Tivoli SecureWay Policy Director
Authorization ADK Developer Reference

Copyright Notice
©Copyright IBM Corporation 2001 All rights reserved. May only be used pursuant to a Tivoli Systems Software License Agreement, an IBM Software License Agreement, or Addendum for Tivoli Products to IBM Customer or License Agreement. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without prior written permission of IBM Corporation. IBM Corporation grants you limited permission to make hardcopy or other reproductions of any machine-readable documentation for your own use, provided that each such reproduction shall carry the IBM Corporation copyright notice. No other rights under copyright are granted without prior written permission of IBM Corporation. The document is not intended for production and is furnished "as is" without warranty of any kind. All warranties on this document are hereby disclaimed, including the warranties of merchantability and fitness for a particular purpose.
U.S. Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corporation.

Trademarks
IBM, the IBM logo, Tivoli, the Tivoli logo, AIX, Policy Director, and SecureWay are trademarks or registered trademarks of International Business Machines Corporation or Tivoli Systems Inc. in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
Other company, product, and service names may be trademarks or service marks of others.

Notices
References in this publication to Tivoli Systems or IBM products, programs, or services do not imply that they will be available in all countries in which Tivoli Systems or IBM operates. Any reference to these products, programs, or services is not intended to imply that only Tivoli Systems or IBM products, programs, or services can be used. Subject to valid intellectual property or other legally protectable right of Tivoli Systems or IBM, any functionally equivalent product, program, or service can be used instead of the referenced product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by Tivoli Systems or IBM, are the responsibility of the user. Tivoli Systems or 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 the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, New York 10504-1785, U.S.A.
1.6.1 Using svrsslcfg.................................................................1-20
1.6.2 Using bassslcfg ..............................................................1-21
1.7 Initializing the Authorization Service .........................................1-22
1.7.1 Specifying an Authorization API Configuration File .................1-23
1.7.2 Specifying the Maximum Number of Handle Groups ...............1-24
1.7.3 Specifying the Type of Cache Mode ......................................1-25
1.7.4 Configuring Local Cache File Names .......................................1-27
1.7.5 Configuring Local Cache Refresh ............................................1-28
1.7.6 Configuring Local Cache Notification Listener ..........................1-29
1.7.7 Configuring Notification Listener Ports ....................................1-30
1.7.8 Configuring SSL from the API Client to Policy Director ..........1-31
1.7.9 Configuring the Authorization API for LDAP Access ................1-38
1.7.10 Configuring LDAP Access over SSL .....................................1-40
1.7.11 Configuring Advanced LDAP Parameters .............................1-41
1.7.12 Specifying LDAP User Registry Replica Access ......................1-42
1.7.13 Enabling the Return of Permission Information ......................1-43
1.7.14 Starting the Authorization Service .......................................1-45
1.8 Verifying the Identity of a User ................................................1-46
1.9 Obtaining User Authorization Credentials ..................................1-47
1.9.1 Specifying the Authorization Authority ...................................1-48
1.9.2 Specifying Authentication User Registry Type .........................1-48
1.9.3 Specifying User Authentication Identity ...................................1-48
1.9.4 Specifying Additional User Information .................................1-49
1.9.5 Placing User Information into an API Buffer ..........................1-50
1.9.6 Obtaining Authorization Credentials For the User ....................1-50
1.10 Obtaining an Authorization Decision .......................................1-52
1.10.1 Mapping the User Operation to a Policy Director Permission ......1-52
1.10.2 Mapping the Requested Resource to a Protected Object ..........1-53
1.10.3 Assigning the User Credentials to a Credentials Handle ..........1-53
1.10.4 Building an Attribute List for Additional Application Information 1-54
1.10.5 Obtaining an Authorization Decision .....................................1-55
1.11 Cleaning Up and Shutting Down ............................................1-56
1.11.1 Releasing Allocated Memory ..............................................1-56
1.11.2 Shutting Down the Authorization API ...................................1-56
1.12 Handling Credentials ................................................................................... 1-57
  1.12.1 Converting Credentials to a Transportable Format ......................... 1-57
  1.12.2 Converting Credentials to the Native Format ................................. 1-58
  1.12.3 Creating a Chain of Credentials ....................................................... 1-58
  1.12.4 Determining the Number of Credentials in a Credentials Chain... 1-59
  1.12.5 Obtaining a Credential from a Chain of Credentials...................... 1-59
  1.12.6 Modifying the Contents of a Credential ......................................... 1-60
  1.12.7 Obtaining an Attribute List from a Credential ............................... 1-61
  1.12.8 Comparing Two Credentials .......................................................... 1-63
  1.12.9 Copying A Credential..................................................................... 1-63

1.13 Deploying Applications with the Authorization API .................................. 1-64
  1.13.1 Software Requirements ................................................................. 1-64
  1.13.2 Example Program authzn_demo .................................................... 1-64

Chapter 2 — Authorization C API Reference

azn_attrlist_add_entry() ................................................................................. 2-3
azn_attrlist_add_entry_buffer() .................................................................... 2-5
azn_attrlist_create() ...................................................................................... 2-7
azn_attrlist_copy() ......................................................................................... 2-8
azn_attrlist_delete() ...................................................................................... 2-9
azn_attrlist_delete_entry() .......................................................................... 2-10
azn_attrlist_get_entry_buffer_value() ......................................................... 2-11
azn_attrlist_get_entry_string_value() ......................................................... 2-13
azn_attrlist_get_names() ............................................................................. 2-15
azn_attrlist_name_get_num() ...................................................................... 2-16
azn_authldap_t .............................................................................................. 2-18
azn_creds_combine() .................................................................................... 2-19
azn_creds_compare() .................................................................................... 2-21
azn_creds_copy() ........................................................................................ 2-23
azn_creds_create() ...................................................................................... 2-24
azn_creds_delete() ...................................................................................... 2-26
azn_creds_for_subject() ............................................................................. 2-28
azn_creds_get_attrlist_for_subject() ......................................................... 2-30
azn_creds_get_pac() .................................................................................... 2-32
azn_creds_modify() .................................................................................... 2-34
azn_creds_num_of_subjects() ..................................................................... 2-36
azn_decision_access_allowed() .................................................................. 2-38
azn_decision_access_allowed_ext() ............................................................ 2-40
azn_entitlement_get_entitlements() ............................................................. 2-42
azn_entitlement_get_entitlements() ............................................................. 2-45
Chapter 3 — Authorization Service Plug-ins

3.1 Supported Types of Service Plug-ins
   3.1.1 Entitlements Services
   3.1.2 Credentials Modification Service
   3.1.3 Privilege Attribute Certificate Service

3.2 Service Plug-in Architecture
   3.2.1 The Authorization Service Dispatcher
   3.2.2 Authorization Service Plug-ins
   3.2.3 Calling Applications

3.3 Initialization and Shutdown of Service Plug-ins
   3.3.1 Initialization
   3.3.2 Shutdown

3.4 Handling of Error Codes
   3.4.1 Major Error Codes
   3.4.2 Minor Error Codes

3.5 Deploying an Authorization Service Plug-in
   3.5.1 Specifying a Service ID
   3.5.2 Specifying the Location of the Plug-in Library
   3.5.3 Specifying Plug-in Parameters
   3.5.4 Service Definition Syntax
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.5</td>
<td>Using a Configuration File Entry</td>
<td>3-16</td>
</tr>
<tr>
<td>3.5.6</td>
<td>Configuring a Service Programmatically</td>
<td>3-17</td>
</tr>
<tr>
<td>3.6</td>
<td>Implementing an Entitlements Service</td>
<td>3-19</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Understanding Entitlements</td>
<td>3-20</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Initializing and Shutting Down Entitlements Service Plug-Ins</td>
<td>3-22</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Obtaining Entitlements for a Specified User</td>
<td>3-22</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Authorizing a Caller to a Specific Entitlements Service Plug-In</td>
<td>3-23</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Service Plug-ins Use of the Authorization API</td>
<td>3-23</td>
</tr>
<tr>
<td>3.6.6</td>
<td>Entitlements Service Error Codes</td>
<td>3-24</td>
</tr>
<tr>
<td>3.6.7</td>
<td>Deploying an Entitlements Service Plug-in</td>
<td>3-25</td>
</tr>
<tr>
<td>3.7</td>
<td>Example Entitlements Service Source Code</td>
<td>3-26</td>
</tr>
<tr>
<td>3.8</td>
<td>Policy Director Entitlements Services</td>
<td>3-29</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Policy Director Protected Objects Entitlements Service</td>
<td>3-29</td>
</tr>
</tbody>
</table>

**Chapter 4 — Authorization Service Plug-Ins: C API**

- azn_service_initialize() ................................................................. 4-2
- azn_service_shutdown() ................................................................. 4-5

**Chapter 5 — External Authorization Service**

- 5.1 Introducing the External Authorization Service ................. 5-2
- 5.2 Using the Remote Procedure Call Interface ......................... 5-3
  - 5.2.1 Interface Definition Language (IDL): extern_auth.idl .... 5-4
- 5.3 Building a Custom External Authorization Server ............... 5-5
  - 5.3.1 Source Files ............................................................... 5-5
  - 5.3.2 Supported Platforms ..................................................... 5-5
  - 5.3.3 Pre-requirements .......................................................... 5-5
  - 5.3.4 Build Process .............................................................. 5-5
- 5.4 Configuring a Custom External Authorization Service .......... 5-6
- 5.5 Managing External Authorization Services ......................... 5-8
  - 5.5.1 Registering an External Authorization Server ............... 5-9
  - 5.5.2 Delete an External Authorization Server ....................... 5-10
- 5.6 Reference: Interface Implementation .................................... 5-12
  - check_authorization() .......................................................... 5-13
Chapter 6 — Authorization Java Classes

6.1 Using Java™ 2 Security with Policy Director ............................................. 6-2
6.2 Java™ Authentication and Authorization Services (JAAS) Model ........... 6-3
   6.2.1 Authenticating Users and Obtaining Credentials ............................. 6-4
   6.2.2 Authorizing Access Requests ......................................................... 6-6
6.3 Software Requirements for Installation ..................................................... 6-8
   6.3.1 Policy Director Requirements ......................................................... 6-8
   6.3.2 JAAS and JRE Requirements .......................................................... 6-8
   6.3.3 Java Archive Requirements ............................................................. 6-9
6.4 Java Classes Overview ............................................................................ 6-10
   6.4.1 com.tivoli.mts.PDLoginModule ..................................................... 6-10
   6.4.2 com.tivoli.mts.PDPrincipal .......................................................... 6-11
   6.4.3 com.tivoli.mts.PDPermission ....................................................... 6-11
   6.4.4 com.tivoli.mts.SvrSslCfg ............................................................. 6-11
6.5 Configuring a Java Application into Policy Director ............................... 6-12
   6.5.1 Establishing an Identity with Proper Credentials ............................ 6-13
   6.5.2 Identifying the Policy Director Server Machines ............................ 6-13
   6.5.3 Using the SvrSslCfg Class ............................................................. 6-14
6.6 Configuring Java Authentication and Authorization Service .................... 6-15
   6.6.1 Creating a Login Configuration File .............................................. 6-15
   6.6.2 Specify the Login File Location .................................................... 6-16
6.7 Developing a Resource Manager ............................................................. 6-17
6.8 Making Authorization Decisions Outside of Java 2 ................................. 6-18

Chapter 7 — Authorization Java Classes Reference

Class PDLoginModule ..................................................................................... 7-2
Class PDPermission ....................................................................................... 7-7
Class PDPrincipal ......................................................................................... 7-13
Class SvrSslCfg ............................................................................................ 7-19

Appendix A — Chapter A — Authorization C API: DCE Support

A.1 Building Applications with the Authorization API ................................... A-2
   A.1.1 Software Requirements ................................................................. A-2
A.2 Initializing the Authorization Service .................................................... A-3
   A.2.1 Configuring Remote Cache Mode ............................................... A-5
A.2.2 Configuring Local Cache Refresh ................................................ A-6
A.2.3 Configuring Local Cache Notification Listener ......................... A-7
A.2.4 Configuring Notification Listener Ports ................................. A-8
A.2.5 Registering Local Cache Location with Management Server ...... A-9
A.2.6 Enabling the Return of Permission Information ...................... A-10
A.3 Authenticating an API Application ................................................. A-11
  A.3.1 Logging in Using a DCE Keytab File .................................... A-12
  A.3.2 Logging in Using a Password ............................................. A-13
A.4 Obtaining User Authorization Credentials .................................. A-14
  A.4.1 Specifying Authentication User Registry Type ..................... A-14
  A.4.2 Specifying User Authentication Identity ............................. A-15
  A.4.3 Specifying Additional User Information ............................ A-15
A.5 Handling Credentials ................................................................. A-16
  A.5.1 Attributes Returned by azn_creds_getattrlist_for_subject() .... A-16
A.6 Deploying Applications with the Authorization API .................... A-16

Appendix B — Chapter B — Authorization C API: DCE API Reference

azn_authdce_t .................................................................................. B-2
azn_util_client_authenticate() ...................................................... B-3
azn_util_server_authenticate() ...................................................... B-4
Preface

This document contains information about how to use the Tivoli SecureWay Policy Director development tools and Authorization API to enable an application to use Policy Director security. This document describes both a C implementation and a Java implementation of a Policy Director Authorization API.

Who Should Read This Guide

The target audience for this installation guide includes:

- Security administrators
- System installation and deployment administrators
- Network system administrators
- IT architects
- Application developers

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

Developers who want to use Policy Director authorization capabilities to enforce a security policy should read the sections of this book that describe the Authorization API.

Developers who want to extend the Policy Director Authorization Service by developing a separate external authorization server should read the External Authorization Service chapter in this book. Developers who want to develop an external authorization server should have basic working knowledge about writing and configuring DCE servers.
What This Guide Contains

This document contains the following chapters:

- **Chapter 1, "Authorization C API"**
  This chapter describes the Policy Director implementation of the Open Group standard Authorization C API.

- **Chapter 2, "Authorization C API Reference"**
  This chapter provides reference pages for each of the function calls and data types in the Policy Director Authorization C API.

- **Chapter 3, "Authorization Service Plug-ins"**
  Describes the Authorization Service Plug-in model, including Entitlements Services.

- **Chapter 4, "Authorization Service Plug-Ins: C API"**
  This chapter provides reference pages for each of the Service Plug-in function calls.

- **Chapter 5, "External Authorization Service"**
  This chapter describes how to write an external authorization server for use with the Policy Director Authorization Service.

- **Chapter 6, "Authorization Java Classes"**
  This chapter describes how to use the Policy Director Java classes to add Policy Director authorization to an application.

- **Chapter 7, "Authorization Java Classes Reference"**
  This chapter provides reference pages for each of the Java classes and methods.

- **Appendix A, "Authorization C API: DCE Support"**
  This chapter provides backwards-compatibility information for implementing Policy Director authorization using DCE.

- **Appendix B, "Authorization C API: DCE API Reference"**
  This chapter provides backwards-compatibility reference pages for each of the functions specific to Policy Director authorization using DCE.
What’s New in This Release

- New C API functions: azn_attrlist_delete_entry, azn_attrlist_copy, azn_creds_compare, azn_creds_copy, azn_util_errcode, azn_util_handle_is_valid.
- Java classes that implement the Policy Director Authorization model based on the Java 2 security model and the Java Authentication and Authorization Services model.
- The Authorization C API no longer contains functionality specific to DCE. The srvsslcfg utility is provided for configuring SSL communication between Authorization API Clients and the Policy Director servers.
- Two appendices are provided in this document to describe the backwards-compatibility for DCE applications built against previous versions of the Authorization API.

Typeface Conventions

This guide uses several typeface conventions for special terms and actions. These conventions have the following meaning:

**Bold** Command names and options, keywords, and other information that you must use literally appear in **bold**.

*Italics* Variables, command arguments, and values you must provide appear in *italics*. Titles of publications and special words or phrases that are emphasized also appear in *italics*.

Monospace Code examples, command lines, screen output, and system messages appear in monospace font.
Related Policy Director Documents

The following table summarizes the available Policy Director documentation:

<table>
<thead>
<tr>
<th>Tivoli SecureWay Policy Director Technical Documents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Installation Guides</strong></td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director Base for AIX Installation Guide</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director Base for HP-UX Installation Guide</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director Base for Solaris Installation Guide</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director Base for Windows Installation Guide</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director WebSEAL Installation Guide</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director NetSEAL Installation Guide</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director Management Console Installation Guide</td>
</tr>
<tr>
<td><strong>Administration Guides</strong></td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director Base Administration Guide</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director WebSEAL Administration Guide</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director NetSEAL Administration Guide</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director Management Console Administration Guide</td>
</tr>
<tr>
<td><strong>Developer References</strong></td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director Authorization ADK Developer Reference (<em>this document</em>)</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director WebSEAL Developer Reference</td>
</tr>
<tr>
<td><strong>Supplemental Documentation</strong> (updated regularly on the Tivoli support site)</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director Release Notes</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director Lotus Domino Registry Supplement</td>
</tr>
<tr>
<td>Tivoli SecureWay Policy Director Performance Tuning Guide</td>
</tr>
</tbody>
</table>
Accessing Online Documentation

The Tivoli Customer Support Web site (http://www.tivoli.com/support/) provides links to the following documentation information:

- Technical information, including release notes, installation and configuration guides, administration guides, and developer references.
- Frequently Asked Questions (FAQs)
- Software download information

You can find the Customer Support Handbook (a guide to support services) at: http://www.tivoli.com/support/getting/.

You can access the index of online Tivoli publications at http://www.tivoli.com/support/documents/. Click on Master Index to find product-specific support pages.

You can locate Policy Director technical documentation, by product version, at: http://www.tivoli.com/support/Prodman/html/AB.html#Security

The documentation for some products is available in PDF and HTML formats. Translated documents are also available for some products.

To access most of the documentation, you need an ID and a password. To obtain an ID for use on the support Web site, go to http://www.tivoli.com/support/getting/.

Resellers should refer to http://www.tivoli.com/support/smb/index.html for more information about obtaining Tivoli technical documentation and support.

Business Partners should refer to the Preface section entitled “Ordering Documentation” for more information about obtaining Tivoli technical documentation.

Ordering Documentation

Order Tivoli documentation online at http://www.tivoli.com/support/Prodman/html/pub_order.html or by calling one of the following telephone numbers:

- U.S. customers: (800) 879-2755
- Canadian customers: (800) 426-4968
Providing Feedback about Product Documentation

We are very interested in hearing about your experience with Tivoli products and documentation, and we welcome your suggestions for improvements. If you have comments or suggestions about our products and documentation, contact us in one of the following ways:

■ Send e-mail to pubs@tivoli.com.
■ Fill out our customer feedback survey at http://www.tivoli.com/support/survey/.

Contacting Customer Support

The Tivoli Customer Support Handbook at:

http://www.tivoli.com/support/handbook/

provides information about all aspects of Tivoli Customer Support, including the following:

■ Registration and eligibility
■ How to contact support, depending on the severity of your problem
■ Telephone numbers and e-mail addresses, depending on the country you are in
■ What information you should gather before contacting support
1

Authorization C API

This chapter contains the following topics:

- 1.1 Introducing the Authorization API
- 1.2 Locating the Authorization API Components
- 1.3 Building Applications with the Authorization API
- 1.4 Understanding the Authorization API Functions and Data Types
- 1.5 Summarizing Authorization API Tasks
- 1.6 Authenticating an API Application
- 1.7 Initializing the Authorization Service
- 1.8 Verifying the Identity of a User
- 1.9 Obtaining User Authorization Credentials
- 1.10 Obtaining an Authorization Decision
- 1.11 Cleaning Up and Shutting Down
- 1.12 Handling Credentials
- 1.13 Deploying Applications with the Authorization API
### 1.1 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).
1.1.1 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.

![Figure 1-1: The ISO 10181-3 Authorization Model](image-url)
Chapter 1: Authorization C API

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

![Diagram of the ISO Authorization Model implementation by Policy Director](image)

Figure 1-2: The Policy Director Implementation of the ISO Authorization Model

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.
1.1.2 The Policy Director Authorization Model

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 a 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 1 in the Policy Director Base Administration Guide.
   - For complete information on access control, see the Policy Director Base 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 1 in the Policy Director Base Administration Guide.

3. Develop your application logic to enforce the security policy.
1.2 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 Tivoli SecureWay Toolbox, refer to the Toolbox README file for installation instructions.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>include</td>
<td>C header files</td>
</tr>
<tr>
<td>lib</td>
<td>A library that implements the API functions.</td>
</tr>
<tr>
<td></td>
<td>On Solaris systems, the library is libpdauthzn.so</td>
</tr>
<tr>
<td></td>
<td>On AIX systems, the library is libpdauthzn.a</td>
</tr>
<tr>
<td></td>
<td>On HP-UX systems, the library is libpdauthzn.sl</td>
</tr>
<tr>
<td></td>
<td>On Microsoft Windows systems, the library to include at runtime is pdauthzn.dll</td>
</tr>
<tr>
<td></td>
<td>On Windows, the library to link is pdauthzn.lib</td>
</tr>
<tr>
<td>authzn_demo</td>
<td>An example program that demonstrates usage of the Authorization API.</td>
</tr>
<tr>
<td></td>
<td>Source files and a MAKEFILE are provided.</td>
</tr>
</tbody>
</table>

For installation instructions for the Authorization ADK, see the installation guide for the Policy Director Base distribution for your platform.

1.2.1 Locating Libraries for Building DCE Applications

The Authorization ADK libraries do not support the creation of DCE-based applications. To create DCE-based applications, you must use libraries that are included with the Policy Director Runtime. For more information on using the Authorization API to create DCE-based applications, see Appendix A.
1.2.2 **Header Files**

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

<table>
<thead>
<tr>
<th>File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ogauthzn.h</td>
<td>The Authorization API standard functions</td>
</tr>
<tr>
<td>aznutils.h</td>
<td>Utility functions (extensions to The Authorization API)</td>
</tr>
</tbody>
</table>

1.2.3 **Error Codes**

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

<table>
<thead>
<tr>
<th>File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ogauthzn.h</td>
<td>Major error codes for the standard Authorization API functions.</td>
</tr>
<tr>
<td>aznutils.h</td>
<td>Major error codes for the Authorization API utility functions.</td>
</tr>
<tr>
<td>dceaclmsg.h</td>
<td>Minor error codes for utility functions and the Policy Director Authorization Service.</td>
</tr>
</tbody>
</table>
1.3 Building Applications with the Authorization API

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

- Software Requirements (Section 1.3.1)
- Linking Required Libraries (Section 1.3.2)

Note: Instructions in this section refer to non-DCE environments. See Appendix A for instructions on developing DCE applications.

