Note
Before using this information and the product it supports, read the information in Appendix D, “Notices”, on page 217.
## Contents

**Preface** ........................................ ix
- Who should read this book ........................................... ix
- What this book contains ............................................. ix

**Publications** ....................................... xi
- Release information ............................................... xi
- Base information .................................................. xii
- WebSEAL information .............................................. xii
- Web security information ......................................... xii
- Developer references ............................................... xiii
- Technical supplements ............................................. xiii
- Related publications ............................................... xiv
- Accessing publications online .................................... xiv
- Ordering publications ............................................. xv
- Accessibility ..................................................... xv
- Contacting software support ...................................... xv
- Conventions used in this book ..................................... xv
- Typeface conventions ............................................... xv
- Operating system differences ..................................... xvi

**Chapter 1. Introducing the Tivoli Access Manager authorization API.** ........... 1
- Introducing the authorization API .................................. 1
- The Open Group Authorization API standard ..................... 1
- The Tivoli Access Manager authorization model .................. 3
- Locating the Tivoli Access Manager authorization API components ............................................. 3
- Building applications with the authorization API ................. 5
- Demonstration programs ........................................... 5
- Deploying applications with the authorization API .............. 6
- Summarizing authorization API tasks ................................ 6

**Chapter 2. Authorization API functions and data types.** ............................... 7
- API functions .................................................... 7
- Attribute lists .................................................... 7
- Credentials ..................................................... 8
- Authorization decisions ........................................... 8
- Initialization, shutdown, and error handling ..................... 8
- API extensions ................................................... 9
- Character strings ............................................... 9
- Buffers ......................................................... 9
- Protected object structures ..................................... 10
- Default user registry information structure ..................... 10
- Unauthenticated user information structure ..................... 11
- Attribute lists .................................................. 12
- Credential lists ................................................ 14
- Status codes and error handling ................................ 14

**Chapter 3. Initializing the authorization API** .......................... 17
- Specifying an authorization API configuration file ............. 17
- Specifying cache mode settings .................................. 18
- Specifying cache mode type ..................................... 18
- Authorization database file location ............................ 19
- Local cache refresh ............................................. 19
- Local cache notification listener ................................ 20
- SSL listener ports .............................................. 20
- Configuring SSL from the API client to Tivoli Access Manager ............................................. 20
- Specifying an SSL key file ...................................... 21
Specifying a stash file ................................................................. 21
Specifying a keyfile label ............................................................. 21
SSL session timeout ................................................................. 21
SSL password expiration ............................................................ 22
Authentication method ............................................................... 22
User name and password ............................................................ 23
Tivoli Access Manager configuration file location ......................... 23
Password for the SSL keyfile ......................................................... 23
Maximum number of worker threads ............................................ 23
Automatic refresh of SSL certificate and keyfile password ................. 24
Connection timeout ................................................................. 24
Specifying communications attributes for the policy server ............... 24
Policy server hostname ............................................................... 24
Policy server port number .......................................................... 25
Policy server distinguished name ............................................... 25
Specifying values for an authorization server replica ....................... 26
Configuring the authorization API for LDAP access ....................... 26
LDAP user registry support ......................................................... 26
LDAP server host name ............................................................. 27
LDAP server port number .......................................................... 27
LDAP User Distinguished Name .................................................. 27
LDAP User Password ............................................................... 27
Configuring LDAP access over SSL .............................................. 28
SSL communication with the LDAP server .................................... 28
SSL keyfile name ................................................................. 28
SSL keyfile distinguished name .................................................. 28
SSL keyfile password ............................................................... 28
Configuring advanced LDAP parameters ..................................... 29
Maximum search buffer size ....................................................... 29
Caching LDAP data ................................................................. 29
LDAP server query preference .................................................... 29
Authentication method ............................................................. 29
Specifying LDAP user registry replica access ................................ 30
Enabling the return of permission information ................................ 30
Configuring auditing ............................................................... 32
Audit file location ................................................................. 32
Configuring audit logging ......................................................... 32
Configuring logging ............................................................... 32
Log size ................................................................. 32
Log flush interval ................................................................. 33
Specifying the host interface on which to listen .............................. 33
Starting the authorization service .............................................. 33

Chapter 4. Using the authorization API ........................................ 35
Authenticating an API application ............................................. 35
Verifying the identity of a user .................................................... 36
Obtaining user authorization credentials .................................... 36
Specifying the authorization authority ....................................... 37
Specifying user authentication identity ....................................... 37
Specifying additional user information ....................................... 37
Placing user information into an API buffer ................................ 38
Obtaining authorization credentials for the user ............................ 38
Obtaining an authorization decision ........................................... 39
Mapping the user operation to a Tivoli Access Manager permission .... 39
Mapping the requested resource to a protected object .................... 40
Assigning the user credentials to a credentials handle .................... 40
Building an attribute list for additional application information ........ 40
Obtaining an authorization decision ........................................... 41
Cleaning up and shutting down .................................................. 41
Releasing allocated memory ..................................................... 41
Shutting down the authorization API ......................................... 42
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix B. Authorization service plug-in API reference</td>
<td>169</td>
</tr>
<tr>
<td>Appendix C. Authorization API client configuration file</td>
<td>185</td>
</tr>
<tr>
<td>Appendix D. Notices</td>
<td>217</td>
</tr>
<tr>
<td>Index</td>
<td>221</td>
</tr>
</tbody>
</table>
Preface

Welcome to the IBM Tivoli Access Manager Authorization C API Developer’s Reference.

The Tivoli Access Manager application development kit (ADK) is an application development kit for IBM Tivoli Access Manager that enables application developers to add Access Manager authorization and security services to applications.

This book describes the C implementation of the Tivoli Access Manager authorization API. See IBM Tivoli Access Manager Authorization Java Classes Developer’s Reference for a description of the Java classes and methods associated with Tivoli Access Manager authentication.

IBM® Tivoli® Access Manager (Tivoli Access Manager) is the base software that is required to run applications in the IBM Tivoli Access Manager product suite. It enables the integration of IBM Tivoli Access Manager applications that provide a wide range of authorization and management solutions. Sold as an integrated solution, these products provide an access control management solution that centralizes network and application security policy for e-business applications.

Note: IBM Tivoli Access Manager is the new name of the previously released software entitled Tivoli SecureWay® Policy Director. Also, for users familiar with the Tivoli SecureWay Policy Director software and documentation, the management server is now referred to as the policy server.

Who should read this book

The target audience for this developer reference includes:

• Security administrators
• Network system administrators
• IT architects
• Application developers

Readers should be familiar with:

• Internet protocols, including HTTP, TCP/IP, file transfer protocol (FTP), and telnet
• Development of applications in a C language environment
• Security management, including authentication and authorization

If you are enabling Secure Sockets Layer (SSL) communication, you also should be familiar with SSL protocol, key exchange (public and private), digital signatures, cryptographic algorithms, and certificate authorities.

What this book contains

This document contains the following chapters:

• Chapter 1, "Introducing the Tivoli Access Manager authorization API", on page 1
  Describes the Tivoli Access Manager implementation of the Open Group standard Authorization C API.
• Chapter 2, “Authorization API functions and data types”, on page 7
  Describes the functions and data types that are provided in the authorization API.

• Chapter 3, “Initializing the authorization API”, on page 17
  Describes how to initialize the authorization API, using either configuration file entries or programmatic structures.

• Chapter 4, “Using the authorization API”, on page 35
  Describes how to perform the main authorization API tasks, such as verifying user identity, obtaining authorization credentials, and obtaining an access decision.

• Chapter 5, “Backwards compatibility and application migration”, on page 49
  Describes backwards compatibility support, and lists deprecated interfaces.

• Chapter 6, “Introducing authorization service plug-ins”, on page 53
  Describes the Authorization Service Plug-in model, provides an overview of how to implement service plug-ins, and describes the supplied implementations.

• Chapter 7, “Implementing entitlements service plug-ins”, on page 75
  Describes how to implement and configure entitlements service plug-ins.

• Chapter 8, “Implementing administration service plug-ins”, on page 81
  Describes how to implement and configure administration service plug-ins.

• Chapter 9, “Implementing external authorization service plug-ins”, on page 89
  Describes how to implement and configure external authorization service plug-ins.

• Appendix A, “Authorization API reference”, on page 101
  Provides a reference page for each authorization API function.

• Appendix B, “Authorization service plug-in API reference”, on page 169
  Provides a reference page for each service plug-in function.

• Appendix C, “Authorization API client configuration file”, on page 185
  Provides a reference to the configuration settings specified in the authorization API client configuration file.

Note: The Tivoli SecureWay Policy Director version of this book contained information on the Java implementation of the authorization API. The Java implementation is now described in a separate book, IBM Tivoli Access Manager Authorization Java Classes Developer’s Reference.

Publications

The Tivoli Access Manager library is organized into the following categories:

• “Release information”
• “Base information” on page xi
• “WebSEAL information” on page xi
• “Web security information” on page xi
• “Developer references” on page xii
• “Technical supplements” on page xii

Release information

• IBM Tivoli Access Manager Read Me First Card
  GI11-4198-00 (am41_readme.pdf)
Provides information for installing and getting started using Tivoli Access Manager.

- **IBM Tivoli Access Manager Release Notes**
  SC32-I130-00 (am41_relnotes.pdf)
  Provides late-breaking information, such as software limitations, workarounds, and documentation updates.

### Base information

- **IBM Tivoli Access Manager Base Installation Guide**
  SC32-I131-01 (am41_install.pdf)
  Explains how to install, configure, and upgrade Tivoli Access Manager software, including the Web Portal Manager interface.

- **IBM Tivoli Access Manager Base Administrator’s Guide**
  SC32-I132-01 (am41_admin.pdf)
  Describes the concepts and procedures for using Tivoli Access Manager services. Provides instructions for performing tasks from the Web Portal Manager interface and by using the `pdadmin` command.

### WebSEAL information

- **IBM Tivoli Access Manager WebSEAL Installation Guide**
  SC32-I133-01 (amweb41_install.pdf)
  Provides installation, configuration, and removal instructions for the WebSEAL server and the WebSEAL application development kit.

- **IBM Tivoli Access Manager WebSEAL Administrator’s Guide**
  SC32-I134-01 (amweb41_admin.pdf)
  Provides background material, administrative procedures, and technical reference information for using WebSEAL to manage the resources of your secure Web domain.

### Web security information

- **IBM Tivoli Access Manager for WebSphere Application Server User’s Guide**
  SC32-I136-01 (amwas41_user.pdf)
  Provides installation, removal, and administration instructions for Tivoli Access Manager for IBM WebSphere® Application Server.

- **IBM Tivoli Access Manager for WebLogic Server User’s Guide**
  SC32-I137-01 (amwlts41_user.pdf)
  Provides installation, removal, and administration instructions for Tivoli Access Manager for BEA WebLogic Server.

- **IBM Tivoli Access Manager Plug-in for Edge Server User’s Guide**
  SC32-I138-01 (amedge41_user.pdf)
  Describes how to install, configure, and administer the plug-in for IBM WebSphere Edge Server application.

- **IBM Tivoli Access Manager Plug-in for Web Servers User’s Guide**
  SC32-I139-01 (amws41_user.pdf)
  Provides installation instructions, administration procedures, and technical reference information for securing your Web domain using the plug-in for Web servers.
Developer references

- **IBM Tivoli Access Manager Authorization C API Developer’s Reference**
  SC32-1140-01 (am41_authC_devref.pdf)
  Provides reference material that describes how to use the Tivoli Access Manager authorization C API and the Access Manager service plug-in interface to add Tivoli Access Manager security to applications.

- **IBM Tivoli Access Manager Authorization Java Classes Developer’s Reference**
  SC32-1141-01 (am41_authJ_devref.pdf)
  Provides reference information for using the Java™ language implementation of the authorization API to enable an application to use Tivoli Access Manager security.

- **IBM Tivoli Access Manager Administration C API Developer’s Reference**
  SC32-1142-01 (am41_adminC_devref.pdf)
  Provides reference information about using the administration API to enable an application to perform Tivoli Access Manager administration tasks. This document describes the C implementation of the administration API.

- **IBM Tivoli Access Manager Administration Java Classes Developer’s Reference**
  SC32-1143-01 (am41_adminJ_devref.pdf)
  Provides reference information for using the Java language implementation of the administration API to enable an application to perform Tivoli Access Manager administration tasks.

- **IBM Tivoli Access Manager WebSEAL Developer’s Reference**
  SC32-1135-01 (amweb41_devref.pdf)
  Provides administration and programming information for the Cross-domain Authentication Service (CDAS), the Cross-domain Mapping Framework (CDMF), and the Password Strength Module.

Technical supplements

- **IBM Tivoli Access Manager Command Reference**
  GC32-1107-01 (am41_cmdref.pdf)
  Provides information about the command line utilities and scripts provided with Tivoli Access Manager.

- **IBM Tivoli Access Manager Error Message Reference**
  SC32-1144-01 (am41_error_ref.pdf)
  Provides explanations and recommended actions for the messages produced by Tivoli Access Manager.

- **IBM Tivoli Access Manager Problem Determination Guide**
  GC32-1106-01 (am41_pdg.pdf)
  Provides problem determination information for Tivoli Access Manager.

- **IBM Tivoli Access Manager Performance Tuning Guide**
  SC32-1145-01 (am41_perftune.pdf)
  Provides performance tuning information for an environment consisting of Tivoli Access Manager with the IBM Directory server defined as the user registry.

Related publications

This section lists publications related to the Tivoli Access Manager library.

The Tivoli Software Library provides a variety of Tivoli publications such as white papers, datasheets, demonstrations, redbooks, and announcement letters. The Tivoli
Software Library is available on the Web at:

The Tivoli Software Glossary includes definitions for many of the technical terms related to Tivoli software. The Tivoli Software Glossary is available, in English only, from the Glossary link on the left side of the Tivoli Software Library Web page:

IBM Global Security Toolkit
Tivoli Access Manager provides data encryption through the use of the IBM Global Security Toolkit (GSKit). GSKit is included on the IBM Tivoli Access Manager Base CD for your particular platform.

The GSKit package installs the iKeyman key management utility, gsk5ikm, which enables you to create key databases, public-private key pairs, and certificate requests. The following document is available on the Tivoli Information Center Web site in the same section as the IBM Tivoli Access Manager product documentation:

- Secure Sockets Layer Introduction and iKeyman User’s Guide
  (gskikm5c.pdf)

  Provides information for network or system security administrators who plan to enable SSL communication in their Tivoli Access Manager environment.

IBM DB2 Universal Database
IBM DB2® Universal Database™ is required when installing IBM Directory Server, z/OS™, and OS/390® LDAP servers. DB2 is provided on the product CDs for the following operating system platforms:

- IBM AIX®
- Microsoft™ Windows™
- Sun Solaris Operating Environment

DB2 information is available at:
http://www.ibm.com/software/data/db2/

IBM Directory Server
IBM Directory Server, Version 4.1, is included on the IBM Tivoli Access Manager Base CD for all platforms except Linux for zSeries™. You can obtain the IBM Directory Server software for Linux for S/390 at:


If you plan to use IBM Directory Server as your user registry, see the information provided at:


IBM WebSphere Application Server
IBM WebSphere Application Server, Advanced Single Server Edition 4.0.3, is included on the Web Portal Manager CDs and installed with the Web Portal Manager interface. For information about IBM WebSphere Application Server, see:

IBM Tivoli Access Manager for Business Integration

IBM Tivoli Access Manager for Business Integration, available as a separately orderable product, provides a security solution for IBM MQSeries®, Version 5.2, and IBM WebSphere® MQ for Version 5.3 messages. IBM Tivoli Access Manager for Business Integration allows WebSphere MQSeries applications to send data with privacy and integrity by using keys associated with sending and receiving applications. Like WebSEAL and IBM Tivoli Access Manager for Operating Systems, IBM Tivoli Access Manager for Business Integration, is one of the resource managers that use the authorization services of IBM Tivoli Access Manager for e-business.

