorization decisions.

Additional References:

- “External Authorization Capability” in Chapter 3 of the *Policy Director Administration Guide.*
- “Chapter 11. Managing the authorization service” in the *Policy Director Administration Guide.*

  This section includes information on how to register an external authorization server with Policy Director.

Using the remote procedure call interface

The Policy Director Authorization Service uses the extern_auth IDL interface to request an authorization decision from an external authorization server.

The extern_auth interface specifies a single remote procedure call (RPC):

- check_authorization()

This RPC is called by the Policy Director Authorization Service whenever an occurrence of the external authorization permission is encountered during an ACL check.

See the following sections for interface details.

Interface Definition Language (IDL): extern_auth.idl

This IDL specifies a single RPC exported by all external authorization servers.

```
[
   uuid(4df55494-e9b8-11d0-bb97-00c078500253),
   pointer_default(ptr),
   version(2.0)
]

interface extern_auth {

   import "auth_base.idl";

   /*
    * FUNCTION NAME
    *    check_authorization
    *
    * DESCRIPTION
    *    This function is called by Policy Director as part of the
   authorization
    *    check, if required by the appropriate ACL.
    *
    * ARGUMENTS
    *    handle        Server binding handle.
    *    principals    Authenticated delegation chain.
    *    obj_name      Protected object name.
    *    req_perm      Requested capabilities.
    *    acl_perm      Capabilities granted by the ACL on the
   protected
    *                  object.
    *    req_state     Opaque protected-object specific state
   information.
    *    qop           Returns minimum acceptable quality of
   protection.
    *    status        Returns status.  Returns error_status_ok if
   request is
    *                  authorized.
    */
   void check_authorization(
      [in]          handle_t              handle,
      [in]          ivprincipal_chain_t   *principals,
      [in, string]  char                  *obj_name,
      [in]          unsigned32            req_perm,
      [in]          unsigned32            acl_perm,
      [in]          ivauthzn_state_t      *req_state,
      [out]         ivqop_t               *qop,
      [out]         error_status_t        *status
   );

}
```

Attribute configuration file

```
interface extern_auth {
   check_authorization([comm_status,fault_status] status);
}
```

Implementing a custom external authorization server

The Policy Director product includes the external authorization service interface and demonstration server source as part of the IVAuthADK installation package. The demonstration server is designed to be used as a a starting point for implementing your own customized external authorization server.

Source files

The demonstration server source is included as an example and starting point for building customized external authorization servers. All the external authorization service source files are located in the eas_adk directory, directly under the Policy Director installation directory.

Supported platforms

The external authorization service source files can be compiled on any platform. The custom built executable must reside on a machine within WebSEAL's secure domain.

Pre-requirements

The external authorization service prerequisites include:

- DCE application development tools must be installed on the build machine.

  These tools are normally included as part of an installation package. Specifically, you must install DCE header files and the IDL compiler.
- A platform-specific C compiler and development environment.

Build process

The external authorization service source directory contains a MAKEFILE that builds appropriate interface files and demonstration files. In most cases, after you install the required packages on the build machine, you will be able to compile the server files with only minor modification to the MAKEFILE.

When building a custom external authorization server, you should not modify any of the interface files, such as the IDL and attribute configuration file (ACF). These files are used to communicate with the Policy Director Security Manager. Any changes to the interface files can potentially disrupt the communication process between the Policy Director Security Manager and the external authorization server and possibly produce undesired results.

Configuring a custom external authorization service

Perform the following sequence of tasks to configure Policy Director to use an external authorization service:

1. Write the server program.

   This program must be a DCE server that exports the extern_auth IDL interface (see "Using the remote procedure call interface" on page 34). Additionally, the server must maintain a DCE login context and be able to accept authenticated RPCs.

   Note: Refer to the *DCE Application Development Guide* for details about writing a DCE server.

   Refer to "Implementing a custom external authorization server" on page 36.

2. Use the DCE program command dcecp to create a DCE account for the external authorization server. In general, this requires the following steps:

   a. Create a new principal representing the external authorization server. For example:

      ```
      dcecp> principal create eas_server
      ```

   b. Add the principal to a group. For example:

      ```
      dcecp> group add none -member eas_server
      ```

   c. Add the principal to an organization: For example:

      ```
      dcecp> organization add none -member eas_server
      ```

   d. Create an account that reflects the above information plus a password. For example (entered as one line):

      ```
      dcecp> account create eas_server -group none -organization \
             none -password dascom
      ```

   Note: Refer to the appropriate DCE documentation for detailed information.

3. Create the RPC entry in the CDS namespace where the external authorization server exports its RPC bindings. For example:

   ```
   dcecp> rpcentry create /.:/subsys/intraverse/eas_server
   ```

   This entry is used by the Policy Director Authorization Service to locate the server.

   - The external authorization server must ensure that its bindings are exported to this CDS entry.
   - If the server is replicated, each replica must also export its bindings to the same CDS location.

   Note: Refer to the appropriate DCE documentation for detailed information.

4. Set the correct permissions on the RPC entry so that the server principal has read (r) and write (w) capabilities. For example (entered as one line):

   ```
   dcecp> acl modify /.:/subsys/intraverse/eas_server -entry
          -add {user eas_server rw}
   ```

   Note: Refer to the appropriate DCE documentation for detailed information.

5. Create a DCE key table (keytab) that the server principal can access when it logs in. For example (entered as one line):

   ```
   dcecp> keytab create eas_server -storage
          /opt/intraverse/eas_adk/eas_server.key -data {eas_server
          plain 1 ibm}
   ```

   Note: Refer to the appropriate DCE documentation for detailed information.

6. Register the service with Policy Director using the **ivadmin server register** command. Use the information created in Steps 2 and 3 above as arguments to this command. For example (entered as one line):

   ```
   ivadmin> server register externauth easserver
   /.:/subsys/intraverse/eas_server none k External-Authorization
   ```

   Refer to “Registering an external authorization service” in Chapter 11 of the *Policy Director Administration Guide* for details on registering an external authorization service.

Reference: interface implementation

- "check_authorization()" on page 39

check_authorization()

Policy Director calls this function as part of an authorization check, if required by an external authorization ACL.

Syntax

```
void check_authorization(
   [in]              handle_t handle,
   [in]              ivprincipal_chain_t *principals,
   [in, string]      char *obj_name,
   [in]              unsigned32 req_perm,
   [in]              unsigned32 acl_perm,
   [in]              ivauthzn_state_t *req_state,
   [out]             ivqop_t *qop,
   [out]             error_status_t *status
);
```

Parameters

handle **- input**
: Server binding handle.

principal **- input**
: Authenticated delegation chain. This data structure can be directly cast into an azn_creds_h_t for use with the Authorization API.

obj_name **- input**
: Protected object name.

req_perm **- input**
: Requested capabilities.

acl_perm **- input**
: Capabilities granted by the ACL on the protected object.

req_state **- input**
: Opaque protected object-specific state information.

qop **- output**
: Minimum acceptable quality of protection.

status **- output**
: Return status. Returns error_status_ok if request is authorized.

Remarks

This function performs an extended authorization check from an external authorization server. This call is made only when required by the specific ACL that controls access to an external authorization server.

Return Values

None.
: Success or failure status is returned in the *status* output parameter.

Chapter 4. Credentials Acquisition Service

This chapter contains:

Introducing the Credentials Acquisition Service

Note: This *Policy Director Programming Guide and Reference* assumes basic working knowledge about writing and configuring DCE servers.

The Policy Director Credentials Acquisition Service (Policy Director CAS) extends the authentication capabilities of Policy Director. A CAS allows authentication and mapping of non-Policy Director user identity information (such as a non-Policy Director username and password, or X.509 client-side certificate) to a Policy Director user (principal). The Policy Director Security Manager (using its default registry) can then return credentials for this principal. A CAS also provides password management services.