1.3.1 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 Base components installed:

- Policy Director Runtime
- Policy Director Management Server
- Policy Director Authorization Server
- Policy Director Application Development Kit

You should also install the Policy Director management utility:

- Policy Director Management Console

When 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 installation guide for the Policy Director Base distribution for your platform.

If you already have a 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 Runtime
- Policy Director Application Development Kit
1.3.2 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.

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. See the sample Makefile for build instructions specific to each supported operating system platform.
1.4 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, and constants are defined as part of the Authorization API:

- API Functions (Section 1.4.1)
- Character Strings (Section 1.4.2)
- Buffers (Section 1.4.3)
- Attribute Lists (Section 1.4.4)
- Credential Handles (Section 1.4.5)
- Status Codes and Error Handling (Section 1.4.6)
1.4.1 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_attrlist_add_entry()</td>
<td></td>
</tr>
<tr>
<td>azn_attrlist_add_entry_buffer()</td>
<td></td>
</tr>
<tr>
<td>azn_attrlist_copy()</td>
<td>Attribute Lists (Section 1.4.4)</td>
</tr>
<tr>
<td>azn_attrlist_create()</td>
<td></td>
</tr>
<tr>
<td>azn_attrlist_delete()</td>
<td></td>
</tr>
<tr>
<td>azn_attrlist_delete_entry()</td>
<td></td>
</tr>
<tr>
<td>azn_attrlist_get_entry_buffer_value()</td>
<td></td>
</tr>
<tr>
<td>azn_attrlist_get_entry_string_value()</td>
<td></td>
</tr>
<tr>
<td>azn_attrlist_get_names()</td>
<td></td>
</tr>
<tr>
<td>azn_attrlist_name_get_num()</td>
<td></td>
</tr>
<tr>
<td>azn_util_handle_is_valid()</td>
<td></td>
</tr>
</tbody>
</table>

### Credentials

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_creds_combine()</td>
<td>Creating a Chain of Credentials (Section 1.12.3)</td>
</tr>
<tr>
<td>azn_creds_compare()</td>
<td>Comparing Two Credentials (Section 1.12.8)</td>
</tr>
<tr>
<td>azn_creds_copy()</td>
<td>Copying A Credential (Section 1.12.9)</td>
</tr>
<tr>
<td>azn_creds_create()</td>
<td>Obtaining User Authorization Credentials (Section 1.9)</td>
</tr>
<tr>
<td>azn_creds_delete()</td>
<td>Releasing Allocated Memory (Section 1.11.1)</td>
</tr>
<tr>
<td>azn_creds_for_subject()</td>
<td>Obtaining a Credential from a Chain of Credentials (Section 1.12.5)</td>
</tr>
<tr>
<td>azn_creds_get_attrlist_for_subject()</td>
<td>Obtaining an Attribute List from a Credential (Section 1.12.7)</td>
</tr>
</tbody>
</table>
## Chapter 1: Authorization C API

### Authorization Decisions

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_creds_get_pac()</code></td>
<td>Converting Credentials to a Transportable Format (Section 1.12.1)</td>
</tr>
<tr>
<td><code>azn_creds_modify()</code></td>
<td>Modifying the Contents of a Credential (Section 1.12.6)</td>
</tr>
<tr>
<td><code>azn_creds_num_of_subjects()</code></td>
<td>Determining the Number of Credentials in a Credentials Chain (Section 1.12.4)</td>
</tr>
<tr>
<td><code>azn_id_get_creds()</code></td>
<td>Obtaining User Authorization Credentials (Section 1.9)</td>
</tr>
<tr>
<td><code>azn_pac_get_creds()</code></td>
<td>Converting Credentials to the Native Format (Section 1.12.2)</td>
</tr>
<tr>
<td><code>azn_util_handle_is_valid()</code></td>
<td>Credential Handles (Section 1.4.5)</td>
</tr>
</tbody>
</table>

### Authorization Decisions

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_decision_access_allowed()</code></td>
<td>Obtaining an Authorization Decision (Section 1.10)</td>
</tr>
<tr>
<td><code>azn_decision_access_allowed_ext()</code></td>
<td></td>
</tr>
</tbody>
</table>
### Initialization, Shutdown, and Error Handling

<table>
<thead>
<tr>
<th>Function or Data Type</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td></td>
</tr>
<tr>
<td>azn_error_major()</td>
<td>Status Codes and Error Handling (Section 1.4.6)</td>
</tr>
<tr>
<td>azn_error_minor()</td>
<td></td>
</tr>
<tr>
<td>azn_error_minor_get_string()</td>
<td></td>
</tr>
<tr>
<td>azn_initialize()</td>
<td>Initializing the Authorization Service (Section 1.7)</td>
</tr>
<tr>
<td>azn_release_buffer()</td>
<td>Releasing Allocated Memory (Section 1.11.1)</td>
</tr>
<tr>
<td>azn_release_string()</td>
<td></td>
</tr>
<tr>
<td>azn_release_strings()</td>
<td></td>
</tr>
<tr>
<td>azn_shutdown()</td>
<td>Shutting Down the Authorization API (Section 1.11.2)</td>
</tr>
<tr>
<td>azn_util_errcode()</td>
<td>Status Codes and Error Handling (Section 1.4.6)</td>
</tr>
</tbody>
</table>

### API Extensions

<table>
<thead>
<tr>
<th>Function or Data Type</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function or Data Type</strong></td>
<td>Task</td>
</tr>
<tr>
<td>azn_util_errcode()</td>
<td>Status Codes and Error Handling (Section 1.4.6)</td>
</tr>
<tr>
<td>azn_util_handle_is_valid()</td>
<td>Attribute Lists (Section 1.4.4) Credential Handles (Section 1.4.5)</td>
</tr>
<tr>
<td>azn_util_password_authenticate()</td>
<td>Verifying the Identity of a User (Section 1.8)</td>
</tr>
<tr>
<td>azn_authldap_t</td>
<td>Obtaining User Authorization Credentials (Section 1.9)</td>
</tr>
<tr>
<td>azn_unauth_t</td>
<td></td>
</tr>
</tbody>
</table>
1.4.2 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:

```c
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`.

1.4.3 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.

```c
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()` 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZN_C_EMPTY_BUFFER</td>
<td>NULL</td>
<td>Empty data value-buffer.</td>
</tr>
<tr>
<td>AZN_C_NO_BUFFER</td>
<td>NULL</td>
<td>No value-buffer is supplied or returned.</td>
</tr>
</tbody>
</table>
1.4.4 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.

The Authorization API defines some names. 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:

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create an attribute list</td>
<td>Use <code>azn_attrlist_create()</code> to complete the following tasks:</td>
</tr>
<tr>
<td></td>
<td>• Allocate a new, empty attribute list.</td>
</tr>
<tr>
<td></td>
<td>• Associate a handle with the attribute list.</td>
</tr>
<tr>
<td></td>
<td>• Return the handle.</td>
</tr>
<tr>
<td>Set an entry in an attribute</td>
<td>Use <code>azn_attrlist_add_entry()</code> to add a string name-value pair of type <code>azn_string_t</code>.</td>
</tr>
<tr>
<td>list.</td>
<td>Use <code>azn_attrlist_add_entry_buffer()</code> to add a buffer name-value pair of type <code>azn_buffer_t</code>.</td>
</tr>
<tr>
<td>Delete an entry from an</td>
<td>Use <code>azn_attrlist_delete_entry()</code> to delete all the values that are assigned to an attribute list.</td>
</tr>
<tr>
<td>attribute list.</td>
<td>Get attribute names from an attribute list.</td>
</tr>
<tr>
<td></td>
<td>Use <code>azn_attrlist_get_names()</code> to get all the names in an attribute list, contained in an array of strings of type <code>azn_string_t</code>.</td>
</tr>
<tr>
<td>Get the number of values for a specified attribute name</td>
<td>Use <code>azn_attrlist_name_get_num()</code> to get the number, as an integer, of the value attributes for a specified name in the attribute list.</td>
</tr>
</tbody>
</table>
### Task Description

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy an attribute list.</td>
<td>Use <code>azn_attrlist_copy()</code> to copy one attribute list to a new attribute list.</td>
</tr>
<tr>
<td>Get a value</td>
<td>Use <code>azn_attrlist_get_entry_string_value()</code> to get the value attribute of a string of type <code>azn_string_t</code> for a specified name in an attribute list.</td>
</tr>
<tr>
<td></td>
<td>Use <code>azn_attrlist_get_entry_buffer_value()</code> to get the value attribute of a buffer of type <code>azn_buffer_t</code> 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.</td>
</tr>
<tr>
<td>Delete an attribute list</td>
<td>Use <code>azn_attrlist_delete()</code> to delete the attribute list associated with a specified attribute list handle.</td>
</tr>
<tr>
<td>Determine if the attribute list handle is valid</td>
<td>Use <code>azn_util_handle_is_valid()</code> to determine if an attribute list handle is associated with valid data.</td>
</tr>
</tbody>
</table>
1.4.5 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 to the Policy Director security framework.

Use the function azn_creds_create() 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 the handle to release the memory allocated for the credential structure.

To determine if a credentials handle is valid, use the Authorization API utility function azn_util_handle_is_valid()
1.4.6 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 a 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()` 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()` 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()` to obtain string values for the minor error codes returned by `azn_error_minor()`.

Use `azn_util_errcode()` to build an `azn_status_t` error code from a major and minor status. Use this to return standardized error codes to Authorization API applications when developing Authorization Service Plug-ins.

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

<table>
<thead>
<tr>
<th>File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>oauthzn.h</td>
<td>Major error codes for the standard Authorization API functions.</td>
</tr>
<tr>
<td>aznutils.h</td>
<td>Major error codes for the Authorization API utility functions.</td>
</tr>
<tr>
<td>dceaclmsg.h</td>
<td>Minor error codes for utility functions and the Policy Director Authorization Service.</td>
</tr>
</tbody>
</table>
1.5 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.

1.5.1 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:

- Authenticating an API Application (Section 1.6)
- Initializing the Authorization Service (Section 1.7)
- Verifying the Identity of a User (Section 1.8)
- Obtaining User Authorization Credentials (Section 1.9)
- Obtaining an Authorization Decision (Section 1.10)
- Cleaning Up and Shutting Down (Section 1.11)

1.5.2 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 (Section 1.12)

1.5.3 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 (Section 1.13)
1.6 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.

Use the `svrsslcfg` utility to accomplish the above tasks. Run this utility before initializing the Authorization API.

1.6.1 Using svrsslcfg

Use the `svrsslcfg` utility to create a user identity for the application, and to configure the SSL communication between the application and the Policy Director Management Server.

The `svrsslcfg` utility performs the following tasks:

- Creates a user identity for the application based on the specified server. For example, demo_user.
- Creates an SSL key file for that user: For example, demo_user.key and demo_user.sth.
- Adds the user `ivacld-servers` group for a server type of `local`, or to the `remote-acl-users` group for a server type of `remote`.
The syntax for svrsslcfg is:

```
svrsslcfg <cfg_file> -config -d kdb_dir -n server_name
-s server_type -P admin_pwd [-S server_password] [-A admin_id] \ 
[-r port] [-t timeout] [-e pwd_life] [-C cert_file]
```

The following example is used by the Authorization Demo program that is distributed as part of the Policy Director Authorization ADK:

```
svrsslcfg aznAPI.conf -config -d
/opt/PolicyDirector/authzn_adk/example -n authzn_demo \ 
-S <svr-password> -s local -P <sec_master password>
```

The above example specifies the following configuration information:

- The configuration file is `aznAPI.conf`
- Configure (`-config`) the application to communicate with the Policy Director Management Server
- The directory for the keyfile is: `/opt/PolicyDirector/authzn_adk/example`
- The name of the application server is `authzn_demo`.
- The application server password is `<svr-password>`.
- The server type is `local`, rather than `remote`.
- The LDAP administrator password is `<sec_master password>`.

For more information, see the reference page for `svrsslcfg` in the Authorization API Reference.

### 1.6.2 Using bassslcfg

Use the `bassslcfg` utility to enable the Authorization API application to communicate over SSL with a Policy Director Authorization server that is located on another machine in the secure domain.

For more information, see the reference page for `bassslcfg` in the Authorization API reference.
Chapter 1: Authorization C API

1.7 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.

There are two ways to specify the initialization data:

- Specify input arguments to `azn_initialize`
- Specify entries in the Authorization API configuration file

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`.

Each input argument to `azn_initialize()` has a corresponding entry in the Authorization API configuration file. Input arguments take precedence over configuration file entries.

You can define entries in the configuration file, and then use input parameters to `azn_initialize()` to override them as needed when each Authorization API application initializes.

In order to use a configuration file, specify the configuration file location as an input parameter to `azn_initialize()`.

Complete the instructions in the following sections:

- Specifying an Authorization API Configuration File (Section 1.7.1)
- Specifying the Maximum Number of Handle Groups (Section 1.7.2)
- Specifying the Type of Cache Mode (Section 1.7.3)
- Configuring Local Cache File Names (Section 1.7.4)
- Configuring Local Cache Refresh (Section 1.7.5)
- Configuring Local Cache Notification Listener (Section 1.7.6)
- Configuring Notification Listener Ports (Section 1.7.7)
- Configuring SSL from the API Client to Policy Director (Section 1.7.8)
- Configuring the Authorization API for LDAP Access (Section 1.7.9)
- Configuring LDAP Access over SSL (Section 1.7.10)
- Configuring Advanced LDAP Parameters (Section 1.7.11)
- Specifying LDAP User Registry Replica Access (Section 1.7.12)
- Enabling the Return of Permission Information (Section 1.7.13)
- Starting the Authorization Service (Section 1.7.14)
1.7.1 Specifying an Authorization API Configuration File

You can specify a configuration file that contains initialization values. The configuration file is a text file consisting of stanzas. Each stanza contains a series of name = value pairs. Each of the pairs corresponds to an input parameter that can get passed to `azn_initialize()`.

If no configuration file is specified, `azn_initialize()` obtains initialization parameters only from the attribute list contained in the `init_data` input parameter.

There is no configuration file specified by default.

To specify the location of a configuration file:

1. Call `azn_attrlist_create()` 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()` to add the attribute `azn_init_cfg_file` and assign it a value:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_init_cfg_file</code></td>
<td><code>&lt;filename&gt;</code></td>
<td>A configuration file containing initialization values for the Policy Director Authorization API. There is no default value. A sample configuration file name could be: <code>aznapi.conf</code>.</td>
</tr>
</tbody>
</table>
### 1.7.2 Specifying the Maximum Number of Handle Groups

You can specify the maximum number of handle groups to allocate, to tune your application for memory usage or performance. Each handle group can contain up to 4096 handles. Handle groups are used to manage the allocation of handles for credentials and attribute lists.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry:</td>
<td>&lt;integer&gt;</td>
<td>Integer value of 0 (zero) means to use the default value. The default number of handle groups is 256. This enables the use of 1,000,000 handles. Minimum number of groups is 1. Maximum number of groups is 1,048,574. Use of the default value is recommended.</td>
</tr>
<tr>
<td>azn_init_max_handle_groups</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration File Entry:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>max-handle-groups</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Using An Attribute List

Use `azn_attrlist_add_entry()` to add the attribute `azn_init_max_handle_groups` and assign it a valid value.

#### Using A Configuration File

Go to the `[aznapi-configuration]` stanza in the configuration file and assign the `max-handle-groups` entry a valid value.
1.7.3 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.

- When using remote mode, the caller of the Authorization API must be a member of the `remote-acl-users` group.
  
  For more information on remote cache mode, see Chapter 1 of the *Policy Director Base Administration Guide*.

- When using local mode, the caller of the Authorization API must be a member of the `ivacld-servers` group.
  
  For more information on local cache mode, see Chapter 1 of the *Policy Director Base Administration Guide*.

The `svrsslcfg` utility creates a user identity (or server principal) for the caller and automatically adds it to the appropriate group. The `svrsslcfg` utility determines which group membership is required, based on whether you specified `local` or `remote` to the `-s` parameter. For more information, see the `svrsslcfg` reference page.

You can specify the cache mode either as an input parameter to `azn_initialize()` or as a configuration file entry. The following table displays the names of the cache mode entries and the valid values.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry: azn_init_mode</td>
<td>local</td>
<td>The Policy Director Authorization Service runs in the same server process as the application using the Authorization API.</td>
</tr>
<tr>
<td>Configuration File Entry: mode</td>
<td>remote</td>
<td>The Policy Director Authorization Service runs as a different server process from the application using the Authorization API.</td>
</tr>
</tbody>
</table>
Using An Attribute List

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

1. Call `azn_attrlist_create()` to create a new attribute list called `init_data`. This function returns a pointer to an attribute list handle.

   **Note:** Skip this step if the list was created in Section 1.7.1: "Specifying an Authorization API Configuration File".

2. Use `azn_attrlist_add_entry()` to add the attribute `azn_init_mode` and assign it a valid value.

Using A Configuration File

Go to the `[aznapi-configuration]` stanza in the configuration file and assign a valid value to the `mode` entry.
1.7.4 Configuring Local Cache File Names

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 either by adding attributes to the init_data attribute list or by setting entries in the Authorization API configuration file.

### Attribute List Entry

#### Attribute List Entry: azn_init_db_file

**Value:** filename

**Description:** Path name to the persistent authorization policy database replica.

#### Attribute List Entry: azn_init_audit_file

**Value:** filename

**Description:** Path and file name for the file that collects Authorization API audit events.

### Using An Attribute List

Use `azn_attrlist_add_entry()` to specify path names for files used by the authorization service.

### Using A Configuration File

Go to the [aznapi-configuration] stanza in the configuration file and specify path names for the db-file and auditlog files used by the authorization service.
1.7.5 Configuring Local Cache Refresh

<table>
<thead>
<tr>
<th>Attribute List Entry:</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_init_cache_refresh_interval</td>
<td>disable</td>
<td>Refreshing of the local authorization policy database disabled.</td>
</tr>
<tr>
<td></td>
<td>default</td>
<td>600 seconds.</td>
</tr>
<tr>
<td></td>
<td>number of seconds</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Using An Attribute List**

Use `azn_attrlist_add_entry()` to enable the Authorization API to poll the master authorization database.

**Using A Configuration File**

Go to the `[aznapi-configuration]` stanza in the configuration file and specify valid values for the `cache-refresh-interval` entry. Note that values can be either `disable`, `default`, or the number of seconds expressed as an integer.
1.7.6 Configuring Local Cache Notification Listener

<table>
<thead>
<tr>
<th>Attribute List Entry: azn_init_listen_flags</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry: azn_init_listen_flags</td>
<td>disable</td>
<td>Disable the notification listener.</td>
</tr>
<tr>
<td>Configuration File Entry: listen-flags</td>
<td>enable</td>
<td>Enable the notification listener.</td>
</tr>
</tbody>
</table>

Using An Attribute List

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

Using A Configuration File

Go to the `[aznapi-configuration]` stanza in the configuration file and specify valid values for the `listen-flags` entry. You can enter a combination of values. For example:

```
listen-flags = enable dynamic_port_selection use_tcp_port
```
1.7.7 Configuring Notification Listener Ports

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration File Entry: ssl-listening-port</td>
<td>port number</td>
<td>Use this value to specify the TCP port on which the application will listen for notifications from the master database that has changed. All communications on this port are SSL encrypted.</td>
</tr>
</tbody>
</table>

**Using A Configuration File**

Go to the [ssl] stanza in the configuration file and specify a port number for the ssl-listening-port. The value should be non-zero and not used by any other service on the computer.
1.7.8 Configuring SSL from the API Client to Policy Director

You can specify a number of attributes or configuration file entries that describe the SSL communications configuration between the Authorization API Client, running in remote mode, and the Policy Director Authorization Server and Policy Director Management Server.

The following configuration file entries in the following table are found in the [ssl] stanza.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>azn_init_ssl_keyfile</td>
<td>&lt;keyfile-path&gt;</td>
<td>This is the keyfile used to communicate with ivmgrd/ivacld. It will be created by the svrsslcfg utility.</td>
</tr>
<tr>
<td>Configuration File Entry:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssl-keyfile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute List Entry:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>azn_init_ssl_stashfile</td>
<td>&lt;keyfile-path&gt;</td>
<td>This the stash file for the keyfile. It is created by the svrsslcfg utility. It is used as an obfuscated password to the keyfile. This file should be appropriately secured.</td>
</tr>
<tr>
<td>Configuration File Entry:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssl-keyfile-stash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration File Entry:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssl-keyfile-label</td>
<td>Any string</td>
<td>The label of the certificate to use within the keyfile. In normal operation this is not used. However it is useful if the keyfiles are constructed outside of the svrsslcfg utility and contain multiple certificates.</td>
</tr>
</tbody>
</table>
### Attribute List Entry: `azn_init_ssl_timeout`

**Configuration File Entry:** `ssl-v3-timeout`

Any non-negative integer. Default value is 7200 seconds.

This is the amount of time before an SSL session will expire. The Policy Director Authorization API client automatically creates a new SSL session with new keys when a session expires. This value only applies to the listening aspect of the authorization API's (when the Policy Director Management Server is calling the application). When the application is calling the Policy Director Management Server or the Policy Director Authorization Server, the session timeout value is dictated by that server.

### Attribute List Entry: `azn_init_ssl_pwd_life`

**Configuration File Entry:** `ssl-pwd-life`

Any non-negative integer. Default value is 186 days.

This is the amount of time before the password or stash file to the keyfile will expire. The Policy Director Authorization API client automatically refreshes the password or stash file before this expiry time, provided it is in operation.
### Configuration File Entry: authn-type

**Value:** The type of authentication. Possible values are:
- certificate
- password
- none

The default value is none.

**Description:** The method that the Policy Director Management Server will use to authenticate the authorization API client. If the value is `certificate` the Policy Director Management Server will map the certificate provided by the authorization API client into an identity and authenticate against it. Note that even if `password` or `none` are specified, the client will still need a certificate to communicate with the server. Furthermore, there are currently no operations that can be performed by the API successfully with an authentication type of `none`.

### Configuration File Entry: authn-user

**Value:** Any string.

**Description:** The user name and password that are used if the authentication type is `password`. It may be unwise to store these in the configuration file, however they can be useful for testing communications.
Using An Attribute List

Use `azn_attrlist_add_entry()` to add attributes that contain configuration information for the SSL connections between the API client and the Policy Director servers.