The following documents associated with IBM Tivoli Access Manager for Business Integration Version 4.1 are available on the Tivoli Information Center Web site:

- IBM Tivoli Access Manager for Business Integration Administrator's Guide (SC23-4831-00)
- IBM Tivoli Access Manager for Business Integration Release Notes (GI11-0957-00)
- IBM Tivoli Access Manager for Business Integration Read Me First (GI11-0958-00)

IBM Tivoli Access Manager for Operating Systems

IBM Tivoli Access Manager for Operating Systems, available as a separately orderable product, provides a layer of authorization policy enforcement on UNIX systems in addition to that provided by the native operating system. IBM Tivoli Access Manager for Operating Systems, like WebSEAL and IBM Tivoli Access Manager for Business Integration, is one of the resource managers that use the authorization services of IBM Tivoli Access Manager for e-business.

The following documents associated with IBM Tivoli Access Manager for Operating Systems Version 4.1 are available on the Tivoli Information Center Web site:

- IBM Tivoli Access Manager for Operating Systems Installation Guide (SC23-4829-00)
- IBM Tivoli Access Manager for Operating Systems Administration Guide (SC23-4827-00)
- IBM Tivoli Access Manager for Operating Systems Problem Determination Guide (SC23-4828-00)
- IBM Tivoli Access Manager for Operating Systems Release Notes (GI11-0951-00)
- IBM Tivoli Access Manager for Operating Systems Read Me First (GI11-0949-00)

Accessing publications online

The publications for this product are available online in Portable Document Format (PDF) or Hypertext Markup Language (HTML) format, or both in the Tivoli Software Library: [http://www.ibm.com/software/tivoli/library](http://www.ibm.com/software/tivoli/library)

To locate product publications in the library, click the Product manuals link on the left side of the Library page. Then, locate and click the name of the product on the Tivoli Software Information Center page.

Product publications include release notes, installation guides, user’s guides, administrator’s guides, and developer’s references.

Note: To ensure proper printing of PDF publications, select the Fit to page check box in the Adobe Acrobat Print window (which is available when you click File ➤ Print).
Ordering publications

You can order many IBM Tivoli publications online at:

You can also order by telephone:
• In the United States: 800-879-2755
• In Canada: 800-426-4968
• In other countries, for a list of telephone numbers, see http://www.ibm.com/software/tivoli/order-lit/

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. With this product, you can use assistive technologies to hear and navigate the interface. You also can use the keyboard instead of the mouse to operate all features of the graphical user interface.

Contacting software support

Before contacting IBM Tivoli Software support with a problem, refer to the IBM Tivoli Software support Web site at:

If you need additional help, contact software support by using the methods described in the IBM Software Support Guide at the following Web site:
http://techsupport.services.ibm.com/guides/handbook.html

The guide provides the following information:
• Registration and eligibility requirements for receiving support
• Telephone numbers and e-mail addresses, depending on the country in which you are located
• A list of information you should gather before contacting customer support

Conventions used in this book

This reference uses several conventions for special terms and actions and for operating system-dependent commands and paths.

Typeface conventions

The following typeface conventions are used in this reference:

Bold Lowercase commands or mixed case commands that are difficult to distinguish from surrounding text, keywords, parameters, options, names of Java classes, and objects are in bold.

Italic Variables, titles of publications, and special words or phrases that are emphasized are in italic.

Monospace Code examples, command lines, screen output, file and directory names that are difficult to distinguish from surrounding text, system messages, text that the user must type, and values for arguments or command options are in monospace.
Operating system differences

This book uses the UNIX convention for specifying environment variables and for directory notation. When using the Windows command line, replace $variable with %variable% for environment variables and replace each forward slash (/) with a backslash (\) in directory paths. If you are using the bash shell on a Windows system, you can use the UNIX conventions.
Chapter 1. Introducing the Tivoli Access Manager authorization API

This chapter contains the following topics:

- “Introducing the authorization API”
- “Locating the Tivoli Access Manager authorization API components” on page 3
- “Building applications with the authorization API” on page 5
- “Deploying applications with the authorization API” on page 6
- “Summarizing authorization API tasks” on page 6

Introducing the authorization API

Using the IBM Tivoli Access Manager (Tivoli Access Manager) authorization application programming interface (API), you can program Tivoli Access Manager applications and third-party applications to query the Tivoli Access Manager authorization service for authorization decisions.

The Tivoli Access Manager 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 Tivoli Access Manager 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 Tivoli Access Manager application development kit (ADK).

The Open Group Authorization API standard

The IBM Tivoli Access Manager 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.

Tivoli Access Manager implements the ADF component of this model and provides the authorization API as an interface to this function.

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

The Tivoli Access Manager 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.

The Tivoli Access Manager 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 Tivoli Access Manager security model.

Complete the following steps in order to deploy an application into an Tivoli Access Manager secure domain:

1. Configure the Tivoli Access Manager secure domain to recognize and support the objects, actions, and users that are relevant to your application.
   - For an introduction to the Tivoli Access Manager authorization model, see the [IBM Tivoli Access Manager Base Administrator’s Guide](#).
   - For complete information on access control, see the [IBM Tivoli Access Manager Base Administrator’s 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 the [IBM Tivoli Access Manager Base Administrator’s Guide](#).

3. Develop your application logic to enforce the security policy.

Locating the Tivoli Access Manager authorization API components

The authorization API is included in an optional installation package (ADK) in the Tivoli Access Manager distribution. The authorization API files are installed in several sub-directories under the Tivoli Access Manager installation directory.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>On Microsoft Windows systems, the library to include at run time is pdauthzn.dll</td>
</tr>
<tr>
<td>include</td>
<td>C header files</td>
</tr>
<tr>
<td>lib</td>
<td>A library that implements the API functions. The name of the library is platform dependent:</td>
</tr>
<tr>
<td></td>
<td><strong>Solaris Operating Environment</strong></td>
</tr>
<tr>
<td></td>
<td>11bpdauthzn.so</td>
</tr>
<tr>
<td></td>
<td><strong>AIX</strong></td>
</tr>
<tr>
<td></td>
<td>1libpdauthzn.a</td>
</tr>
<tr>
<td></td>
<td><strong>HP-UX</strong></td>
</tr>
<tr>
<td></td>
<td>1libpdauthzn.sl</td>
</tr>
<tr>
<td></td>
<td><strong>Microsoft Windows</strong></td>
</tr>
<tr>
<td></td>
<td>pdauthzn.lib</td>
</tr>
</tbody>
</table>
Directory | Contents
---|---
example | This directory contains an example program that demonstrates usage of the authorization API. Source files and a MAKEFILE are provided.

For installation instructions for the ADK, see the IBM Tivoli Access Manager Base Installation Guide.

- Header Files

The header files are found in the include directory, located directly under the Tivoli Access Manager 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>
<tr>
<td>azn_svc_protos.h</td>
<td>Prototypes for generic authorization service plug-in functions. Contains prototypes for the <code>azn_serviceInitialize()</code> and <code>azn_serviceShutdown()</code> calls. This can optionally be included by a plug-in programmer to prototype the calls defined in the service.</td>
</tr>
<tr>
<td>azn_admin_svc_protos.h</td>
<td>Prototypes for plug-in functions for the authorization administration service.</td>
</tr>
<tr>
<td>azn_deprecated.h</td>
<td>Prototypes and declarations for the functions, variables and attributes that are deprecated in this version of Tivoli Access Manager. Programmers should avoid including this header file as the symbols that are contained there will not be supported in future releases of the product.</td>
</tr>
<tr>
<td>ivadminapi.h</td>
<td>Function prototypes for the Tivoli Access Manager administration API. This API is described in IBM Tivoli Access Manager Administration C API Developer’s Guide.</td>
</tr>
<tr>
<td>pdb*msg.h</td>
<td>Minor error codes.</td>
</tr>
</tbody>
</table>

- 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>pdb*msg.h</td>
<td>Minor error codes for utility functions and the Tivoli Access Manager authorization service are found in a number of error message files, such as pdbaclmsg.h</td>
</tr>
</tbody>
</table>
Building applications with the authorization API

To develop applications that use the Tivoli Access Manager authorization API, you must install and configure a IBM Tivoli Access Manager secure domain.

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

- Tivoli Access Manager runtime environment
- Tivoli Access Manager policy server
- Tivoli Access Manager application development kit

When the Tivoli Access Manager secure domain uses an LDAP or Lotus Domino server user registry, the application development system must have an LDAP client installed.

If you already have a Tivoli Access Manager secure domain installed, and want to add a development system to the domain, the minimum Tivoli Access Manager installation consists of the following components:

- Tivoli Access Manager runtime environment
- Tivoli Access Manager application development kit

Note: For Tivoli Access Manager installation instructions refer to the IBM Tivoli Access Manager Base Installation Guide.

In order to compile applications that use the authorization API, you must install the Tivoli Access Manager ADK on the build machine.

When compiling your application, make sure you add the include directory for the Tivoli Access Manager 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.

Note: The Tivoli Access Manager authorization API is compiled as a 32-bit application.

Demonstration programs

The Tivoli Access Manager authorization API is provided with several example programs. The authzn_demo directory contains examples programs that demonstrate use of the authorization API. A C language example is included. The C example contains a sample Makefile. See the sample Makefile for build instructions specific to each supported operating system platform. Refer to the README file, located in the same directory, for information regarding the use of this example program.

An example of the administration service plug-in is provided in the admin_svc_demo directory. See the sample Makefile for build instructions.

An example of an external authorization service plug-in is provided in the eas_demo directory. See the sample Makefile for build instructions.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>authzn_demo</td>
<td>Authorization API demonstration program</td>
</tr>
</tbody>
</table>
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.

Applications that have been developed with the Tivoli Access Manager authorization API must be run on systems that are configured into a Tivoli Access Manager secure domain. When the Tivoli Access Manager secure domain uses an LDAP user registry, the application deployment system must have an LDAP client installed.

The minimum Tivoli Access Manager installation required on a system that will run an application is the Tivoli Access Manager runtime environment component.

For deployment examples, see the demonstration programs described in "Demonstration programs" on page 5.

Summarizing authorization API tasks

The primary task of the authorization API is to obtain an authorization decision from the Tivoli Access Manager authorization service.

Use the authorization API to present information about the user, operation, and requested resource to the Tivoli Access Manager authorization service, and receive the authorization decision. Your application is responsible for enforcing the decision, as appropriate.

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

- Chapter 3, “Initializing the authorization API”, on page 17
- “Authenticating an API application” on page 35
- “Verifying the identity of a user” on page 36
- “Obtaining user authorization credentials” on page 36
- “Obtaining an authorization decision” on page 39
- “Cleaning up and shutting down” on page 41

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

- “Working with credentials” on page 42
Chapter 2. Authorization API functions and data types

The IBM Tivoli Access Manager (Tivoli Access Manager) 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” on page 7
- “Character strings” on page 9
- “Buffers” on page 9
- “Protected object structures” on page 10
- “Default user registry information structure” on page 10
- “Attribute lists” on page 12
- “Credential handles” on page 14
- “Status codes and error handling” on page 14

API functions

The following tables list the authorization API functions and provide both a link to the reference page for the function and a link to the section in this document that describes each function’s task.

**Note:** The Tivoli Access Manager authorization API functions are 32-bit only.

### Attribute lists

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>“azn_attrlist_add_entry()” on page 103</td>
<td>“Attribute lists” on page 7</td>
</tr>
<tr>
<td>“azn_attrlist_add_entry_buffer()” on page 104</td>
<td></td>
</tr>
<tr>
<td>“azn_attrlist_add_entry_pobj()” on page 105</td>
<td></td>
</tr>
<tr>
<td>“azn_attrlist_add_entry_ulong()” on page 106</td>
<td></td>
</tr>
<tr>
<td>“azn_attrlist_create()” on page 108</td>
<td></td>
</tr>
<tr>
<td>“azn_attrlist_copy()” on page 107</td>
<td></td>
</tr>
<tr>
<td>“azn_attrlist_delete()” on page 109</td>
<td></td>
</tr>
<tr>
<td>“azn_attrlist_delete_entry()” on page 110</td>
<td></td>
</tr>
<tr>
<td>“azn_attrlist_get_entry_buffer_value()” on page 111</td>
<td></td>
</tr>
<tr>
<td>“azn_attrlist_get_entry_ulong_value()” on page 116</td>
<td></td>
</tr>
<tr>
<td>“azn_attrlist_get_entry_pobj_value()” on page 113</td>
<td></td>
</tr>
<tr>
<td>“azn_attrlist_get_entry_string_value()” on page 114</td>
<td></td>
</tr>
<tr>
<td>“azn_attrlist_get_names()” on page 117</td>
<td></td>
</tr>
<tr>
<td>“azn_attrlist_name_get_num()” on page 118</td>
<td></td>
</tr>
<tr>
<td>“azn_release_buffer()” on page 157</td>
<td></td>
</tr>
<tr>
<td>“azn_release_pobj()” on page 158</td>
<td></td>
</tr>
<tr>
<td>“azn_release_string()” on page 159</td>
<td></td>
</tr>
<tr>
<td>“azn_release_strings()” on page 160</td>
<td></td>
</tr>
<tr>
<td>“azn_util_handle_is_valid()” on page 163</td>
<td></td>
</tr>
</tbody>
</table>
Credentials

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;azn_creds_combine()&quot; on page 119</td>
<td>&quot;Creating a chain of credentials&quot; on page 43</td>
</tr>
<tr>
<td>&quot;azn_creds_copy()&quot; on page 121</td>
<td>&quot;Copying a credential&quot; on page 47</td>
</tr>
<tr>
<td>&quot;azn_creds_create()&quot; on page 122</td>
<td>&quot;Obtaining user authorization credentials on page 36</td>
</tr>
<tr>
<td>&quot;azn_creds_delete()&quot; on page 123</td>
<td>&quot;Releasing allocated memory&quot; on page 41</td>
</tr>
<tr>
<td>&quot;azn_creds_equal()&quot; on page 124</td>
<td>&quot;Comparing two credentials&quot; on page 47</td>
</tr>
<tr>
<td>&quot;azn_creds_for_subject()&quot; on page 125</td>
<td>&quot;Obtaining a credential from a chain of credentials&quot; on page 43</td>
</tr>
<tr>
<td>&quot;azn_creds_get_attrlist_for_subject()&quot; on page 128</td>
<td>&quot;Obtaining an attribute list from a credential&quot; on page 44</td>
</tr>
<tr>
<td>&quot;azn_creds_set_attr_value_string()&quot; on page 137</td>
<td>&quot;Setting and getting string attribute values for a credential&quot; on page 46</td>
</tr>
<tr>
<td>&quot;azn_creds_get_attr_value_string()&quot; on page 127</td>
<td></td>
</tr>
<tr>
<td>&quot;azn_creds_get_pac()&quot; on page 130</td>
<td>&quot;Converting credentials to a transportable format&quot; on page 42</td>
</tr>
<tr>
<td>&quot;azn_creds_modify()&quot; on page 132</td>
<td>&quot;Modifying the contents of a credential&quot; on page 44</td>
</tr>
<tr>
<td>&quot;azn_creds_num_of_subjects()&quot; on page 135</td>
<td>&quot;Determining the number of credentials in a credentials chain&quot; on page 43</td>
</tr>
<tr>
<td>&quot;azn_id_get_creds()&quot; on page 150</td>
<td>&quot;Obtaining user authorization credentials&quot; on page 36</td>
</tr>
<tr>
<td>&quot;azn_pac_get_creds()&quot; on page 155</td>
<td>&quot;Converting credentials to the native format&quot; on page 43</td>
</tr>
<tr>
<td>&quot;azn_util_handle_is_valid()&quot; on page 163</td>
<td>&quot;Credential handles&quot; on page 14</td>
</tr>
</tbody>
</table>

Authorization decisions

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;azn_decision_access_allowed()&quot; on page 139</td>
<td>&quot;Obtaining an authorization decision&quot; on page 39</td>
</tr>
<tr>
<td>&quot;azn_decision_access_allowed_ext()&quot; on page 141</td>
<td></td>
</tr>
</tbody>
</table>

Initialization, shutdown, and error handling

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;azn_error_major()&quot; on page 147</td>
<td>&quot;Status codes and error handling&quot; on page 14</td>
</tr>
<tr>
<td>&quot;azn_error_minor()&quot; on page 148</td>
<td></td>
</tr>
<tr>
<td>&quot;azn_error_minor_get_string()&quot; on page 149</td>
<td></td>
</tr>
<tr>
<td>&quot;azn_error_get_string()&quot; on page 146</td>
<td></td>
</tr>
<tr>
<td>&quot;azn_initialize()&quot; on page 152</td>
<td>Chapter 3, &quot;Initializing the authorization API&quot;, on page 17</td>
</tr>
</tbody>
</table>
## API extensions

<table>
<thead>
<tr>
<th>Function or Data Type</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_util_errcode()</code> on page 162</td>
<td>“Status codes and error handling” on page 14</td>
</tr>
<tr>
<td><code>azn_util_handle_is_valid()</code> on page 163</td>
<td>“Attribute lists” on page 12 “Credential handles” on page 14</td>
</tr>
<tr>
<td><code>azn_util_password_authenticate()</code> on page 164</td>
<td>“Verifying the identity of a user” on page 36</td>
</tr>
<tr>
<td><code>azn_util_password_change()</code> on page 166</td>
<td></td>
</tr>
</tbody>
</table>

### 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`.

### 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;
    unsigned char *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>

### Protected object structures

This data structure is available for applications that want to track information about protected objects.

```c
typedef struct azn_pobj_desc_struct {
    azn_string_t name;
    unsigned int type;
    azn_string_t description;
    azn_boolean_t is_policy_attachable;
} azn_pobj_desc, *azn_pobj_t
```

The variables in the structure above contain the following information:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A string containing the protected object path.</td>
</tr>
<tr>
<td>type</td>
<td>The protected object type, expressed as an unsigned int. This can be defined by the application developer.</td>
</tr>
<tr>
<td>description</td>
<td>A string description of the protected object.</td>
</tr>
<tr>
<td>is_policy_attachable</td>
<td>A boolean value that indicates whether authorization policy can be attached to this object.</td>
</tr>
</tbody>
</table>

### Default user registry information structure

The authorization API uses this structure to pass information for building Tivoli Access Manager credentials to the `azn_id_get_creds()` call.