The specifics of this authentication and mapping service are determined entirely by the CAS developer or designer. Mapping rules are stored in a database external to Policy Director.

To allow setup of a CAS, Policy Director provides:

- The IDL interface between WebSEAL and the CAS.
- The general server framework that handles CAS server functions, such as startup, server registration, and signal handling.

It is the CAS developer's responsibility to extend the CAS framework to implement the identity mapping functions required by the particular application.

Additional References:

- "Credentials acquisition" in Chapter 2 of the *Policy Director Administration Guide.*
- "Credentials acquisition service overview" in Chapter 2 of the *Policy Director Administration Guide.*
- "X.509 certificate mapping mode" in Chapter 2 of the *Policy Director Administration Guide.*
- "Username mapping mode" in Chapter 2 of the *Policy Director Administration Guide.*
- "Policy Director Credentials Acquisition Service" in Chapter 13 of the *Policy Director Administration Guide.*

Using the remote procedure call interface

WebSEAL uses the **cdas** IDL interface to request identity mapping or password management services from a credentials acquisition server.

The **cdas** interface specifies two remote procedure calls (RPC):

- cdas_get_identity()
- cdas_change_password()

These RPCs are called by the Policy Director Authorization Service whenever the CAS is called to perform authentication.

See the following section for interface details.

IDL: cas_auth.idl

This IDL specifies two RPCs and four data structures that are exported by all external credentials acquisition servers. The following authentication styles are currently supported by this interface:

- No Authentication
- Username, Password / Passticket
- Public Key Certificates

```
[
   uuid(04f8642a-0fae-11d3-b3df-0a0000c6aa77),
   pointer_default(ptr),
   version(1.0)
]


interface cdas {

   /*
    * Authentication style constants
    */
   const unsigned32 IVAUTHN_STYLE_NONE = (0);/* No authn
                                              *  information
                                              */
   const unsigned32 IVAUTHN_STYLE_PASSWORD = (1); /* Secret key
                                                   * authn
                                                   */
   const unsigned32 IVAUTHN_STYLE_CERT       = (2); /* Public key
                                                     * authn
                                                     */
   const unsigned32 IVAUTHN_STYLE_TOKEN_CARD = (3); /* SecurID-style
                                                     * authn
                                                     */
   const unsigned32 IVAUTHN_STYLE_ANONYMOUS = (4); /* Username only
                                                    * authn
                                                    */

   /*
    * cdas_authn_info_t
    *
    * This data structure conveys all client authentication
   information
    * required by a CDAS.
    */
   typedef struct {
      union switch (unsigned32 authn_style) data {
         case IVAUTHN_STYLE_NONE:
         ;                                  /* No data */
```

```
        case IVAUTHN_STYLE_PASSWORD:
            struct {
                [string] char *username;      /* Client username */
                [string] char *password;      /* Client password */
            } password_data;
        case IVAUTHN_STYLE_CERT:
            struct {
                unsigned32 cert_chain_len; /* Length of cert chain */
                    [size_is(cert_chain_len)]
                byte    *cert_chain;          /* Certificate chain */
            } cert_data;
        case IVAUTHN_STYLE_TOKEN_CARD:
            struct {
                [string] char *username;     /* Client username */
                [string] char *pin;          /* Client PIN number */
                [string] char *token;        /* Current valid token */
            } token_card_data;
        case IVAUTHN_STYLE_ANONYMOUS:
            struct {
                [string] char *username;      /* Client username */
            } anonymous_data;
    } authn_data;
    unsigned32     ipaddr;        /* Client IP address */
    [string] char  *qop;          /* Client quality of protection */
    [string] char  *browser_info; /* Client browser type
                                   *(if present)
                                   */
} cdas_authn_info_t;


/*
 * Constants that represent how a resulting client identity is
 * conveyed by the CDAS.
 */
const unsigned32 IVAUTHN_PRIN_TYPE_NAME = (0); /* Principal name */
const unsigned32 IVAUTHN_PRIN_TYPE_DN   = (1); /* Distinguished
                                                * name
                                                */

/*
 * cdas_xattr_t
 *
 * An extended attribute for use in an attribute list.
 *
 * Fields:
 *
 * name    string name for the attribute.
 * value   string value for the attribute.
 *
 */
typedef struct {
    [string] char    *name;
    [string] char    *value;
} cdas_xattr_t;


/*
 * cdas_xattr_list_t;
 *
 * A list of extended attributes
 *
 * Fields:
 *
 * count     number of attribute structs in list.
 * list      a list of attribute structs.
 *
 */
typedef struct {
    unsigned32          count;
        [size_is(count)]
    cdas_xattr_t        *list;
} cdas_xattr_list_t;
```

```
/*
 * cdas_identity_t
 *
 * This data structure conveys the resulting client identity
 * information returned from a CDAS upon successful authentication.
 */
typedef struct {
   union switch (unsigned32 prin_type) data {
       case IVAUTHN_PRIN_TYPE_NAME:
           [string] char    *name;    /* Resulting principal name */
       case IVAUTHN_PRIN_TYPE_DN:
           [string] char   *dn;       /* Resulting principal DN */
   } prin;
   [string] char   *user_info;       /* Audit information */
   [string] char   *authnmech_info;  /* Authn mechanism info */
   cdas_xattr_list_t xattrs;         /* Extended attributes */
} cdas_identity_t;


/*
 * cdas_get_identity()
 *
 * This function performs client authentication on behalf of
WebSEAL
 * and returns the resulting client identity upon success. If the
 * client has presented a certificate to Policy Director, it is the
role of
 * the CDAS to perform the relevant client identity mapping.
 *
 * [in] h:
 *  RPC binding handle.
 *
 * [in] authn_info:
 *  Client authentication information. This will be one of:
 *    1) username/password pair
 *    2) an ASN1 encoded chain of certificates.
 *    3) A tokencard username/pin/token
 *      NOTE: Not supported in the current version of WebSEAL.
 *
 *    4) Anonymous (username only)
 *      NOTE: Not supported in the current version of WebSEAL.
 *
 *      Other useful client-related information is also included.
 *      Please see cas_auth.idl for more details about the
 *      cdas_authn_info_t structure
 *
 * [out] client_id:
 *  The resulting client identity (upon successful authentication).
 *  Set to point to NULL on failure
 *
 * [out] st:
 *  Used for reporting RPC communication errors and server errors
 *  processing the request. For now it is assumed that
 *  error_status_ok will be returned, otherwise another
 *  binding/server interface will be tried for the same request
 *  until one works or none are left.
 */
void cdas_get_identity(
   [in]handle_t         h,
   [in]cdas_authn_info_t *authn_info,
   [out]cdas_identity_t   **client_id,
   [out]error_status_t   *st
);


/*
 * cdas_change_password()
 *
 * This function enables the caller to manage a given client's
 * password information.
 *
 * [in] h:
    *  RPC binding handle.
    *
    * [in] username:
    * Username associated with the client account upon which the
    * password modification should be made.
    *
    * [in] old_password:
    *  Password associated with the client account that should be
    *  changed.
    *
    * [in] new_password:
    *  New password to associate with the client account.
    *
    * [out] st:
    *  Used for reporting both RPC communication errors and server
    *  errors processing the request. For now it is assumed that
    *  error_status_ok will be returned, otherwise another
    *  binding/server interface will
    *  be tried for the same request until one works or none are left.
    */
   void cdas_change_password(
      [in]           handle_t          h,
      [in, string]   char              *username,
      [in, string]   char              *old_password,
      [in, string]   char              *new_password,
      [out]          error_status_t    *st
   );
}
```

Attribute configuration file

```
interface cdas
{
   cdas_get_identity(
      [comm_status,fault_status] st
   );

   cdas_change_password(
      [comm_status,fault_status] st
   );
}
```

Implementing a custom Credentials Acquisition Service

Policy Director WebSEAL includes the Policy Director CAS interface and the Policy Director CAS server files. The source files provided with the ADK are a starting point for implementing your own customized CAS server.