Using A Configuration File

1. Go to the `[ssl]` stanza in the configuration file.
2. Add entries as specified in the preceding table.
## Communications Attributes for the Management Server

The configuration file entries described in the following table are found in the [manager] stanza.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry: azn_init_master_host</td>
<td>&lt;Management Server hostname&gt;</td>
<td>Specifies the hostname of the Policy Director Management Server. This entry and stanza can be in either the authorization API client's configuration file (aznAPI.conf) or the Policy Director Runtime Environment configuration file (pd.conf). The pd.conf value overrides that in aznAPI.conf.</td>
</tr>
<tr>
<td>Configuration File Entry: master-host</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute List Entry: azn_init_master_port</td>
<td>&lt;port number&gt;</td>
<td>Specifies the port of the Policy Director Management Server. This entry and stanza can be in either the authorization API client's configuration file (aznAPI.conf) or the Policy Director Runtime Environment configuration file (pd.conf). The pd.conf value overrides that in aznAPI.conf.</td>
</tr>
<tr>
<td>Configuration File Entry: master-port</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using A Configuration File

1. Go to the [manager] stanza in the configuration file.

2. Assign values for the Management Server, as specified in the preceding table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry:</td>
<td>&lt;distinguished name&gt;</td>
<td>Specifies the Distinguished Name (DN) of the Policy Director Management Server. The authorization API client can match this value against that provided by the Policy Director Management Server at runtime to prevent spoofing. If the value is empty, no checking will be performed. This entry and stanza can be in either the authorization API client's configuration file (aznAPI.conf) or the Policy Director Runtime Environment configuration file (pd.conf). The pd.conf value overrides that in aznAPI.conf.</td>
</tr>
<tr>
<td>azn_init_master_dn</td>
<td></td>
<td>Configuration File Entry: master-dn</td>
</tr>
</tbody>
</table>
Specifying an Authorization Server Replica

The following configuration file entry is found in the [manager] stanza:

<table>
<thead>
<tr>
<th>Configuration File Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Configuration File Entry:</strong> replica</td>
</tr>
<tr>
<td><strong>Value:</strong></td>
</tr>
<tr>
<td>&lt;Authorization Server values&gt;</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
</tr>
<tr>
<td>Each of these entries specifies a Policy Director Authorization Server. These are usually added to pd.conf via the bassslcfg command. They can be manually added, and there can be more than one entry. Note that the separator for the fields is a colon (&quot;:&quot; ) and not a comma (&quot;,&quot;) like the LDAP replicas use.</td>
</tr>
<tr>
<td>The replicas are of the format:</td>
</tr>
<tr>
<td>&lt;replica hostname&gt;:&lt;port&gt;:&lt;preference&gt;:&lt;replica cert dn&gt;</td>
</tr>
<tr>
<td>where:</td>
</tr>
<tr>
<td>• &lt;replica hostname&gt; is the network name of the server.</td>
</tr>
<tr>
<td>• &lt;port&gt; is the port on the server.</td>
</tr>
<tr>
<td>• &lt;preference&gt; is a ranking for attempting contact from 1 to 10 where 10 is the highest preference.</td>
</tr>
<tr>
<td>• &lt;DN&gt; is the DN for the ivacld server.</td>
</tr>
<tr>
<td>For example,</td>
</tr>
<tr>
<td>&quot;rweber.bball.com:7137:5:cn=ivacld/rweber,o=Policy Director,C=US&quot;</td>
</tr>
</tbody>
</table>

Using A Configuration File

1. Go to the [manager] stanza in the configuration file.
2. Assign values for the Authorization Server, using the format specified in the preceding table.
### 1.7.9 Configuring the Authorization API 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.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration File Entry:</td>
<td>yes</td>
<td>Enable LDAP user registry support. This is the default value. This entry is not used when building an attribute list. LDAP access is automatically enabled when the attribute <code>azn_init_ldap_port</code> is not null.</td>
</tr>
<tr>
<td><code>enable</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute List Entry:</td>
<td><code>host name</code></td>
<td>Host name of LDAP server.</td>
</tr>
<tr>
<td><code>azn_init_ldap_host</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration File Entry:</td>
<td><code>port number</code></td>
<td>Port number for communicating with the LDAP server.</td>
</tr>
<tr>
<td><code>host</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute List Entry:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>azn_init_ldap_port</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration File Entry:</td>
<td><code>LDAP DN</code></td>
<td>Distinguished Name of the LDAP user. Created by the <code>svrsslcfg</code> utility.</td>
</tr>
<tr>
<td><code>port</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute List Entry:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>azn_init_ldap_bind_dn</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration File Entry:</td>
<td><code>password</code></td>
<td>Password for the LDAP user. Created by the <code>svrsslcfg</code> utility.</td>
</tr>
<tr>
<td><code>bind-dn</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute List Entry:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>azn_init_ldap_bind_pwd</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration File Entry:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bind-pwd</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using An Attribute List

Use `azn_attrlist_add_entry()` to add the attributes to the `init_data` attribute list.

Using A Configuration File

1. Go to the `[ldap]` stanza in the configuration file.

2. Add entries for the LDAP server, as described in the preceding table.
1.7.10 Configuring LDAP Access over SSL

If the communication between the Policy Director Authorization server and the LDAP server is over Secure Sockets Layer (SSL), set the following values:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration File Entry: ssl-enabled</td>
<td>yes</td>
<td>Enables SSL communication with the LDAP server. This entry is not used when building attribute lists. If azn_init_ldap_ssl_keyfile is not null, then SSL is automatically configured.</td>
</tr>
<tr>
<td>Attribute List Entry: azn_init_ldap_ssl_keyfile</td>
<td>filename</td>
<td>Name of the SSL key file.</td>
</tr>
<tr>
<td>Configuration File Entry: ssl-keyfile-dn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute List Entry: azn_init_ldap_ssl_keyfile_dn</td>
<td>KeyLabel</td>
<td>Key label to identify the client certificate that is presented to the LDAP server.</td>
</tr>
<tr>
<td>Configuration File Entry: ssl-keyfile-dn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute List Entry: azn_init_ldap_ssl_keyfile_pwd</td>
<td>password</td>
<td>Password to access the SSL key file.</td>
</tr>
<tr>
<td>Configuration File Entry: ssl-keyfile-pwd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the Policy Director Authorization API client must use two key files: one for communicating with the LDAP server and one for the Policy Director servers.

Using An Attribute List

Use `azn_attrlist_add_entry()` to add the attributes to the `init_data` attribute list.

Using a Configuration File

1. Go to the `[ldap]` stanza in the configuration file.
2. Add the SSL values as described in the preceding table.
### 1.7.11 Configuring Advanced LDAP Parameters

<table>
<thead>
<tr>
<th>Entry</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry: azn_init_ldap_max_search_size</td>
<td>Optional. Limit for the maximum search buffer size returned from the LDAP server in entries. Note that this value can also be limited by the LDAP server itself.</td>
<td></td>
</tr>
<tr>
<td>Configuration File Entry: max-search-size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute List Entry: azn_init_ldap_cache</td>
<td>true</td>
<td>Optional. Enable LDAP client-side caching of user, group and LDAP policy data to improve performance for similar LDAP queries.</td>
</tr>
<tr>
<td>Configuration File Entry: cache-enabled</td>
<td>false</td>
<td>Optional. Disable LDAP client-side caching. This is the default value.</td>
</tr>
<tr>
<td>Attribute List Entry: azn_init_prefer_rw_svr</td>
<td>true</td>
<td>Optional. The client attempts to query the read/write LDAP server (see ldap-replica configuration option) before querying any read-only servers that are configured in the domain.</td>
</tr>
<tr>
<td>Configuration File Entry: prefer-readwrite-server</td>
<td>false</td>
<td>Optional. Do not query read/write LDAP server first</td>
</tr>
<tr>
<td>Attribute List Entry: azn_init_auth_using_compare</td>
<td>true</td>
<td>Optional. Choose whether ldap_compare() is used instead of ldap_bind() to authenticate LDAP users. This option changes the method used by the Policy Director Authorization API call and azn_util_password_authenticate().</td>
</tr>
<tr>
<td>Configuration File Entry: auth-using-compare</td>
<td>false</td>
<td>Optional. Use ldap_bind().</td>
</tr>
</tbody>
</table>

#### Using An Attribute List

Use `azn_attrlist_add_entry()` to add advanced LDAP parameters to the attribute list.

#### Using a Configuration File

Go to the [ldap] stanza in the configuration file and specify the advanced LDAP parameters as described in the table above.
1.7.12 Specifying LDAP User Registry Replica Access

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry:azzi_init_ldap_replica</td>
<td>&lt;ldap-server&gt;</td>
<td>The network name of the LDAP server.</td>
</tr>
<tr>
<td>Configuration File Entry:ldap-replica</td>
<td>&lt;port&gt;</td>
<td>The port on the LDAP server.</td>
</tr>
<tr>
<td></td>
<td>&lt;type&gt;</td>
<td>Either “readonly” or “readwrite”</td>
</tr>
<tr>
<td></td>
<td>&lt;pref&gt;</td>
<td>A preference or priority level to assign to accessing this replica. The minimum value is 1. The maximum value is 10. A higher number denotes a higher preference.</td>
</tr>
</tbody>
</table>

Using An Attribute List

Use `azon_attrlist_add_entry()` to add the attributes that define the LDAP user registry replicas in the domain. Add the attributes to the `init_data` attribute list.

Using a Configuration File

1. Go to the `[ldap]` stanza in the configuration file.
2. Define the LDAP user registry replicas in the domain by specifying valid values for `ldap-replica`.
   Assign multiple values to `ldap-replica` by entering a list consisting of entries that are separated by commas. For example:
   ```
   ldap-replica = barney,391,readwrite,2
   ldap-replica = fred,391,readonly,3
   ```
1.7.13 Enabling the Return of Permission Information

You can specify information to be returned by the `azn_decision_access_allowed_ext()` function call. This call returns a pointer to an attribute list (azn_attrlist_h_t) named `permission_info`.

This attribute list is empty by default but can optionally be used to return additional information about the access control decision. The Authorization API defines a number of permission information attributes that can be returned. In order for these attributes to be returned, they must be set when the Authorization API is initialized.

**Note:** You can also specify that user-defined attributes be returned in `permission_info`. See the `azn_decision_access_allowed_ext` reference page for more information.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry: <code>azn_init_set_perminfo_attr</code></td>
<td><code>azn_perinfo_all_attrs</code></td>
<td>Include all of the permission information attributes</td>
</tr>
<tr>
<td>Configuration File Entry: <code>permission-info-returned</code></td>
<td><code>azn_perinfo_wm</code></td>
<td>Warning mode. When warning mode is enabled, access is always granted. If access should not have been granted then the access is logged.</td>
</tr>
<tr>
<td></td>
<td><code>azn_perinfo_wm_permitted</code></td>
<td>Access permitted because of warning mode. The boolean indicator is used to tell the caller that access was granted because warning mode is enabled.</td>
</tr>
<tr>
<td></td>
<td><code>azn_perinfo_al</code></td>
<td>Auditing events that are performed for this authorization check.</td>
</tr>
</tbody>
</table>
Using An Attribute List Entry as Initialization Data

Use `azn_attrlist_add_entry()` to add the `azn_init_set_perminfo_attrs` attributes to the `init_data` attribute list.

Using A Configuration File

1. Go to the `[aznapi-configuration]` stanza in the configuration file.
2. Set values for the permission-info-returned entry.
1.7.14 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:

   ```c
   /* Start the service */
   status = azn_initialize(init_data, &init_info);
   if (status != AZN_R_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()`.

   When using this function on Windows NT, do not call it from dllMain(). For more information, see the reference page for `azn_initialize()`.
1.8 Verifying the Identity of a User

The application must verify 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 the LDAP User registry. The user is authenticated, and information about the user can be obtained. This information includes, for example, the LDAP Distinguished Name.

- **Unauthenticated**
  In this case, the user’s identity in the secure domain is not specifically registered in the LDAP 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 (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 the reference page for `azn_util_password_authenticate()`.

After the application has obtained identity information for the user, you can use the Authorization API to obtain authorization credentials for the user.
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 LDAP user registry 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 (Section 1.12.2).

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

1. Specifying the Authorization Authority (Section 1.9.1)
2. Specifying Authentication User Registry Type (Section 1.9.2)
3. Specifying User Authentication Identity (Section 1.9.3)
4. Specifying Additional User Information (Section 1.9.4)
5. Placing User Information into an API Buffer (Section 1.9.5)
6. Obtaining Authorization Credentials For the User (Section 1.9.6)
1.9.1 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.

1.9.2 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 Section 1.8: “Verifying the Identity of a User”.

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:

<table>
<thead>
<tr>
<th>User Registry Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAP User Registry</td>
<td>IV_LDAP</td>
</tr>
<tr>
<td>Unauthenticated</td>
<td>IV_UNAUTH</td>
</tr>
</tbody>
</table>

1.9.3 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 string in the data structure that corresponds to the user registry type.

<table>
<thead>
<tr>
<th>User Identity Type</th>
<th>Data Structure</th>
<th>String</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAP User Registry</td>
<td>azn_authldap_t</td>
<td>ldap dn</td>
<td>cn=root</td>
</tr>
<tr>
<td>Unauthenticated User</td>
<td>azn_unauth_t</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

If the user is unauthenticated, you do not have to load an identity into `azn_unauth_t`. 
### 1.9.4 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_authldap_t` and `azn_unauth_t`.

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

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth_method</td>
<td>Indicates that the user was authenticated through the LDAP user registry. This value can be any string that is useful to the application. Not available in <code>azn_unauth_t</code>.</td>
</tr>
<tr>
<td>authnmech_info</td>
<td>Additional authentication information. This value can be any string that is useful to the application. Not available in <code>azn_unauth_t</code>.</td>
</tr>
<tr>
<td>user_info</td>
<td>Additional user information for auditing purposes. This string can contain any information that is useful to the application.</td>
</tr>
<tr>
<td>browser_info</td>
<td>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.</td>
</tr>
<tr>
<td>ipaddr</td>
<td>The IP address of the user. This is optional information for use by the application.</td>
</tr>
</tbody>
</table>
1.9.5 Placing User Information into an API Buffer

Place the data structure you filled out in Specifying User Authentication Identity (Section 1.9.3) and Specifying Additional User Information (Section 1.9.4) on page 45 into an Authorization API buffer.

Complete the following steps:

1. Declare a buffer of type `azn_buffer_t`:

   ```c
   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()`.

1.9.6 Obtaining Authorization Credentials For the User

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>authority</code></td>
<td>The authorization authority, as described in Section 1.9.1: “Specifying the Authorization Authority”.</td>
</tr>
<tr>
<td><code>mechanism_id</code></td>
<td>The authentication mechanism, as described in Section 1.9.2: “Specifying Authentication User Registry Type”.</td>
</tr>
<tr>
<td><code>mechanism_info</code></td>
<td>User information, as described in the following sections:</td>
</tr>
<tr>
<td></td>
<td>• Specifying User Authentication Identity (Section 1.9.3)</td>
</tr>
<tr>
<td></td>
<td>• Specifying Additional User Information (Section 1.9.4)</td>
</tr>
<tr>
<td></td>
<td>• Placing User Information into an API Buffer (Section 1.9.5)</td>
</tr>
</tbody>
</table>

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.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.
"); continue;
}

For more information, see the reference page for `azn_id_get_creds()`. Refer also to the Authorization API demonstration program. See Example Program authzn_demo (Section 1.13.2).

The application is now ready to submit the authorization request. See Obtaining an Authorization Decision (Section 1.10).
1.10 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 a Policy Director Permission (Section 1.10.1)
- Mapping the Requested Resource to a Protected Object (Section 1.10.2)
- Assigning the User Credentials to a Credentials Handle (Section 1.10.3)
- Building an Attribute List for Additional Application Information (Section 1.10.4)
- Obtaining an Authorization Decision (Section 1.10.5)

1.10.1 Mapping the User Operation to a Policy Director Permission

The operation requested by the user must correspond to one of the operations for which a 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 ogauthzn.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()`.
Obtaining an Authorization Decision

1.10.2 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().

1.10.3 Assigning the User Credentials to a Credentials Handle

The authorization credentials for a user obtained in Section 1.9: “Obtaining User Authorization Credentials” 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 the function azn_pac_get_creds(). For more information, see Section 1.12.2: “Converting Credentials to the Native Format”.
1.10.4 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 the reference page for `azn_decision_access_allowed_ext()`.
1.10.5 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. 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td><code>permission</code></td>
<td>The result of the access request. Consists of one of the following constants:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>AZN_C_PERMITTED</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>AZN_C_NOT_PERMITTED</code></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_attrlist_h_t</code></td>
<td><code>*permission_info</code></td>
<td>Application-specific context information contained in attribute list.</td>
</tr>
</tbody>
</table>

For more information, see the reference pages for the following functions:

- `azn_decision_access_allowed()`
- `azn_decision_access_allowed_ext()`
1.11 Cleaning Up and Shutting Down

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

- Releasing Allocated Memory (Section 1.11.1)
- Shutting Down the Authorization API (Section 1.11.2)

1.11.1 Releasing Allocated Memory

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

- **azn_attrlist_delete()**
  Use this function to release memory that is allocated for attribute lists.

- **azn_attrlist_deleteEntry()**
  Use this function to delete an entry from an attribute list.

- **azn_creds_delete()**
  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()**
  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()**
  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()**
  Use this function to release memory allocated for an array of strings of type `azn_string_t`.

1.11.2 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()` to disconnect from and shut down the Authorization API.
1.12 Handling Credentials

In addition to the credentials handling tasks described earlier in this chapter, the Authorization API provides functions to accomplish the following optional tasks:

- Converting Credentials to a Transportable Format (Section 1.12.1)
- Converting Credentials to the Native Format (Section 1.12.2)
- Creating a Chain of Credentials (Section 1.12.3)
- Determining the Number of Credentials in a Credentials Chain (Section 1.12.4)
- Obtaining a Credential from a Chain of Credentials (Section 1.12.5)
- Modifying the Contents of a Credential (Section 1.12.6)
- Obtaining an Attribute List from a Credential (Section 1.12.7)

1.12.1 Converting Credentials to a Transportable Format

Use the function `azn_creds_get_pac()` 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.
1.12.2 Converting Credentials to the Native Format

Use the function `azn_pac_get_creds()` 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 of type `azn_creds_h_t`, for access by other Authorization API functions.

1.12.3 Creating a Chain of Credentials

Use the function `azn_creds_combine()` 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`. 
1.12.4 Determining the Number of Credentials in a Credentials Chain

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

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

1.12.5 Obtaining a Credential from a Chain of Credentials

Use the function `azn_creds_for_subject()` 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`.
1.12.6 Modifying the Contents of a Credential

Use the function `azn_creds_modify()` 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 (Section 1.4.4).

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`.
1.12.7 Obtaining an Attribute List from a Credential

Use the function `azn_creds_get_attrlist_for_subject()` 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.

The Authorization API defines a number of attributes that can be returned in the attribute list. Some of the attributes might not be present in an attribute list according to the type of authenticated user and the information that was used when the credential was built.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_cred_version</code></td>
<td>The credential version</td>
</tr>
<tr>
<td><code>azn_cred_mech_id</code></td>
<td>The mechanism ID for this credential</td>
</tr>
<tr>
<td><code>azn_cred_principal_uuids</code></td>
<td>The UUID for the entity</td>
</tr>
<tr>
<td><code>azn_cred_principal_name</code></td>
<td>The string name of the entity</td>
</tr>
<tr>
<td><code>azn_cred_group_uuids</code></td>
<td>The string group UUID memberships of this entity</td>
</tr>
<tr>
<td><code>azn_cred_group_names</code></td>
<td>The string group name memberships of this entity</td>
</tr>
<tr>
<td><code>azn_cred_ldap_dn</code></td>
<td>The LDAP DN used to build these authorization credentials</td>
</tr>
<tr>
<td><code>azn_cred_uraf_name</code></td>
<td>The URAF name used to build these authorization credentials</td>
</tr>
<tr>
<td><code>azn_cred_user_info</code></td>
<td>Any user information that was passed in the <code>mechinfo</code> structure</td>
</tr>
<tr>
<td><code>azn_cred_auth_method</code></td>
<td>Any authentication method information that was passed in the <code>mechinfo</code> structure.</td>
</tr>
<tr>
<td><code>azn_cred_authmech_info</code></td>
<td>Any authentication mechanism information that was passed in the <code>mechinfo</code> structure.</td>
</tr>
<tr>
<td><code>azn_cred_ip_address</code></td>
<td>The IP address information passed in the <code>mechinfo</code> structure</td>
</tr>
<tr>
<td><code>azn_cred_browser_info</code></td>
<td>The browser information passed in the <code>mechinfo</code> structure</td>
</tr>
</tbody>
</table>
Chapter 1: Authorization C API

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`. 

1.12.8 Comparing Two Credentials

Use the function `azn_creds_compare()` to compare the contents of two credentials.

1. Identify the credentials handles (`azn_creds_h_t`) for each credential to be compared.

2. Pass the credentials handles as input parameters to `azn_creds_compare()`.
   
   The function `azn_creds_compare()` returns a boolean value of type `azn_boolean_t`. The value is `true` if the credentials are identical or `false` if the credentials differ.

1.12.9 Copying A Credential

Use the function `azn_creds_copy()` to copy the contents of one credentials to another, new credentials.

1. Identify the credentials handles (`azn_creds_h_t`) for the credential to be copied.

2. Pass the credentials handle as the input parameter to `azn_creds_copy()`.
   
   The function `azn_creds_copy()` returns a credentials handle to the new credential.
   
   When the application is finished using the new credential, use `azn_creds_delete()` to release the memory that was allocated.