When this structure is used in conjunction with a NULL mechanism ID in a function call to `azn_id_get_creds()`, the client credential is built from information stored in the default user registry. It will use the registry option that was selected when the Tivoli Access Manager secure domain was installed. The client does not need to know the registry type.

This structure replaces the deprecated structures `azn_authdce_t`, `azn_authldap_t`, and `azn_authuraf_t`.

```c
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 authmech_info;
    void *reserved[9];
} azn_authdefault_t;
```
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>principal</td>
<td>The client’s principal name in the registry. This value is mandatory. Example: sec_master</td>
</tr>
<tr>
<td>auth_method</td>
<td>The user registry used to authenticate this user credential. This string is optional. This field is deprecated and should be set to NULL.</td>
</tr>
<tr>
<td>ipaddr</td>
<td>A network order unsigned long IP address. For example: 0x56780012. This value is mandatory for IP authorization. Otherwise, it is optional.</td>
</tr>
<tr>
<td>qop</td>
<td>Quality of protection required for requests made by this user. Supplied for use by applications to keep track of the user’s current level of protection. This value is optional. Note: This entry is not used by the authorization engine for the purpose of returning the Quality of Protection required for access to an object. Examples: none integrity privacy</td>
</tr>
<tr>
<td>user_info</td>
<td>Additional user information that may be required for auditing. Supplied for use by the application to keep track of any additional user information it may need. This value is optional.</td>
</tr>
<tr>
<td>browser_info</td>
<td>The browser employed by the user. Supplied for use by web applications. This value is optional.</td>
</tr>
<tr>
<td>authnmech_info</td>
<td>Additional authentication mechanism information. Supplied for use by the application to keep track of any additional information on the method of authentication used for the user. Example: X.509 certificate This value is optional.</td>
</tr>
<tr>
<td>reserved</td>
<td>Fields reserved for future use.</td>
</tr>
</tbody>
</table>

**Unauthenticated user information structure**

This data structure contains information for use in building an unauthenticated authorization credential for a user within the Tivoli Access Manager secure domain.

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipaddr</td>
<td>IP address of the requesting user in network byte order.</td>
</tr>
<tr>
<td>qop</td>
<td>Quality of protection required for requests made by this user.</td>
</tr>
<tr>
<td>user_info</td>
<td>Additional user information that may be required for auditing.</td>
</tr>
<tr>
<td>browser_info</td>
<td>The browser employed by the user. This field is optional.</td>
</tr>
</tbody>
</table>

**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.

**CAUTION:**
Always initialize attribute list handles to AZN_C_INVALID_HANDLE when declaring them on the stack. If an application attempts to access an uninitialized attribute list handle, the access request may result in memory corruption or undefined behavior.

Many authorization API functions use 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 values under each attribute name are stored in the order in which they were inserted into the list. Each value entry can be retrieved using the azn_attrlist_get_entry_*() functions with an index number starting at 0 for the first value in the list. There is no checking performed upon values in the list so duplicate values are permitted.

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>
</table>
| Create an attribute list | Use `azn_attrlist_create()` to complete the following tasks:  
  • Allocate a new, empty attribute list.  
  • Associate a handle with the attribute list.  
  • Return the handle. |
| Set an entry in an attribute list | Use `azn_attrlist_add_entry()` to add a string name-value pair of type `azn_string_t`.  
Use `azn_attrlist_add_entry_buffer()` to add a buffer name-value pair of type `azn_buffer_t`.  
Use `azn_attrlist_add_entry_ulong()` to add a name-value pair of type `azn_ulong_t`.  
Use `azn_attrlist_add_entry_pobj()` to add a name-value pair of type `azn_pobj_t`. |
| Delete an entry from an attribute list. | Use `azn_attrlist_delete_entry()` to delete all the values that are assigned to an attribute in an attribute list. |
| Get attribute names from an attribute list | Use `azn_attrlist_get_names()` to get all the names in an attribute list. The names are returned in a NULL-terminated array of strings of type `azn_string_t`. |
| Get the number of values for a specified attribute name | Use `azn_attrlist_name_get_num()` to get the number, as an integer, of the value attributes for a specified name in the attribute list. |
| Copy an attribute list. | Use `azn_attrlist_copy()` to make a copy of an attribute list. |
| Get a value | Use `azn_attrlist_get_entry_string_value()` to get the value attribute of a string of type `azn_string_t` for a specified name in an attribute list.  
Use `azn_attrlist_get_entry_buffer_value()` to get the value attribute of a buffer of type `azn_buffer_t` for a specified name in an attribute list.  
Use `azn_attrlist_get_entry_ulong_value()` to get the value attribute of type `azn_ulong_t`.  
Use `azn_attrlist_get_entry_pobj_value()` to get the value attribute of type `azn_pobj_t`.  
For each of the functions described above, the specified attribute list entry can have multiple values. For a multi-valued attribute, specify an index to get that instance of the value. For a single-valued attribute, the index should be set to 0. |
| Delete an attribute list | Use `azn_attrlist_delete()` to delete the attribute list associated with a specified attribute list handle. |
| Determine if the attribute list handle is valid | Use `azn_util_handle_is_valid()` to determine if an attribute list handle is associated with valid data. |
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.

By default, the credentials generated by the `azn_id_get_creds()` interface and Tivoli Access Manager components contain only a single Tivoli Access Manager credential. The intermediary credentials are not referenced by Tivoli Access Manager when making an authorization decision. Only the primary credential, referred to as the “initiator” is used in authorization decisions. Chains can be used for customer API applications but an External authorization service would need to be developed to authorize the remaining credentials in a chain.

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 Tivoli Access Manager security framework.

**CAUTION:**
Always initialize credential handles to AZN_C_INVALID_HANDLE when declaring them on the stack. If an application attempts to access an uninitialized credential handle, the access request may result in memory corruption or undefined behavior.

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()`.

For more information on functions that use credentials handles to access credential information, see [“Working with credentials” on page 42](#).

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 Tivoli Access Manager authorization server.

Use `azn_error_minor_get_string()` to obtain string values for the minor error codes returned by `azn_error_minor()`.

Use `azn_error_get_string()` to return the Tivoli Access Manager serviceability message string from a authorization API status structure of type `azn_status_t`. This function will automatically select the major or minor code and map the error value to an error message string.

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>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>pdb*msg.h</td>
<td>Minor error codes for utility functions and the Tivoli Access Manager authorization service are found in a number of files. The filenames for these files all share the prefix <code>pdb</code> and the suffix <code>msg.h</code></td>
</tr>
</tbody>
</table>
Chapter 3. Initializing the authorization API

To use the IBM Tivoli Access Manager (Tivoli Access Manager) 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 \texttt{azn\_initialize()}
- Specify entries in the authorization API configuration file

The authorization API initialization function \texttt{azn\_initialize()} takes as an input parameter an attribute list named \texttt{init\_data}. To specify initialization data, you must add the necessary attributes to \texttt{init\_data}.

Each input argument to \texttt{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 \texttt{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 \texttt{azn\_initialize()}.

Complete the instructions in the following sections:

- “Specifying an authorization API configuration file” on page 17
- “Specifying cache mode settings” on page 18
- “Configuring SSL from the API client to Tivoli Access Manager” on page 20
- “Specifying communications attributes for the policy server” on page 24
- “Specifying values for an authorization server replica” on page 26
- “Configuring the authorization API for LDAP access” on page 26
- “Configuring LDAP access over SSL” on page 28
- “Configuring advanced LDAP parameters” on page 29
- “Specifying LDAP user registry replica access” on page 30
- “Enabling the return of permission information” on page 30
- “Starting the authorization service” on page 33

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 \texttt{name = value} pairs. Each of the pairs corresponds to an input parameter that can get passed to \texttt{azn\_initialize()}

If no configuration file is specified, \texttt{azn\_initialize()} obtains initialization parameters only from the attribute list contained in the \texttt{init\_data} input parameter. There is no configuration file specified by default.
Specify a configuration file by using the `azn_init_cfg_file` attribute.

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, as described in the table below.

<table>
<thead>
<tr>
<th><code>azn_init_cfg_file</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;filename&gt;</code></td>
<td>A configuration file containing initialization values for the Tivoli Access Manager authorization API. There is no default value. The sample configuration file distributed with the Authorization demonstration program is named <code>aznapi.conf</code>.</td>
</tr>
</tbody>
</table>

### Specifying cache mode settings

The cache mode determines if the authorization API talks to a Tivoli Access Manager authorization service 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.
- When using local mode, the caller of the authorization API must be a member of the `ivacld-servers` group.

**Note:** For more information on remote cache mode or local cache mode, see the [IBM Tivoli Access Manager Base Administrator's Guide](#).

The `svrsslcfg` utility creates a user identity 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 [IBM Tivoli Access Manager Command Reference](#).

### Specifying cache mode type

You can specify the cache mode by using either an attribute list entry or a configuration file entry:

- Attribute List Entry: `azn_init_mode`
- Configuration File Entry: `mode`
- Configuration File Stanza: `[aznapi-configuration]`
- Modified by `svrsslcfg`? Yes. Must be specified.

Use the option `-s [local | remote]`. 

The following table displays the valid values for cache modes.

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>The Tivoli Access Manager authorization service runs in the same server process as the application using the authorization API.</td>
</tr>
<tr>
<td>remote</td>
<td>The Tivoli Access Manager authorization service runs as a different server process from the application using the authorization API.</td>
</tr>
</tbody>
</table>

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. You can use this configuration to only receive updates when the `pdadmin server replicate` command is issued.

Note that in order for the `pdadmin server replicate` command to function correctly, a valid non-zero listening port (`azn_ssl_listening_port`) must be set.

### Authorization database file location

You can specify the location of the database file used by the authorization service by either using attribute list entries or by using configuration file entries.

- Attribute List entry: `azn_init_db_file`
- Configuration File Entry: `db-file`
- Configuration File Stanza: `[aznapi-configuration]`
- Modified by `svrsslcfg`?: No.

<table>
<thead>
<tr>
<th>Authorization Database File Location</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>filename</td>
<td>Path name to the persistent authorization policy database replica.</td>
</tr>
</tbody>
</table>

### Local cache refresh

You can specify the interval for the local cache refresh by either using attribute list entries or by using configuration file entries.

- Attribute List entry: `azn_init_cache_refresh_interval`
- Configuration File entry: `cache-refresh-interval`
- Configuration File Stanza: `[aznapi-configuration]`
- Modified by `svrsslcfg`?: No.

The following table shows that you can disable or enable the cache refresh. When you enable the cache refresh, you can specify the cache refresh interval in number of seconds.
<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disable</td>
<td>Refreshing of the local authorization policy database disabled.</td>
</tr>
<tr>
<td>default</td>
<td>disabled</td>
</tr>
</tbody>
</table>

| number of seconds | Number of seconds between refreshes of the local authorization policy database. Set appropriate values to ensure that the replicated database is updated in a timely manner to reflect changes made to the master database. |

**Local cache notification listener**

You can configure the notification listener by either using attribute list entries or by using configuration file entries.

- Attribute List entry: `azn_init_listen_flags`
- Configuration File entry: `listen-flags`
- Configuration File Stanza: `[aznapi-configuration]`
- Modified by `svrsslcfg`?: Yes.

Use the option `-l [yes | no ]`. If not specified the default is disable.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disable</td>
<td>Disable the notification listener. This is the default.</td>
</tr>
<tr>
<td>enable</td>
<td>Enable the notification listener.</td>
</tr>
</tbody>
</table>

**SSL listener ports**

This port is also used to listen for administration requests from the policy server. It is mandatory to specify this port when running `svrsslcfg`.

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

You can specify the notification listener port by using a configuration file entry.

- Attribute List entry: `azn_init_ssl_listening_port`
- Configuration File entry: `ssl-listening-port`
- Configuration File stanza: `[ssl]`
- Modified by `svrsslcfg`?: Yes.

Use the option `-r [ <port-number> ]`. This must be specified.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<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 is has changed. All communications on this port are SSL encrypted. The value should be non-zero and not used by any other service on the computer.</td>
</tr>
</tbody>
</table>

**Configuring SSL from the API client to Tivoli Access Manager**

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 Tivoli Access Manager authorization server and Tivoli Access Manager policy server.
Specifying an SSL keyfile

- Attribute List Entry: `azn_init_ssl_keyfile`
- Configuration File Entry: `ssl-keyfile`
- Configuration File Stanza: `[ssl]`
- Modified by `svrsslcfg`?: Yes. Must be specified.

This uses the `svrsslcfg` options `-d [kdb-directory]` and the root from `-n [server-name]` to create an entry value: `<kdb-directory>/<server-name>.kdb`.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;keyfile-path&gt;</code></td>
<td>This is the keyfile used to communicate with the Tivoli Access Manager policy server and the Tivoli Access Manager authorization server. It is created by the <code>svrsslcfg</code> utility. This can be any relative or fully qualified filename.</td>
</tr>
</tbody>
</table>

Specifying a stash file

- Attribute List Entry: `azn_init_ssl_stashfile`
- Configuration File Entry: `ssl-keyfile-stash`
- Configuration File Stanza: `[ssl]`
- Modified by `svrsslcfg`?: Yes. Must be specified.

This uses the `svrsslcfg` options `-d [kdb-directory]` and the root from `-n [server-name]` to create an entry value: `<kdb-directory>/<server-name>.sth`.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;keyfile-path&gt;</code></td>
<td>This the stash file for the keyfile. It is created by the <code>svrsslcfg</code> utility. It is used as an obfuscated password to the keyfile. This file should be appropriately secured. This can be any relative or fully qualified filename.</td>
</tr>
</tbody>
</table>

Specifying a keyfile label

- Attribute List Entry: `azn_init_ssl_keyfile_label`
- Configuration File Entry: `ssl-keyfile-label`
- Configuration File Stanza: `{ssl}`
- Modified by `svrsslcfg`?: No.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<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 <code>svrsslcfg</code> utility and contain multiple certificates.</td>
</tr>
</tbody>
</table>

SSL session timeout

- Attribute List Entry: `azn_init_ssl_timeout`
- Configuration File Entry: `ssl-v3-timeout`
- Configuration File Stanza: `[ssl]`
- Modified by `svrsslcfg`?: Yes.