Source files

The CAS server's source files are located in the cdas_adk directory, located directly under the Policy Director installation directory.

If you install the Policy Director IVNet package, the Policy Director CAS is automatically installed. See the Policy Director README file available on the Web for information on how to configure it for use.

In addition, source files are included with Policy Director as an example and starting point for building customized CAS servers. CAS error messages are contained in the file dceiasmsg.h.

If you create a customized CAS server from the CAS source files, replace the Policy Director CAS server binary with your own custom server binary. Name your server cdas_server. Place it in the bin directory located in the cdas_server directory. The cdas_server directory is located directly under the Policy Director installation directory.

Supported platforms

The CAS source files can be compiled on any platform. The custom built executable must reside on a machine within WebSEAL's secure domain.

Pre-requirements

The Credentials Acquisition Service prerequisites include:

- DCE application development tools must be installed on the build machine

  These tools are normally included as part of an installation package. Specifically, you must install DCE header files and the IDL compiler.
- A platform-specific C compiler and development environment

Build process

The CAS source directory contains a MAKEFILE that builds appropriate interface files and demonstration files. In most cases, after you install the required packages on the build machine, you can compile the CAS server files with only minor modification to the MAKEFILE.

When building a custom CAS server, you should not modify any of the interface files (IDL and ACF). These files are used to communicate with the Policy Director Security Manager. Any changes to the interface files can potentially disrupt the communication process between the Policy Director Security Manager and the CAS server and possibly produce undesired results.

Deploying a custom Credentials Acquisition Service

Perform the following sequence of tasks to deploy a credentials acquisition server in a Policy Director environment.

1. Install the IVAuthADK package to obtain the CAS source files (located in the **cdas_adk** directory).
2. Modify the CAS server source files as needed.
3. Build a new binary server file.

   See "Build process" on page 46..
4. Copy this new server file to the **cdas_server/bin** directory.
5. Configure WebSEAL to point to the appropriate CAS CDS location.

   See “Policy Director Credentials Acquisition Service” in Chapter 13 of the *Policy Director Administration Guide*.

Note: Note that the level of accountability in a many-to-one mapping is not fine-grained. Auditing services can track only the Policy Director user (principal), not the individual users mapped to this principal.

Reference: interface implementation

- "cdas_get_identity()" on page 48
- "cdas_change_password()" on page 50

cdas_get_identity()

Performs client authentication on behalf of WebSEAL and returns the resulting client identity upon success.

Syntax

```
void cdas_get_identity(
   [in]    handle_t           h,
   [in]    cdas_authn_info_t  *authn_info,
   [out]   cdas_identity_t    **client_id,
   [out]   error_status_t     *st
);
```

Parameters

h **- input**
RPC binding handle.

authn_info **- input**
Client authentication information. This will be one of:

- Username and password pair
- An ASN1 encoded chain of certificate

client_id **- output**
The resulting client identity (upon successful authentication).

st **- output**
Used for reporting both RPC communication errors and server errors processing the request. Returns error_status_ok upon success.

Remarks

This remote procedure call performs client authentication on behalf of Policy Director and returns the resulting client identity upon success. When the client presents a certificate to Policy Director, it is the role of the CAS to perform the relevant client identity mapping.

Return Values

Success or failure status is returned in the “st” output parameter.

The following error messages are included in dceiasmsg.h:

ivauthn_account_expired
The configured expiry period associated with the client's account has elapsed.

ivauthn_authentication_failure
An authentication mechanism failed a client request to authenticate. The reason for the failure is specific to the authentication method, but often is due to the presentation of incorrect authentication information.

ivauthn_bad_authentication_info
The authentication information format is incorrect.

ivauthn_internal_error
The authentication switch encountered an unexpected internal error.

ivauthn_invalid_username
Could not locate the client's username in the authentication registry.

ivauthn_out_of_memory
A request to allocate a memory buffer was denied by the operating system.

ivauthn_password_expired
The configured expiry period associated with the client's password has elapsed.

ivauthn_retry_limit_reached
Client has reached the allowable limit of invalid consecutive invalid authentication attempts.

ivauthn_unknown_error
An unexpected error code.

cdas_change_password()

Enables the application to manage a given client’s password information.

Syntax

```
void cdas_change_password(
   [in]            handle_t       h,
   [in, string]    char           *username,
   [in, string]    char           *old_password,
   [in, string]    char           *new_password,
   [out]           error_status_t *st
);
```

Parameters

h **- input**
RPC binding handle.

username **- input**
User name associated with the client account upon which the password modification should be made.

old_password **- input**
Password associated with the client account that should be changed.

new_password **- input**
New password to associate with the client account.

st **- output**
Used for reporting both RPC communication errors and server errors when processing the request. Returns **error_status_ok** upon success.

Remarks

This remote procedure call enables the caller to change a given client’s password information.

Return Values

Success or failure status is returned in the *st* output parameter.The following error messages are included in dceiasmsg.h:

ivauthn_account_expired
The configured expiry period associated with the client's account has elapsed.

ivauthn_general_chpass_fail
Could not change user password.

ivauthn_incorrect_curr_passwd
Old password does not match existing password.

ivauthn_internal_error
The authentication switch encountered an unexpected internal error.

ivauthn_invalid_username
Could not locate the client's username in the authentication registry.

ivauthn_out_of_memory
A request to allocate a memory buffer was denied by the operating system.

ivauthn_unknown_error
An unexpected error code.

Chapter 5. Authorization API Manual Pages

This section discusses the following Authorization API:

- azn_attrlist_add_entry()
- azn_attrlist_add_entry_buffer()
- azn_attrlist_create()
- azn_attrlist_delete()
- azn_attrlist_entry_get_num()
- azn_attrlist_get_entry_buffer_value()
- azn_attrlist_get_entry_string_value()
- azn_attrlist_get_names()
- azn_authdce_t
- azn_authldap_t
- azn_creds_combine()
- azn_creds_create()
- azn_creds_delete()
- azn_creds_for_subject()
- azn_creds_get_attrlist_for_subject()
- azn_creds_get_pac()
- azn_creds_modify()
- azn_creds_num_of_subjects()
- azn_decision_access_allowed()
- azn_decision_access_allowed_ext()
- azn_error_major()
- azn_error_minor()
- azn_error_minor_get_string()
- azn_id_get_creds()
- azn_initialize()
- azn_pac_get_creds()
- azn_release_buffer()
- azn_release_string()
- azn_release_strings()
- azn_shutdown()
- azn_unauth_t
- azn_util_client_authenticate()
- azn_util_password_authenticate()
- azn_util_server_authenticate()

azn_attrlist_add_entry()

Adds a name or string-value entry to an attribute list

Syntax

```
azn_status_t
azn_attrlist_add_entry(
    azn_attrlist_h_t attr_list,
    azn_string_t attr_name,
    azn_string_t string_value
);
```

Parameters

attr_list **- input**
Handle to an attribute list.

attr_name **- input**
Name attribute of the entry to be added.

string_value **- input**
Value (string) attribute of the entry to be added.

Remarks

This function adds an entry to the attribute list *attr_list*. The added entry will have name *attr_name* and value *string_value*.