---

**Note:** To add a credential to an existing credential, use `azn_creds_combine()`.
1.13 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 (Section 1.13.1)
- Example Program authzn_demo (Section 1.13.2)

1.13.1 Software Requirements

Applications that have been developed with the Policy Director Authorization API must be run on systems that are configured into a 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 Runtime

1.13.2 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.
This section discusses the following Authorization API:

- `azn_attrlist_add_entry()`
- `azn_attrlist_add_entry_buffer()`
- `azn_attrlist_copy()`
- `azn_attrlist_create()`
- `azn_attrlist_delete()`
- `azn_attrlist_delete_entry()`
- `azn_attrlist_get_entry_buffer_value()`
- `azn_attrlist_get_entry_string_value()`
- `azn_attrlist_get_names()`
- `azn_attrlist_name_get_num()`
- `azn_authldap_t`
- `azn_creds_combine()`
- `azn_creds_compare()`
- `azn_creds_copy()`
- `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()`
This section includes reference pages for the following utilities:

- bassslcfg
- srvsslcfg
azn_attrlist_add_entry()

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

Syntax

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

Parameters

**Input**

- `attr_list` Handle to an attribute list.
- `attr_name` Name attribute of the entry to be added.
- `string_value` Value (string) attribute of the entry to be added.

Description

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

```c
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

- **Input**
  - `attr_list` Handle to an attribute list.
  - `attr_name` Name attribute of the entry to be added.
  - `buffer_value` Value (buffer) attribute of the entry to be added.

Description

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

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

Parameters

**Output**

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

Description

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_copy()

Copies a valid attribute list to a new attribute list.

Syntax

```c
azn_attrlist_h_t azn_attrlist_copy(azn_attrlist_h_t attr_list);
```

Parameters

- **Input**
  - `attr_list` The handle of the attribute list to be copied.

Description

This function copies an existing attribute list to another new attribute list and returns a handle to the new 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_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

Input

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

Output

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

Description

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_atrlist_delete_entry()

Deletes the attribute list associated with the attribute list handle.

**Syntax**

```c
azn_status_t azn_atrlist_delete(
    azn_atrlist_h_t attr_list,
    azn_string_t attr_name
);
```

**Parameters**

- **Input**
  - `attr_list` An existing attribute list handle.
  - `attr_name` The name of an attribute contained within the attribute list that is referenced by `attr_list`.

**Description**

This function deletes the specified attribute `attr_name` from the attribute list associated with the handle `attr_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_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

```c
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

**Input**

*attr_list* 
Handle to an attribute list.

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

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

**Output**

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

Description

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

```c
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

**Input**

- `attr_list` Handle to an existing attribute list.
- `attr_name` Name attribute of the entry from which the value attribute is to be returned.
- `value_index` Index within the entry of the value attribute to be returned.

**Output**

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

Description

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

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

Parameters

- **Input**
  - `attr_list` Handle to an existing attribute list

- **Output**
  - `attr_names` 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.

Description

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

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

Description

Input

*attr_list*  Handle to an existing attribute list.

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

Output

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

Description

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_authldap_t

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

**Syntax**

```c
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.

**Description**

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 returns a pointer to a handle to the resulting combined credentials chain.

Syntax

```c
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

Input

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

creds_to_add
Handle to the credentials chain to be appended to creds.

Output

combined_creds
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.

Description

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_compare()

Compares the contents of two credentials.

Syntax

```c
azn_boolean_t
azn_creds_compare(
    const azn_creds_h_t cred1,
    const azn_creds_h_t cred2
);
```

Parameters

Input

- `cred1` Handle to a credential.
- `cred2` Handle to a credential.

Description

This function compares the contents of two credentials. The function returns `true` when the credentials are identical and `false` when the credentials differ.

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 `cred1` or `cred2` 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_copy()

Copies the contents of one credential to a new credential.

Syntax

```
azn_creds_h_t
azn_creds_copy(
    const azn_creds_h_t creds
);
```

Parameters

- **Input**
  - `cred` Handle to a credential.

Description

This function compares the contents of a credential to a new credential.

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 `cred` 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_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

<table>
<thead>
<tr>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>creds</td>
</tr>
</tbody>
</table>

Pointer to the new credentials handle that is returned.

Description

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

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

Parameters

**Input**

creds

Pointer to the handle of the credentials chain to be deleted.

**Output**

cred

NULL pointer to a credentials handle that is invalid upon return.

Description

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

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

Parameters

**Input**

- **creds**
  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**
  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).

**Output**

- **new_creds**
  Pointer to the handle to the new credentials structure that is returned.

Description

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**

```c
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**

**Input**

- `creds` Handle to a credentials chain.

- `subject_index` 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).

**Output**

- `creds_attrlist` Pointer to the handle of an attribute list that holds the specified subject’s attribute information on return.

**Description**

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

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

Parameters

**Input**

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

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

**Output**

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

Description

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_SVC_SERVICE_NOT_FOUND**
  The service with the specified identification number was not found in the list of configured services.

- **AZN_S_SVC_AUTHORIZATION_FAILED**
  The caller is not authorized to make calls to the service. The minor error code contains additional information about the reason for the failure.

- **AZN_S_SVC_DISPATCHER_FAILURE**
  The service dispatcher failed. This can be caused by incorrect initialization of the Authorization API.

- **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

```c
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

Input

- `creds` Handle to the authorization credentials chain to be modified.
- `mod_svc_id` Identification (id) of the credential modification service.
- `mod_info` 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.

Output

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

Description

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_SVC_SERVICE_NOT_FOUND**
  The service with the specified identification number was not found in the list of configured services.

- **AZN_S_SVC_AUTHORIZATION_FAILED**
  The caller is not authorized to make calls to the service. The minor error code contains additional information about the reason for the failure.

- **AZN_S_SVC_DISPATCHER_FAILURE**
  The service dispatcher failed. This can be caused by incorrect initialization of the Authorization API.

- **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.
Chapter 2: Authorization C API Reference

azn_creds_num_of_subjects()

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

Syntax

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

Parameters

**Input**

- `creds` Handle to a credentials chain.

**Output**

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

Description

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**

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

**Parameters**

**Input**

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

*protected_resource*  
Name of the request's target.

*operation*  
Name of the requested operation.

**Output**

*permission*  
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()`.

**Description**

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.
Chapter 2: Authorization C API Reference

azn_decision_access_allowed_ext()

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

Syntax

```c
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

**Input**

- `creds` Handle to the initiator's credentials chain.

- `protected_resource` Name of the target of the request.

- `operation` Name of the requested operation.

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

- `permission_info` 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.
Output

**permission**

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**

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.”

Description

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_entitlement_get_entitlements()

Obtains entitlements from the specified entitlements service plug-in.

Syntax

```
azn_status_t
azn_entitlement_get_entitlements(  
    azn_creds_h_t creds,
    azn_string_t svc_id,
    azn_attrlist_h_t app_context,
    azn_attrlist_h_t *entitlements
);
```

Description

The Entitlements Service dispatcher calls this interface to request entitlements from the Entitlements Service Plug-in. This function takes as input a users credentials, a service ID specifying the Service Plug-in, and an attribute list specifying application-specific information.

This function returns an attribute list containing the entitlements.

Parameters

**Input**

- **creds**
  The credentials of the user for whom the calling application needs the list of applicable entitlements.

- **svc_id**
  The service identification (ID) of the Entitlements Service Plug-in.

- **app_context**
  The requested action and the protected resource for which the user’s entitlements are needed.

**Output**

- **entitlements**
  A pointer to an attributes list that contains the user’s entitlements. The entitlements are presented as a series of name-value pairs.
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_INVALID_CREDS_HDL
  The credentials handle supplied is invalid.

- AZN_S_INVALID_ENTITLEMENTS_SVC
  The Entitlements Service identifier is invalid.

- AZN_S_INVALID_APP_CONTEXT_HDL
  The attribute list handle for the application context is invalid.

- AZN_S_INVALID_ENTITLEMENTS_HDL
  The attribute list handle for the entitlements is invalid.

- AZN_S_SVC_SERVICE_NOT_FOUND
  The service with the specified identification number was not found in the list of configured services.

- AZN_S_SVC_AUTHORIZATION_FAILED
  The caller is not authorized to make calls to the service. The minor error code contains additional information about the reason for the failure.

- AZN_S_SVC_DISPATCHER_FAILURE
  The service dispatcher failed. This can be caused by incorrect initialization of the Authorization API.

- AZN_S_FAILURE
  An implementation specific error or failure has occurred. An implementation specific minor error code should be returned in the status code for the caller to extract with azn_error_minor().
azn_error_major()

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

Syntax

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

Parameters

Input

`status_code`

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

Description

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

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

Parameters

**Input**

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

Description

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**

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

**Parameters**

**Input**

- `minor_error` Minor error code previously returned by `azn_error_minor()`.

**Output**

- `minor_error_string` A string describing the condition that triggered the generation of the `minor_error` code.

**Description**

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_MINOR_CODE_MESSAGE_NOT_FOUND**
    
    The specified minor code has not corresponding error message in the catalog. This message also appears when a minor code of 0 (success) is passed.
  - **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

```c
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

Input

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

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

- **mechanism_info**
  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.

Output

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

Description

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.
Chapter 2: Authorization C API Reference

**azn_initialize()**

Initializes the authorization service.

**Syntax**

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

**Parameters**

**Input**

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

**Output**

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

**Description**

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.

When using this function on Windows NT, do not call it from dllMain(). The function dllMain() is run as one thread, which prevents azn_initialize() from spawning other threads. When programming in C++ on Windows NT, do not call azn_initialize() from a global or statically defined class constructor because these constructors are run from dllMain().
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_SVC_DEFINITION_ERROR**
  A service has been defined incorrectly. The error condition is caused either by invalid entries in the service definition attributes that are passed as input to azn_initialize(), or by invalid entries in the configuration file.

- **AZN_S_SVC_INIT_FAILED**
  A service failed to initialize correctly. The minor error code contains additional information about the reason for the failure.

- **AZN_S_SVC_AUTHORIZATION_FAILED**
  The caller is not authorized to make calls to the service. The minor error code contains additional information about the reason for the failure.

- **AZN_S_SVC_DLL_LOAD_FAILED**
  The DLL for a service failed to load correctly

- **AZN_S_SVC_INITIALIZE_NOT_FOUND**
  The azn_service_initialize() interface was not found in the DLL module for a service.

- **AZN_S_SVC_SHUTDOWN_NOT_FOUND**
  The azn_service_shutdown() interface was not found in the DLL module for a service.
AZN_S_SVC_ENT_FUNC_NOT_FOUND
The azn_entitlement_get_entitlements() interface was not found in the DLL module for an entitlements service.

AZN_S_SVC_PAC_FUNC_NOT_FOUND
The azn_pac_get_creds() or azn_creds_get_pac() interface was not found in the DLL module for a PAC service.

AZN_S_SVC_CRED_MOD_FUNC_NOT_FOUND
The azn_creds_modify() interface was not found in the DLL module for a credentials modification service.

AZN_S_SVC_SERVICE_IS_REGISTERED
The service ID cannot be registered because it is already listed as registered by the Service Dispatcher.

AZN_S_SVC_DISPATCHER_FAILURE
The service dispatcher failed. This can be caused by incorrect initialization of the Authorization API.

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

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

Parameters

**Input**

- `pac`  
  Buffer structure that holds the supplied PAC.

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

**Output**

- `new_creds`  
  Pointer to a handle to the returned credentials chain.

Description

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.
Chapter 2: Authorization C API Reference

- **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_SVC_SERVICE_NOT_FOUND**
  The service with the specified identification number was not found in the list of configured services.

- **AZN_S_SVC_AUTHORIZATION_FAILED**
  The caller is not authorized to make calls to the service. The minor error code contains additional information about the reason for the failure.

- **AZN_S_SVC_DISPATCHER_FAILURE**
  The service dispatcher failed. This can be caused by incorrect initialization of the Authorization API.

- **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

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

Parameters

Input

`buffer`

Pointer to the buffer whose memory is to be released.

Output

`buffer`

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

Description

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

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

Parameters

**Input**

*string*  
Pointer to the string to be released.

**Output**

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

Description

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

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

Parameters

**Input**

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

**Output**

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

Description

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**

```c
azn_status_t azn_shutdown();
```

**Description**

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_SVC_SHUTDOWN_FAILED**
  A service failed to shutdown correctly. The minor error code contains additional information about the reason for the failure.

- **AZN_S_SVC_AUTHORIZATION_FAILED**
  The caller is not authorized to make calls to the service. The minor error code contains additional information about the reason for the failure.

- **AZN_S_SVC_DISPATCHER_FAILURE**
  The service dispatcher failed. This can be caused by incorrect initialization of the Authorization API.
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.

Description

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_errcode()

Builds an azn_status_t error code from a major and minor status.

Syntax

```c
azn_status_t
azn_util_errcode(
    const unsigned int major,
    const unsigned int minor
);
```

Description

Builds an azn_status_t error code from a major and minor status.

Parameters

Input

- **major**
  
  The major component of the error status.

- **minor**
  
  The minor component of the error status.

Return Values

Returns complete azn_status_t error code.
**azn_util_handle_is_valid()**

Determine if the handle references a valid data structure.

**Syntax**

```c
azn_boolean_t azn_util_handle_is_valid(
    long handle
);
```

**Parameters**

**Input**

`handle` A handle to an attribute list.

**Description**

This function determines if the specified handle references a valid data structure.

**Return Values**

Returns AZN_S_COMPLETE on success, or an error code on failure.
azn_util_password_authenticate()

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

Syntax

```c
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

Input

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

`password`  Password for the user.

Output

`mechanism_id`  Pointer to a string identifying the authentication mechanism with which the user is authenticated.

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

Description

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.
basslcfg

Configure a machine to communicate over SSL with a remote Authorization Server.

Syntax

basslcfg -add_replica -p <port> -h <Authorization Server host> -r 10

Description

The add_replica option may be used by the Administrator to configure a machine to communicate with the Authorization Server (PDAcld) on another machine using over an SSL connection. This utility adds the location of the Authorization Server (ivacld) to the pd.conf file.

Replicas may be ranked in order of preference. When establishing a connection, Policy Director will try each replica in order of rank from lowest number to highest. If a connection cannot be established with a replica, then a connection will be made with the PDMgr server in the secure domain.

Check the ivacld.conf file to determine the port on which the Authorization Server is listening. Look for the ssl-listening-port entry.

The following command assumes that the Authorization Server is running on the machine acld_host on port 7136.

basslcfg -add_replica -p 7136 -h acld_host -r 10

Type basslcfg -? -add_replica to see the required and optional parameters for the -add_replica option.
Chapter 2: Authorization C API Reference

**srvsslcfg**

Configure an Authorization API application to communicate with the Policy
Director Management Server or Authorization Server.

**Syntax**

```
 srvsslcfg <cfg_file> -config -d kdb_dir -n server_name -s server_type
    -P admin_pwd [-S server_password]
    [-A admin_id] [-r port] [-t timeout]
    [-e pwd_life] [-C cert_file]
```

**Description**

The `srvsslcfg` program performs the following tasks:
- Creates a user with a name specified by `<server_name>`. For example, demo_user.
- Creates an SSL key file for that user: For example, demo_user.key and demo_user.sth.
- Adds the user to the ivacld-servers group for when the `<server_type>` is local, or to the remote-acl-users group when the `<server_type>` is remote.

Administrators use the `-config` option to configure an application that has been written with the Authorization API. This option enables the application to communicate with the either the Management Server or the Authorization Server. (Note that the `bassslcfg` command might also need to be operated manually.) The Policy Director Runtime Environment must be installed before using this utility.

This utility creates an SSL stanza or modifies an existing stanza for the SSL parameters in the configuration file. The utility also creates a key database in the specified directory. The database contains an SSL certificate signed by the Management Server.

You must provide a unique Policy Director name for the server, the directory in which the key ring database files are created, the application authorization type (local or remote) and a TCP port number if the application will listen for database update notifications (local mode only).

To view the required and optional parameters for configuration, type:

```
 srvsslcfg -? -config
```
Options

-\texttt{d\ kdb\_dir}
  The directory that is to contain the keyring database files for the server. This directory must already exist, it will not be created by svrsslcfg. This is a required parameter.

-\texttt{n\ server\_name}
  The name of the server. The name may be specified as either server\_name/hostname or server\_name, in which case the local hostname will be appended to form name/hostname. The names "ivacld", "secmgrd", "ivnet", "ivweb" are reserved for Policy Director servers. This is a required parameter.

-\texttt{s\ server\_type}
  The type of server being configured. The value must be either local or remote. This is a required parameter.

-\texttt{P\ admin\_pwd}
  The Policy Director Administrator password. When the User Registry type is LDAP, then this must be the SecurityMaster principal's password. This is a required parameter. If this parameter is not specified, the password will be read from stdin.

-\texttt{S\ server\_pwd}
  The server's password. This password must conform to the Policy Director password policy. This parameter is required if the User Registry type is LDAP. A user object will be created in the User Registry for this server using this password and the server name. If this parameter is not specified and it is needed, the server password will be read from stdin.

-\texttt{A\ admin\_id}
  The Policy Director Administrator name. When the User Registry type is LDAP, then this parameter is ignored, and the value "sec_master" will be used instead.

-\texttt{e\ pwd\_life}
  The keyring file's password expiration time in days. This parameter is optional. If not specified during initial configuration, a default of 183 days is used.
Chapter 2: Authorization C API Reference

\-r port_num
The listening port number for the server if the server will process notifications from the Management server. A value of 0 will disable the notification support. This parameter is optional with the -config action, and if not specified, a default of 0 is used. This parameter is required with the -chgport action. This parameter is only valid with local mode (-s local).

\-t ssl_timeout
Specify the SSL session timeout in seconds. The value must be in the range 1-86400. This parameter is optional. If it is not specified during initial configuration, a default value of 7200 is used. This parameter is only valid with local mode (-s local).

\-C cert_file
Specify the fully qualified name of the file containing the base-64 encoded SSL certificate used when the server authenticates directly with LDAP. This is optional.

Example

For example:

```
svrsslcfg aznAPI.conf -config -d
/opt/PolicyDirector/authzn_adk/example -n authzn_demo -S <svr-password> -s local -A cn=sec_master -P <admin-password>
```
Authorization Service Plug-ins

The Authorization API supports a Service Plug-in model. This model enables developers to write plug-in modules that extend the capabilities of the Policy Director Authorization Service. Developers of third party applications can use Authorization API functions that access the Service Plug-in interface to perform authorization operations that are specific to the Policy Director secure domain.

This chapter contains the following sections:

- 3.1 Supported Types of Service Plug-ins
- 3.2 Service Plug-in Architecture
- 3.3 Initialization and Shutdown of Service Plug-ins
- 3.4 Handling of Error Codes
- 3.5 Deploying an Authorization Service Plug-in
- 3.6 Implementing an Entitlements Service
- 3.7 Example Entitlements Service Source Code
- 3.8 Policy Director Entitlements Services
Chapter 3: Authorization Service Plug-ins

3.1 Supported Types of Service Plug-ins

The Authorization Service supports three types of Service Plug-ins:

- Entitlements Services
- Credentials Modification Service
- Privilege Attribute Certificate (PAC) Service

The Authorization API provides common initialization and shutdown interface calls for use by the Service Plug-ins. The Authorization API also provides additional interfaces that are specific to each of the Service Plug-ins.

3.1.1 Entitlements Services

An Entitlements Service Plug-in enables domain-specific Authorization API applications to retrieve the entitlements for a user from a domain-specific policy repository. The application can use this entitlements information as needed. For example:

- An application can allow or deny a user request for access to a protected action or protected resource, based on the user’s entitlements.
- A graphical user interface application can use entitlements information to construct a graphical view of the Policy Director secure domain contains only those protected objects that the user is authorized to view.

3.1.2 Credentials Modification Service

A Credentials Modification Service plug-in enables domain-specific Authorization API applications to perform modifications on a Policy Director credential. The Credentials Modification Service can then return this modified credential for use by the calling application. Applications can use this service to add additional information to a user’s credential. For example, this additional information could include the user’s credit card number and the user’s credit limit.
3.1.3 Privilege Attribute Certificate Service

A Privilege Attribute Certificate (PAC) Service plug-in gives domain-specific Authorization API applications the ability to move Policy Director credentials back and forth between the native Policy Director credentials format and an alternate format called Privilege Attribute Certificates (PAC).

Applications can convert user credentials to PACs for use within other authorization domains. Applications can then pass the PACs to a server in another authorization domain and perform an operation.
3.2 Service Plug-in Architecture

The Authorization Service Plug-in Architecture features the following objects:

- Authorization Service Plug-in Dispatcher
- Service Plug-in modules
- Calling applications

When an external application needs authorization information, it sends a request to the Service Dispatcher. The Service Dispatcher vectors the request to the appropriate Service Plug-in.

![Diagram of Authorization Service Plug-in Architecture](image)

Figure 3-1: Authorization Service Plug-in Architecture
3.2.1 The Authorization Service Dispatcher

The Authorization Service Dispatcher is a service of the Authorization API library. The Dispatcher is the management layer between Authorization API interfaces and the Service Plug-in modules. The Dispatcher manages the location, configuration and loading of available service plug-ins.

The Service Dispatcher performs the following tasks:

- Initializes each configured Service Plug-in.
- Directs Authorization API interface calls to the specified Service Plug-in.
- Receives returned information from the Service Plug-in and returns it to the calling application.
- Shuts down each configured Service Plug-in when the Authorization API is shut down.

3.2.2 Authorization Service Plug-ins

Authorization Service Plug-ins are shared libraries written by application developers. Developers create these libraries to implement a domain-specific task for the domain-specific application. The types of data passed between the Service Plug-in and the application are also domain-specific. This means that the only restrictions on the data types are the parameter definitions in the Authorization API service functions.

The data can be in a format that is unknown to the Policy Director Authorization Server. The data is passed unchanged through the Authorization Service Dispatcher to the Authorization Service Plug-ins.

Authorization Service Plug-ins are identified by a unique identification number (ID). The Service Dispatcher uses the unique ID number to load the Service Plug-in. The Service Dispatcher can optionally pass initialization parameters to the Service Plug-in. The Service Plug-in can optionally return service information, such as the plug-in version number, to the Service Dispatcher.
3.2.3 Calling Applications

Applications may choose to use information received from Service Plug-ins to make authorization decisions without requiring access to databases maintained by the Policy Director Authorization Server. The Service Plug-in framework operates independently from the Authorization API access_decision_* call environment.

The application invokes an Authorization API function that is specific to the type of service. For example, to obtain entitlements information, the calling application calls `azn_entitlements_get_entitlements()`.

The API call is processed by the Service Dispatcher and forwarded to the appropriate Service Plug-in. The Service performs basic error checking on the passed parameters, to ensure that all handles are valid.