Use the option `-t [ssl-timeout]`. If not specified, the default is 7200 seconds.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any non-negative integer. Default value is 7200 seconds</td>
<td>This is the amount of time before an SSL session will expire. The Tivoli Access Manager 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 Tivoli Access Manager policy server is calling the application). When the application is calling the Tivoli Access Manager policy server or the Tivoli Access Manager authorization server, the session timeout value is dictated by the most restrictive of the values for that client and server.</td>
</tr>
</tbody>
</table>

### SSL password expiration

- Attribute List Entry: `azn_init_ssl_pwd_life`
- Configuration File Entry: `ssl-pwd-life`
- Configuration File Stanza: `[ssl]`
- Modified by `svrsslcfg`: Yes.

Use the option `-e [password-life]`. If not specified, the default is 183 days.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any non-negative integer. Default value is 183 days.</td>
<td>This is the amount of time before the password or stash file to the keyfile will expire. The Tivoli Access Manager authorization API client automatically refreshes the password or stash file before this expiry time, if automatic refresh is enabled.</td>
</tr>
</tbody>
</table>

### Authentication method

- Attribute List Entry: `azn_init_ssl_authn_type`
- Configuration File Entry: `ssl-authn-type`
- Configuration File Stanza: `[ssl]`
- Modified by `svrsslcfg`: Yes.

This is always set by `svrsslcfg` to `certificate`.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The type of authentication. Possible values are: certificate, password, none. The default value is none.</td>
<td>The method that the Tivoli Access Manager policy server will use to authenticate the authorization API client. The <code>svrsslcfg</code> utility automatically sets this to certificate. If the value is <code>certificate</code> the Tivoli Access Manager policy server will map the certificate provided by the authorization API client into an identity and authenticate against it. Note that even if <code>password</code> or <code>none</code> are specified, the client will still use SSL to communicate with the server, and as such will still require a keyring database file that has the Tivoli Access Manager Certificate Authority (CA) certificate as a signer certificate or trusted certificate. There are currently no operations that can be performed by the API successfully with an authentication type of <code>none</code>.</td>
</tr>
</tbody>
</table>
User name and password

- Attribute List Entry: `azn_init_ssl_authn_user`
- Attribute List Entry: `azn_init_ssl_authn_pwd`
- Configuration File Entry: `ssl-authn-user`
- Configuration File Entry: `ssl-authn-password`
- Configuration File Stanza: `[ss1]`
- Modified by `svrsslcfg`?: No.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any string.</td>
<td>The user name and password that are used if the authentication type is <code>password</code>. It may be unwise to store these in the configuration file, however they can be useful for testing communications.</td>
</tr>
</tbody>
</table>

Tivoli Access Manager configuration file location

- Attribute List Entry: `azn_init_ssl_mgr_config`
- Configuration File Entry: `ssl-mgr-config`
- Configuration File Stanza: `[ss1]`

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The relative or fully qualified pathname to the pd.conf file.</td>
<td>This entry is used to point to the configuration file that was created as part of configuring the Tivoli Access Manager runtime environment (pd.conf). If it is specified, the values for <code>master-host</code>, <code>master-port</code>, and <code>master-dn</code> will come from the manager stanza of pd.conf and override any values specified in the authorization API client’s configuration file. Furthermore, if entries or values are not found in pd.conf for any of these entries, empty values will be used. The pd.conf usually lives in the Tivoli Access Manager installation directory, under <code>./lib/pd.conf</code></td>
</tr>
</tbody>
</table>

Password for the SSL keyfile

- Attribute List Entry: `azn_init_ssl_keyfile_pwd`
- Configuration File Entry: `ssl-keyfile-pwd`
- Configuration File Stanza: `[ss1]`
- Modified by `svrsslcfg`?: No.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;password string&gt;</code></td>
<td>Password used to protect keys in the keyfile. When both <code>ssl-keyfile-pwd</code> and <code>ssl-keyfile-stash</code> are specified, the value in <code>ssl-keyfile-pwd</code> is used.</td>
</tr>
</tbody>
</table>

Maximum number of worker threads

- Attribute List Entry: `azn_init_ssl_max_worker_threads`
- Configuration File Entry: `ssl-maximum-worker-threads`
- Configuration File Stanza: `[ss1]`
- Modified by `svrsslcfg`?: No.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;maximum number of worker threads&gt;</code></td>
<td>Maximum number of worker threads that are created by the server to handle incoming requests. The value is an integer. The minimum number is 1. The maximum number is determined by the amount of system resources available. The default number is 50.</td>
</tr>
</tbody>
</table>

**Automatic refresh of SSL certificate and keyfile password**

- Attribute List Entry: `azn_init_ssl_auto_refresh`
- Configuration File Entry: `ssl-auto-refresh`
- Configuration File Stanza: `[ssl]`
- Modified by `svrsslcfg`?: Yes.
  
  Use the option `-a [ yes | no ]`. If not specified the default is yes.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes or no</td>
<td>Enables or disables automatic refresh of the SSL certificate and key database file password. When enabled, the certificate is renewed when it is close to expiration. This means that the same public key gets a renewed signature from the certificate authority, which results in an extended lifetime for the certificate. A value of yes enables automatic refresh. A value of no disables automatic refresh.</td>
</tr>
</tbody>
</table>

**Connection timeout**

- Attribute List Entry: `azn_init_ssl_io_inactivity_timeout`
- Configuration File Entry: `ssl-io-inactivity-timeout`
- Configuration File Stanza: `[ssl]`
- Modified by `svrsslcfg`?: No.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;number of seconds&gt;</code></td>
<td>Inactivity timeout for the input/output connection, expressed in seconds. The value is an integer. The minimum value is 0. When the value is 0, no timeout is enforced. The default value is 0 seconds. The recommended value is 90 seconds. Note that this value is set in the supplied aznapi.conf file.</td>
</tr>
</tbody>
</table>

**Specifying communications attributes for the policy server**

Use the attributes described in this section to specify the location, port number, and distinguished name of the Tivoli Access Manager policy server. Authorization API Clients use this information to communicate with the policy server.

**Policy server hostname**

- Attribute List Entry: `azn_init_master_host`
- Configuration File Entry: `master-host`
Configuration File Stanza: [manager]

Modified by svrsslcfg?: Yes.

The value is taken from the pd.conf file. The pd.conf file is created when the Tivoli Access Manager runtime component is configured on the machine.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;policy server hostname&gt;</td>
<td>Specifies the hostname of the Tivoli Access Manager policy server.</td>
</tr>
<tr>
<td></td>
<td>This entry and stanza can be in either the authorization API client's configuration file or the file pointed to by the azn_init_ssl_mgr_config attribute (if specified). If the azn_init_ssl_mgr_config attribute is specified, its value overrides that in aznAPI.conf.</td>
</tr>
</tbody>
</table>

Policy server port number

- Attribute List Entry: azn_init_master_port
- Configuration File Entry: master-port
- Configuration File Stanza: [manager]
- Modified by svrsslcfg?: Yes.

The value is taken from the pd.conf file. The pd.conf file is created when the Tivoli Access Manager runtime component is configured on the machine.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;port number&gt;</td>
<td>This entry and stanza can be in either the authorization API client's configuration file or the file pointed to by the azn_init_ssl_mgr_config attribute (if specified). If the azn_init_ssl_mgr_config attribute is specified, its value overrides that in the authorization API client's configuration file.</td>
</tr>
</tbody>
</table>

Policy server distinguished name

- Attribute List Entry: azn_init_master_dn
- Configuration File Entry: master-dn
- Configuration File Stanza: [manager]
- Modified by svrsslcfg?: Yes.

The value is taken from the pd.conf file. The pd.conf file is created when the Tivoli Access Manager runtime component is configured on the machine.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;distinguished name&gt;</td>
<td>Specifies the Distinguished Name (DN) of the Tivoli Access Manager policy server. The authorization API client can match this value against that provided by the Tivoli Access Manager policy server at runtime to prevent spoofing. If the value is empty, no checking will be performed.</td>
</tr>
<tr>
<td></td>
<td>This entry and stanza can be in either the authorization API client's configuration file or the file pointed to by the azn_init_ssl_mgr_config attribute (if specified). If the azn_init_ssl_mgr_config attribute is specified, its value overrides that in the authorization API client's configuration file.</td>
</tr>
</tbody>
</table>
Specifying values for an authorization server replica

You can specify a series of values that describe the location and communication values for an authorization server replica. All values are assigned to one configuration file entry.

Each configuration file entry describes one Tivoli Access Manager authorization server. You can add one or more entries for each authorization server to the configuration file.

The replicas are of the format:
<replica hostname> :<port> :<preference> :<replica cert dn>

For example:
"rweber.bball.com:7137:5:cn=ivacld/rweber,o=Access Manager,C=US"

Note that the separator for the fields is a colon (":") and not a comma (",") like the LDAP replicas use.

- Attribute List Entry: azn_init_replica
- Configuration File Entry: replica
- Configuration File Stanza: [manager]
- Modified by svrsslcfg: Can be.

Replicas can be added to the configuration file by using the svrsslcfg -add_replica option.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;replica hostname&gt;</td>
<td>The fully qualified domain name of the server.</td>
</tr>
<tr>
<td>&lt;port&gt;</td>
<td>The port on the server.</td>
</tr>
<tr>
<td>&lt;preference&gt;</td>
<td>A ranking for attempting contact. Valid values are from 1 to 10. The lowest preference is 1, the highest preference is 10.</td>
</tr>
<tr>
<td>&lt;replica certificate distinguished name&gt;</td>
<td>The LDAP distinguished name for the authorization server.</td>
</tr>
</tbody>
</table>

Configuring the authorization API for LDAP access

When your application runs in a Tivoli Access Manager 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 Tivoli Access Manager was installed on the local system.

LDAP user registry support

- Attribute List Entry: None.
- Configuration File Entry: enable
- Configuration File Stanza: [ldap]
- Modified by svrsslcfg? Yes.

The value is taken from the pd.conf file. The pd.conf file is created when the Tivoli Access Manager runtime component is configured on the machine.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<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. When <code>azn_init_ldap_port</code> is null, LDAP access is automatically disabled.</td>
</tr>
</tbody>
</table>

**LDAP server host name**

- Attribute List Entry: `azn_init_ldap_host`
- Configuration File Entry: `host`
- Configuration File Stanza: `[ldap]`
- Modified by `svrsslcfg`? Yes.

The value is taken from the pd.conf file. The pd.conf file is created when the Tivoli Access Manager runtime component is configured on the machine.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host name</td>
<td>Host name of LDAP server.</td>
</tr>
</tbody>
</table>

**LDAP server port number**

- Attribute List Entry: `azn_init_ldap_port`
- Configuration File Entry: `port`
- Configuration File Stanza: `[ldap]`
- Modified by `svrsslcfg`? Yes.

The value is taken from the pd.conf file. The pd.conf file is created when the Tivoli Access Manager runtime component is configured on the machine.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>port number</td>
<td>Port number for communicating with the LDAP server.</td>
</tr>
</tbody>
</table>

**LDAP User Distinguished Name**

- Attribute List Entry: `azn_init_ldap_bind_dn`
- Configuration File Entry: `bind-dn`
- Configuration File Stanza: `ldap`
- Modified by `svrsslcfg`? Yes.

The value is created based on the server name that was specified with the `-n server_name` flag and the local host of the machine.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAP DN</td>
<td>Distinguished Name of the LDAP user. Created by the svrsslcfg utility.</td>
</tr>
</tbody>
</table>

**LDAP User Password**

- Attribute List Entry: `azn_init_ldap_bind_pwd`
- Configuration File Entry: `bind-pwd`
- Configuration File Stanza: `ldap`
- Modified by `svrsslcfg`? Yes.
The value is created based on the password that was specified with the -S password flag.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>password</td>
<td>Password for the LDAP user. Created by the svrsslcfg utility.</td>
</tr>
</tbody>
</table>

**Configuring LDAP access over SSL**

If the communication between the Tivoli Access Manager Authorization server and the LDAP server is over Secure Sockets Layer (SSL), you must specify the communication settings.

Note that the Tivoli Access Manager authorization API client must use two key files: one for communicating with the LDAP server and one for communicating with the Tivoli Access Manager servers.

**SSL communication with the LDAP server**

- Attribute List Entry: None.
- Configuration File Entry: ssl-enabled
- Configuration File Stanza: [1dap]

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<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. When azn_init_ldap_ssl_keyfile is null, SSL is not automatically configured</td>
</tr>
</tbody>
</table>

**SSL keyfile name**

- Attribute List Entry: azn_init_ldap_ssl_keyfile
- Configuration File Entry: ssl-keyfile
- Configuration File Stanza: [1dap]

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;filename&gt;</td>
<td>Name of the SSL key file.</td>
</tr>
</tbody>
</table>

**SSL keyfile distinguished name**

- Attribute List Entry: azn_init_ldap_ssl_keyfile_dn
- Configuration File Entry: ssl-keyfile-dn
- Configuration File Stanza: [1dap]

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyLabel</td>
<td>Key label to identify the client certificate that is presented to the LDAP server.</td>
</tr>
</tbody>
</table>

**SSL keyfile password**

- Attribute List Entry: azn_init_ldap_ssl_keyfile_pwd
- Configuration File Entry: ssl-keyfile-pwd
• Configuration File Stanza: [ldap]

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>password</td>
<td>Password to access the SSL key file. When the keyfile is specified but the password is not specified or is set to NULL, this indicates that there is no password required for access to the SSL key file. When the keyfile is specified, and the password is a non-NULL string, the password is used by the LDAP client to access the SSL key file.</td>
</tr>
</tbody>
</table>

## Configuring advanced LDAP parameters

### Maximum search buffer size

- Attribute List Entry: `azn_init_ldap_max_search_size`
- Configuration File Entry: `max-search-size`
- Configuration File Stanza: [ldap]

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;search-size&gt;</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>
</tr>
</tbody>
</table>

## Caching LDAP data

- Attribute List Entry: `azn_init_ldap_cache`
- Configuration File Entry: `cache-enabled`
- Configuration File Stanza: [ldap]

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<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>false</td>
<td>Optional. Disable LDAP client-side caching. This is the default value.</td>
</tr>
</tbody>
</table>

## LDAP server query preference

- Attribute List Entry: `azn_init_prefer_rw_svr`
- Configuration File Entry: `prefer-readwrite-server`
- Configuration File Stanza: [ldap]

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<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>false</td>
<td>Optional. Do not query read/write LDAP server first</td>
</tr>
</tbody>
</table>

## Authentication method

- Attribute List Entry: `azn_init_ldap_using_compare`
- Configuration File Entry: `auth-using-compare`
• Configuration File Stanza: [ldap]

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<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 Tivoli Access Manager authorization API call and azn_util_password_authenticate().</td>
</tr>
<tr>
<td>false</td>
<td>Optional. Use ldap_bind().</td>
</tr>
</tbody>
</table>

**Specifying LDAP user registry replica access**

You can add an attribute or configuration file entry that defines the LDAP user registry replicas in the domain.

- Attribute List Entry: `azn_init_ldap_replica`
- Configuration File Entry: `replica`
- Configuration File Stanza: [ldap]

Assign multiple values to `replica` by entering a list consisting of entries that are separated by commas. For example:

```
replica = barney,391,readwrite,2
replica = fred,391,readonly,3
```

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

**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 attributes list contains more detailed information on the result or reasoning behind the access decision that was made. It may also be used to return additional information that is applicable to the authorization decision. For example, this could include the quality of protection (QOP) that is required for communication between client and server entities.

You can specify the information returned in the `permission_info` attribute list by adding values to the `azn_init_set_perminfo_attrs` attribute during initialization of the authorization API. The `azn_init_set_perminfo_attrs` attribute is set in the `init_data` attribute list.

The `init_data` attribute list is passed as an input parameter to the authorization API initialization function, `azn_initialize()`. You can add multiple values to `azn_init_set_perminfo_attrs`. The defined values are listed in the table below.

To enable the return of permission information you must supply values for one of the following:
The default configuration file for the authorization server is ivacld.conf. This configuration file has the permission-info-returned parameter commented out (disabled) by default:

```
# permission-info-returned = azn_perminfo_qop azn_perminfo_qop_ulong
#
```

Uncomment the line in order to enable the return of permission info.