This call can be issued multiple times with the same *attr_list* and the same *attr_name* but with different string values. When this is done, *attr_list* contains multiple values for the specified name.

The *attr_name* and *string_value* input parameters are copied into a new attribute list entry.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major()**.**

AZN_S_COMPLETE
Successful completion.

AZN_S_INVALID_ATTRLIST_HDL
Attribute list handle is invalid.

AZN_S_INVALID_ATTR_NAME
Attribute name is invalid.

AZN_S_INVALID_STRING_VALUE
Attribute value is invalid.

AZN_S_FAILURE
An error or failure has occurred. Use azn_minor_error() to derive specific minor error codes from the returned status code.

azn_attrlist_add_entry_buffer()

Adds a name/buffer value entry to an attribute list.

Syntax

```
azn_status_t
azn_attrlist_add_entry_buffer(
    azn_attrlist_h_t attr_list,
    azn_string_t attr_name,
    azn_buffer_t buffer_value
);
```

Parameters

attr_list **- input**
Handle to an attribute list.

attr_name **- input**
Name attribute of the entry to be added.

buffer_value **- input**
Value (buffer) attribute of the entry to be added.

Remarks

This function adds an entry to the attribute list, *attr_list*. The added entry will have name *attr_name* and value *buffer_value*.

This function can be issued multiple times with the same *attr_list* and the same *attr_name*, but with different *buffer_values*. When this is done, *attr_list* contains multiple values for the specified name.

The *attr_name* and *buffer_value* input parameters are copied into a new attribute list entry.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major.

AZN_S_COMPLETE
Successful completion.

AZN_S_INVALID_ATTRLIST_HDL
Attribute list handle is invalid.

AZN_S_INVALID_ATTR_NAME
Attribute name is invalid.

AZN_S_INVALID_BUFFER
Attribute buffer is invalid.

AZN_S_FAILURE
An error or failure has occurred. Use azn_minor_error() to derive specific minor error codes from the returned status code.

azn_attrlist_create()

Creates a valid, empty attribute list, assigns it a handle, and returns the handle.

Syntax

```
azn_status_t
azn_attrlist_create(
    azn_attrlist_h_t *new_attr_list
);
```

Parameters

new_attr_list **- output**
: Pointer to the new attribute list handle that is returned.

Remarks

This function creates a new and empty attribute list, assigns it a handle *new_attr_list*, and returns a pointer to the handle.

When *new_attrlist* is no longer needed, its storage should be released by calling azn_attrlist_delete().

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
: Successful completion.

AZN_S_INVALID_ATTRLIST_HDL
: Attribute list handle is invalid.

AZN_S_FAILURE
: An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

azn_attrlist_delete()

Deletes the attribute list associated with the attribute list handle.

Syntax

```
azn_status_t
azn_attrlist_delete(
    azn_attrlist_h_t *old_attr_list
);
```

Parameters

old_attr_list **- input**
On input, a pointer to an existing attribute list handle.

old_attr_list **- output**
On output, a NULL pointer containing an invalid value.

Remarks

This function deletes the attribute list associated with the handle *old_attr_list*. This function will set the input attribute list handle to an invalid value to ensure that it cannot be used in future functions.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_INVALID_ATTRLIST_HDL
Attribute list handle is invalid.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

azn_attrlist_get_entry_buffer_value()

Returns a single specified value attribute for a name attribute that has multiple values that are contained in buffers.

Syntax

```
azn_status_t
azn_attrlist_get_entry_buffer_value(
   azn_attrlist_h_t attr_list,
   azn_string_t attr_name,
   unsigned int value_index,
   azn_buffer_t *buffer_value
);
```

Parameters

attr_list **- input**
Handle to an attribute list.

attr_name **- input**
Name attribute of the entry from which the value attribute is to be returned.

value_index **- input**
Index within the entry of the value attribute to be returned.

buffer_value **- output**
Pointer to an allocated buffer that holds the value of the returned attribute.

Remarks

This function returns one buffer-type value attribute in *buffer_value*. The returned value attribute is the one at position *value_index* within the entry whose name attribute is specified by *attr_name*. The first value attribute for any particular name attribute within an attribute list has index 0.

When buffer_value is no longer needed, its storage should be released by calling azn_release_buffer().

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_major_error().

AZN_S_COMPLETE
Successful completion.

AZN_S_INVALID_ATTRLIST_HDL
Attribute list handle is invalid.

AZN_S_INVALID_ATTR_NAME
Attribute name is invalid.

AZN_S_INVALID_BUFFER_REF
Buffer reference is not valid.

AZN_S_ATTR_VALUE_NOT_BUFFER_TYPE
The value attributes of this entry are not of type buffer.

AZN_S_ATTR_INVALID_INDEX
Index is not valid (no value exists for this index).

AZN_S_FAILURE

An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

azn_attrlist_get_entry_string_value()

Returns a single specified value attribute for a name attribute that has multiple values that are strings.

Syntax

```
azn_status_t
azn_attrlist_get_entry_string_value(
    azn_attrlist_h_t attr_list,
    azn_string_t attr_name,
    unsigned int value_index,
    azn_string_t *string_value
);
```

Parameters

attr_list **- input**
Handle to an existing attribute list.

attr_name **- input**
Name attribute of the entry from which the value attribute is to be returned.

value_index **- input**
Index within the entry of the value attribute to be returned.

string_value **- output**
Pointer to a string that holds the returned value attribute.

Remarks

This function returns one string-type value attribute in *string_value*. The returned value attribute is the one at position *value_index* within the set of value attributes belonging to the name attribute that is specified by *attr_name*.The first value attribute for a specified name attribute within an attribute list has index 0.

When *string_value* is no longer needed, call azn_release_string() to release its storage.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_INVALID_ATTRLIST_HDL
Attribute list handle is invalid.

AZN_S_INVALID_ATTR_NAME
Attribute name is invalid.

AZN_S_INVALID_STRING_REF
String reference is invalid.

AZN_S_ATTR_VALUE_NOT_STRING_TYPE
Value attributes of this entry are not of type string.

AZN_S_ATTR_INVALID_INDEX
Index is invalid (no value exists for this index).

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

azn_attrlist_get_names()

Returns the list of all name attributes appearing in entries of the attribute list.

Syntax

```
azn_status_t
azn_attrlist_get_names(
   azn_attrlist_h_t attr_list,
   azn_string_t *attr_names[]
);
```

Parameters

attr_list **- input**

Handle to an existing attribute list

attr_names **- output**

Pointer to an array of NULL-terminated strings that hold the returned list of name attributes. The last entry in the array is denoted by a NULL azn_string_t.

Remarks

This function returns a list of names attributes as an array of NULL terminated strings. When the *attr_names* array is no longer required, call azn_release_strings() to release its storage.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE

Successful completion.

AZN_S_INVALID_ATTRLIST_HDL

Attribute list handle is invalid.

AZN_S_INVALID_STRING_REF

String reference is invalid.

AZN_S_FAILURE

An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

azn_attrlist_name_get_num()

Returns the number of value attributes for a specified name attribute in a specified attribute list.

Syntax

```
azn_status_t
azn_attrlist_name_get_num(
   azn_attrlist_h_t attr_list,
   azn_string_t attr_name,
   unsigned int *num_values
);
```

Parameters

attr_list **- input**
Handle to an existing attribute list.

attr_name **- input**
Name attribute for the entry whose number of value attributes is to be returned.

num_values **- output**
Pointer to an integer through which the number of value attributes (in the entry whose name attribute is specified by *attr_name*) is returned.

Remarks

This function returns the number of value attributes for a specified name attribute in a specified attribute list.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_INVALID_ATTRLIST_HDL
Attribute list handle is invalid.