The following table lists the handles that must be valid:

<table>
<thead>
<tr>
<th>Service</th>
<th>API function</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entitlements</td>
<td><code>azn_entitlements_get_entitlements()</code></td>
<td>entitlements</td>
</tr>
<tr>
<td>Credentials Modification</td>
<td><code>azn_creds_modify()</code></td>
<td>creds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>new_creds</td>
</tr>
<tr>
<td>Privilege Attribute Certificate</td>
<td><code>azn_pac_get_creds()</code></td>
<td>new_pac</td>
</tr>
<tr>
<td></td>
<td><code>azn_creds_get_pac()</code></td>
<td>pac</td>
</tr>
</tbody>
</table>

Applications are responsible for initializing and shutting down the Authorization API. To do this, applications call the Authorization API functions `azn_initialize()` and `azn_shutdown()`. These functions call the appropriate functions for initializing and shutting down the Service Plug-ins.

For some services, Policy Director provides a default service. For all other services, the application must specify a Service ID as a parameter to `azn_service_initialize()`.
3.3 Initialization and Shutdown of Service Plug-ins

All Service Plug-ins must provide service interfaces to perform the tasks described in the following sections:

- Section 3.3.1: “Initialization”
- Section 3.3.2: “Shutdown”

3.3.1 Initialization

Applications initialize the Authorization API by calling `azn_initialize()`. This function checks the Services stanzas in either the specified authorization configuration file or in data contained in the `init_data` attribute list parameter. When a service is defined, `azn_initialize()` loads the Service Plug-in and calls the `azn_service_initialize()` interface.

The `azn_service_initialize()` interface contains parameters named `argc` and `argv`. These parameters contain the values specified in the service definition within the configuration file. The service definition defines all entries after the character `&` to be initialization parameters. Unlike the C language `argv`, the `argv[0]` array entry is the first parameter, and not the calling sequence for the Service Plug-in.

The `azn_service_initialize()` interface must also take the `svc_init` input parameter, which specifies an attribute list. The Service Dispatcher may use the attribute list to pass additional information to the Service Plug-in.

Note: Currently, there are no attributes defined for the `svc_init` parameter.

The Service Plug-in can return information to the Service Dispatcher through the optional output parameter `svc_info`. This parameter specifies an attribute list that can be used to return information, such as the version number, that is specific to the Service Plug-in.

For more information, see the reference page for `azn_shutdown()`.
3.3.2 Shutdown

Applications shut down the Service Plug-ins as part of shutting down the Authorization API. The application calls `azn_shutdown()`. If a Service is initialized, `azn_shutdown()` calls `azn_service_shutdown()`.

The `azn_service_shutdown()` interface is called with the same `argc` and `argv` parameters that were passed to the `azn_service_initialize()` interface when the Service Plug-in was first initialized.

The `azn_service_shutdown()` also takes the optional input parameter `svc_init`, which specifies an attribute list. The Service Plug-in developer can use the attribute list to pass additional information to the Service Plug-in. Currently, there are no defined attributes for `svc_init`.

The Service Plug-in can return information to the Service Dispatcher through the optional output parameter `svc_info`.

For more information, see the reference page for `azn_service_shutdown()`.
3.4 Handling of Error Codes

The Authorization API Service Plug-in interface defines a number of major error codes. The interface also defines a minor error code mask and an error code creation utility which Service Plug-ins must use to construct error codes.

Use of this error mask ensures that the 32 bit major and minor error codes are correctly translated to a 32 bit error code for return to the calling application.

The calling application parses the returned error code into its major and minor error code components by the using the `azn_error_major()` and `azn_error_minor()` function calls.

Developers of Service Plug-ins should use the Authorization API utility function `azn_util_errcode()` to construct valid error codes to return to the calling application. The function `azn_util_errcode()` takes major and minor integer error code values and converts them in to an `azn_status_t` value. The minor error code must be defined as described in Section 3.4.2: “Minor Error Codes”.

3.4.1 Major Error Codes

Authorization Service Plug-ins should return all applicable major error codes that are defined by the Authorization API. Service Plug-ins should also return the major error codes defined for a particular type of Service Plug-in, such as the Entitlements Service.

**Note:** Some error codes do not apply to all Service Plug-in types.

The following additional major error codes are defined for the Authorization API Service modules in `oauthzn.h`. The Service Dispatcher returns only some of the error codes, while other error codes are returned only by the Service Plug-in.
The following table shows the error codes that the Service Dispatcher returns.

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
</table>
| AZN_S_SVC_DEFINITION_ERROR           | The Service Dispatcher returns this error when the service definition is constructed incorrectly in the Service stanza of the configuration file or in a value that is passed in the `init_data` attribute list of `azn_initialize()`.
| AZN_S_SVC_NOT_FOUND                  | The Service Dispatcher returns this error when it cannot locate the Authorization API Service Plug-in. The Service Dispatcher also returns this error when it cannot load the Service Plug-in.                     |
| AZN_S_SVC_INTERFACE_NOT_FOUND        | The Service Dispatcher returns this error when it encounters an error while either locating or loading a Service interface within a specific Service Plug-in. For example, the Service Dispatcher returns this error if the `azn_service_initialize()` interface could not be found in the loaded Service Plug-in. |
The following table shows the major error codes that Service Plug-ins return:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZN_S_SVC_INIT_FAILED</td>
<td>The Service Plug-in returns this error when it encounters an error during initialization.</td>
</tr>
<tr>
<td>AZN_S_SVC_SHUTDOWN_FAILED</td>
<td>The Service Plug-in returns this error when it encounters an error during shutdown.</td>
</tr>
<tr>
<td>AZN_S_SVC.AUTHORIZATION_FAILURE</td>
<td>The Service Plug-in returns this error when the calling application lacks authorization to invoke the Service.</td>
</tr>
</tbody>
</table>

In all of the above cases, the application developer may also insert an implementation specific minor error code into the status code. The minor error code provides the calling application with further information on the error that the Service Plug-in encountered. The calling application uses `azn_error_minor()` to extract the minor error code.
3.4.2 Minor Error Codes

The Service Plug-in modules define minor error codes. Each minor error code is specific to a particular Service Plug-in. Developers should define the minor error codes in the interface file for the Service Plug-in. The interface file also contains other details specific to the Service Plug-in, such as the names of the attributes recognized by the service. Applications that use the Service Plug-in will include and reference the service interface file.

The Authorization API encodes all minor error codes by using a table of known message catalogue prefixes. There is one prefix for all Service Plug-ins of the same type. For example, all Entitlements Service Plug-ins use the same prefix.

The file ogauthzn.h defines the following prefixes:

<table>
<thead>
<tr>
<th>Service</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entitlements</td>
<td>azn_ent_svc_err_prefix</td>
</tr>
<tr>
<td>Credentials Modification</td>
<td>azn_mod_svc_err_prefix</td>
</tr>
<tr>
<td>Privilege Attribute Certificate</td>
<td>azn_pac_svc_err_prefix</td>
</tr>
</tbody>
</table>

The Service Plug-in developer uses these prefixes to define the minor error codes for each Service Plug-in type. The azn_util_errcode() interface uses these minor codes to properly encode each error. The azn_util_errcode() function removes the prefix from the minor code and cross-references it against the known prefixes. When the prefix is identified, it is converted into a mask. The azn_util_errcode() function then inserts the mask into the major error portion of an azn_status_t structure.

For example, the prefix for Entitlement Services is defined as follows in the file ogauthzn.h:

extern unsigned int azn_ent_svc_err_prefix;

Entitlement Service developers should define their error messages in terms of azn_ent_svc_err_prefix.

For example, minor error codes for Authorization API Entitlement Service plug-ins are defined in the interface file as follows:

#define ENT_SVC_CORE_DUMP (azn_ent_svc_err_prefix | 0x1)
#define ENT_SVC_INVALID_ATTRIBUTE (azn_ent_svc_err_prefix | 0x2)
#define ENT_SVC_BAD_FILENAME (azn_ent_svc_err_prefix | 0x3)
The Service Plug-in developer has 16 bits of the minor error code to use for minor error codes specific to a service. The 16 bit error code can be personalized for the service, to avoid conflicts with other Service Plug-ins of the same type.

For example:

```c
#define MY_ENTSVC_ERR (0xE000)
#define ENT_SVC_CORE_DUMP
   (azn_ent_svc_err_prefix | (MY_ENTSVC_ERR | 0x1))
```
3.5 Deploying an Authorization Service Plug-in

You can deploy an Authorization Service Plug-in by registering it with the Authorization Service. To register a service plug-in, construct a service definition. The service definition can be entered into a configuration file, or passed in programmatically to `azn_service_initialize()`.

To construct a service definition, complete the following tasks:

1. **Specifying a Service ID (Section 3.5.1)**
2. **Specifying the Location of the Plug-in Library (Section 3.5.2)**
3. **Specifying Plug-in Parameters (Section 3.5.3)**

After constructing your service definition, choose one of the following methods for registering your service definition:

- Using a Configuration File Entry (Section 3.5.5)
- Configuring a Service Programmatically (Section 3.5.6)

### 3.5.1 Specifying a Service ID

The Service ID is a unique string that identifies the service to the calling application. The Service ID can be any string that is accepted as a valid key name by the stanza file parsing code. The Service ID is case insensitive.

### 3.5.2 Specifying the Location of the Plug-in Library

The Plug-in location is the fully qualified pathname for the shared library or DLL module that contains the plug-in for the given Service ID. The fully qualified path name is case sensitive.

If the library or DLL is located in a directory that is normally searched for system libraries or DLLs, you can just use the library name. Examples of this type of location are `/usr/lib` on UNIX systems and `%PATH%` on Windows NT systems.

You can specify a “short form” library name if you want the library name to be platform independent. This enables the library to be loaded on any supported Policy Director platform.

The Authorization API will prepend and append the library with known library prefixes and suffixes, and search for each possibility in turn.

For example, the Authorization API will search for a library with the short form name of `azn_ent_user` by looking for each of the following files:
3.5.3 Specifying Plug-in Parameters

The entry of additional parameters is optional. The `azn_service_initialize()` function can pass these parameters to the Service Plug-in as initialization information in the form of arguments.

To add optional initialization parameters to the service definition, insert the `&` character and then specify the arguments.

3.5.4 Service Definition Syntax

Each Service Plug-in entry uses the following syntax:

```
<service-id> = <plug-in location> [ & <plug-in parameters> ]
```

The following line is an example entry for an Entitlements Service Plug-in:

```
entsvc = /lib/libentsvc.so & -server barney
```

In the above example:

- The Service ID is `entsvc`.
- The Plug-in location is `/lib/libentsvc.so`
- The optional arguments are `-server barney`. In this example, the optional arguments tell the Service Plug-in the name of the server where the plug-in executes.

When the plug-in has been successfully loaded, the Service Dispatcher calls the service initialization interface `azn_service_initialize()`. The Service Dispatcher passes to `azn_service_initialize()` the parameters specified in the Service ID entry after the `&` character.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Library File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows NT</td>
<td>azn_ent_user.dll</td>
</tr>
<tr>
<td>AIX</td>
<td>libazn_ent_user.so, libazn_ent_user.a</td>
</tr>
<tr>
<td>Solaris</td>
<td>libazn_ent_user.so</td>
</tr>
<tr>
<td>HP/UX</td>
<td>libazn_ent_user.sl</td>
</tr>
</tbody>
</table>
3.5.5 Using a Configuration File Entry

The Policy Director Authorization Service recognizes and registers Service Plug-ins by reading entries in the configuration file `aznapi.conf`.

When the application initializes the Authorization API, the Authorization Server parses the configuration file. The Dispatcher resolves the location of each Service Plug-in, and loads each Service Plug-in. The `azn_service_initialize()` function returns an error if a Service Plug-in is not configured correctly or if the Service Plug-in module cannot be located.

Each type of Service has a separate section within the configuration file. The default configuration file contains the following entries:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Service Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[aznapi-entitlement-services]</td>
<td>Entitlements Service Plug-ins</td>
</tr>
<tr>
<td>[aznapi-pac-services]</td>
<td>Privilege Attribute Certificate Service Plug-ins</td>
</tr>
<tr>
<td>[aznapi-cred-modification-services]</td>
<td>Credentials Modification Service Plug-ins</td>
</tr>
</tbody>
</table>

For example, each entry for a Entitlement Service Plug-ins is declared in the configuration file under the following stanza entry:

```
[azn-api-entitlements-service]
```
3.5.6 Configuring a Service Programmatically

You can use the `init_data` input parameter to pass a service definition to `azn_initialize()`. Service definitions that are defined programmatically do not have to be defined in the configuration file `aznapi.conf`.

To use `init_data`, build an attribute list that contains the service definition entries. The file `ogauthzn.h` defines the following strings to use for the attribute list names for each type of service.

```c
/* Entitlements service definition attribute */
AZN_DECLSPEC azn_string_t azn_init_ent_svc;

/* PAC service definition attribute */
AZN_DECLSPEC azn_string_t azn_init_pac_svc;

/* Credential modification service definition attribute */
AZN_DECLSPEC azn_string_t azn_init_cred_mod_svc;
```

Use each of the above attributes to define a service of that particular type. The format of the string attribute values is described in Section 3.5.4: “Service Definition Syntax”. Each part of service definition, such as `<service id>`, represents a separate string value for the attribute.

Each attribute can have multiple service definitions. Use the function `azn_attrlist_add_entry()` to add the attribute and string values to the attribute list. For more information on attribute lists see Section 1.4.4: “Attribute Lists”.

Example of Configuring a Service Programmatically

```c
azn_status_t status;
azn_attrlist_h_t init_data, init_info;
azn_string_t service_entry;

azn_attrlist_create(&init_data);
azn_attrlist_create(&init_info);

/*
 * Load an Entitlements Service programmatically. The service
 * entry will load the DLL "mysvc" and associate it with the
 * service ID "MYSVC". The dispatcher will automatically search
 * the library search path for the platform specific DLL name
 * variations of this name. i.e. on NT: mysvc.dll, and on Unix:
 * libmysvc.so. The parameters -server and -port will be passed
 * to the azn_service_initialize() interface of the service in the
 * argv array.
 */

service_entry = "MYSVC=mysvc & -server barney -port 1234";
status = azn_attrlist_add_entry(init_data,
  azn_init_ent_svc,
  service_entry);

if (status != AZN_S_COMPLETE) {
    fprintf(stderr, "azn_attrlist_add_entry_failed: [%08X:%08X]\n",
            azn_error_major(status),
            azn_error_minor(status));
    exit -1;
}

/*
 * Call initialize
 */
status = azn_initialize(init_data, &init_info);

if (status != AZN_S_COMPLETE) {
    fprintf(stderr, "\nanz_initialize failed: [%08X:%08X]\n",
            azn_error_major(status),
            azn_error_minor(status));
    exit -1;
}
```

3.6 Implementing an Entitlements Service

The Authorization API Services framework provides a modular, plug-in style architecture that enables application developers to supplement Policy Director authorization with their own entitlements models. Developers can build a shared library object that implements an entitlements service. Developers then configure the Authorization API to use their module by exposing the appropriate entitlements interface.

The entitlements shared library object is a plug-in to the Authorization API. Application developers build their plug-in as an Authorization API client. The Entitlements Service Plug-in must know how to manipulate Authorization API structures such as user credentials and attribute lists. In most cases, the Service Plug-in functions as an Authorization API client.

![Figure 3-2: Entitlements Service Plug-in Architecture](image-url)
Chapter 3: Authorization Service Plug-ins

3.6.1 Understanding Entitlements

An entitlement is a data structure that contains externalized policy information. An entitlement is policy data that is formatted in a way that is understandable to a specific application. The application uses the Authorization API to instruct the Policy Director Authorization Service to obtain and return the entitlements.

For example, an application for processing stock market trading can define an entitlement called *trading limit*. This entitlement defines the maximum amount that a specific trader can trade in one transaction. The entitlement can be set independently for each trader known to the application. The application can request the trading limit for a specific trader. The Authorization Service returns the entitlement in a format, such as United States dollars, that the application can understand.

Entitlements to be brokered by the service are designed with the target application in mind. The policy to be modeled using entitlements is first identified and then the possible values for these entitlements are specified.

The Authorization API uses attributes within an attribute list to represent individual entitlements. Each attribute within the list consists of an attribute name and a multi-valued list of values for that attribute. Attributes can consist of values represented by data of type *azn_string_t* or by data of type *azn_buffer_t*.

**Note:** For more information on Authorization API attribute lists and data types, see Section 1.4: "Understanding the Authorization API Functions and Data Types".

Entitlements of type *azn_string_t*

An example of an entitlement is the "time of day" access restrictions for a particular resource. This refers to the time periods during a day that a particular user is permitted to access the resource to which the entitlement is attached. The attribute name defined by the entitlement service for this entitlement is simply a string:

```
extern azn_string_t time_of_day_restriction;
```

The value of the data returned is specific to the implementation and so may be returned as multiple strings identifying times of the day in which the resource may be accessed.
For example:
"mon-fri: 9am-12pm"
"mon-fri: 1pm-5pm"
"sat: 9am-12pm"

These string values for time_of_day_restriction define valid access times.
The entitlement service defines how these strings are formatted and interpreted
by the calling application.

**The Policy Director Protected Object Entitlements Service**

Another example of an entitlement is the data returned by the Policy Director
Protected Object Entitlement Service.

This service returns a list of objects within the protected object space for which
the given user credential has the specified access privileges.

An application could use this information to create a portal interface
specifically for each user that invokes the application. The services that each
user can employ are returned to the application as a list of strings. Each string
represents a protected object within the Policy Director protected object space
that the caller with the specified privileges can access.

**Entitlements of type azn_buffer_t**

Entitlement services may also define an entitlement as a blob of data using an
azn_buffer_t value for the attribute. This may be used by distributed object
based applications such as CORBA to retrieve encoded CORBA objects from
the Entitlements Service that can then be used to perform an operation for the
calling application. The contents of the azn_buffer_t data type are opaque to
the Authorization Service.

For more information on data type azn_buffer_t, see Section 1.4.3: “Buffers”.
3.6.2 Initializing and Shutting Down Entitlements Service Plug-Ins

Each Entitlements Service Plug-in is a standalone module that is dynamically loaded into the Authorization Service.

The Authorization API Service Plug-in model provides standard interfaces to the Service Dispatcher to initialize and shut down all types of Service Plug-ins. Developers of Entitlements Service Plug-ins should provide the standard functions. See the following sections:

- Section 3.3.1: “Initialization”
- Section 3.3.2: “Shutdown”

For more information, see the reference pages for `azn_shutdown()` and `azn_service_shutdown()`.

3.6.3 Obtaining Entitlements for a Specified User

The Authorization API provides the `azn_entitlements_get_entitlements()` interface for obtaining entitlements for a specified user. Both the calling application and the Service Plug-in must provide this interface. The `azn_entitlements_get_entitlements()` interface has the following input parameters:

<table>
<thead>
<tr>
<th>Input Parameter Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_creds_h_t</code></td>
<td><code>creds</code></td>
<td>The credentials of the user for whom the calling application wants the list of entitlements.</td>
</tr>
<tr>
<td><code>azn_string_t</code></td>
<td><code>svc_id</code></td>
<td>The service identification string of the Entitlements Service Plug-in.</td>
</tr>
<tr>
<td><code>azn_attrlist_h_t</code></td>
<td><code>app_context</code></td>
<td>The resource within the protected object space, and the action requested.</td>
</tr>
</tbody>
</table>

The Authorization API Service Dispatcher directs the request to the appropriate Service Plug-in, based on the `svc_id` parameter.

The Service Plug-in also takes the above parameters as input, and returns the requested entitlements in the attribute list `entitlements`. The Service Plug-in is not required to use the input parameters, but it is required to return valid data in the `entitlements` output parameter.
Entitlements Service Plug-ins share the address space of both the calling application and the Authorization API shared library. Service Plug-ins can assume that pointers passed in through the app_context parameter are valid within the address space of the Service Plug-in. Credential and attribute list handles are also valid to use within the Service Plug-in.

### 3.6.4 Authorizing a Caller to a Specific Entitlements Service Plug-In

Entitlements Service Plug-ins typically supply information in a directory service, such as LDAP, or in a data store, such a DB2 database, to the application caller. These directory services or data stores may require a proprietary authentication step before permitting access to stored information. The developer of each Entitlements Service Plug-in must implement these proprietary authentication steps. The Authorization API does not provide functions to implement proprietary authentication models.

The Authorization API does, however, provide the plug-in initialization interface `azn_service_initialize()`. This interface permits the passing of proprietary data in parameters to the service initialization function.

Alternately, the developer can use the initialization parameters to authenticate the Authorization API application before loading the Entitlements Service Plug-in. In this case, the Entitlements Service Plug-in can inherit privileges from the Authorization API application.

### 3.6.5 Service Plug-ins Use of the Authorization API

Entitlements Service Plug-ins should access the contents of passed parameters using the Authorization API. The Entitlements Service Plug-ins are not required to make use of any other features or interfaces of the Authorization API other than those that provide access to these data types.

The Entitlements Service Plug-in becomes part of the Authorization API application’s address space. The Service Plug-in can assume that if it is denied access to a particular Authorization API interface that the calling application was not allowed to perform the operation.

The Service Plug-in should not use either the `azn_util_server_authenticate()` or the `azn_util_client_authenticate()` calls to override these privileges as they will automatically propagate to the calling application as well.
3.6.6 Entitlements Service Error Codes

The Entitlements Service can generate error codes from the following functions:

- `azn_service_initialize()`
- `azn_service_shutdown()`
- `azn_entitlements_get_entitlements()`

For information on error codes for `azn_service_initialize()` and `azn_service_shutdown()`, see Section 3.4: “Handling of Error Codes”.

The `azn_entitlements_get_entitlements()` function returns the following error codes:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AZN_S_INVALID_CREDS_HDL</code></td>
<td>The credentials handle supplied is invalid.</td>
</tr>
<tr>
<td><code>AZN_S_INVALID_ENTITLEMENTS_SVC</code></td>
<td>The Entitlements Service identifier is invalid.</td>
</tr>
<tr>
<td><code>AZN_S_INVALID_APP_CONTEXT_HDL</code></td>
<td>The attribute list handle for the application context is invalid.</td>
</tr>
<tr>
<td><code>AZN_S_INVALID_ENTITLEMENTS_HDL</code></td>
<td>The attribute list handle for the entitlements is invalid.</td>
</tr>
<tr>
<td><code>AZN_S_FAILURE</code></td>
<td>An implementation specific error or failure has occurred. An implementation</td>
</tr>
<tr>
<td></td>
<td>specific minor error code should be returned in the status code for the</td>
</tr>
<tr>
<td></td>
<td>caller to extract with <code>azn_error_minor()</code>.</td>
</tr>
</tbody>
</table>

Each Entitlements Service Plug-in can optionally return minor error codes that express errors specific to the Service Plug-in implementation. For more information on implementing minor error codes, see Section 3.4.2: “Minor Error Codes”.
3.6.7 Deploying an Entitlements Service Plug-in

The Policy Director Authorization Service recognizes and registers Entitlements Service Plug-ins with the Service Dispatcher by reading entries in the configuration file `aznapi.conf`.