The following table contains the values that you can specify.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_perminfo_all_attrs</td>
<td>Include all of the permission information attributes</td>
</tr>
<tr>
<td>azn_perminfo_wm_ulong</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>azn_perminfo_wm_permitted_ulong</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>azn_perminfo_al_ulong</td>
<td>Auditing events that are performed for this authorization check.</td>
</tr>
<tr>
<td>&lt;pop-extended-attribute-name&gt;</td>
<td>Any attribute of a protected object policy (pop) that is used when making an authorization decision.</td>
</tr>
<tr>
<td>&lt;acl-extended-attribute-name&gt;</td>
<td>Any extended attribute of an access control list (acl) that is used when making an authorization decision.</td>
</tr>
<tr>
<td>azn_acl_ext_attr_loc</td>
<td>The value of this attribute is the location within the object space to which an ACL with extended attributes is attached. The extended attributes must be defined in the &lt;acl-extended-attribute-name&gt; value above.</td>
</tr>
<tr>
<td>azn_pop_ext_attr_loc</td>
<td>The value of this attribute is the location within the object space to which a POP with extended attributes is attached. The extended attributes must be defined in the &lt;pop-extended-attribute-name&gt; value above.</td>
</tr>
<tr>
<td>azn_perminfo_qop</td>
<td>The quality of protection level.</td>
</tr>
</tbody>
</table>

This is a string version of the value in azn_perminfo_qop_ulong. Valid values are integrity or privacy. There is no string equivalent for the quality of protection setting of none. The status of no quality of protection is indicated by the absence of this attribute.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_perminfo_qop_ulong</td>
<td>The quality of protection level.</td>
</tr>
<tr>
<td></td>
<td>The values are:</td>
</tr>
<tr>
<td></td>
<td>1. No quality of protection</td>
</tr>
<tr>
<td></td>
<td>2. Integrity</td>
</tr>
<tr>
<td></td>
<td>3. Privacy</td>
</tr>
<tr>
<td>azn_perminfo_stepup_level</td>
<td>The level required for step-up authentication. When this value is not specified, no step-up authentication is required.</td>
</tr>
</tbody>
</table>

### Configuring auditing

#### Audit file location

You can specify the location of the audit file used by the authorization service by either using attribute list entries or by using configuration file entries.

- Attribute List entry: azn_init_audit_file
- Configuration File Entry: auditlog
  
  This entry is located in the [aznapi-configuration] stanza.

<table>
<thead>
<tr>
<th>Audit File</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;filename&gt;</td>
<td>Path and file name for the file that collects authorization API audit events.</td>
</tr>
</tbody>
</table>

### Configuring audit logging

- Attribute List Entry: azn_init_auditcfg
- Configuration File Entry: auditcfg
- Configuration File Stanza: [aznapi-configuration]

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn or authn</td>
<td>To disable or enable component specific audit records, you can add or remove either authentication (authn) or authorization (azn).</td>
</tr>
</tbody>
</table>

### Configuring logging

#### Log size

- Attribute List Entry: azn_init_log_size
- Configuration File Entry: logsize
- Configuration File Stanza: [aznapi-configuration]
Log flush interval

- Attribute List Entry: `azn_init_log_flush_interval`
- Configuration File Entry: `logflush`
- Configuration File Stanza: `[aznapi-configuration]`

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;flush interval in seconds&gt;</code></td>
<td>The auditing and logging service flush interval in seconds. A value of 0 sets the default flush interval of 20 seconds. The maximum value is 600 seconds.</td>
</tr>
</tbody>
</table>

Specifying the host interface on which to listen

Machines that have more than one network interface card installed and configured can have more than one hostname. When more than one network interface card is configured, use this authorization API initialization attribute to specify the hostname on which the authorization API application listens.

When this attribute is not specified, the default hostname is used.

- Attribute List Entry: `azn_app_host`
- Configuration File Entry: `azn-app-host`
- Configuration File Stanza: `[aznapi-configuration]`

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;hostname&gt;</code></td>
<td>The hostname on which the application is running. This value can be used to specify a hostname on machines that support multiple hostnames (by using multiple network interface cards). When this is not specified, the default hostname is used. <strong>Note:</strong> This cannot be an IP address.</td>
</tr>
</tbody>
</table>

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_S_COMPLETE)
    return(status);
```
In the example code above, \texttt{azn\_initialize()} returns the attribute list \texttt{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.

\textbf{Note:} To re-initialize the API, use \texttt{azn\_shutdown()} and then call \texttt{azn\_initialize()}.

When using this function on Windows \texttt{NT}, do not call it from \texttt{dllMain()}. For more information, see the reference page for "\texttt{azn\_initialize()}" on page 152.
Chapter 4. Using the authorization API

This chapter contains the following topics:

- “Authenticating an API application”
- “Verifying the identity of a user” on page 36
- “Obtaining user authorization credentials” on page 36
- “Obtaining an authorization decision” on page 39
- “Cleaning up and shutting down” on page 41
- “Working with credentials” on page 42

Authenticating an API application

The API application must establish its own authenticated identity within the IBM Tivoli Access Manager (Tivoli Access Manager) secure domain, in order to request authorization decisions from the Tivoli Access Manager authorization service.

Before you run the authorization API application for the first time, you must create a unique identity for the application in the Tivoli Access Manager 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.

The svrsslcfg utility creates a user identity for the application, and configures the SSL communication between the application and the Tivoli Access Manager policy server.

The svrsslcfg utility performs the following tasks:

- Creates a user identity for the application by combining the server name with the local TCP/IP host name.
- Creates an SSL key file for that user: For example, demo_user.kdb 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.

For more information, including the svrsslcfg syntax, see the IBM Tivoli Access Manager Command Reference.
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 Tivoli Access Manager 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`. This buffer 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()` on page 164.

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

Obtaining user authorization credentials

In order to submit an authorization request to the Tivoli Access Manager 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 Tivoli Access Manager secure domain.
The authorization API function `azn_id_get_creds()` takes user identity information as input parameters and returns user authorization credentials.

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

**Note:** Identity information can also be obtained from a privilege attribute certificate (PAC). See “Converting credentials to the native format” on page 43.

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

1. “Specifying the authorization authority”
2. “Specifying user authentication identity”
3. “Specifying additional user information”
4. “Placing user information into an API buffer” on page 38
5. “Obtaining authorization credentials for the user” on page 38

**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 `mechanism_id` to `azn_id_get_creds()`. Set `mechanism_id` to NULL to specify Tivoli Access Manager authorization.

**Specifying user authentication identity**

For each user to be authenticated, information is loaded into the data structure `azn_authdefault_t`. If the user is authenticated, you must load the user’s identity into the `principal` string (of type `azn_string_t`) in the `azn_authdefault_t` data structure.

If the user is unauthenticated, information is loaded into the data structure `azn_unauth_t`. You do not have to load an identity into `azn_unauth_t`.

**Specifying additional user information**

When the application authenticates the user, the application can optionally obtain additional information about the user. This additional information is for use by the application as needed. The Tivoli Access Manager 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_authdefault_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>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>
</tbody>
</table>
### Placing user information into an API buffer

Place the data structure you filled out in "Specifying user authentication identity" on page 37 and "Specifying additional user information" on page 37 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;
   unsigned char *value;
   } azn_buffer_desc, *azn_buffer_t;
   ```

2. Determine the length of your data structure and assign that value to `length`.

3. Set the pointer value to point to the address of your data structure.

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

### Obtaining authorization credentials for the user

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>authority</td>
<td>The authorization authority, as described in &quot;Specifying the authorization authority&quot; on page 37</td>
</tr>
<tr>
<td>mechanism_id</td>
<td>The authentication mechanism.</td>
</tr>
<tr>
<td>mechanism_info</td>
<td>User information, as described in the preceding sections:</td>
</tr>
<tr>
<td></td>
<td>• &quot;Specifying user authentication identity&quot; on page 37</td>
</tr>
<tr>
<td></td>
<td>• &quot;Specifying additional user information&quot; on page 37</td>
</tr>
<tr>
<td></td>
<td>• &quot;Placing user information into an API buffer&quot;</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 the default user registry, and calls `azn_id_get_creds()` to obtain authorization credentials:

```c
azn_authdefault_t user_info;
azn_buffer_desc buf = {0, 0};
azn_creds_h_t creds = AZN_C_INVALID_HANDLE;
azn_creds_create(&creds);
```
/* Specify LDAP user name */
user_info.principal = "testuser";

/* Set LDAP user information. Note: these values are just placeholders */
user_info.auth_method = "ldap_auth_method";
user_info.authmec_method = "ldap_authmec_method";
user_info.user_info = "ldap_user_info";
user_info.browser_info = "ldap_browser_info";
user_info.ipaddr = 0x0a000002;

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

/* Obtain an authorization credential. */
/* Specify the authority as NULL */
/* Specify the authentication registry (mech) as NULL */

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

For more information, see the reference page for "azn_id_get_creds()" on page 150.
Refer also to the authorization API demonstration program.

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

### 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 Tivoli Access Manager permission”
- “Mapping the requested resource to a protected object” on page 40
- “Assigning the user credentials to a credentials handle” on page 40
- “Building an attribute list for additional application information” on page 40
- “Obtaining an authorization decision” on page 41

### Mapping the user operation to a Tivoli Access Manager permission

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

**Note:** For a complete list of defined operations, see the file ogauthzn.h.

Alternatively, the operation can be a custom operation.

- Assign the operation to a string named `operation`. 
• Pass this string as an input parameter to `azn_decision_access_allowed()`.

**Mapping the requested resource to a protected object**

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

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

Complete the following steps:
• Assign the protected object to the string `protected_resource`.
• Pass this string as an input parameter to `azn_decision_access_allowed()`.

**Assigning the user credentials to a credentials handle**

The authorization credentials for a user obtained in “Obtaining user authorization credentials” on page 36 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 “Converting credentials to the native format” on page 43.

**Building an attribute list for additional application information**

The Tivoli Access Manager 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 Tivoli Access Manager 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()`” on page 141.
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 Tivoli Access Manager 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>int</td>
<td>permission</td>
<td>The result of the access request. Consists of one of the following constants: <code>AZN_C_PERMITTED</code> <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>*permission_info</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()`” on page 139
- “`azn_decision_access_allowed_ext()`” on page 141

Cleaning up and shutting down

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

- “Releasing allocated memory”
- “Shutting down the authorization API” on page 42

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_delete_entry()`
  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_pobj()
  Use this function to release memory that is allocated for the azn_pobj_t structure
  that is returned by a call to azn_attrlist_entry_get_pobj_value().

- 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.

**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.

**Working with 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”
- “Converting credentials to the native format” on page 43
- “Creating a chain of credentials” on page 43
- “Determining the number of credentials in a credentials chain” on page 43
- “Obtaining a credential from a chain of credentials” on page 43
- “Modifying the contents of a credential” on page 44
- “Obtaining an attribute list from a credential” on page 44

**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.
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.

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`.

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.

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`.

**Modifying the contents of a credential**

Use the function `azn_creds_modify()` and the default credential modification service 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, delete, or modify application-specific information in a user’s credentials. The effect of this call is to replace the existing credential attribute list with the new attribute list supplied.

Complete the following steps:

1. Use the attribute list functions to create an attribute list containing the information to be added. (Alternatively, you can use `azn_creds_get_attrlist_for_subject()` to obtain a handle to an existing attribute list.) Assign the attribute list handle `mod_info` to the new attribute list. For more information on attribute lists, see the section “Attribute lists” on page 12.

2. Set the credential modification service `mod_svc_id` to NULL. This invokes the default credential modification service, which replaces the existing attribute list in the credential with the new attribute list passed as `mod_info`.

3. Assign the credentials handle `creds` to point to the credentials to be modified.

4. Call `azn_creds_modify()`, passing the credential to be modified in the `creds` parameter and the attribute list that will replace the credential’s existing attribute list in the `mod_info` parameter.

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

Note that the following attributes are considered read-only and must not be modified by `azn_creds_modify()`:

- `azn_cred_principal_uuid`
- `azn_cred_principal_name`
- `azn_cred_principal_domain`
- `azn_cred_version`
- `azn_cred_mech_id`
- `azn_cred_group_uuids`
- `azn_cred_group_names`
- `azn_cred_authzn_id`
- `azn_cred_ldap_dn`

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

Entire attribute lists can be added or modified within the credential structures by using the `azn_creds_modify()` function. Single string attributes can be set and
retrieved by using the `azn_creds_set_attr_value_string()` and `azn_creds_get_attr_value_string()` functions, respectively.

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_credential_version</code></td>
<td>The credential version. For Tivoli Access Manager 4.1.0, the version is 0x410.</td>
</tr>
<tr>
<td><code>azn_credential_mech_id</code></td>
<td>The mechanism ID for this credential. The ID depends on the registry from which the credential was built.</td>
</tr>
<tr>
<td></td>
<td>For example, when build from LDAP it is <code>IV_LDAP_V3.0</code>. When built from Domino or AD, it is <code>IV_URAF_V3.0</code>. When the user credentials are unauthenticated, it is <code>IV_UNAUTH_V3.0</code>.</td>
</tr>
<tr>
<td><code>azn_credential_principal_uuid</code></td>
<td>The UUID for the entity. For example:</td>
</tr>
<tr>
<td></td>
<td><code>eb529fec-6498-11d7-b236-000629ba5445</code></td>
</tr>
<tr>
<td><code>azn_credential_principal_name</code></td>
<td>The string name of the entity. For example:</td>
</tr>
<tr>
<td></td>
<td><code>fred_smith</code></td>
</tr>
<tr>
<td><code>azn_credential_group_uuids</code></td>
<td>The string group UUID memberships of this entity. Each value is a UUID, similar to the value shown in the <code>azn_credential_principal_uuid</code> example.</td>
</tr>
<tr>
<td><code>azn_credential_group_names</code></td>
<td>The string group name memberships of this entity. Values are strings. For example,</td>
</tr>
<tr>
<td></td>
<td><code>engineering</code></td>
</tr>
<tr>
<td><code>azn_credential_ldap_dn</code></td>
<td>The LDAP DN used to build these authorization credentials. Note that this is used only for credentials built from LDAP user registry. For example:</td>
</tr>
<tr>
<td></td>
<td><code>cn=fred_smith,o=tivoli,c=us</code></td>
</tr>
<tr>
<td><code>azn_credential_uraf_name</code></td>
<td>The URAF name used to build these authorization credentials. For example:</td>
</tr>
<tr>
<td></td>
<td><code>fred_smith</code></td>
</tr>
<tr>
<td><code>azn_credential_user_info</code></td>
<td>Any user information that was passed in the mechinfo structure that was defined by the caller. For example:</td>
</tr>
<tr>
<td></td>
<td><code>Dr Fred Oscar Smith III</code></td>
</tr>
<tr>
<td><code>azn_credential_auth_method</code></td>
<td>Any authentication method information that was passed in the mechinfo structure. This is specified by caller of the API. For example:</td>
</tr>
<tr>
<td></td>
<td><code>X509 certificate</code></td>
</tr>
<tr>
<td><code>azn_credential_authnmech_info</code></td>
<td>Any authentication mechanism information that was passed in the mechinfo structure. This is specified by caller of the API. For example:</td>
</tr>
<tr>
<td></td>
<td><code>IBM Tivoli Demo Client v4.1.0</code></td>
</tr>
</tbody>
</table>
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.

### Setting and getting string attribute values for a credential

The azn_creds_get_attr_value_string() and azn_creds_set_attr_value_string() functions are provided for manipulating a single string attribute within a credential.

Use the function azn_creds_get_attr_value_string() to obtain the string value of a specified attribute in a credential. This function accesses the attribute list for a specified credential, and returns the string value of the specified attribute.

Use the function azn_creds_set_attr_value_string() to set the value of a specified attribute in the user credential. This function edits the attribute list of the specified credential and sets the attribute to the specified string value.

Note that the following attributes are considered read-only and must not be modified by azn_creds_set_attr_value_string():

- azn_cred_principal_uuid
- azn_cred_principal_name
- azn_cred_principal_domain
- azn_cred_version
- azn_cred_mech_id
- azn_cred_group_uuids
- azn_cred_group_names
- azn_cred_authzn_id

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_cred_ip_address</td>
<td>The IP address information passed in the mechinfo structure. Specified by caller. This is the string representation of the IP address in network order. For example: 0x12657834</td>
</tr>
<tr>
<td>azn_cred_browser_info</td>
<td>The browser information passed in the mechinfo structure. This is specified by caller of the API. For example: Mozilla</td>
</tr>
</tbody>
</table>
Comparing two credentials

Use the function `azn_creds_equal()` 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_equal()`.