AZN_S_INVALID_ATTR_NAME
Attribute name is invalid.

AZN_S_INVALID_INTEGER_REF
Integer reference is invalid.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

azn_authdce_t

Contains information for use in building an authenticated authorization credential for a user within the Policy Director secure domain.

Syntax

```
typedef struct {
   azn_string_t principal;
   azn_string_t auth_method;
   unsigned int ipaddr;
   azn_string_t qop;
   azn_string_t user_info;
   azn_string_t browser_info;
   azn_string_t authnmech_info;
} azn_authdce_t;
```

Values

principal
: Name of the DCE user (principal).

auth_method
: String that indicates use of the DCE authentication method. The content of the string is defined by the application.

ipaddr
: IP address of requesting user.

qop
: Quality of protection that is required for requests that are made by this user.

user_info
: Additional user information that might be required for auditing.

browser_info
: Browser (if any) employed by the user.

authnmech_info
: Additional authentication mechanism information. Supplied and used as needed by the application.

Remarks

This DCE information structure is passed into the azn_id_get_creds() interface. Authenticated DCE users must provide a DCE user name (principal) that can be used to retrieve more user-specific authorization credentials. Values in all fields, except for *principal***,** are specified by the application for use, as needed, by the application.

azn_authldap_t

Contains information for use in building an authenticated authorization credential for a user within the Policy Director secure domain.

Syntax

```
typedef struct {
   azn_string_t ldap_dn;
   azn_string_t auth_method;
   unsigned int ipaddr;
   azn_string_t qop;
   azn_string_t user_info;
   azn_string_t browser_info;
   azn_string_t authnmech_info;
} azn_authldap_t;
```

Values

ldap_dn
: LDAP distinguished name.

auth_method
: String that indicates use of the LDAP authentication method. The content of the string is defined by the application.

ipaddr
: IP address of requesting user.

qop
: Quality of protection that is required for requests that are made by this user.

user_info
: Additional user information that might be required for auditing.

browser_info
: Browser (if any) that is employed by the user.

authnmech_info
: Additional authentication mechanism information. Supplied and used as needed by the application.

Remarks

This LDAP information structure is passed into the azn_id_get_creds() interface. Authenticated LDAP users must provide a LDAP distinguished name that can be used to retrieve more user-specific authorization credentials. Values in all fields, except for *ldap_dn***,** are specified by the application for use, as needed, by the application.

azn_creds_combine()

Combines two authorization credentials chains and a returns a pointer to a handle to the resulting combined credentials chain.

Syntax

```
azn_status_t
azn_creds_combine(
   azn_creds_h_t creds,
   azn_creds_h_t creds_to_add,
   azn_creds_h_t *combined_creds
);
```

Parameters

creds **- input**
: Handle to credentials chain whose first indexed entry is the credential of the initiator of the request.

creds_to_add **- input**
: Handle to the credentials chain to be appended to *creds*.

combined_creds **- output**
: Pointer to a handle to the returned new credentials chain, which consists of the credentials chain referenced by *creds* followed by the credentials chain referenced by *creds_to_add*.

Remarks

This function takes a credential handle *creds_to_add*, which refers to a credentials chain, and adds it to the end of a chain of one or more credentials, which are referenced by the credential handle *creds*. The credentials chain referenced by *creds* must contain as its first indexed credential the credentials of the initiator. The credentials chain referenced by *creds* might also contain the (previously combined) credentials of one or more of the initiator's proxies. A handle to the combined credentials is returned through *combined_creds*.

The input credential handles and the credentials chains to which they refer are not modified in any way by this call. Later changes to these structures, including the releasing of their storage, will have no effect on *combined_creds*.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
: Successful completion.

AZN_S_API_UNINITIALIZED
: This function has been called before azn_initialize().

AZN_S_INVALID_CREDS_HDL
: Handle passed as *creds* is invalid.

AZN_S_INVALID_ADDED_CREDS_HDL
Credentials handle passed as ***creds_to_add*** is invalid.

AZN_S_INVALID_COMB_CREDS_HDL
Credentials handle passed as *combined_creds* is invalid.

AZN_S_UNIMPLEMENTED_FUNCTION
This function is not supported by the implementation.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_creds_create()

Creates a new, empty credentials chain, assigns it a handle, and returns a pointer to the handle.

Syntax

```
azn_status_t
azn_creds_create(
    azn_creds_h_t *creds
);
```

Parameters

creds **- output**

Pointer to the new credentials handle that is returned.

Remarks

This function creates a new, empty credentials chain, assigns it a handle, and returns a pointer to the handle.

When *creds* is no longer required, call azn_creds_delete() to release its storage.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE

Successful completion.

AZN_S_API_UNINITIALIZED

This function has been called before azn_initialize().

AZN_S_INVALID_CREDS_HDL

The credentials handle supplied is invalid.

AZN_S_FAILURE

An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_creds_delete()

Deletes the credentials chain associated with the credential handle.

Syntax

```
azn_status_t
azn_creds_delete(
   azn_creds_h_t *creds
);
```

Parameters

creds **- input**
Pointer to the handle of the credentials chain to be deleted.

creds **- output**
NULL pointer to a credentials handle that is invalid upon return.

Remarks

This function deletes the credentials chain associated with the handle *creds*. This function sets the input credentials handle to an invalid value to ensure that it cannot be used in future functions.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_API_UNINITIALIZED
This function has been called before azn_initialize().

AZN_S_INVALID_CREDS_HDL
The credentials handle supplied is invalid.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_creds_for_subject()

Returns a pointer to a handle to a credentials chain. The handle is used to extract an individual credentials chain from a longer chain containing the combined credentials chains of several subjects.

Syntax

```
azn_status_t
azn_creds_for_subject(
   azn_creds_h_t creds,
   unsigned int subject_index,
   azn_creds_h_t *new_creds
);
```

Parameters

creds **- input**

Handle to a credentials structure representing the combined credentials chain of several subjects. The combined credentials chain contains a list of 1 or more individual credentials chains.

When this function returns, the structure referred to by *creds* is unchanged[

subject_index **- input**

Index of the requested individual credentials chain within the combined credentials chain. The index of the first credentials chain in the combined credentials chain, which should be that of the initiator, is zero (0).

new_creds **- output**

Pointer to the handle to the new credentials structure that is returned.

Remarks

This function returns a handle, *new_creds*, to a credentials chain for the individual credential at index *subject_index* within the credentials chain *creds.* The chain *creds* contains the combined credentials of several subjects.

This function does not modify the input handle *creds* and the credentials chain to which it refers. Later changes to this structure, including the release of its storage, have no effect on *new_creds.*

Combined credentials chains are created by azn_creds_combine(). The first credential chain in a combined credentials chain is that of the initiator, and its index is zero (0). Callers can retrieve the credentials of the initiator by passing the constant AZN_C_INITIATOR_INDEX as the value of *subject_index*.

When *new_creds* is no longer required, use azn_creds_delete() to release its storage.

Use azn_creds_num_of_subjects() to determine the total number of credentials chains in a combined credentials chain.

Return Values

If successful, the function will returns AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_API_UNINITIALIZED
This function has been called before azn_initialize().

AZN_S_INVALID_CREDS_HDL
The credentials handle supplied as *creds* is invalid.

AZN_S_INVALID_NEW_CREDS_HDL
The pointer to the new credentials handle supplied as *new_creds* is invalid.

AZN_S_INVALID_SUBJECT_INDEX
The supplied index is invalid.

AZN_S_UNIMPLEMENTED_FUNCTION
This function is not supported by the implementation.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_creds_get_attrlist_for_subject()

Returns information from a specified subject's credentials chain within a specified (and possibly combined) credentials chain.