Entitlement service plug-ins are declared in the configuration file under the following stanza entry:

```
[entitlements-service]
```

The Policy Director Authorization Service also recognizes and registers Entitlements Service Plug-ins through arguments passed to the `init_data` parameter of the `azn_initialize()` function.

For complete configuration specifications, see Section 3.5: “Deploying an Authorization Service Plug-in”.

3.7 Example Entitlements Service Source Code

/*
 * FILENAME
 * mysvc.cpp
 * 
 * DESCRIPTION
 * Example entitlements service for the aznAPI.
 */

#include <stdio.h>
#include <ogauthzn.h>
#include <aznutils.h>

#ifdef _WIN32
#define AZN_DECLSPEC __declspec(dllexport)
#else
#define AZN_DECLSPEC
#endif

#define MY_SVC_VER "Example Entitlements Service v1.0"

#ifdef __cplusplus
extern "C" {
#endif

/**********************************************************************
 * Interface function definitions.
***********************************************************************/

/*
 * FUNCTION NAME
 * azn_service_initialize
 * 
 * DESCRIPTION
 * init the entitlements service
 * 
 * ARGUMENTS
 * [in] argc The count of arguments to the service.
 * [in] argv The array of argument strings.
 * [in] svc_init List of initialization attributes for the service.
 * [out] svc_info attr list ptr for attributes returned by the service.
 * 
 * RETURN VALUE
 * AZN_S_COMPLETE on success, error code on failure
 */

AZN_DECLSPEC
azn_status_t
AZN_CALLTYPE
azn_service_initialize:

int argc, /* in */
char **argv, /* in */
azn_attrlist_h_t svc_init, /* in */
azn_attrlist_h_t *svc_info /* out */
)
{  
  azn_status_t st;
  azn_boolean_t freeAttrlist = FALSE;

  /* svc_info must not be NULL */
  if (svc_info == NULL) {
    return (azn_util_errcode(AZN_S_FAILURE,
                           AZN_PAC_EPAC_INVALID_SVCINFO_HDL));
  }

  /* ensure argc is valid */
  if (argc < 0 || (argc == 0 && argv != NULL)) {
    return (azn_util_errcode(AZN_S_FAILURE,
                           AZN_PAC_EPAC_INVALID_ARG_COUNT));
  }
  if (argc > 0 && argv == NULL) {
    return (azn_util_errcode(AZN_S_FAILURE,
                           AZN_PAC_EPAC_INVALID_ARG_ARRAY));
  }

  /* Process arguments and initialize service. */

  /* return the service version to the dispatcher */
  if (*svc_info == AZN_C_INVALID_HANDLE) {
    azn_attrlist_create(svc_info);
    freeAttrlist = TRUE;
  }
  st = azn_attrlist_add_entry(*svc_info, azn_svc_version, MY_SVC_VER);
  if (st != AZN_S_COMPLETE) {
    if (freeAttrlist) {
      azn_attrlist_delete(svc_info);
    }
    return (st);
  }
  return (AZN_S_COMPLETE);
}

/*
* FUNCTION NAME
* azn_service_shutdown
*
* DESCRIPTION
* Shutdown the entitlements service.
* The initialization parameters are passed in
* again on shutdown but are ignored. No version
* info is returned.
*
* ARGUMENTS
* [in] argc The count of arguments to the service.
* [in] argv The array of argument strings.
* [in] svc_init List of initialization attributes for
* the service.
* [out] svc_info attr list ptr for attributes returned by the
* service.
*
* RETURN VALUE
* AZN_S_COMPLETE
*/
Chapter 3: Authorization Service Plug-ins

```c
AZN_DECLSPEC
azn_status_t
AZN_CALLTYPE
azn_service_shutdown(
    int argc, /* in */
    char **argv, /* in */
    azn_attrlist_h_t svc_init, /* in */
    azn_attrlist_h_t *svc_info /* out */
)
{
    return (AZN_S_COMPLETE);
}

/*
* FUNCTION NAME
*   azn_entitlement_get_entitlements
*
* DESCRIPTION
*   Returns entitlements information associated with
*   the credential and context info passed in.
*
* ARGUMENTS
*   [in] creds The credentials of the caller
*   [in] svc_id The id of the entitlements service
*   [in] app_context attribute list containing information
*       regarding the type of object we are
*       operating on
*   [out] entitlements attribute list containing the
*       entitlements associated with the specified
*       object
*
* RETURN VALUE
*   AZN_S_COMPLETE on success, error code on failure
*/
AZN_DECLSPEC
azn_status_t
AZN_CALLTYPE
azn_entitlement_get_entitlements(
    azn_creds_h_t creds, /* in */
    azn_string_t svc_id, /* in */
    azn_attrlist_h_t app_context, /* in */
    azn_attrlist_h_t *entitlements /* out */
)
{
    /* Authenticate the call or authorize the caller */
    /* Obtain entitlements data from back-end database */

    return (AZN_S_COMPLETE);
}
#endif
```

# ifndef __cplusplus
#endif
3.8 Policy Director Entitlements Services

Policy Director Version 3.7 provides the following default entitlements services:

- Policy Director Protected Objects Entitlements Service (Section 3.8.1)

3.8.1 Policy Director Protected Objects Entitlements Service

Policy Director Version 3.7 provides an entitlement service specific to the Policy Director environment. This entitlements service is provided in the Authorization API distribution package.

This Entitlements Service should be configured in secure domains with Policy Director servers that have been configured and are running.

The Policy Director Protected Objects Entitlements Service provides the following features:

- The Service returns a list of protected resources in the database for which the given credential has the specified access privilege. For example, the application can request the Service to return a list of HTML objects under /WebSEAL/srvA/staff for which a specified user has read permission. A graphical user interface application can use the returned entitlements (protected object names) to determine which buttons the specified user can see when the GUI application is loaded.

- The Entitlements Service accepts a single, multi-valued string attribute that specifies the root node(s) for searching the Policy Director protected object namespace. This enables the application to limit its search to a particular set of protected objects in the web space, rather than search the entire web space. Because the attribute can contain multiple values, the application can specify multiple root nodes for the search.

- Entitlements data is returned as a multi-valued attribute list of protected objects for each search tree root node passed in to the `azn_entitlements_get_entitlements()` call. The list contains only objects underneath the specified root node.
Authorization Service Plug-Ins: 
C API

This chapter contains the following reference pages:

- `azn_service_initialize()`
- `azn_service_shutdown()`
azn_service_initialize()

Initialize the specified Authorization Service Plug-in.

Syntax

```c
azn_status_t
azn_service_initialize(
    int argc,
    const char **argv,
    azn_attrlist_h_t svc_init,
    azn_attrlist_h_t *svc_info
);
```

Description

Use this initialization function to initialize a Service Plug-in. The Service Dispatcher calls this function prior to calling the function `azn_entitlements_get_entitlements()` to obtain entitlements information from the Service Plug-in.

The input parameters `argc` and `argv` are built from the parameters that are specified in the Service definition for the service instance.

A sample service definition for an Entitlements service named `entsvc` is:

```
entsvc = /lib/libentsvc.so & -server barney
```

For the service definition above `azn_service_initialize()` is called with an `argc` value of 2. The `argv` array contains the following values:

```
argv[0] = "-server"
argv[1] = "barney"
```

In this example, the `argv` values are used to specify the server system where the Entitlements Service Plug-in is to be loaded.

The service information returned by the Service Plug-in in `svc_info` can contain the version number of the service. This value is defined in `ogauthzn.h`:

```
extern azn_string_t azn_svc_version;
```

The Service plug-in can assume that the Service Dispatcher will release the attribute list returned in `svc_info` when it has finished with it.

Plug-ins are only required to return service information when the dispatcher specifically requests this information.

Input parameters should not be modified.
Parameters

Input

argc
The number of arguments contained in the argv array.

argv
The string arguments passed in the Service definition for this service instance.

svc_init
The svc_init parameter is an attribute list containing attributes that are specified by the Service Dispatcher to either configure or request initialization information from the Service Plug-in.

Output

svc_info
The svc_info parameter is a list of attributes returned by the Service Plug-in to request specific treatment by the Service Dispatcher or to inform the Service Dispatcher of the operational parameters of service.

Examples of the information listed in the attributes are the version of the service and the largest minor error code that maybe returned by the service.

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().

Each of the following error messages also returns an implementation-specific minor error code. Use azn_error_minor() to derive specific minor error codes from the returned status code.

- AZN_S_SVC_DEFINITION_ERROR
  Returned by the Service Dispatcher when an error has been found in the service definition in the Authorization API configuration file aznapi.conf.

- AZN_S_SVC_NOT_FOUND
  Returned by the Service Dispatcher when an error occurs either while locating or loading an Authorization API service plug-in.
Chapter 4: Authorization Service Plug-Ins: C API

- **AZN_S_SVC_INTERFACE_NOT_FOUND**
  Returned by the Service Dispatcher when an error occurs either locating or loading a service interface within a particular plug-in. For example, the dispatcher returns this error if the `azn_service_initialize()` interface was not found in the loaded plug-in.

- **AZN_S_SVC_INIT_FAILED**
  Returned by the entitlements service plug-in if an error occurs when the entitlements service is initializing.

- **AZN_S_SVC_AUTHORIZATION_FAILURE**
  Returned when the calling application does not have sufficient authority to invoke the services of this Service plug-in.

- **AZN_S_FAILURE**
  An implementation specific error or failure has occurred. An implementation specific minor error code should be returned in the status code for the caller to extract with `azn_error_minor()`.
azn_service_shutdown()

Shut down the specified Service Plug-In.

Syntax

```c
azn_status_t
azn_service_shutdown(
    int argc,
    const char **argv,
    azn_attrlist_h_t svc_init,
    azn_attrlist_h_t *svc_info
);
```

Description

The Authorization API Service Dispatcher calls this interface to shut down the specified Authorization Service Plug-in as part of shutting down the Authorization API.

The input parameters `argc` and `argv` are built from the parameters that were specified in the service definition for this service instance.

A sample configuration file entry for an entitlements service named `entsvc` is:

```plaintext
entsvc = /lib/libentsvc.so & -server barney
```

For the service definition above `azn_service_shutdown()` is called with an `argc` value of 2. The `argv` array contains the following values:

```plaintext
argv[0] = "-server"
argv[1] = "barney"
```

The Service Plug-in can assume that the Service Dispatcher will release the attribute list returned in `svc_info` when it has finished with it.

Information that may be requested of the service during shutdown is not currently defined but may be defined by the interface in future
Chapter 4: Authorization Service Plug-Ins: C API

Parameters

Input

argc
The number of arguments in the argv array.

argv
The string arguments contained in the service definition for this service instance.

svc_init
The svc_init parameter is an attribute list containing attributes that are specified by the dispatcher to either unconfigure or request information from the service plug-in after shutdown is complete.

Output

svc_info
The svc_info parameter is a list of attributes returned by the entitlements service to request specific treatment by the Service Dispatcher or to inform the Service Dispatcher of the results of service shutdown.

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_SVC_SHUTDOWN_FAILED
  Returned by the authorization service plug-in when an error occurs while it is shutting down.

- AZN_S_FAILURE
  An implementation specific error or failure has occurred. An implementation specific minor error code should be returned in the status code for the caller to extract with azn_error_minor().
5

External Authorization Service

This chapter contains:

■ 5.1 Introducing the External Authorization Service
■ 5.2 Using the Remote Procedure Call Interface
■ 5.3 Building a Custom External Authorization Server
■ 5.4 Configuring a Custom External Authorization Service
■ 5.5 Managing External Authorization Services
■ 5.6 Reference: Interface Implementation
5.1 Introducing the External Authorization Service

Note: This Policy Director Application Development Kit 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.

5.2 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.
Chapter 5: External Authorization Service

5.2.1 Interface Definition Language (IDL): extern_auth.idl

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

```
{uuid(4df5494-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);
}
```
5.3 Building a Custom External Authorization Server

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

5.3.1 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.

5.3.2 Supported Platforms

The external authorization service source files can be compiled on any platform. The custom built executable must reside on a machine within the Policy Director secure domain.

5.3.3 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.

5.3.4 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.
Chapter 5: External Authorization Service

5.4 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 (Section 5.2). Additionally, the server must maintain a DCE login context and be able to accept authenticated RPCs.

   **Note:** Refer to your DCE application development guide for details about writing a DCE server.

   Refer to Building a Custom External Authorization Server (Section 5.3)

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):

```bash
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):

```bash
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 `pdadmin server register` command. Use the information created in Steps 2 and 3 above as arguments to this command. For example (entered as one line):

```bash
pdadmin> 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.
5.5 Managing External Authorization Services

An External Authorization Service allows you to impose additional authorization controls and conditions to supplement the standard Policy Director authorization process. These additional controls and conditions are dictated by a separate authorization server program.

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

Two 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. Register the external authorization server with the Authorization Service.
   Refer to Section 5.5.1: “Registering an External Authorization Server”.

Once the server is registered, a new permission that represents this extended service will appear in the Management Console. You can now use this permission in any ACL entry.

When the permission is encountered during an authorization check, the external authorization server is referenced for additional authorization decisions.
5.5.1 Registering an External Authorization Server

Use the `ivadmin server register externauth` command to inform the Authorization Service of the existence and location of an external authorization server. The following syntax applies:

```
ivadmin> server register externauth <server-name> <ns-location> <server-group> <character> <description>
```

where:

- **server-name**: A name (or label) for this external authorization service. 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 external authorization server exports its RPC bindings.
- **server-group**: The name of the DCE group whose membership includes the external authorization server process.
- **character**: The permission character used in an ACL to dictate the use this external authorization service for supplemental authorization decisions.
- **description**: The descriptive label that appears to the right of the character in the Management Console display (ACLs tab).

This command produces a default ACL organization category called **External Authorization** that appears in the ACL Entry area of the Management Console. The permissions for all registered external authorization servers appear under this category.

For example:

```
ivadmin> server register externauth timechecker
./:/subsys/timechk t-checks k time-check
```

registers an external authorization server named **timechecker** with the Authorization Service.

The RPC entry in the CDS namespace where timechecker exports its RPC bindings is `./:/subsys/timechk`.

The DCE group, in which the server is a member, is **t-checks**.

The permission associated with this server is **k (time-check)**.
Chapter 5: External Authorization Service

The permission for this registered external authorization server would appearing the Management Console as follows:

```
ivadmin> server delete /ExternAuthzn/<server-name>
```

where:

- **server-name**: A name (or label) for this external authorization service. This is the name that appears in the display of the object space on the Management Console.

For example:

```
ivadmin> server delete /ExternAuthzn/timechecker
```
Example 1

A third-party bulletin board/news service has time restrictions placed on its operation. Users can only view the information provided by this service between the hours of 8AM and 5PM. An external authorization service is written to perform a time check on all requests made to the bulletin board/news service.

The external authorization service is set up through `ivadmin` almost exactly like the example presented above.

The following figure illustrates the possible scenarios for the authorization processing. The user of course must have the “r” permission to view the bulletin board/news information. The ACL on the news service also includes the “k” permission, which dictates to the Authorization Service that the `timechecker` authorization server be included in the final decision.

The final authorization decision is based on the summation of all authorization server decisions.

![Figure 5-1: Timechecker Example](image-url)
Example 2

This example is the same as Example 1 with the addition of a second external authorization service that audits activity on the bulletin board/news service.

Note that when viewing is not permitted by the timechecker authorization service, auditing of the activity still occurs because the presence of the “h” permission (which requires the involvement of the auditing authorization server during the ACL check).

![Diagram of authorization services](image)

**Figure 5-2: Timechecker-Audit Example**

5.6 Reference: Interface Implementation

- `check_authorization()`
check_authorization()

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

Syntax

```c
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

**Input**

handle

Server binding handle.

principal

Authenticated delegation chain. This data structure can be directly cast into an `azn_creds_h_t` data type for use with the Authorization API.

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.

**Output**

qop

Minimum acceptable quality of protection.

status

Return status. Returns `error_status_ok` if request is authorized.
Chapter 5: External Authorization Service

Description

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.
Authorization Java Classes

The Policy Director Authorization Java Classes provide an implementation of Java security code that is fully compliant with the Java 2 security model and the Java Authentication and Authorization Services (JAAS) extensions.

The Policy Director Authorization Java Classes are described in the following sections:

- 6.1 Using Java™ 2 Security with Policy Director
- 6.2 Java™ Authentication and Authorization Services (JAAS) Model
- 6.3 Software Requirements for Installation
- 6.4 Java Classes Overview
- 6.5 Configuring a Java Application into Policy Director
- 6.6 Configuring Java Authentication and Authorization Service
- 6.7 Developing a Resource Manager
- 6.8 Making Authorization Decisions Outside of Java 2
6.1 **Using Java™ 2 Security with Policy Director**

The Java™ 2 security architecture is policy-based, and allows for fine-grained access control. When code is loaded, it is assigned "permissions" based on the security policy currently in effect. Each permission specifies a permitted access to a particular resource, such as "read" access to a specified file, or "connect" access to a specified host and port. The policy specifies which permissions are available for code from various signers and locations. The policy can be initialized from an external configuration file.

Code can access a resource only if the permission that guards the resource gives the code explicit permission. These new concepts of permission and policy enable the Java™ 2 to offer fine-grained, highly configurable, flexible, and extensible access control. Such access control can now be specified for all Java™ code, including applications, beans, and servlets.

In Policy Director 3.7, the Authorization Server provides an SSL-based access mode for handling remote authorization calls. The Policy Director Java Authorization API uses this socket-based capability to provide functionality equivalent to that provided in the C API by azn_decision_access_allowed().

The azn_decision_access_allowed() function requires the following information:

- Authentication information
- Resource name
- Access mode

The Java 2 permission model provides the resource name and the access mode. The Java Authentication and Authorization Services (JAAS) extensions to the Java 2 model provide the authentication information.

Policy Director functions as a back-end for normal Java 2 permission checks by providing:

- A custom JAAS LoginModule that manufactures authentication credentials.
- A custom permission class that knows how to locate and call Policy Director.
6.2 **Java™ Authentication and Authorization Services (JAAS) Model**

The Java 2 permission model takes into account the following information:

- The physical origin (the directory or URL) of the classes that are currently active.
- The logical origin of those classes.
- The identity of the organization that produced the classes, as proved by digital signature.

This model serves well the browsers that first popularized Java, as it deals effectively with the issues of mobile code.

JAAS augments the current Java™ 2 runtime to add knowledge of the user who is trying to run the application. This knowledge provides the authentication information needed when implementing the security model.

JAAS augments the Java™ 2 security model to enable the following features:

- Specification of permissions based on a user's identity.
- Enforcement of those permissions at application runtime.

These two features provide the authorization functionality needed when implementing the security model.

The following sections describe how Policy Director Authorization Java Classes use the JAAS model:

- **Authenticating Users and Obtaining Credentials (Section 6.2.1)**
- **Authorizing Access Requests (Section 6.2.2)**
6.2.1 Authenticating Users and Obtaining Credentials

The Policy Director Java-based authentication feature is built around the Java™ Authentication and Authorization Services (JAAS) model.

Note: For more information on the JAAS, access the following URL:
http://java.sun.com/products/jaas

Policy Director provides one JAAS LoginModule. You can use the module in two different ways. You can use it to authenticate a user and obtain the user’s credentials. Alternatively, you can use it just to obtain the user’s credentials.

Authenticating with a Username and Password

In order to authenticate a user, the LoginModule requires that the calling application provide the following:

- A principal name, specified as either a short name or a X500Name (DN)
- A password

The LoginModule authenticates the principal and returns the Policy Director credential. The LoginModule expects the calling application to provide the following information:

- The username, through a javax.security.auth.callback.NameCallback
- The password, through a javax.security.auth.callback.PasswordCallback.

When the Policy Director credential is successfully retrieved, the JAAS LoginModule creates a Subject and a PDPrincipal.

Retrieving Credentials without Authenticating

To retrieve credentials without authenticating, the calling application can call the JAAS Login Module with only a principal name as a short name or a X500Name (DN).

The LoginModule will expect the calling application to provide the username through a javax.security.auth.callback.NameCallback.
Using the Login Configuration File

You can use an entry in the login configuration file to specify which of two login modes your application uses. You can configure the module to either require both a user name and a password, or just a user name.

This configuration takes the form of an optional keyword, `nameOnly=true`. If `nameOnly` is omitted or specified to be `false`, both the user name and the password are required.
6.2.2 **Authorizing Access Requests**

The Policy Director Authorization Java Classes are built around JAAS and the Java™ 2 security model. The Policy Director API closely follows the Java 2 permission model.

---

**Note:** For more information on the Java 2 security model, see: http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html

---

The Policy Director Java Classes provide a new permission class named PDPermission. This class extends the abstract class `com.ibm.IBMPPermission`, which extends the abstract class `java.security.Permission.java.security.Permission`. PDPermission has a static initializer that establishes the SSL-protected socket communications protocol which is used to talk to Policy Director.

An entry needs to be made in the Java 2 policy file to insure that the Java™ 2 security code calls the implies() method in the PDPermission class described below. This entry could be made specific to particular codebases, as desired.

```java
grant signedBy "xxx" codeBase "file:/E:/Program Files/aaa/bbb/ccc"
principal com.tivoli.mts.PDPrincipal "*" {
  permission com.ibm.mts.PdPermission "ignoreme";
};
```

The contents of the action string `ignoreme` above are unimportant because the PDPermission class ignores them. This is because Policy Director acts as the repository for security policy. The intent of this entry is to get the Java security code to call PDPermission's `implies()` method when some resource manager checks to see if a PDPermission is held.

PDPermission implements `implies()`, `equals()`, `getActions()` and `hashCode()`, as well as two constructors for the class.

The `implies()` method flow consists of the following steps:

1. Use the static `getSubject()` method to retrieve the current Subject. (Subject was created by the LoginModule, and placed on the current thread of execution by the resource manager.)