The function `azn_creds_equal()` returns a boolean value of type `azn_boolean_t`. The value is `true` if the credentials are identical or `false` if the credentials differ.

Copying a credential

Use the `azn_creds_copy()` function to copy the contents of one credentials to another, new credentials. The `azn_creds_copy()` function does not actually make another copy of the credentials, it merely creates a new handle to the credentials. Tivoli Access Manager releases the storage associated with the credentials after the last handle to the credentials is released.

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 `azn_creds_copy()` function returns a new credentials handle to the credential.

When the application is finished using the new credential, use `azn_creds_delete()` to release the handle. The storage for the credentials is released after all handles have been released.

Note: To add a credential to an existing credential, use the `azn_creds_combine()` function.
Chapter 5. Backwards compatibility and application migration

IBM Tivoli Access Manager (Tivoli Access Manager) Base Version 4.1 is binary backwards compatible to existing Tivoli Access Manager Version 3.9 and Tivoli SecureWay Policy Director Version 3.8 non-DCE authorization API clients.

Existing authorization API clients that are being migrated to use the Tivoli Access Manager Base Version 4.1 ADK must run the `svrsslcfg` utility again. You should use the sample configuration file `aznAPI.conf`, which is provided with the authorization API demonstration program, as the base to create the authorization API configuration file.

You might receive errors for some interfaces when compiling existing authorization API clients with the Tivoli Access Manager Version 4.1 ADK. This is because some of the authorization API interfaces from Version 3.8 and Version 3.9 have been deprecated for Version 4.1.

The deprecated interfaces are located in the `azn_deprecated.h` header file. You can look in the `azn_deprecated.h` file to determine the new interface that replaces each deprecated interface. You can also review the section "Deprecated API elements".

In the short term, you can compile with the `azn_deprecated.h` header file to continue to use the deprecated Version 3.8 and Version 3.9 interfaces. However, you should switch as soon as possible to the new interfaces, to avoid further problems.

Binary backwards compatibility

Tivoli Access Manager Version 4.1 supports binary backwards compatibility for Tivoli SecureWay Policy Director Version 3.8, and Tivoli Access Manager Version 3.9 applications as follows:

- The Tivoli Access Manager Version 4.1 runtime environment supports applications compiled against the Tivoli SecureWay Policy Director ADK Version 3.8, Tivoli Access Manager ADK Version 3.9, and Tivoli Access Manager ADK Version 4.1 with the following exception:
  - Tivoli Access Manager Version 4.1 runtime environment on the Solaris Operating Environment only supports applications compiled against either the Version 3.9 or Version 4.1 Tivoli Access Manager ADK.
- Tivoli Access Manager Version 4.1 runtime environment on the Solaris Operating Environment does not support applications compiled against the Tivoli SecureWay Policy Director ADK Version 3.8, due to a Solaris compiler problem.

Deprecated API elements

This section describes elements of the authorization API that have been deprecated. The deprecated elements are supported for backwards compatibility in Tivoli Access Manager Version 4.1. Application developers should not use any deprecated elements in new applications.
Permission info attribute values

The following values for the `azn_init_set_perminfo_attrs` attribute have been deprecated:

- `azn_perminfo_wm`
  Replaced by `azn_perminfo_wm_ulong`.
- `azn_perminfo_wm_permitted`
  Replaced by `azn_perminfo_wm_permitted_ulong`
- `azn_perminfo_al`
  Replaced by `azn_perminfo_al_ulong`

The above attributes are passed as input parameters to `azn_initialize()`.

Each of the deprecated attributes has a corresponding attribute that is returned by the `permission_info` output parameter of the `azn_decision_access_allowed_ext()` function call. The corresponding attributes have been deprecated also.

The deprecated attributes do not work when the remote authorization API client and the Authorization server are on operating system platforms that implement different Endian ordering.

Remote authorization API clients no longer receive the `azn_perminfo_qop` and `azn_perminfo_qop_int` attributes by default. To enable this support, you must specify these attributes in the authorization server’s configuration file `ivacld.conf`. To specify these attributes, uncomment the following line in the configuration file:

```
#permission-info-returned = azn_perminfo_qop azn_perminfo_qop_ulong
```

Deprecated API for comparing credentials

The API for comparing credentials have been deprecated.

- `azn_creds_compare()`
  The `azn_creds_compare()` API has been replaced by `azn_creds_equal()`. You should use `azn_creds_equal()` because it returns an `azn_status_t` return code, which can be used to handle API execution failures.

Obtaining the user’s authorization identification

The new attribute `azn_cred_authzn_id` replaces the following attributes:

- `azn_cred_dce_name`
- `azn_cred_uraf_name`

The attribute `azn_cred_ldap_dn` remains a valid attribute specific to LDAP but does not now perform the same function as `azn_cred_authzn_id`. The `azn_cred_ldap_dn` attribute is now, for LDAP credentials only, defined to be the LDAP Distinguished Name (DN) of the principal.

Authorization API initialization attributes

The following attributes have been deprecated. These attributes formerly were passed as parameters to `azn_initialize()`.

- `azn_init_namespace_location`
- `azn_init_tcp_port`
- `azn_init_udp_port`
- `azn_init_qop`
• azn_init_remote_ns_loc
• azn_init_max_handle_groups

The following configuration file entries have been deprecated. These configuration file entries correspond to each of the attributes in the list above.
• namespace-location
• tcp-port
• udp-port
• remote-qop
• remote-ns-loc
• max-handle-groups

DCE authentication APIs
The following DCE authentication APIs have been deprecated:
• azn_util_server_authenticate()
• azn_util_client_authenticate()

When called, these functions now return the error code AZN_S_UNIMPLEMENTED_FUNCTION.

Applications are now authenticated as part of azn_initialize().

User registry information
The new data structure azn_authdefault_t contains credential and other information used within the Tivoli Access Manager secure domain. This data structure replaces the following deprecated data structures:
• azn_authdce_t
• azn_authldap_t
• azn_authuraf_t

Deprecated return codes
The following minor error codes have been deprecated:
• AZN_ENT_PDPOBJ_INVALID_SVCINFO_HDL
• AZN_ENT_PDPOBJ_INVALID_ARG_COUNT
• AZN_ENT_PDPOBJ_ARG_ARRAY
• AZN_ENT_PDPOBJ_OUT_OF_MEMORY
• AZN_ENT_PDPOBJ_INVALID_ARGUMENT
• AZN_PAC_EPAC_INVALID_SVCINFO_HDL
• AZN_PAC_EPAC_INVALID_ARG_COUNT
• AZN_PAC_EPAC_ARG_ARRAY
• AZN_PAC_EPAC_OUT_OF_MEMORY
• AZN_PAC_EPAC_INVALID_ARGUMENT

The following return codes have been replaced by major error codes defined in ogauthzn.h.
• AZN_S_SVC_ENT_INVALID_SVCINFO_HDL
• AZN_S_SVC_ENT_INVALID_ARG_COUNT
• AZN_S_SVC_ENT_ARG_ARRAY
- AZN_S_SVC_ENT_OUT_OF_MEMORY
- AZN_S_SVC_ENT_INVALID_ARGUMENT
- AZN_S_SVC_PAC_INVALID_SVCINFO_HDL
- AZN_S_SVC_PAC_INVALID_ARG_COUNT
- AZN_S_SVC_PAC_ARG_ARRAY
- AZN_S_SVC_PAC_OUT_OF_MEMORY
- AZN_S_SVC_PAC_INVALID_ARGUMENT
Chapter 6. Introducing authorization service plug-ins

The IBM Tivoli Access Manager (Tivoli Access Manager) authorization API supports a service plug-in model. This model enables developers to write plug-in modules that extend the capabilities of the Tivoli Access Manager 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 Tivoli Access Manager secure domain.

This chapter contains the following sections:
- “Service plug-in architecture”
- “Implementing a service plug-in” on page 58
- “Supplied implementations for service plug-ins” on page 69

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.
The architecture for each type of service plug-in may expand the architecture model illustrated above. For example, the administration service presents some extensions, as shown in Figure 4 on page 82.

**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.

**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 Tivoli Access Manager 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.

**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 Tivoli Access Manager authorization server. In general, the service plug-in framework operates independently from the authorization functionality. The exception to this is External Authorization Services which can only be invoked in the course of making an access decision from an `azn_decision_*` interface.

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_entitlement_get_entitlements()`.

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

Handles can be either input parameters or output parameters. An input parameter is valid when the handle was either:
- Created by a call to `azn_creds_create()`
- Created by a call to `azn_attrlist_create()`
- Returned as output from an authorization API function

If the handle is an output parameter, then its valid values also include an initialization to `AZN_C_INVALID_HANDLE`.

The following table lists the handles that must be valid.

<table>
<thead>
<tr>
<th>Service</th>
<th>API Function Invoked By the Calling Application</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entitlements</td>
<td><code>azn_entitlement_get_entitlements()</code></td>
<td><code>entitlements</code></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, Tivoli Access Manager provides a default service. For all other services, the application must specify a service ID as a parameter to `azn_service_initialize()`.

**Supported types of service plug-ins**

The authorization service supports these types of service plug-ins:

- Entitlements service
- Credentials modification service
- Privilege attribute certificate (PAC) service
- Administration service
- External authorization service

---

<table>
<thead>
<tr>
<th>Service</th>
<th>API Function Invoked By the Calling Application</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Credentials Modification</strong></td>
<td><code>azn_creds_modify()</code></td>
<td><code>creds</code></td>
</tr>
<tr>
<td></td>
<td><strong>Input parameter.</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>new_creds</code></td>
<td><strong>Output parameter.</strong></td>
</tr>
<tr>
<td><strong>Privilege Attribute Certificate</strong></td>
<td><code>azn_pac_get_creds()</code></td>
<td><code>new_creds</code></td>
</tr>
<tr>
<td></td>
<td><strong>Output parameter.</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>azn_creds_get_pac()</code></td>
<td><code>pac</code></td>
</tr>
<tr>
<td></td>
<td><strong>Output parameter.</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Administration</strong></td>
<td><code>ivadmin_protobj_get2()</code></td>
<td><code>creds</code></td>
</tr>
<tr>
<td></td>
<td><code>ivadmin_protobj_list3()</code></td>
<td><code>indata</code></td>
</tr>
<tr>
<td></td>
<td><code>ivadmin_server_gettasklist()</code></td>
<td><code>outdata</code></td>
</tr>
<tr>
<td></td>
<td><code>ivadmin_server_performtask()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>The <code>creds</code> and <code>indata</code> parameters are input parameters. The <code>outdata</code> parameter is an output parameter.</strong></td>
<td></td>
</tr>
<tr>
<td><strong>External authorization</strong></td>
<td><code>azn_decision_access_allowed_ext()</code></td>
<td><code>creds</code></td>
</tr>
<tr>
<td></td>
<td><code>app_context</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>permission_info</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>The <code>creds</code> and <code>app_context</code> parameters are input parameters. The <code>permission_info</code> parameter is an output parameter.</strong></td>
<td></td>
</tr>
</tbody>
</table>
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.

**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 Tivoli Access Manager secure domain that contains only those protected objects that the user is authorized to view.

**Credentials modification service**
A credentials modification service plug-in enables domain-specific authorization API applications to perform modifications on a Tivoli Access Manager 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.

**Privilege attribute certificate service**
A privilege attribute certificate (PAC) service plug-in gives domain-specific authorization API applications the ability to move Tivoli Access Manager credentials back and forth between the native Tivoli Access Manager 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.

**Administration service**
An administration service plug-in enables applications to perform application-specific administration tasks on protected object resources that are secured in the Tivoli Access Manager secure domain. The administration service provides functions that enable a plug-in to obtain the contents of a defined portion of the protected object hierarchy. Additional functions enable a plug-in to define application-specific administration tasks, and to return commands that perform those tasks.

The administration service plug-in is accessed by a calling application that sends Tivoli Access Manager administration API calls. The calling application can be either an administrative utility such as the Tivoli Access Manager pdadmin command or the Tivoli Access Manager Web Portal Manager, or can be a custom-built application. The administration service maps the administration API calls to the corresponding administration service API calls, and carries out the requested action.

**External authorization service**
An external authorization service plug-in is an optional extension of the Tivoli Access Manager authorization service that allows you to impose additional authorization controls and conditions. You can use an external authorization service plug-in to force authorization decisions to made based on application-specific criteria that are not known to the Tivoli Access Manager authorization service.
Implementing a service plug-in

This section describes how to implement a service plug-in. The service plug-in architecture supports standard methods for initialization, configuration, error handling and shutdown. Each service plug-in requires implementation of interfaces that are specific to the service type.

This section contains the following topics:

- “Initialization and configuration of service plug-ins”
- “Implementing service interfaces” on page 61
- “Using error codes” on page 62
- “Shut down” on page 66
- “Example service source code” on page 66

Initialization and configuration of service plug-ins

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 initialization attributes defined for the `svc_init` parameter.

The service plug-in can, but is not required to, 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_service_initialize()`” on page 179.

Constructing a service definition

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, you must define several elements and then combine them into a service definition. To do this, complete the following tasks:

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.
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 Tivoli Access Manager 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:

<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. 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.

4. Building a service definition

Each service plug-in entry uses the following syntax:

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

Parameters specified after the ampersand (&) are passed to the service plug-in’s shared library. The shared library takes the remainder of the string following the ampersand &, breaks the string up into white space separated tokens, and passes the tokens directly to the administration service’s initialization interface, azn_svc_initialize(), in the argv array parameter. The number of strings in the argv array is indicated by the argc function parameter.

Parameters specified before the & are processed by the authorization API. For example, the external authorization service can have an optional weight parameter, and the administration service can optionally take the name of a protected object hierarchy name.

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.

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

- "Configuring a service by using a configuration file" on page 60
- "Configuring a service programmatically" on page 60

### Configuring a service by using a configuration file

The Tivoli Access Manager authorization service recognizes and registers service plug-ins by reading entries in the authorization API client configuration file.

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>
<tr>
<td>[aznapi-admin-services]</td>
<td>Administration service plug-ins</td>
</tr>
<tr>
<td>[aznapi-extern-authzn-services]</td>
<td>External Authorization service plug-ins</td>
</tr>
</tbody>
</table>

### 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.

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_DECLSPECM defazn_string_t defazn_init_ent_svc;
/* PAC service definition attribute */
AZN_DECLSPECM defazn_string_t defazn_init_pac_svc;
/* Credential modification service definition attribute */
AZN_DECLSPECM defazn_string_t defazn_init_cred_mod_svc;
/* Administration service definition attribute */
AZN_EXTERNPECM defazn_string_t defazn_init_admin_svc;
/* External authorization service definition attribute */
AZN_EXTERNPECM defazn_string_t defazn_initExtern_authzn_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 "Constructing a service definition" on page 58. 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.