Syntax

```
azn_status_t
azn_creds_get_attrlist_for_subject (
   azn_creds_h_t creds,
   unsigned int subject_index,
   azn_attrlist_h_t *creds_attrlist
);
```

Parameters

creds **- input**
: Handle to a credentials chain.

subject_index **- input**
: Index of the requested individual subject within the credentials chain. The index of the first credential in the combined credentials chain, which should be that of the initiator, is zero (0).

creds_attrlist **- output**
: Pointer to the handle of an attribute list that holds the specified subject's attribute information on return.

Remarks

This function returns an attribute list containing privilege attribute information from the credentials chain for the individual subject at index *subject_index* within a credentials chain *creds*.

Combined credentials chains are created by azn_creds_combine(). The first credential chain in a combined credentials chain is that of the initiator, and its index will be zero (0). Callers can retrieve the attributes of the credentials chain of the initiator by passing the constant AZN_C_INITIATOR_INDEX as the value of *subject_index*.

This function does not modify the input handle *creds* and the credentials chain to which it refers. Later changes to *creds*, including releasing its storage, will have no effect on *creds_attrlist*.

Use the *azn_attrlist** functions to retrieve individual attribute values from *creds_attrlist*. See ogauthzn.h for a list of attribute names.

The audit identifier associated with the specified credentials structure is present in the returned attribute list. It is the value attribute of an entry whose name attribute is AZN_C_AUDIT_ID.

When *creds_attrlist* is no longer required, call azn_attrlist_delete() to release its storage.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_API_UNINITIALIZED
This function has been called before azn_initialize().

AZN_S_INVALID_CREDS_HDL
The credentials handle supplied is invalid.

AZN_S_INVALID_SUBJECT_INDEX
The supplied index is invalid.

AZN_S_INVALID_ATTRLIST_HDL
The attribute list handle supplied is invalid.

AZN_S_UNIMPLEMENTED_FUNCTION
This function is not supported by the implementation.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_creds_get_pac()

Creates and returns a privilege attribute certificate (PAC) by invoking a specified PAC service on the supplied credentials chain.

Syntax

```
azn_status_t
azn_creds_get_pac(
   azn_creds_h_t creds,
   azn_string_t pac_svc_id,
   azn_buffer_t *pac
);
```

Parameters

creds **- input**
Handle to the credentials chain whose information is used to build the PAC.

pac_svc_id **- input**
Identification (id) of the PAC service that produces the PAC.

pac **- output**
Pointer to the buffer structure that contains the returned PAC.

Remarks

This function uses the PAC service whose identification is supplied as *pac_svc_id* to build a new PAC. The PAC service uses the information in the supplied credentials chain to build the PAC. Different PAC services might produce PACs with different formats. Some PAC services can cryptographically protect or sign the PACs they produce.

When *pac_svc_id* is NULL, the default PAC service returns an architecture-independent and network-independent encoding of the specified credentials chain. This PAC can be safely transmitted. The receiver of the PAC can use azn_pac_get_creds() to decode the PAC and obtain a valid copy of the original credentials chain.

This function takes as an input parameter a handle to an existing credentials structure, and returns a pointer to the output PAC in an Authorization API buffer.

This function does not modify the input handle *creds* and the credentials chain to which it refers. Later changes to *creds*, including releasing its storage, will have no effect on *pac*.

When *pac* is no longer required, call azn_release_buffer() to release its storage.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_API_UNINITIALIZED
This function has been called before azn_initialize().

AZN_S_INVALID_CREDS_HDL
The credentials handle supplied is invalid.

AZN_S_INVALID_PAC_SVC
The privilege attribute certificate service identifier is invalid.

AZN_S_UNIMPLEMENTED_FUNCTION
This function is not supported by the implementation.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_creds_modify()

Modifies an existing credentials chain and returns a pointer to the handle to a new credentials chain containing the modifications.

Syntax

```
azn_status_t
azn_creds_modify(
   azn_creds_h_t creds,
   azn_string_t mod_svc_id,
   azn_attrlist_h_t mod_info,
   azn_creds_h_t *new_creds
);
```

Parameters

creds **- input**

Handle to the authorization credentials chain to be modified.

mod_svc_id **- input**

Identification (id) of the credential modification service.

mod_info **- input**

Attribute list containing modification service-specific or application-specific data that describes the desired credential modifications. Attribute lists that are empty are inserted into the credentials.

new_creds **- output**

Pointer to a handle to a credentials chain that contains the modified credentials chain upon return.

Remarks

This function uses the specified modification service *mod_svc_id*, and optionally an attribute list *mod_info* which contains modification information provided by the caller, to modify a copy of the supplied credentials chain *creds*. The function returns a pointer to a handle to a new credentials chain *new_creds* containing the requested modifications. The supplied credentials chain is unchanged.

When *mod_svc_id* is NULL, this function modifies an existing credential chain *creds* by adding the attribute list *mod_info* to the credentials chain, and returning the modified credential in *new_creds.*

If the input *creds* handle references a combined credentials chain with more than one element, only the first element will be modified. This is the default behavior when *mod_svc_id* is NULL. In this case, the output chain consists of the modified first element followed by unmodified copies of the remaining elements in the input combined credentials chains. The elements in the output credentials chain are kept in the same order as their counterparts in the input credentials chain.

When *new_creds* is no longer required, call azn_creds_delete() to release its storage.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_API_UNINITIALIZED
This function has been called before azn_initialize().

AZN_S_INVALID_CREDS_HDL
The credentials handle supplied is invalid.

AZN_S_ INVALID_MOD_FUNCTION
The supplied modification service identifier is invalid.

AZN_S_INVALID_ATTRLIST_HDL
The attribute list handle is invalid.

AZN_S_INVALID_NEW_CREDS_HDL
The pointer to the new credentials handle that references the new output credentials chain is invalid.

AZN_S_UNIMPLEMENTED_FUNCTION
This function is not supported by the implementation.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_creds_num_of_subjects()

Returns the number of individual subjects' credentials chains in a combined credentials chain.

Syntax

```
azn_status_t
azn_creds_num_of_subjects(
   azn_creds_h_t creds,
   unsigned int *num_of_subjects
);
```

Parameters

creds **- input**
Handle to a credentials chain.

num_of_subjects **- output**
Number of subjects whose credentials appear in the input credentials chain *creds*.

Remarks

This function returns the number of individual subjects, *num_of_subjects*, whose credentials appear in a credentials chain *creds*. Combined credentials chains are created by the azn_creds_combine() function.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_API_UNINITIALIZED
This function has been called before azn_initialize().

AZN_S_INVALID_CREDS_HDL
The credentials handle supplied is invalid.

AZN_S_ATTR_INVALID_INTEGER_REF
The integer reference is invalid.

AZN_S_UNIMPLEMENTED_FUNCTION
This function is not supported by the implementation.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_decision_access_allowed()

Makes an access control decision.

Syntax

```
azn_status_t
azn_decision_access_allowed(
   azn_creds_h_t creds,
   azn_string_t protected_resource,
   azn_string_t operation,
   int *permission
);
```

Parameters

creds **- input**
Handle to the initiator's credential chain.

protected_resource **- input**
Name of the request’s target.

operation **- input**
Name of the requested operation.

permission **- output**
Value of the returned permission.

When the returned status value is AZN_S_COMPLETE, the returned permission is either AZN_C_PERMITTED or AZN_C_NOT_PERMITTED. When the returned status code is not AZN_S_COMPLETE, the returned permission is set to AZN_C_NOT_PERMITTED.

If additional information beyond a boolean result is needed, use azn_decision_access_allowed_ext().