2. If the Subject contains a Principal of type `com.tivoli.mts.PDPrincipal`, then the appropriate credentials are secured for the call to Policy Director.
The example below illustrates one way a resource manager (WebServer, EJB container, etc.) could place the Subject on the current thread of execution.

```java
Subject.doAs(whoami, new java.security.PrivilegedAction() {
    public java.lang.Object run() {
    }
});
```

At this point the PDPermission class has all the information required to make the authorization call to Policy Director.

Following is an example of a typical authorization check that invokes the Policy Director through the PDPermission class implementation. The checkPermission() method returns quietly unless it fails, in which case it throws a java.lang.SecurityException.

```java
PDPermission perm = new PDPermission("/MyResourceManager/private",
        "[simple]rT[newActionGroup1]Z");

SecurityManager.checkPermission(perm);
```
6.3 Software Requirements for Installation

The Policy Director Authorization Java Classes software requirements that must be met for installation are described in the following sections:

- Policy Director Requirements (Section 6.3.1)
- JAAS and JRE Requirements (Section 6.3.2)
- Java Archive Requirements (Section 6.3.3)

6.3.1 Policy Director Requirements

The Policy Director Authorization Java Classes are distributed in the Policy Director Authorization ADK package. In general, this package depends on the Policy Director Runtime Environment package. However, the Policy Director Authorization Java Classes code does not use any files from the Policy Director Runtime Environment. Therefore, this Java code can be used on systems that have no other Policy Director code present.

Note: The Policy Director Authorization Java Classes support use of remote cache mode for accessing the Policy Director authorization database. Local cache mode is not supported.

6.3.2 JAAS and JRE Requirements

The Policy Director Authorization Java Classes require the following Java software:

- The Java Authentication and Authorization Services (JAAS) standard extension
- A supporting Java Runtime Environment (JRE)

These requirements are satisfied by either of the following software distributions:

- Java 1.3 with the JAAS standard extension
- An IBM JVM (Java 1.2.2 or higher) that includes the JAAS

The security technologies in this Policy Director package that are outside of the JAAS standard require Java 2. They been tested on both Java 1.2.2 and Java 1.3.
6.3.3 Java Archive Requirements

The Policy Director Authorization Java Classes are packaged in the Java Archive (JAR) file PDPerm.jar. This JAR file is distributed as part of the Policy Director Authorization ADK.

This JAR file must be copied to the JRE's extension directory. Typically this is:

```
<java.home>/jre/lib/ext
```

Installation of the JAR file in the JRE extension directory enables it to be granted java.security.AllPermission. If the JAR file is not moved to this directory, you must complete a number of manual steps in order to set a variety of permissions which are needed for full functionality.

The Policy Director Authorization ADK distributes other supporting archives that must be copied to the same JRE extensions directory.

<table>
<thead>
<tr>
<th>Java Archive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibmjsse.jar</td>
<td>SSL support</td>
</tr>
<tr>
<td>ibmjcefw.jar</td>
<td>Part of Java Cryptography Extension</td>
</tr>
<tr>
<td>ibmjceprovider.jar</td>
<td></td>
</tr>
<tr>
<td>local_policy.jar</td>
<td></td>
</tr>
<tr>
<td>US_export_policy.jar</td>
<td></td>
</tr>
<tr>
<td>ibmpkcs.jar</td>
<td>Public Key Cryptography Support</td>
</tr>
<tr>
<td>jaas.jar</td>
<td>Java Authentication and Authorization Services.</td>
</tr>
<tr>
<td></td>
<td>There is a version of this file for each of the</td>
</tr>
<tr>
<td></td>
<td>following Java releases:</td>
</tr>
<tr>
<td></td>
<td>• Java 1.2.2</td>
</tr>
<tr>
<td></td>
<td>• Java 1.3</td>
</tr>
</tbody>
</table>

The JAR files are distributed in two separate directories, one for Java 1.2.2, and one for Java 1.3.
6.4 Java Classes Overview

The Policy Director Authorization Java Classes consists of the following classes:

- com.tivoli.mts.PDLoginModule (Section 6.4.1)
- com.tivoli.mts.PDPrincipal (Section 6.4.2)
- com.tivoli.mts.PDPermission (Section 6.4.3)
- com.tivoli.mts.SvrSslCfg (Section 6.4.4)

See Chapter 7, ”Authorization Java Classes Reference” for detailed information about each of these classes.

6.4.1 com.tivoli.mts.PDLoginModule

This class knows how to authenticate to Policy Director using a username and password. This class expects to be run inside the JAAS framework.

```java
public class PDLoginModule implements javax.security.auth.spi.LoginModule{
    public login()
    public logout()
    public abort()
    public commit()
    public initialize(javax.security.auth.Subject subject,
                      javax.security.auth.callback.CallbackHandler callbackHandler,
                      java.util.Map sharedState,
                      java.util.Map options)
}
```

For more information, see the Class PDLoginModule reference page.
6.4.2 com.tivoli.mts.PDPrincipal

This class represents the identity of a Policy Director user.

```java
class PDPrincipal implements Principal, PrincipalComparator, java.io.Serializable {
    public PDPrincipal(String name)
    public PDPrincipal(String name, String password)
    public String getName()
    public String toString()
    public boolean equals(Object o)
    public int hashCode()
    public boolean implies(Subject subject)
    public boolean implies(PDPermission permission)
}
```

For more information, see the Class PDPrincipal reference page.

6.4.3 com.tivoli.mts.PDPermission

This class knows how to check Policy Director for specified actions. Resource managers and applications can create a subclass or use PDPermission directly to get a Permission class. The Permission class goes to Policy Director on a java.security.SecurityManager.checkPermission(perm) call.

```java
class PDPermission extends Permission {
    public PDPermission(String rname, String actions)
    public boolean implies(Permission p)
    public boolean equals(Object obj)
    public String getActions()
    public int hashCode()
    public boolean implies(PDPrincipal principal)
}
```

For more information, see the Class PDPermission reference page.

6.4.4 com.tivoli.mts.SvrSslCfg

This class configures SSL communication with remote Policy Director servers.

```java
class SvrSslCfg extends java.lang.Object {
    public static void main(java.lang.String[] argv)
}
```

For more information, see the Class SvrSslCfg reference page.
6.5 Configuring a Java Application into Policy Director

To configure a Java application to run within the Policy Director secure domain, you must establish an identity for the application, and configure the location of the Policy Director Management Server and Policy Director Authorization Server.

Policy Director uses a self-generated and self-signed certificate to authenticate its Secure Sockets Layer (SSL) communications. Therefore, the Java SSL support contained in the Policy Director Authorization Java Classes must determine which certificate Policy Director is using. This is accomplished by using the SvrSslCfg class.

To configure a Java application into the Policy Director secure domain, complete the instructions in the following sections:

- Establishing an Identity with Proper Credentials (Section 6.5.1)
- Identifying the Policy Director Server Machines (Section 6.5.2)
- Using the SvrSslCfg Class (Section 6.5.3)
6.5.1 Establishing an Identity with Proper Credentials

Complete the following instructions:

1. Use the `pdadmin` command to authenticate as the Policy Director administrative user. For example, enter the following command at a command prompt:
   ```
   >pdadmin -a sec_master -p secretpw
   ```
   The LDAP administrative user is sec_master. The `-p` parameter specifies the password for sec_master.

2. Use the `pdadmin` command to create an account for this machine:
   ```
   pdadmin> user create PdPermission9999 \
   cn=PdPermission/scalawag,o=ibm,c=us PdPermission/scalawag \
   PdPermission secretpw9999
   ```
   The above command illustrates creation of a user on a machine named scalawag. The machine scalawag will use Policy Director for permission checks.
   
   An account should be set up for each machine that will be using the PDPPermission class. The name must be unique, since a certificate is created to represent this account, and the certificate will be registered with Policy Director.

3. Use `pdadmin` to make the user’s account valid.
   ```
   pdadmin> user modify PdPermission9999 account-valid yes
   ```

4. Use `pdadmin` to add the new user to the group necessary for performing remote permissions checks.
   ```
   pdadmin> group modify remote-acl-users add PdPermission9999
   ```

6.5.2 Identifying the Policy Director Server Machines

Identify the machines that are running the Policy Director Management Server (PDMgrd) and the Policy Director Authorization Server (PDAcld). Use a fully qualified domain name.

For example:

- flyer.austin.lab.tivoli.com

  For this example, the machine running the Policy Director Management Server

- timon.austin.lab.tivoli.com

  The machine running the Policy Director Authorization Server.
6.5.3 Using the SvrSslCfg Class

Based on the example shown above in Section 6.5.1: “Establishing an Identity with Proper Credentials” and Section 6.5.2: “Identifying the Policy Director Server Machines”, use the SvrSslCfg Class as follows:

```
java com.tivoli.mts.SvrSslCfg cn=PdPermission/scalawag,o=ibm,c=us \
  secretpw flyer.austin.lab.tivoli.com timon.austin.lab.tivoli.com
```

SvrSslCfg Usage Syntax

The use of SvrSslCfg can be summarized as follows:

```
java com.tivoli.mts.SvrSslCfg userDN pw mgrd acld
```

The above command includes the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userDN</td>
<td>Distinguished name of a Policy Director user. This account must already exist</td>
</tr>
<tr>
<td>pw</td>
<td>The password for sec_master</td>
</tr>
<tr>
<td>mgrd</td>
<td>The machine running the Policy Director Management Server.</td>
</tr>
<tr>
<td>acld</td>
<td>The machine running the Policy Director Authorization Server.</td>
</tr>
</tbody>
</table>

The result of this configuration is that a `PdPerm.properties` file ends up in the `java.home` directory and a `pdperm.ks` file ends up in `java.home/jre/lib/security`.

**Note:** If either of these files is damaged, the configuration steps must be repeated. Creating backups of these two files is recommended.

The SvrSslCfg class supports additional parameters. For more information, see the Class SvrSslCfg reference page.
6.6 Configuring Java Authentication and Authorization Service

This section describes how to setup and use a login configuration file with the Policy Director Authorization Java Classes. The Policy Director configuration steps follow the configuration methods supported by the Java Authentication and Authorization Service (JAAS).

This section does not provide an overview of all of the JAAS configuration options. We recommend that you review the JAAS configuration information that can be accessed through the following URL:

http://java.sun.com/products/jaas

Complete the instructions in the following sections:

- Creating a Login Configuration File (Section 6.6.1)
- Specify the Login File Location (Section 6.6.2)

6.6.1 Creating a Login Configuration File

To create a login file, review the following example: The example login configuration file described below is named:

d:\java\j122ibm\jre\lib\security\config.pd

Following is an example of a standard JAAS login configuration file.

```plaintext
/// config.pd: login configuration file for PDLoginModule

pd-debug {
  com.tivoli.mts.PDLoginModule required debug=true;
};

pd {
  com.tivoli.mts.PDLoginModule required;
};

pd-nopass {
  com.tivoli.mts.PDLoginModule required nameOnly=true;
};
```

Note that the last stanza allows applications that use `pd-nopass` in their LoginContext constructor to simply supply usernames but not passwords. For more information, see the Class PDLoginModule reference page or review the javadoc HTML page for com.tivoli.mts.PDLoginModule.
6.6.2 Specify the Login File Location

Choose one of the following ways to specify the login file location:

- Point to the login configuration file from the file
  \texttt{java.home\jre\lib\security\java.security}

  Following is a sample entry from the \texttt{java.security} file:
  \texttt{login.config.url.1=file:d:/Java/j122ibm/jre/lib/security/config.pd}

- Specify the appropriate -D options on the \texttt{java} command line invocation.
  For more information, see JAAS configuration documentation.
Developing a Resource Manager

A resource manager is a Java application that uses the Java Authentication and Authorization Service and the Policy Director Authorization Java Classes to make access control decisions. The following code sample illustrates the tasks that the resource manager must accomplish:

```java
// identify configuration status and callback routine
lc = new LoginContext("pd-debug", np);

// drive LoginModule's login and commit methods
lc.login();
whoami = lc.getSubject();
System.out.println(whoami);

// become that user
Subject.doAs(whoami, new java.security.PrivilegedAction() {
    public java.lang.Object run() {
        boolean worked;
        java.security.Permission perm = new PDPermission("/test/private", ";a");
        try {
            // sm is a reference to a SecurityManager
            sm.checkPermission(perm);
            worked = true;
        } catch (AccessControlException e) {
            if (VERBOSE) e.printStackTrace();
            worked = false;
        }
        if (worked) {
            System.out.println("use user "+ user + " has "+
                perm.getActions()+" permission(s) to target "+perm.getName());
        } else {
            System.out.println("use user "+ user + " DOES NOT HAVE "+
                perm.getActions()+" permission(s) to target "+perm.getName());
        }
    }
});
```

6.7 Developing a Resource Manager
6.8 Making Authorization Decisions Outside of Java 2

The Policy Director Authorization Java Classes also support a completely Java-compliant usage of Policy Director's authorization check that is outside of the Java 2 and JAAS framework.

The usage is simple.

The PDPrincipal class has one constructor that takes a name and password and authenticates to Policy Director as part of the construction of the object. The PDPrincipal class also has a constructor that simply takes a name.

A security check is performed on the current environment when one is using the no-password version of the constructor. The permission that must be held is 'permission javax.security.auth.AuthPermission "createPDPrincipal"'.

If authorized, the constructor will retrieve authentication information from Policy Director for that entity. The names that are supported on these constructors can either be Policy Director shortnames, or distinguished names.

When you have constructed a PDPrincipal for the specified entity, you can construct a PDPermission with the name of the requested resource (protected object) and the requested action to be performed on that object.

You can then invoke PDPrincipal.implies(PDPermission) to see if the specified access is allowed on the specified object by the specified entity.

The following code shows an example of how to achieve these tasks:

```java
PDPrincipal whoIsIt = new PDPrincipal("tom", "letmein");
PDPermission whatTheyWant = new PDPermission("/everything", "abT");
boolean haveAccess = whoIsIt.implies(whatTheyWant);
if (haveAccess) {
    // let them proceed...
} else {
    // deny the requested access
}
```
This section contains reference pages for the following mts.tivoli.com classes:

- Class PDLoginModule
- Class PDPermission
- Class PDPrincipal
- Class SvrSslCfg
Class PDLoginModule

com.tivoli.mts.PDLoginModule

public class PDLoginModule extends java.lang.Object
implements javax.security.auth.spi.LoginModule

Description

This PDLoginModule represents the JAAS authentication mechanism for Policy Director.

A CallbackHandler is required to retrieve a Policy Director user name and password pair.

The objectives of the PDLoginModule class are to:

- Authenticate a Policy Director user
- Create a PDPrincipal based on the username
- Obtain the user's Policy Director credentials if authentication is successful.

This LoginModule recognizes the debug option. If set to true in the login Configuration, debug messages will be output to the output stream, System.out.

This LoginModule recognizes the nameOnly option. If set to true in the login Configuration, the LoginModule will not prompt for a password through the Callback mechanism, but will only prompt for a name.

To use this module with the nameOnly option, the calling application must have the following permission:

'permission javax.security.auth.AuthPermission "createPDPrincipal"'
Constructor Summary

- PDLoginModule()

Method Summary

  Initialize this LoginModule.
- boolean login()
  Authenticate the user (first phase).
- boolean commit()
  Commit the authentication (second phase).
- boolean abort()
  Abort the authentication (second phase).
- boolean logout()
  Logout the user

Methods inherited from class java.lang.Object
- equals
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait
- wait
- wait

Constructor Detail

PDLoginModule

public PDLoginModule()
Chapter 7: Authorization Java Classes Reference

Method Detail

**initialize**

```java
public void initialize(javax.security.auth.Subject subject, 
                        javax.security.auth.callback.CallbackHandler callbackHandler, 
                        java.util.Map sharedState, 
                        java.util.Map options)
```

Initialize this LoginModule.

Specified by:

initialize() in interface javax.security.auth.spi.LoginModule

Parameters:

- **subject**
  - The Subject to be authenticated.

- **callbackHandler**
  - A CallbackHandler for communicating with the end user. For example, prompting for user names and passwords.

- **sharedState**
  - Shared LoginModule state.

- **options**
  - Options specified in the login configuration for this particular LoginModule.

**login**

```java
public boolean login(
                    throws javax.security.auth.login.LoginException
```

Authenticate the user (first phase).

This method attempts to authenticate the user to Policy Director and retrieve the user's credentials.

Specified by:

login() in interface javax.security.auth.spi>LoginModule

Returns:

true in all cases (this LoginModule should not be ignored).

Throws:

- javax.security.auth.login.FailedLoginException
  - When attempts to retrieve the underlying system information fail.
commit
public boolean commit()
    throws javax.security.auth.login.LoginException
    
    Commit the authentication (second phase).

    This method is called if the LoginContext's overall authentication
    succeeded (the relevant REQUIRED, REQUISITE, SUFFICIENT and
    OPTIONAL LoginModules succeeded).

    If this LoginModule's own authentication attempt succeeded (the
    importing of the Policy Director authentication information succeeded),
    then this method associates the Policy Director Principals with the
    Subject currently tied to the LoginModule. If this LoginModule's
    authentication attempt failed, then this method removes any state that
    was originally saved.

    Specified by:
    commit() in interface javax.security.auth.spi.LoginModule

    Returns:
    true if this LoginModule's own login and commit attempts
    succeeded, or false otherwise.

    Throws:
    javax.security.auth.login.LoginException
    When the commit() fails.
Chapter 7: Authorization Java Classes Reference

abort
public boolean abort()
    throws javax.security.auth.login.LoginException

Abort the authentication (second phase).
This method is called if the LoginContext's overall authentication failed.
(the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules did not succeed).
This method cleans up any state that was originally saved as part of the authentication attempt from the login and commit methods.

Specified by:
abort() in interface javax.security.auth.spi.LoginModule

Returns:
false if this LoginModule's own login() or commit() attempts failed, and true otherwise.

Throws:
javax.security.auth.login.LoginException
When the abort() fails.

logout
public boolean logout()
    throws javax.security.auth.login.LoginException

Logout the user.
This method removes the Principals associated with the Subject.

Specified by:
logout() in interface javax.security.auth.spi.LoginModule

Returns:
true in all cases (this LoginModule should not be ignored).

Throws:
javax.security.auth.login.LoginException
When logout() fails.
Class PDPermission

com.tivoli.mts.PDPermission

public class PDPermission
extends java.security.Permission

Description

This class represents an authorization permission for accessing a resource
object in the Policy Director secure domain.

PDPermission allows usage of Policy Director as the authorization back-end
for normal Java 2 permission checks. In PD 3.7, the Policy Director
Authorization Server offers an SSL-based access mode for remote ACL
checking.

Permission check is done by the implies() method which sets up an SSL-based
connection to a remote Policy Director Authorization Server for the
authorization check.

Permissions are created with resource name and actions. They are immutable
once they are created.

The actions must correspond to one of the operations for which a Policy
Director permission has been defined. The actions strings use this format: [].
Action group name is optional. Action lists consist of action names which are
one character in length. Examples of valid actions strings are: "[primary]rwc"
"rw" "[java]rc[primary]rwx".

When you create new actions for use with the PDPermission class, you must
also define them in the Policy Director secure domain. You can use either the
Policy Director Management Console or the pdadmin command to create new
actions. See the Policy Director Base Administration Guide for instructions on
creating new actions, and for a complete list of actions that Policy Director
supports by default.

The requested resource to query for must correspond to a resource that has
been defined as a protected object within Policy Director's protected object
namespace.
Chapter 7: Authorization Java Classes Reference

Constructor Summary

- `PDPermission(java.lang.String rname, java.lang.String actions)`
  Creates a new PDPermission object with the specified actions and resource.

Method Summary

- `boolean equals(java.lang.Object obj)`
  Checks two PDPermission objects for equality.

- `java.lang.String getActions()`
  Returns the canonical string representation of the actions.

- `int hashCode()`
  Returns the hash code value for this object.

- `boolean implies(java.security.Permission p)`
  Determine if Policy Director grants the specified permission.

- `boolean implies(PDPrincipal principal)`
  Determine if the current PDPrincipal has the specified PDPermission.

- Methods inherited from class `java.security.Permission`:
  - `checkGuard`
  - `getName`
  - `toString`
  - `newPermissionCollection`

- Methods inherited from class `java.lang.Object`:
  - `getClass`
  - `notify`
  - `notifyAll`
  - `wait`
  - `wait`
  - `wait`
Constructor Detail

PDPermission

public PDPermission(java.lang.String rname,
                    java.lang.String actions)

Creates a new PDPermission object with the specified actions and
resource.

Action names can only be one character in length. Policy Director
supports a default list of action names. Administrators may add additional
action groups and additional permissions.

Parameters:

rname
   Name of the target resource object.

actions
   Name of the requested operations. The actions can be
concatenated together to form complex operation strings. The
format is "[""""< action-name>.. "["""".... Example action
strings for the Policy Director Authorization API would be:
"[primary]rwc" "rw" "[java]re[primary]rwx". "primary" is the
default action group if an action group name is not specified.
Method Detail

**implies**

```java
public boolean implies(java.security.Permission p)
```

Checks to see if Policy Director grants the specified permission. More specifically, this method returns true if:

- `p` is an instance of PDPermission,
- `p`'s actions are a proper subset of this object's action map, and
- `p` is a granted permission by ivacld for the specified principal

*Overrides:*  
implies() in class com.ibm.IBMPermission

*Parameters:*  
- `p`  
The permission to check.

*Returns:*  
true if the specified permission is granted by Policy Director, false if not.
implies

public boolean implies(PDPrincipal princ)

Determines if Policy Director grants the specified permissions to the PDPrincipal.

This method returns true if:

- princ is non-null.
- This PDPermission's actions are a proper subset of this object's action map, and
- The specified actions are allowed by the Policy Director Authorization Server for the specified principal.

Parameters:
    princ
        The PDPrincipal whose permissions will be checked.

Returns:
    true if the specified permission is implied by this object,
    false if not.

equals

public boolean equals(java.lang.Object obj)

Checks two PDPermission objects for equality. Checks that obj is a PDPermission, and has the same name and actions as this object.

Overrides:
    equals() in class com.ibm.IBMPermission

Parameters:
    obj
        The object that is being tested for equality with this object.

Returns:
    true if obj is a PDPermission, and has the same name and actions as this PDPermission object.
Chapter 7: Authorization Java Classes Reference

---

**getActions**

```java
public java.lang.String getActions()
```

Returns the canonical string representation of the actions.

**Overrides:**

getActions() in class java.security.Permission

**Returns:**

The canonical string representation of the actions.

---

**hashCode**

```java
public int hashCode()
```

Returns the hash code value for this object.