The following example shows source code for configuring an entitlements service programmatically.

```c
azn_status_t status;
azn_attrlist_h_t init_data = AZN_C_INVALID_HANDLE;
azn_attrlist_h_t init_info = AZN_C_INVALID_HANDLE;
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. 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, "azn_initialize failed: [%08X:%08X]\n",
            azn_error_major(status),
            azn_error_minor(status));
    exit -1;
}

Implementing service interfaces

The implementation of a service plug-in consists of building a shared library. The shared library must contain interfaces that are specific to the type of service. These interfaces perform the work that is specific to the service. A calling application sends a request to the authorization API, which vectors the request to the appropriate service plug-in. The service plug-in shared library must contain an implementation of the authorization API interface that will perform the tasks specific to the service plug-in.

The following table shows the authorization API interface that is invoked by the calling application, and the corresponding authorization API interface that is implemented in the service plug-in shared library.
<table>
<thead>
<tr>
<th>Service</th>
<th>API function invoked by the calling application</th>
<th>Function implemented by the service plug-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entitlements service</td>
<td>azn_entitlement_get_entitlements()</td>
<td>azn_entitlement_get_entitlements()</td>
</tr>
<tr>
<td>Credentials modification service</td>
<td>azn_creds_modify()</td>
<td>azn_creds_modify()</td>
</tr>
<tr>
<td>Privilege attribute certificate service</td>
<td>azn_creds_get_pac()</td>
<td>azn_creds_get_pac()</td>
</tr>
<tr>
<td></td>
<td>azn_pac_get_creds()</td>
<td>azn_pac_get_creds()</td>
</tr>
<tr>
<td>Administration service</td>
<td>ivadmin_protobj_get2()</td>
<td>azn_admin_get_object()</td>
</tr>
<tr>
<td></td>
<td>ivadmin_protobj_list3()</td>
<td>azn_admin_get_objectlist()</td>
</tr>
<tr>
<td></td>
<td>ivadmin_server_gettasklist()</td>
<td>azn_admin_get_tasklist()</td>
</tr>
<tr>
<td></td>
<td>ivadmin_server_performtask()</td>
<td>azn_admin_perform_task()</td>
</tr>
<tr>
<td>External authorization service</td>
<td>azn_decision_access_allowed()</td>
<td>azn_decision_access_allowed()</td>
</tr>
<tr>
<td></td>
<td>azn_decision_access_allowed_ext()</td>
<td>azn_decision_access_allowed_ext()</td>
</tr>
</tbody>
</table>

Note that the name of the authorization API function that is invoked by the calling application and the name of the authorization API function that is invoked by the service plug-in shared library are often the same. Note, however, that these are two separate implementations of the interface (function). The service plug-in shared library implements its own version of the interface, in order to perform service type-specific tasks.

For example, calling applications often call `azn_decision_access_allowed_ext()` to request an authorization decision. When the authorization API has been configured to recognize an external authorization service, this call to `azn_decision_access_allowed_ext()` is routed through the service dispatcher to the appropriate external authorization service plug-in shared library. Within this library the plug-in developer will have implemented a local version of `azn_decision_access_allowed_ext()` that can make authorization decisions based on conditions or rules that are potentially unknown to the core Tivoli Access Manager authorization engine.

As long as the implementation of the interface within the service plug-in satisfies the interface (function) signature of the authorization API function, the service plug-in shared library can augment the default authorization process with code specific to the plug-in.

Note that the interfaces described in this section are in addition to the interfaces that are generic to all types of service plug-ins. These generic interfaces include initialization, shutdown, and error handling functions.

**Using 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 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 to an `azn_status_t` value. The minor error code must be defined as described in “Minor error codes” on page 65.

**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 `ogauthzn.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_SERVICE_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_INITIALIZE_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.
| AZN_S_INVALID_MOD_FUNCTION | The supplied modification service identifier is invalid.
| AZN_S_INVALID_PAC_SVC | The identifier (id) of the PAC service is invalid.
| AZN_S_SVC_DISPATCHER_FAILURE | The service dispatcher failed. This can be caused by incorrect initialization of the authorization API.
| AZN_S_SVC_DLL_LOAD_FAILED | The DLL for a service plug-in failed to load correctly.
| AZN_S_SVC_SERVICE_IS_REGISTERED | The service ID cannot be registered because it is already listed as registered by the service dispatcher.
The following table shows the major error codes that service plug-ins return:

<table>
<thead>
<tr>
<th>Error Code Prefix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZN_S_SVC_ADMIN_*</td>
<td>Administration service plug-in major error codes.</td>
</tr>
<tr>
<td>AZN_S_SVC_ENT_*</td>
<td>Entitlements service plug-in major error codes.</td>
</tr>
<tr>
<td>AZN_S_SVC_EAS_*</td>
<td>External authorization service plug-in major error codes.</td>
</tr>
<tr>
<td>AZN_S_SVC_CRED_MOD_*</td>
<td>Credentials modification service plug-in major error codes.</td>
</tr>
</tbody>
</table>

In addition to the generic major codes that apply to all types of service plug-ins, there exist generic major codes that apply to a specific service plugin. These major codes are defined in ogauthzn.h.
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.

**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 catalog 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 `oauthzn.h` defines the following prefixes:

<table>
<thead>
<tr>
<th>Service</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entitlements</td>
<td><code>azn_ent_svc_err_prefix</code></td>
</tr>
<tr>
<td>Credentials Modification</td>
<td><code>azn_mod_svc_err_prefix</code></td>
</tr>
<tr>
<td>Privilege Attribute Certificate</td>
<td><code>azn_pac_svc_err_prefix</code></td>
</tr>
<tr>
<td>Administration service</td>
<td><code>azn_admin_svc_err_prefix</code></td>
</tr>
<tr>
<td>External authorization service</td>
<td><code>azn_eas_svc_err_prefix</code></td>
</tr>
</tbody>
</table>

The service plug-in developer must use these prefixes to define the minor error codes for each service plug-in. The `azn_util_errcode()` interface uses these prefixes to properly encode each minor error into an `azn_status_t`.

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

```c
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:

```c
#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))
```

**Shut down**

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()`” on page 182.

**Example service source code**

This section contains source code for an example implementation of an entitlements service plug-in. This code demonstrates configuration, initialization, shutdown, and implementation of interfaces specific to the service type.

```c
/*
 * 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"

/*
 * Define a mask to identify minor errors that originate from this
 * service. All 3rd party entitlements services must use the
 * azn_ent_svc_err_prefix to prefix minor code definitions. The lower
```
* 16 bits should be differentiated from other installed entitlements
* services. eg. in this case 0x8000 identifies the error as originating
* from this service.
*/
#define AZN_MYSVC_ERROR_MASK (azn_ent_svc_err_prefix | 0x8000)
#define AZN_MYSVC_INVALID_SVCINFO_HDL (AZN_MYSVC_ERROR_MASK | 1)
#define AZN_MYSVC_INVALID_ARG_COUNT (AZN_MYSVC_ERROR_MASK | 2)
#define AZN_MYSVC_INVALID_ARG_ARRAY (AZN_MYSVC_ERROR_MASK | 3)
#ifdef __cplusplus
extern "C" {
#endif

/* Interface function definitions.
******************************************************************************/
/*
* INTERFACE NAME
* azn_service_initialize
*
* INTERFACE DESCRIPTION
* init the entitlements service
*
* INTERFACE 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_MYSVC_INVALID_SVCINFO_HDL));
    }

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

    if (argc > 0 && argv == NULL) {
        return (azn_util_errcode(AZN_S_FAILURE,
                                  AZN_MYSVC_INVALID_ARG_ARRAY));
    }

Chapter 6. Introducing authorization service plug-ins  67
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
   */

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

Supplied implementations for service plug-ins

The Tivoli Access Manager authorization ADK supplies implementations for a number of different service plug-in types. Some of these implementations are built into the authorization API. Others consist of separate shared libraries.

The name for each implementation that consists of a shared library is specified as the short name in the table in each of the following sections. You can use the short name when specifying the plug-in location, as part of constructing the service definition. For more information, see “Constructing a service definition” on page 58.

The following sections describe the supplied service implementations:

- “Tivoli Access Manager entitlements services”
- “Credentials modification service” on page 71
- “Privilege attribute certificate service” on page 72
- “External authorization service” on page 72

Tivoli Access Manager entitlements services

Tivoli Access Manager provides two entitlements services:

- Tivoli Access Manager protected objects entitlements service
  This is the default entitlements service.
- Extended attributes entitlements service
These services are described in the following tables:

<table>
<thead>
<tr>
<th>Service Plug-in Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Category</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Name</td>
<td>Protected objects entitlements service</td>
</tr>
<tr>
<td>Service ID</td>
<td>NULL</td>
</tr>
<tr>
<td>Plug-in Short Name</td>
<td>Not applicable (implemented internally)</td>
</tr>
</tbody>
</table>
| Parameters             | **app_context** (input) The entitlements service accepts a single, multi-valued string attribute that specifies the root node(s) for searching the Tivoli Access Manager 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. The **app_context** attribute list contains these attributes:
  • **azn_ent_svc_pd_pobj_path** (multi-valued)
  • **azn_ent_svc_pd_pobj_reqd_ops** (single value)
    This attribute contains a string to denote the set of operations that the credential must have upon the protected object. For example: `rxT`.

  **entitlements** (output) Entitlements data is returned as a multi-valued attribute list of protected objects for each search tree root node passed in to the `azn_entitlement_get_entitlements()` call. The list only contains objects underneath the specified root node that have an ACL explicitly attached to them.

    The attribute list is appended with the attribute `azn_ent_svc_pd_pobj_matches`, which contains string protected object names. The attribute list may contain `AZN_C_INVALID_HANDLE` if there are no protected objects that match.

| Description | The service returns a list of protected resources in the database that have an ACL explicitly attached to them and for which the ACL allows the specified credential 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. This entitlements service should be configured in secure domains with Tivoli Access Manager servers that have been configured and are running. |

The following table describes the extended attributes entitlements service:

<table>
<thead>
<tr>
<th>Service Plug-in Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Category</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Name</td>
<td>Extended attributes entitlements service</td>
</tr>
<tr>
<td>Service ID</td>
<td>User specified in the service definition.</td>
</tr>
<tr>
<td>Plug-in Short Name</td>
<td><code>azn_ent_ext_attr</code></td>
</tr>
</tbody>
</table>
### Credentials modification service

The Tivoli Access Manager authorization ADK provides two implementations of a credentials modification service. One implementation modifies attribute lists. The other implementation modifies group memberships.

The service described in the table below is built-in to the authorization API.

<table>
<thead>
<tr>
<th>Service Plug-in Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Category</strong></td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Service ID</td>
</tr>
<tr>
<td>Plug-in Short Name</td>
</tr>
</tbody>
</table>
| Parameters | **creds** The credentials with which to work.  
**list** The attribute list to replace.  
**outcred** The credential resulting from the operation. |
| Description | This service will replace the attribute list in credential **creds** with the attribute list **list**. Read-only attributes, such as group UUIDs, cannot be changed by this service. The original credential is left unchanged and a new credential containing **list** is returned in **outcred**. |

The service described in the table below is not built into the authorization API.

<table>
<thead>
<tr>
<th>Service Plug-in Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Category</strong></td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Service ID</td>
</tr>
<tr>
<td>Plug-in Short Name</td>
</tr>
</tbody>
</table>
| Parameters | **creds** The credentials to modify  
**mod_info** Input attribute: AZN_MOD_RAD_GROUP_NAMES Contains the names of the groups to which the resulting credential **creds** will effectively be added.  
**newcreds** A new credential based on **creds** and built to include the groups in **mod_info**. |

---

Chapter 6. Introducing authorization service plug-ins 71
Privilege attribute certificate service

This service is built in to the authorization API.

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Tivoli Access Manager privilege attribute certificate (PAC) encoding service</td>
</tr>
<tr>
<td>Service ID</td>
<td>NULL</td>
</tr>
<tr>
<td>Plug-in Name</td>
<td>not applicable (built-in)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters: azn_creds_get_pac()</th>
<th>creds</th>
<th>Input parameter. The credentials to be encoded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pac</td>
<td>An azn_buffer_t that will contain the encoded credentials PAC for creds on completion.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters: azn_pac_get_creds()</th>
<th>pac</th>
<th>An encoded PAC to be decoded into credentials.</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_creds</td>
<td>A pointer to an azn_creds_h_t that will refer to the credential decoded by the service from pac.</td>
<td></td>
</tr>
</tbody>
</table>

Description

Encodes and decodes a Tivoli Access Manager credential to or from a format that is transmissible in a text only environment. The format is a combination of ASN1 and MIME encoding.

External authorization service

The Authorization ADK provides a sample implementation of an external authorization service (EAS) plug-in. This implementation is not built-in to the authorization API.

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Example external authorization service</td>
</tr>
<tr>
<td>Service ID</td>
<td>User specified in the service definition.</td>
</tr>
<tr>
<td>Plug-in Short Name</td>
<td>azn_eas_demo</td>
</tr>
<tr>
<td>Parameters</td>
<td>not applicable</td>
</tr>
</tbody>
</table>
### Service Plug-in Summary

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>This is an example EAS plug-in which simply returns AZN_C_PERMITTED for all authorizations in which it is called to participate. Configure the EAS to be called for a decision that would normally return AZN_C_NOT_PERMITTED. The demonstration program performs some perfunctory checking of input parameters but does not attempt to make any authorization related decisions based upon the incoming access decision information (ADI).</td>
</tr>
</tbody>
</table>
Chapter 7. Implementing entitlements service plug-ins

The IBM Tivoli Access Manager (Tivoli Access Manager) authorization API services framework provides a modular, plug-in style architecture that enables application developers to supplement Tivoli Access Manager 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.

This chapter contains the following topics:

- “Understanding entitlements”
- “Initialization, configuration, and shut down” on page 76
- “Obtaining entitlements for a specified user” on page 77
- “Authorizing a caller to a specific entitlements service plug-in” on page 78
- “Using authorization API interfaces” on page 78
- “Entitlements service error codes” on page 78

Understanding entitlements

An entitlement is a data structure that contains externalized policy information. An entitlement is policy data or capabilities that is formatted in a way that is understandable to a specific application. The application uses the authorization API to instruct the Tivoli Access Manager 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, by data of type azn_buffer_t, or by any other valid attribute type.

Note: For more information on authorization API attribute lists and data types, see Chapter 2, “Authorization API functions and data types”, on page 7.
Entitlements of type \texttt{azn\_string\_t}

An example of an entitlement is the \textit{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 entitlements service for this entitlement is simply a string:

\texttt{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-5pm"

"mon-fri: 1pm-5pm"

"sat: 9am-12pm"

These string values for \textit{time of day restriction} define valid access times.

The entitlements service defines how these strings are formatted and interpreted by the calling application.

The Tivoli Access Manager protected object entitlements service

Another example of an entitlement is the data returned by the Tivoli Access Manager protected object entitlements 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 Tivoli Access Manager protected object space that the caller with the specified privileges can access.

Entitlements of type \texttt{azn\_buffer\_t}

Entitlements services may also define an entitlement as a blob of data using an \texttt{azn\_buffer\_t} value for the attribute. This may be used by distributed object based applications such as Common Object Request Broker Architecture (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 \texttt{azn\_buffer\_t} data type are opaque to the authorization service.

For more information on data type \texttt{azn\_buffer\_t}, see "Buffers" on page 9.

Initialization, configuration, and shut down

Each entitlements service plug-in is a standalone module that is dynamically loaded into the authorization service.

The Tivoli Access Manager authorization service recognizes and registers entitlements service plug-ins with the service dispatcher by reading entries in the aznapi.conf configuration file.
Entitlements service plug-ins are declared in the configuration file under the following stanza entry:

```
[aznapi-entitlement-services]
```

The Tivoli Access Manager 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 “Initialization and configuration of service plug-ins” on page 58.

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:

For more information, see the following sections:

- “Initialization and configuration of service plug-ins” on page 58
- “Shut down” on page 66

See also the reference pages for “azn_service_initialize()” on page 179 and “azn_service_shutdown()” on page 182.

### Obtaining entitlements for a specified user

The authorization API provides the `azn_entitlement_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_entitlement_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.
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.

Using authorization API interfaces

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.

Entitlements service error codes

The entitlements service can generate error codes from the following functions:

- azn_service_initialize()
- azn_service_shutdown()
- azn_entitlement_get_entitlements()

For information on error codes for azn_service_initialize() and azn_service_shutdown(), see “Using error codes” on page 62.

The azn_entitlement_get_entitlements() function returns the following error codes:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZN_S_INVALID_CREDS_HDL</td>
<td>The credentials handle supplied is invalid.</td>
</tr>
<tr>
<td>AZN_S_INVALID_ENTITLEMENTS_SVC</td>
<td>The entitlements service identifier is invalid.</td>
</tr>
<tr>
<td>AZN_S_INVALID_APP_CONTEXT_HDL</td>
<td>The attribute list handle for the application context is invalid.</td>
</tr>
<tr>
<td>AZN_S_INVALID_ENTITLEMENTS_HDL</td>
<td></td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>The attribute list handle for the entitlements is invalid.</td>
</tr>
<tr>
<td>AZN_S_FAILURE</td>
<td>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().</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 "Minor error codes" on page 65.
Chapter 8. Implementing administration service plug-ins

This chapter contains the following topics:

- “Understanding administration service plug-ins”
- “Configuring administration service plug-ins” on page 83
- “Initializing and shutting down administration service plug-ins” on page 84
- “Using an administration service plug-in” on page 85
- “Error codes” on page 86
- “Deploying an administration service plug-in” on page 88

Understanding administration service plug-ins

The administration service makes available to applications several administrative functions that operate on objects in the IBM Tivoli Access Manager (Tivoli Access Manager) protected object namespace. Application developers can write a plug-in to the administration service that applies these administrative functions to objects in a section of the namespace that is specific to the application. Through the plug-in, application developers can also specify administrative actions that are specific to objects in the application-specific namespace.