Remarks

This function decides whether the initiator specified by credentials *creds* is authorized to perform the operation *operation* on the target *protected_resource*. The decision is returned through *permission*.

azn_decision_access_allowed() is semantically equivalent to azn_decision_access_allowed_ext() when app_context=NULL and permission_info=NULL.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_API_UNINITIALIZED
This function has been called before azn_initialize().

AZN_S_INVALID_CREDS_HDL
The credentials handle supplied is invalid.

AZN_S_INVALID_PROTECTED_RESOURCE
The target name is invalid.

AZN_S_INVALID_OPERATION
The operation has no meaning for the specified target.

AZN_S_INVALID_PERMISSION_REF
The integer reference to return the permission is invalid.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_decision_access_allowed_ext()

Makes an access control decision using application-specific context information; returns information about why the decision was made.

Syntax

```
azn_status_t
azn_decision_access_allowed_ext(
   azn_creds_h_t creds,
   azn_string_t protected_resource,
   azn_string_t operation,
   azn_attrlist_h_t app_context,
   int *permission,
   azn_attrlist_h_t *permission_info
);
```

Parameters

creds **- input**
: Handle to the initiator's credentials chain.

protected_resource **- input**
: Name of the target of the request.

operation **- input**
: Name of the requested operation.

app_context **- input**
: Attribute list containing application-specific context access control information. A NULL value indicates there is no context access control information.

permission_info **- input**
: Pointer to an attribute list through which the implementation might return implementation-specific information about the decision. If a NULL value is passed as input, then no information will be returned.

permission **- output**
: Value of the returned permission.

When the returned status value is AZN_S_COMPLETE, the returned permission is either AZN_C_PERMITTED or AZN_C_NOT_PERMITTED. When the returned status code is not AZN_S_COMPLETE, the returned permission is set to AZN_C_NOT_PERMITTED.

permission_info **- output**
: Pointer to an attribute list through which the implementation can return implementation-specific information about the decision. When a NULL pointer is passed as input, no information is returned.

The output parameter *permission_info* can be used to return implementation-specific qualifiers to AZN_C_NOT_PERMITTED. The qualifiers can be used to assist the calling application or the initiator in formulating a request which will be authorized. Examples of such qualifiers might include: “not permitted yet,” “requires additional privilege attributes,” or “permissible with restrictions.”

Remarks

This function decides whether the initiator specified by the credentials chain *creds* is authorized to perform the operation *operation* on the target *protected_resource*. Optionally, callers can supply application-specific context access control information using the *app_context* argument. The decision is returned through *permission*.

Optionally, the implementation can return implementation-specific information about the decision through *permission_info*. For example, the information can indicate which rule was responsible for granting or denying access.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_API_UNINITIALIZED
This function has been called before azn_initialize().

AZN_S_INVALID_CREDS_HDL
The credentials handle supplied is invalid.

AZN_S_INVALID_PROTECTED_RESOURCE
The target name is invalid.

AZN_S_INVALID_OPERATION
The operation has no meaning for the specified target.

AZN_S_INVALID_PERMISSION_REF
The integer reference to return the permission is invalid.

AZN_S_INVALID_APP_CONTEXT_HDL
The attribute list handle for the context access control information (ACI) is invalid.

AZN_S_INVALID_ATTRLIST_HDL
The attribute list handle for the returned permission information is invalid.

AZN_S_UNIMPLEMENTED_FUNCTION
This function is not supported by the implementation.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_error_major()

Returns the major error code that is associated with a returned status code.

Syntax

```
unsigned int
azn_error_major(
   azn_status_t status_code
);
```

Parameters

status_code **- input**

Previously returned status code by any of the azn_* routines.

Remarks

This function returns the major error code associated with a previously returned status code.

Return Values

Any of the defined major error codes, AZN_S_*. For a list of error codes, see ogauthzn.h and aznutils.h.

azn_error_minor()

Returns the implementation-specific minor error code that is associated with a returned status code.

Syntax

```
unsigned int
azn_error_minor(
   azn_status_t status_code
);
```

Parameters

status_code **- input**

Previously returned status code by any of the azn_* routines.

Remarks

The function returns the minor error code associated with a previously returned status code.

Return Values

An implementation specific minor error code is returned. For a complete list of minor error codes, see the file dceaclmsg.h.

azn_error_minor_get_string()

Returns a string describing the implementation-specific minor error code.

Syntax

```
azn_status_t
azn_error_minor_get_string(
   unsigned int minor_error
   azn_string_t *minor_error_string
);
```

Parameters

minor_error **- input**
: Minor error code previously returned by azn_error_minor().

minor_error_string **- output**
: A string describing the condition that triggered the generation of the *minor_error* code.

Remarks

This function returns a string that describes the error corresponding to a previously returned minor error status code.

When *minor_error_string* is no longer needed, use azn_release_string() to release its storage.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
: Successful completion.

AZN_S_FAILURE
: The specified *minor_error* code is invalid, or no string describing the specified *minor_error* can be returned.

azn_id_get_creds()

Returns a handle to the credentials chain associated by a specified authorization authority with a specified identity.

Syntax

```
azn_status_t
azn_id_get_creds(
   azn_string_t authority,
   azn_string_t mechanism_id,
   azn_buffer_t mechanism_info,
   azn_creds_h_t *new_creds
);
```

Parameters

authority **- input**
: Identification (id) of the authorization authority to be used to build the credential. A NULL input value selects a default.

mechanism_id **- input**
: Authentication mechanism that is used to generate the identity passed through *mechanism_info*. A NULL input value selects a default authentication mechanism.

mechanism_info **- input**
: Buffer containing initiator access control information, which consists of identity information obtained from an authentication service. The authentication service used to produce this information should be identified using the *mechanism_id* argument. A NULL input value denotes the default identity for the selected authentication mechanism from the environment.

new_creds **- output**
: Handle to a new, empty credentials chain that will hold the returned credentials.

Remarks

This function builds an authorization credentials chain, referenced by the returned handle *new_creds*, for the identity corresponding to the initiator access control information *mechanism_info* produced by an authentication mechanism *mechanism_id*.

Specifying a NULL value for *authority* causes the default authority to be used. The default authority is Policy Director, which is the only authority supported by this release of the Policy Director Authorization API.

Specifying NULL values for *mechanism_id* and *mechanism_info* causes the default authentication mechanism and the default identity to be the authentication mechanism used in the Policy Director secure domain.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
: Successful completion.

AZN_S_API_UNINITIALIZED
: This function has been called before azn_initialize().

AZN_S_INVALID_AUTHORITY
The authorization authority identification (id) is invalid.

AZN_S_INVALID_MECHANISM
The security mechanism identification (id) is not supported by the selected authorization authority.

AZN_S_INVALID_MECHANISM_INFO
The security mechanism information is invalid.

AZN_S_INVALID_NEW_CREDS_HDL
The credentials handle supplied for the new credentials chain is invalid.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_initialize()

Initializes the authorization service.

Syntax

```
azn_status_t
azn_initialize(
   azn_attrlist_h_t init_data,
   azn_attrlist_h_t *init_info
);
```

Parameters

init_data **- input**

Handle to an attribute list containing implementation-specific initialization data.

init_info **- output**

Pointer to a handle to an attribute list through which implementation-specific information is returned from initialization.

Remarks

This function must be called before calling most other Authorization API functions. The exceptions to this rule are the attribute list functions (azn_attrlist_*) and the error handling functions (azn_error_*).

The attribute list referenced by *init_info* contains the Authorization API version number, which is returned as the value for the attribute AZN_C_VERSION.

When *init_info* is no longer required, use azn_attrlist_delete() to release its storage.

Return Values

If successful, the function will return AZN_S_COMPLETE. If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE

Successful completion.