**Overrides:**

hashCode() in class com.ibm.IBMPermission

**Returns:**

A hash code value for this object.
Class PDPrincipal

```java
com.tivoli.mts.PDPrincipal
public class PDPrincipal
extends java.lang.Object
implements java.security.Principal,
com.ibm.security.auth.PrincipalComparator, java.io.Serializable
```

Description

This class implements the Principal interface and represents general information about a Policy Director user.

 Principals such as this PDPrincipal may be associated with a particular Subject to augment that Subject with an additional identity. Refer to the Subject class for more information on how to achieve this. Authorization decisions can then be based upon the Principals associated with a Subject.

Constructor Summary

- `PDPrincipal(java.lang.String name, java.lang.String password)`
  Create a PDPrincipal with an identifying name.

- `PDPrincipal(java.lang.String name)`
  Create a PDPrincipal with an identifying name.

Method Summary

- `boolean equals(java.lang.Object o)`
  Compare the specified Object with this PDPrincipal for equality.

- `java.lang.String getName()`
  Return a string name of this PDPrincipal.

- `int hashCode()`
  Return a hash code for this PDPrincipal.

- `boolean implies(PDPermission perm)`
  Determine if the current PDPrincipal has the specified PDPermission.

- `boolean implies(javax.security.auth.Subject subject)`
  Check if the specified Subject is implied by this object.
Chapter 7: Authorization Java Classes Reference

- `java.lang.String toString()`
  Return a string representation of this PDPrincipal.

- Methods inherited from class `java.lang.Object`
  - `getClass`
  - `notify`
  - `notifyAll`
  - `wait`
  - `wait`
  - `wait`

### Constructor Detail

#### PDPrincipal

```java
public PDPrincipal(java.lang.String name, java.lang.String password)
```

Create a PDPrincipal with an identifying name.

**Parameters:**

- `name`
  The identifying name.

- `password`
  The password for the identifying name.

**Throws:**

- `NullPointerException`
  When the name is null.

- `java.lang.IllegalArgumentException`
  When the name has zero length.

- `java.lang.SecurityException`
  When the caller does not have permission to create PDPrincipals.

- `java.lang.IllegalStateException`
  When Policy Director could not obtain this user’s credentials.
PDPrincipal

public PDPrincipal(java.lang.String name)

Create a PDPrincipal with an identifying name.

Parameters:

   name
       The identifying name.

Throws:

   NullPointerException
       When the name is null.

   java.lang.IllegalArgumentException
       When the name has zero length.

   java.lang.SecurityException
       When the caller does not have permission to create PDPrincipals.

   java.lang.IllegalStateException
       When Policy Director could not obtain this user's credentials.
Method Detail

implies

public boolean implies(PDPermission perm)

Determine if the current PDPrincipal has the specified PDPermission.

Parameters:

perm

The specified PDPermission.

Returns:

ture if the current PDPrincipal has the specified PDPermission and false otherwise.

getName

public java.lang.String getName()

Return a string name of this PDPrincipal.

Specified by:

getName() in interface java.security.Principal.

Returns:

A string name of this PDPrincipal.

toString

public java.lang.String toString()

Return a string representation of this PDPrincipal.

Specified by:

toString() in interface java.security.Principal

Overrides:

toString() in class java.lang.Object

Returns:

A string representation of this PDPrincipal.
equals

public boolean equals(java.lang.Object o)

    Compares the specified Object with this PDPrincipal for equality. Returns true if the given object is also a PDPrincipal and the two PDPrincipals have the same String representation.

    Specified by:
    equals() in interface java.security.Principal

    Overrides:
    equals() in class java.lang.Object

    Parameters:
    o
    
    Object to be compared for equality with this PDPrincipal.

    Returns:
    true if the specified Object is equal to this PDPrincipal.

hashCode

public int hashCode()

    Return a hash code for this PDPrincipal.

    Specified by:
    hashCode() in interface java.security.Principal

    Overrides:
    hashCode() in class java.lang.Object

    Returns:
    A hash code for this PDPrincipal.
implies

public boolean implies(javax.security.auth.Subject subject)

Check if the specified Subject is implied by this object.

Specified by:
implies() in interface
com.ibm.security.auth.PrincipalComparator

Parameters:

subject
A Subject to check for implication.

Returns:

true if the specified Subject is implied by this object, or
false otherwise.
Class SvrSslCfg

com.tivoli.mts.SvrSslCfg
public class SvrSslCfg
extends java.lang.Object

Description

This program configures this machine for participation with Policy Director in resolving Java security checks that involve the PDPermission class or subclasses of PDPermission.

The output of a successful run of this program is a set of properties in a file named PdPerm.properties and a JCE KeyStore that has been initialized to both recognize the certificate that Policy Director is currently using and to securely store a client certificate. The client certificate allows PDPermission to make authenticated usage of Policy Director services to check authorization.

Constructor Summary

- SvrSslCfg ()

Method Summary

- static void main(java.lang.String[] argv)
  This routine configures this machine for use with Policy Director.

- Methods inherited from class java.lang.Object:
  - equals
  - getClass
  - hashCode
  - notify
  - notifyAll
  - toString
  - wait
  - wait
  - wait
Chapter 7: Authorization Java Classes Reference

Constructor Detail

**SvrSslCfg**

public SvrSslCfg()

Method Detail

**main**

public static void main(java.lang.String[] argv)

This routine configures this machine for use with Policy Director.

To illustrate how this configuration might be done, consider the following example. The administrator will have run the following set of commands on the Policy Director Management Server. In this example, the Policy Director Management Server is on the host timon.i.com:

```
pdadmin -a sec_master -p xxxpassword
user create PdPermission9993
cn=PdPermission/gordo,o=myCompany PdPermission/gordo
PdPermission somePW9993
user modify PdPermission9993 account-valid yes
group modify remote-acl-users add PdPermission9993
```

(This example assumes that the LDAP root for the users is anchored at o=myCompany, and uses the convention that the "distinguishing" part of the distinguished name will in fact be the hostname of the machine, which in this example is "gordo".)

The command that could be issued now on gordo is:

```
java com.tivoli.mts.SvrSslCfg cn=PdPermission/gordo,o=myCompany xxxpassword timon.i.com rafiki.i.com
```

This example command takes the default SSL port numbers for the Policy Director Authorization Server and the Policy Director Management Server, and is configuring gordo to use the Policy Director Authorization Server on the host rafiki.i.com.
Parameters:

dn
The distinguished name of the principal that this machine is to use.

pw
The password for sec_master.

ivmgrd-hostname
The name of the machine that is running the Policy Director Management Server.

ivacld-hostname
The name of the machine that is running the instance of the Policy Director Authorization Server with which this machine is to communicate.

ivmgrd-port
The SSL port number for the Policy Director Management Server, if other than the default (7135).

ivacld-port
The SSL port number for Policy Director Authorization Server, if other than the default (7136).

Throws:

java.lang.IllegalArgumentException
When it appears that invalid data was passed in.

java.lang.IllegalStateException
When this request could not be processed.
Authorization C API: DCE Support

This appendix contains information specific to the creation of DCE-based Policy Director applications. Information in this Appendix supplements or replaces information in Chapter 1, "Authorization C API".

Topic Index:
- A.1 Building Applications with the Authorization API
- A.2 Initializing the Authorization Service
- A.3 Authenticating an API Application
- A.4 Obtaining User Authorization Credentials
- A.5 Handling Credentials
- A.6 Deploying Applications with the Authorization API
A.1 Building Applications with the Authorization API

The Policy Director Version 3.7 Authorization API supports DCE in a backwards compatibility mode only. Developers of DCE-based Policy Director applications are encouraged to migrate their applications to a non-DCE environment.

The Policy Director Authorization ADK distributes a non-DCE library as part of the Authorization ADK. The version of this library that supports DCE is distributed in the Policy Director Runtime. This library is provided for use when linking DCE-based Policy Director applications.

The following information supplements Section 1.2: “Locating the Authorization API Components”.

The Policy Director libraries that implement the DCE-supported versions of API functions are:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris</td>
<td>libivauthzn.so</td>
</tr>
<tr>
<td>AIX</td>
<td>libivauthzn.a</td>
</tr>
<tr>
<td>HP-UX</td>
<td>libivauthzn.sl</td>
</tr>
<tr>
<td>Microsoft Windows</td>
<td>ivauthzn.dll -- include at run time</td>
</tr>
<tr>
<td></td>
<td>ivauthzn.lib -- library to link against</td>
</tr>
</tbody>
</table>

A.1.1 Software Requirements

The following information supplements Section 1.3.1: “Software Requirements”.

When developing DCE-based Policy Director applications, the development environment must include a DCE runtime.

When the Policy Director secure domain uses a DCE user registry, the DCE runtime (client) must be installed as a prerequisite to the Policy Director Runtime Environment installation.

On all platforms, DCE libraries are needed at application runtime.
A.2 Initializing the Authorization Service

Information in this section supplements Section 1.7: “Initializing the Authorization Service”. Initializing the Authorization Service for DCE-based Policy Director applications is similar to initializing for non-DCE based applications.

The following table describes which sections in Section 1.7 apply to DCE-based applications. Some sections apply unchanged. Some sections are replaced by text in this Appendix. Some sections are supplemented by text in this Appendix. Some sections are not used with DCE-based applications.

<table>
<thead>
<tr>
<th>Section</th>
<th>Usage with DCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifying an Authorization API Configuration File (Section 1.7.1)</td>
<td>Use unchanged.</td>
</tr>
<tr>
<td>Specifying the Maximum Number of Handle Groups (Section 1.7.2)</td>
<td></td>
</tr>
<tr>
<td>Specifying the Type of Cache Mode (Section 1.7.3)</td>
<td></td>
</tr>
<tr>
<td>(deleted remote cache mode configuration section)</td>
<td>Replace with the following section: Configuring Remote Cache Mode (Section A.2.1)</td>
</tr>
<tr>
<td>Configuring Local Cache File Names (Section 1.7.4)</td>
<td>Use unchanged.</td>
</tr>
<tr>
<td>Configuring Local Cache Refresh (Section 1.7.5)</td>
<td>Replace with the following section: Configuring Local Cache Refresh (Section A.2.2)</td>
</tr>
<tr>
<td>Configuring Local Cache Notification Listener (Section 1.7.6)</td>
<td>Replace with the following section: Configuring Local Cache Notification Listener (Section A.2.3)</td>
</tr>
<tr>
<td>Configuring Notification Listener Ports (Section 1.7.7)</td>
<td>Replace with the following section: Configuring Notification Listener Ports (Section A.2.4)</td>
</tr>
<tr>
<td>Configuring SSL from the API Client to Policy Director (Section 1.7.8)</td>
<td>Not used.</td>
</tr>
<tr>
<td>Section</td>
<td>Usage with DCE</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>----------------------------------------------------</td>
</tr>
<tr>
<td>Configuring the Authorization API for LDAP Access (Section 1.7.9)</td>
<td>Use unchanged.</td>
</tr>
<tr>
<td>Configuring LDAP Access over SSL (Section 1.7.10)</td>
<td></td>
</tr>
<tr>
<td>Configuring Advanced LDAP Parameters (Section 1.7.11)</td>
<td></td>
</tr>
<tr>
<td>Specifying LDAP User Registry Replica Access (Section 1.7.12)</td>
<td></td>
</tr>
<tr>
<td>Enabling the Return of Permission Information (Section 1.7.13)</td>
<td>Replace with the following section:</td>
</tr>
<tr>
<td></td>
<td>Enabling the Return of Permission Information</td>
</tr>
<tr>
<td></td>
<td>(Section A.2.6)</td>
</tr>
<tr>
<td>Authenticating an API Application (Section 1.6)</td>
<td>Replace with the following section:</td>
</tr>
<tr>
<td></td>
<td>Authenticating an API Application</td>
</tr>
<tr>
<td></td>
<td>(Section A.3)</td>
</tr>
<tr>
<td>Starting the Authorization Service (Section 1.7.14)</td>
<td>Use unchanged.</td>
</tr>
</tbody>
</table>
A.2.1 Configuring Remote Cache Mode

<table>
<thead>
<tr>
<th>Attribute List Entry:</th>
<th>azn_init_qop</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>none</td>
<td></td>
<td>No protection.</td>
</tr>
<tr>
<td></td>
<td>integrity</td>
<td></td>
<td>Data stream integrity. The data can be seen but not modified or replayed by a third party.</td>
</tr>
<tr>
<td></td>
<td>privacy</td>
<td></td>
<td>Data stream privacy. The data cannot be seen, modified, or replayed by a third party.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration File Entry:</th>
<th>remote-qop</th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Attribute List Entry:</th>
<th>azn_init_remote_ns_loc</th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Configuration File Entry:</th>
<th>remote-ns-loc</th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry:</td>
<td>azn_init_remote_ns_loc</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration File Entry:</th>
<th>remote-ns-loc</th>
<th></th>
</tr>
</thead>
</table>

Using An Attribute List

1. If you specified remote cache mode, use `azn_attrlist_add_entry()` to add the attribute `azn_init_qop` and assign it a value.

2. Use `azn_attrlist_add_entry()` to add the attribute `azn_init_remote_ns_loc` and assign it a value.

Using A Configuration File

1. Go to the `[aznapi-configuration] stanza in the configuration file.

2. Edit the entries for remote-qop and remote-ns-loc.

Initialization of remote cache mode is now complete.
Appendix A: Authorization C API: DCE Support

A.2.2 Configuring Local Cache Refresh

This section replaces Section 1.7.5: “Configuring Local Cache Refresh”.

<table>
<thead>
<tr>
<th>Attribute List Entry:</th>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_init_cache_refresh_interval</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration File Entry:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cache-refresh-interval</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>disable</td>
<td>Refreshing of the local authorization policy database disabled.</td>
</tr>
<tr>
<td></td>
<td>default</td>
<td>600 seconds.</td>
</tr>
<tr>
<td></td>
<td>number of seconds</td>
<td>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.</td>
</tr>
</tbody>
</table>

Using An Attribute List

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

Using A Configuration File

Initialization of remote cache mode is now complete.

1. Go to the `[aznapi-configuration]` stanza in the configuration file.
2. Edit the entries for `cache-refresh-interval`. 
A.2.3 Configuring Local Cache Notification Listener

The following section replaces Section 1.7.6: “Configuring Local Cache Notification Listener”.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry: azn_init_listen_flags</td>
<td>disable</td>
<td>Disable the notification listener.</td>
</tr>
<tr>
<td></td>
<td>enable</td>
<td>Enable the notification listener.</td>
</tr>
</tbody>
</table>

When you select enable, you can also specify any combination of the following values. The values are logically OR’d together.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>use_tcp_port</td>
<td>Enable the listener to use Transmission Control Protocol (TCP).</td>
<td></td>
</tr>
<tr>
<td>use_udp_port</td>
<td>Enable the listener to use User Datagram Protocol (UDP).</td>
<td></td>
</tr>
<tr>
<td>dynamic_port_selection</td>
<td>Instruct the listener to use randomly assigned ports.</td>
<td></td>
</tr>
</tbody>
</table>

Using An Attribute List

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

Using A Configuration File

Go to the [aznapi-configuration] stanza in the configuration file and specify valid values for the `listen-flags` entry. You can enter a combination of values. For example:

```
listen-flags = enable dynamic_port_selection use_tcp_port
```
A.2.4 Configuring Notification Listener Ports

This section replaces Section 1.7.7: “Configuring Notification Listener Ports”

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry: azn_init_tcp_port</td>
<td>port number</td>
<td>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.</td>
</tr>
<tr>
<td>Configuration File Entry: tcp-port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute List Entry: azn_init_udp_port</td>
<td>port number</td>
<td>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.</td>
</tr>
<tr>
<td>Configuration File Entry: udp-port</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Using An Attribute List**

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

**Using A Configuration File**

Go to the [aznapi-configuration] stanza in the configuration file and specify port numbers for either the tcp-port or udp-port entry.
**A.2.5 Registering Local Cache Location with Management Server**

This section replaces the use of `srvsslcfg` to specify local or remote mode. See Section 1.6.1: “Using `srvsslcfg`”.

If you enable the notification listener, you must use the `pdadmin` command to inform the Policy Director Management server (`pdmgrd`) of your location in order to receive notification of updates. Use the `pdadmin 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:

```
pdadmin>server register dbreplica server-name ns-location \ 
       server-principal server-host
```

Where

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server-name</td>
<td>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 <code>ivadmin server list</code> command.</td>
</tr>
<tr>
<td>ns-location</td>
<td>The RPC entry in the CDS namespace where the application exports its RPC bindings.</td>
</tr>
<tr>
<td>server-principal</td>
<td>The name of the DCE principal representing this application process.</td>
</tr>
<tr>
<td>server-host</td>
<td>The Domain Name System (DNS) name or IP address of the machine where this application process resides.</td>
</tr>
</tbody>
</table>
A.2.6 Enabling the Return of Permission Information

The following section supplements Section 1.7.13: “Enabling the Return of Permission Information”.

In addition to the perminfo attribute values described in the section mentioned above, the attribute value \textit{azn\_perminfo\_qop} is valid in a DCE environment.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List Entry:</td>
<td>\textit{azn_init_set_perminfo_attrs} \textit{azn_perminfo_qop}</td>
<td>Include all of the permission information attributes</td>
</tr>
</tbody>
</table>
| Configuration File Entry:   | \textit{permission\_info\_returned} \textit{azn\_perminfo\_qop} | Quality of Protection, which can be one of the following values: *
|                             |                    | • \textit{azn\_perminfo\_qop\_uint\_none}                  |
|                             |                    | • \textit{azn\_perminfo\_qop\_uint\_integrity}             |
|                             |                    | • \textit{azn\_perminfo\_qop\_uint\_privacy}               |

Using An Attribute List Entry as Initialization Data

Use \texttt{azn\_attrlist\_add\_entry()} to add the \textit{azn\_perminfo\_qop} value to the \textit{init\_data} attribute list.

Using A Configuration File

1. Go to the \texttt{[aznapi\_configuration]} stanza in the configuration file.
2. Edit the values for permission\_info\_returned to include the \textit{azn\_perminfo\_qop} value.
Authenticating an API Application

A.3 Authenticating an API Application

The following section replaces Authenticating an API Application (Section 1.6).

Note: The following authentication steps must be taken after the Authorization API is started, as described in Section 1.7.14: “Starting the Authorization Service”

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 (Section A.3.1)
- Logging in Using a Password (Section A.3.2)
A.3.1 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`:

```c
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 the `azn_util_server_authenticate()` reference page.
A.3.2 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”:

```c
/* 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 the `azn_util_client_authenticate()` reference page.
A.4 Obtaining User Authorization Credentials

The following sections replace sections contained in Obtaining User Authorization Credentials (Section 1.9).

<table>
<thead>
<tr>
<th>API Section</th>
<th>Appendix Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifying Authentication User Registry Type (Section 1.9.2)</td>
<td>The following section replaces the API section: Specifying Authentication User Registry Type (Section A.4.1)</td>
</tr>
<tr>
<td>Section 1.9.3: “Specifying User Authentication Identity”</td>
<td>The following section replaces the API section: Specifying User Authentication Identity (Section A.4.2)</td>
</tr>
<tr>
<td>Section 1.9.4: “Specifying Additional User Information”</td>
<td>The following section supplements the API section: Specifying Additional User Information (Section A.4.3)</td>
</tr>
</tbody>
</table>

A.4.1 Specifying Authentication User Registry Type

The following section replaces Section 1.9.2: “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.

If the user was not authenticated in a DCE 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:

<table>
<thead>
<tr>
<th>User Registry Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCE User Registry</td>
<td>IV_DCE</td>
</tr>
<tr>
<td>LDAP User Registry</td>
<td>IV_LDAP</td>
</tr>
<tr>
<td>Unauthenticated</td>
<td>IV_UNAUTH</td>
</tr>
</tbody>
</table>
A.4.2 Specifying User Authentication Identity

The following section replaces Specifying Authentication User Registry Type (Section 1.9.2).

For each user to be authenticated, information is loaded into the data structure that corresponds to the DCE user registry, 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 azn_authdce_t data structure.

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

<table>
<thead>
<tr>
<th>User Identity Type</th>
<th>Data Structure</th>
<th>String</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCE User Registry</td>
<td>azn_authdce_t</td>
<td>principal</td>
<td>cell_admin</td>
</tr>
<tr>
<td>LDAP User Registry</td>
<td>azn_authldap_t</td>
<td>ldap_dn</td>
<td>cn=root</td>
</tr>
<tr>
<td>Unauthenticated User</td>
<td>azn_unauth_t</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

A.4.3 Specifying Additional User Information

The following information supplements the information in Specifying Additional User Information (Section 1.9.4).

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.

For authenticated DCE users, the application can store the additional user information in the azn_authdce_t data structure. For unauthenticated users, the application can store the data in azn_unauth_t.

In addition to the elements described in Section 1.9.4: “Specifying Additional User Information”, the following element can be specified:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qop</td>
<td>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.</td>
</tr>
</tbody>
</table>
A.5 Handling Credentials

The following information supplements Section 1.12: “Handling Credentials”

A.5.1 Attributes Returned by azn_creds_getattrlist_for_subject()

The following information supplements Section 1.12.7: “Obtaining an Attribute List from a Credential”.

The following attributes are only returned in an attribute lists for credentials built from a DCE user registry:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_creds_dce_name</td>
<td>The DCE name used to build these authorization credentials</td>
</tr>
<tr>
<td>azn_creds_qop_info</td>
<td>The Quality of Protection information passed in the mechinfo structure. Valid values are none, integrity or privacy.</td>
</tr>
</tbody>
</table>

A.6 Deploying Applications with the Authorization API

The following information supplements Section 1.13: “Deploying Applications with the Authorization API”.

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.
This section contains reference pages for the following API:

- `azn_authdce_t`
- `azn_util_client_authenticate()`
- `azn_util_server_authenticate()`
Appendix B: Authorization C API: DCE API Reference

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.

Description

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_util_client_authenticate()

Performs a login from a user name and password.

Syntax

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

Parameters

Input

- `principal_name` Name of the principal (user) to be logged in.
- `password` Text password for the user.

Description

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_serverAuthenticate()

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_serverAuthenticate(
    const azn_string_t principal_name,
    const azn_string_t keytab_path
);
```

**Parameters**

**Input**

- `principal_name` Name of the user (principal) to log in.
- `keytab_path` Path to the keytab file containing the principal's key.

**Description**

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.