AZN_S_API_ALREADY_INITIALIZED

azn_initialize() has been called twice without an intervening call to azn_shutdown().

AZN_S_INVALID_INIT_DATA_HDL

The attribute list handle for the initialization information is invalid.

AZN_S_INVALID_INIT_INFO_HDL

The attribute list handle for the output initialization information is invalid.

AZN_S_FAILURE

An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_pac_get_creds()

Returns a handle to new credentials chain that is derived from a privilege attribute certificate (PAC) by a specified PAC service.

Syntax

```
azn_status_t
azn_pac_get_creds(
   azn_buffer_t pac,
   azn_string_t pac_svc_id,
   azn_creds_h_t *new_creds
);
```

Parameters

pac **- input**
Buffer structure that holds the supplied PAC.

pac_svc_id **- input**
Identification (id) of the PAC service that produces the credentials chain.

new_creds **- output**
Pointer to a handle to the returned credentials chain.

Remarks

This function uses the identified PAC service (*pac_svc_id*) to build a new credentials chain using the information in the supplied PAC (*pac*). Some PAC services will cryptographically verify the protection or signature on the received PAC, and will return an error if the PAC cannot be verified.

This function decodes PACs that are built by azn_creds_get_pac().

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_API_UNINITIALIZED
This function has been called before azn_initialize().

AZN_S_INVALID_PAC
The PAC is invalid or could not be verified by the PAC service.

AZN_S_INVALID_PAC_SVC
The id of the PAC service is invalid.

AZN_S_INVALID_NEW_CREDS_HDL
The credentials handle supplied for *new_creds* is invalid.

AZN_S_UNIMPLEMENTED_FUNCTION
This function is not supported by the implementation.

AZN_S_FAILURE

An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_release_buffer()

Frees storage associated with a buffer.

Syntax

```
azn_status_t
azn_release_buffer(
   azn_buffer_t *buffer
);
```

Parameters

buffer **- input**
Pointer to the buffer whose memory is to be released.

buffer **- output**
Pointer to the buffer whose memory is released. The pointer is set to an invalid value.

Remarks

This function releases the specified *azn_buffer_t* structure. The input buffer pointer is set to an invalid value to ensure that it cannot be used in future function calls.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_INVALID_BUFFER_REF
The pointer to the buffer is invalid.

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

azn_release_string()

Frees storage that is associated with a string.

Syntax

```
azn_status_t
azn_release_string(
    azn_string_t *string
);
```

Parameters

string **- input**

Pointer to the string to be released.

string **- output**

Pointer to the string whose memory is released. The pointer is set to an invalid value.

Remarks

This function releases the specified *azn_string_t* structure. The input string pointer is set to an invalid value to ensure that it cannot be used in future function calls.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE

Successful completion.

AZN_S_INVALID_STRING_REF

The pointer to the string is invalid.

AZN_S_FAILURE

An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

azn_release_strings()

Frees storage that is associated with an array of strings.

Syntax

```
azn_status_t
azn_release_strings(
    azn_string_t *strings[]
);
```

Parameters

strings **- input**
: Pointer to the array of azn_string_t structures to be released.

string **- output**
: Pointer to the array of strings whose memory is released. The pointer is set to an invalid value.

Remarks

This function releases a NULL-terminated array of *azn_string_t* structures. The input string pointer is set to an invalid value to ensure that it cannot be used in future function calls.

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
: Successful completion.

AZN_S_INVALID_STRING_REF
: Pointer to the array of strings is invalid.

AZN_S_FAILURE
: An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

azn_shutdown()

Cleans up internal authorization service state in preparation for shutdown.

Syntax

```
azn_status_t
azn_shutdown();
```

Remarks

Use **azn_shutdown()** to clean up the Authorization API's memory and other internal implementation state before the application exits. This function shuts down the implementation state created by azn_initialize().

The only authorization API functions that can be used after calling azn_shutdown(), prior to calling azn_initialize() again, are the attribute list functions (azn_attrlist_*), the error handling functions (azn_error_*), and the memory release functions (azn_*_delete and azn_release_*).

Return Values

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().

AZN_S_COMPLETE
Successful completion.

AZN_S_API_UNINITIALIZED
This function has been called before azn_initialize().

AZN_S_FAILURE
An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_unauth_t

Contains information for use in building an unauthenticated authorization credential for a user within the Policy Director secure domain.

Syntax

```
typedef struct {
   unsigned int ipaddr;
   azn_string_t qop;
   azn_string_t user_info;
   azn_string_t browser_info;
} azn_unauth_t;
```

Values

ipaddr
IP address of requesting user.

qop
Quality of protection that is required for requests that are made by this user.

user_info
Additional user information that might be required for auditing.

browser_info
Browser (if any) that is employed by the user.

Remarks

This data structure is used to pass information about an unauthenticated user into the azn_id_get_creds() interface. The content of each element of this structure is determined by the application, based on application requirements.

azn_util_client_authenticate()

Performs a login from a user name and password.

Syntax

```
azn_status_t
azn_util_client_authenticate(
   const azn_string_t principal_name,
   const azn_string_t password
);
```

Parameters

principal_name **- input**

Name of the principal (user) to be logged in.

password **- input**

Text password for the user.

Remarks

Performs a login from a user name and password pair. Starts a background thread to refresh the login context as necessary.

The Authorization API must be initialized before this function is called. Use azn_initialize() to initialize the Authorization API.

Return Values

Returns AZN_S_COMPLETE on success, or an error code on failure.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_util_password_authenticate()

Performs a login for a user name and password pair, and returns authentication information if the login was successful.

Syntax

```
azn_status_t
azn_util_password_authenticate(
   const azn_string_t principal_name,
   const azn_string_t password,
   azn_string_t *mechanism_id,
   azn_buffer_t *authinfo
);
```

Parameters

principal_name **- input**
Name of the user (principal) used to log in. If LDAP authentication is used, this will be a DN string.

password **- input**
Password for the user.

mechanism_id **- output**
Pointer to a string identifying the authentication mechanism with which the user is authenticated.

authinfo **- output**
Pointer to a buffer that is loaded with the authentication information that is returned by a successful login attempt.

Remarks

This function performs a login for a user name and password pair, and returns authentication information when the login is successful.

The authentication mechanism used depends upon the underlying authentication mechanism that was configured when the Authorization API was installed. Policy Director supports DCE and LDAP authentication. For LDAP Authorization API authentication, the azn_initialize() function must have completed successfully.

This function does not establish a security context for the application.

The *mechanism_id* and *authinfo* returned can be appended with data specific to the principal and passed into the azn_id_get_creds() function. The *mechanism_id* string is allocated by the utility function and must be freed using azn_release_string() when no longer needed. The *authinfo* buffer must be freed using azn_release_buffer().

Return Values

Returns AZN_S_COMPLETE on success, or an error code on failure.

The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See dceaclmsg.h for a complete list of minor error codes that describe access control problems.

azn_util_server_authenticate()

Performs a login from a keytab file, and starts a background thread to refresh the login context as necessary.

Syntax

```
azn_status_t
azn_util_server_authenticate(
   const azn_string_t principal_name,
   const azn_string_t keytab_path
);
```

Parameters

principal_name **- input**
Name of the user (principal) to log in.

keytab_path **- input**
Path to the keytab file containing the principal's key.

Remarks

This function performs a login from a keytab file, and starts a background thread to refresh the login context as necessary.

In order to use this utility function, applications that operate in a Policy Director secure domain that uses an LDAP user registry must use DCE commands to create a keytab file.

The Authorization API must be initialized before this function is called. Use azn_initialize() to initialize the Authorization API.

Return Values

Returns AZN_S_COMPLETE on success, error code on failure.

Appendix. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks

Index

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.