The administration service differs from other authorization API services in that it does not provide direct access to the administrative function to the calling applications. Instead, the calling applications use the Tivoli Access Manager administration API to send a request.

For example, the administration service functions are called in response to administrative operations that are issued by a calling application such as the pdadmin command line interface, the Tivoli Access Manager Web Portal Manager graphical console, or a third-party application that has been written to use the Tivoli Access Manager administration API.

The administration service maps the administration API calls to the corresponding administration service API calls, and carries out the requested action. There is a one-to-one mapping between several administration API functions and a corresponding administration service function.

The administration service supports dynamic object creation and task execution by adding an externalized administration family of functions to the authorization API. The name of each of these functions is prefixed by azn_admin. You implement these functions within libraries that are registered with the Authorization administration service by an authorization API client application.
In Figure 4, the user application or GUI sends an administrative operation through the Tivoli Access Manager administration API to the Tivoli Access Manager policy server. The Tivoli Access Manager policy server forwards the call to the administration service interface of the authorization API resource manager application that is configured to administer the target protected object space for the administration request. The service dispatcher within the authorization API resource manager application calls the applicable service interface of all administration service plug-ins that have registered to service this type of request. Each plug-in’s service interface is called in turn by the dispatcher and the results are returned to the Tivoli Access Manager policy server. The Tivoli Access Manager policy server then returns the results to the calling application.

Note that the diagram above shows three different administration service plug-ins. You do not need to implement more than one plug-in, but multiple plug-ins are supported.

Do not confuse the Tivoli Access Manager authorization administration service with the Tivoli Access Manager administration API. The Tivoli Access Manager

Figure 4. The administration service plug-in to the authorization API
administration API provides a series of programmatic interfaces that a calling application can use to send requests to the Tivoli Access Manager policy server.

In most cases, applications can use the administration API independent from any use of the authorization administration service. However, application developers can use the authorization administration service plug-in to provide “back-end” authorization functions that can leverage administration API functions to execute application-specific administrative commands. This is described in more detail in “Using an administration service plug-in” on page 85.

Most of the Tivoli Access Manager administration API functions provide programmatic equivalents to each of the pdadmin command line interfaces. The names of the administration API functions begin with the ivadmin_ prefix. The functions are described in the ivadminapi.h header file. For more information on the Tivoli Access Manager administration API, see the IBM Tivoli Access Manager Administration C API Developer’s Reference.

Configuring administration service plug-ins

You can configure an administration service plug-in either manually or programmatically. Manual configuration is accomplished by setting values in a configuration file. Programmatic configuration is accomplished by passing attributes to the administration API at API initialization. These methods correspond directly to the registration steps used by all types of authorization service plug-ins, as described in “Constructing a service definition” on page 58.

Configuration of an administration service plug-in is described in the following sections:

- “Creating a configuration file entry for an administration service”
- “Configuring an administration service programmatically” on page 84

Creating a configuration file entry for an administration service

Administration service plug-ins are typically configured in aznapi.conf, under the following stanza:

[aznapi-admin-services]

The administration services definition syntax is as follows:

<service-id> = <plug-in location> [-pobj <protected object hierarchy name>] [& <plug-in parameters>]

The protected object hierarchy name is an optional parameter. This parameter refers to either the name of a protected object space (hierarchy) or simply to a protected object. The authorization service takes the remainder of the string following the ampersand &, breaks the string up into white space separated tokens, and passes the tokens directly to the administration service’s initialization interface, azn_svc_initialize(), in the argv array parameter. The number of strings in the argv array is indicated by the argc function parameter.

Here is an example configuration file entry:

[aznapi-admin-services]
adminsvc1 = /lib/libadminsvc1.so -pobj /Printers/printer1 & -printer sequoia
adminsvc2 = /lib/libadminsvc2.so -pobj /Printers/printer2 & -printer sequoia
adminsvc3 = /lib/libadminsvc3.so & -printer sequoia
adminsvc4 = /lib/libadminsvc4.so

Chapter 8. Implementing administration service plug-ins 83
As shown in the example above, an authorization API application can register more than one administration service plug-in, but each must have a unique service ID.

Each authorization service plug-in passes the administration service definitions to the Tivoli Access Manager policy server during the `azn_initialize()` call. This enables the Tivoli Access Manager policy server to know the mapping of a protected object hierarchy name to the authorization API client application that has registered an administration service plug-in for it. This is important for object space oriented administration API interfaces which operate on the objects within the object space. Since this mapping is global, care must be taken to specify protected object space mappings that do not conflict across authorization API client applications.

Thus, protected object hierarchy names must be unique for each administration service plug-in within the scope of an authorization API application. Multiple authorization API application instances, however, can register to service the same protected object hierarchy name(s). This can be used to provide failover support for administration of an object space in the event that a particular authorization API application server fails.

### Configuring an administration service programmatically

The authorization API header file `ogauthzn.h` contains a service definition attribute:

```c
azn_string_t azn_init_admin_svc
```

Complete the following steps to use this service definition attribute to configure an administration service plug-in.

1. Use the `azn_attrlist_add_entry()` API to assign to the `init_data` attribute list as many values to the `azn_init_admin_svc` attribute as the number of Administrative service plug-ins needed. These values are expressed as strings, and need to conform to the administration service definition syntax specified in “Initialization and configuration of service plug-ins” on page 58. Ensure that protected object hierarchies that are specified as part of the service definition attribute are unique. See the discussion just above in “Creating a configuration file entry for an administration service” on page 83.

2. Pass the `init_data` attribute to the `azn_initialize()` call to ensure that the specified administration service plug-ins are loaded.

### Initializing and shutting down administration service plug-ins

Each administration 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 administration service plug-ins should provide the standard functions. See the following sections:

- “Initialization and configuration of service plug-ins” on page 58
- “Shut down” on page 66

For more information, see the reference pages for “`azn_service_initialize()`” on page 179 and “`azn_service_shutdown()`” on page 182.
Using an administration service plug-in

The administration service supports the following authorization API functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_admin_get_object()</code></td>
<td>Retrieves a potential protected object. Tests for the existence of a protected object.</td>
</tr>
<tr>
<td><code>azn_admin_get_objectlist()</code></td>
<td>Accesses the administration service to provide a list of all potential protected objects that are children of the specified parent object.</td>
</tr>
<tr>
<td><code>azn_admin_perform_task()</code></td>
<td>Instructs the service to perform an administration task. The service returns the results of the task.</td>
</tr>
<tr>
<td><code>azn_admin_get_tasklist()</code></td>
<td>Returns a list of all the supported administration tasks.</td>
</tr>
</tbody>
</table>

Each of the functions above maps to an equivalent administration API function and an equivalent `pdadmin` command line interface. The mappings are shown in the table below.

<table>
<thead>
<tr>
<th>Authorization API administration Function</th>
<th>Administration API function</th>
<th>pdadmin command line interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_admin_get_object()</code></td>
<td><code>ivadmin_protobj_get2()</code></td>
<td><code>pdadmin object show object_name</code></td>
</tr>
<tr>
<td><code>azn_admin_get_objectlist()</code></td>
<td><code>ivadmin_protobj_list3()</code></td>
<td><code>pdadmin object list object_name</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>pdadmin object listandshow object_name</code></td>
</tr>
<tr>
<td><code>azn_admin_perform_task()</code></td>
<td><code>ivadmin_server_performtask()</code></td>
<td><code>pdadmin server task server_name task</code></td>
</tr>
<tr>
<td><code>azn_admin_get_tasklist()</code></td>
<td><code>ivadmin_server_gettasklist()</code></td>
<td><code>pdadmin server listtasks server_name</code></td>
</tr>
</tbody>
</table>

The Tivoli Access Manager policy server uses these mappings to redirect the administration API calls (`ivadmin_*`) and the `pdadmin` command lines that correspond to the `azn_admin_get_object()` and `azn_admin_get_objectlist()` APIs to the appropriate administration service plug-in. When processing the administration APIs (`ivadmin_*`) and the `pdadmin` command lines that correspond to the `azn_admin_perform_task()` and `azn_admin_get_tasklist()` APIs, the Tivoli Access Manager policy server just forwards them as a single unparsed string to the corresponding authorization API client application.

An authorization API administration service should provide a server task to handle help requests for every command it supports. A valid response should be produced for any `pdadmin server task server-name help command-name` command for a supported `command-name`. However, in response to a `pdadmin server task server-name help` command, a major error code of AZN_S_SVC_ADMIN_INVALID_TASK should be returned, as the service does not know about other administration services that could be registered for the same server.

Not all of the authorization API administration service interfaces are relevant to every Authorization administration service plug-in. Therefore, a result of not supported is returned for each administration service interface not implemented by the authorization service plug-in.

The administration service plug-ins make use of the following attributes in the `outdata` attribute list that the authorization API administration service interfaces
Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_admin_svc_pobj</td>
<td>Administration service protected object attribute, of type azn_string_t.</td>
</tr>
<tr>
<td>azn_admin_svc_task</td>
<td>Administration service task attribute, of type azn_string_t.</td>
</tr>
<tr>
<td>azn_admin_svc_results</td>
<td>Administration service results attribute, of type azn_string_t.</td>
</tr>
</tbody>
</table>

For more information on the authorization API administration functions, see the following reference pages:

- “azn_admin_get_object()” on page 170
- “azn_admin_get_objectlist()” on page 172
- “azn_admin_get_tasklist()” on page 174
- “azn_admin_perform_task()” on page 176

**Error codes**

This section contains the following topics:

- “Errors when registering the administration service plug-in”
- “Errors when registering administration definitions” on page 87
- “Major errors from administration service functions” on page 87
- “Minor errors from administration service functions” on page 88
- “Error codes specific to an authorization services plug-in” on page 88

**Errors when registering the administration service plug-in**

The authorization API initialization function azn_initialize() fails when the Authorization administration service definitions made by a single authorization API application result in following error conditions:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZN_S_SERVICE_IS_REGISTERED</td>
<td>Attempted to register more than one service definition with the same service-id</td>
</tr>
<tr>
<td>AZN_S_SVC_ADMIN_POBJ_ALREADY_REGISTERED</td>
<td>Attempted to register more than one service definition with the same protected object hierarchy name</td>
</tr>
<tr>
<td>AZN_S_SVC_ADMIN_POBJ_FUNC_NOT_FOUND</td>
<td>The -pobj option is specified for an administration service definition, but the specified plug-in does not contain the azn_admin_get_objectlist() and the azn_admin_get_object() functions. When the -pobj option is not specified, the plug-in is not required to support either of these functions.</td>
</tr>
<tr>
<td>AZN_S_SVC_ADMIN_TASK_FUNC_NOT_FOUND</td>
<td></td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>The administration service plug-in supports only one of the task related functions azn_admin_get_tasklist() and azn_admin_perform_task(). Each Administration service plug-in must support either both the task-related functions or support none of them.</td>
</tr>
<tr>
<td>AZN_S_INVALID_LISTENING_PORT</td>
<td>When the administration service plug-in registration succeeds, but the ssl-listening-port has not been specified as part of the authorization API initialization, azn_initialize() returns this error.</td>
</tr>
</tbody>
</table>

**Errors when registering administration definitions**

The authorization API registers the administration service definitions with the Tivoli Access Manager policy server during the azn_initialize() call. The policy server stores the registered administration service definitions in persistent store and overwrites an existing registration when it receives a new registration from the same authorization API application.

When the authorization API cannot contact the Tivoli Access Manager policy server during azn_initialize(), it skips registration of the specified administration service definitions, logs a message describing that failure, loads the associated plug-ins, and completes the azn_initialize() call.

The failure of the authorization API to contact the Tivoli Access Manager policy server can occur within one of the following contexts:

- The administration services have already been registered with the policy server. In this case, when the policy server becomes operational again, any administration API functions or pdadmin command lines that are handled by the administration service definitions will automatically succeed again.
- The administration service definitions have not been registered with the policy server, because this was the first attempt to register them. In this case, the authorization API must be re-initialized so that it can register the administration service definitions. Once contact with the policy server succeeds, the administration API functions and pdadmin commands will succeed.
- The authorization API has already registered the administration services with the policy server. However, the service definitions being registered now differ from definitions that are already registered.

In this case, the new service definitions have not successfully registered, and the administration API functions and pdadmin commands will fail once the policy server starts. The authorization API needs to be restarted. This causes the authorization API to be re-initialized, which results in the policy server registering the new service definitions and deleting the old ones.

**Major errors from administration service functions**

The authorization service APIs are not invoked directly by the end user. They are invoked by policy server in response to certain administration API functions or pdadmin commands. The administration service plug-ins can return the major error codes specified for the authorization API, as well as those specified generically for all authorization service plug-ins.
The authorization service APIs can also return the AZN_S_SVC_ADMIN_* major status codes that are generic for all Authorization administration service plug-ins. The authorization API forwards these return codes to the policy server, which returns them to the administration API or pdadmin command. The final return code sent to the end-user conforms to the error codes returned by the administration API functions or the pdadmin commands.

**Minor errors from administration service functions**

The administration service functions conform to the authorization services plug-in model for using azn_util_errorcode() to return plug-in specific error return codes within the 16-bit minor error code portion of azn_status_t. The returned minor error code must be a valid Tivoli Access Manager minor error code, in order for the Tivoli Access Manager policy server to generate a message for it. If the returned minor code is invalid, an error message of unknown error is returned to the pdadmin command or administration API function that issued the request.

For more information on authorization service minor error codes, see “Minor error codes” on page 65.

**Error codes specific to an authorization services plug-in**

Administration service plug-ins can also return implementation-specific return codes in the outdata output parameter for the azn_admin_* calls. For example, the functions can return status codes of type unsigned long by using the azn_admin_svc_results() API to add unsigned long values to a pre-defined attribute in the outdata attribute list.

The administration service functions can return results strings (which are displayed at the pdadmin CLI) as values for the azn_admin_svc_results attribute of the outdata attribute list. These results strings are forwarded by the policy server to the pdadmin command line interface or to the application using the administration API functions.

The administration service functions can also return other attributes of their choice. These attributes are added to the outdata attribute list and directly forwarded to the caller of the administration API.

**Deploying an administration service plug-in**

When you have developed your administration service plug-in, and are ready to deploy it, complete the following steps:

- Run svrsslcfg for the authorization API application. Specify the appropriate input parameters.

  For more information on the svrsslcfg command line utility, see the IBM Tivoli Access Manager Command Reference.

- When configuring the authorization API calling application, use pdadmin or the administration API to create the application-specific portion of the protected object namespace.

- Create the appropriate protected objects under the application-specific portion of the protected object namespace.

- Register the administration service plug-ins for the protected objects created in the previous step. See “Configuring administration service plug-ins” on page 83.
Chapter 9. Implementing external authorization service plug-ins

This sections consists of the following topics:

- “Introducing the external authorization service”
- “Understanding the external authorization service” on page 90
- “Configuring an external authorization service plug-in” on page 94
- “Initializing and shutting down external authorization service plug-ins” on page 95
- “Obtaining an authorization decision” on page 96
- “Error codes” on page 97

Introducing the external authorization service

The external authorization service is a modular authorization service plug-in that allows system designers to supplement IBM Tivoli Access Manager (Tivoli Access Manager) authorization with their own authorization models. Developers can build a shared library object that implements an external authorization service. You can configure the authorization API client to use this plug-in by ensuring that the appropriate interfaces are exposed within the library. The external authorization service is implemented as an authorization API client so that the plug-in can manipulate authorization API data structures such as the credentials and attribute lists.

External authorization service plug-ins are called only by local-mode authorization API client. Remote-mode authorization API clients do not have a local authorization server and thus cannot call out to an external authorization service. The configuration of external authorization service plug-ins is applicable only to local-mode authorization API clients.

The external authorization service plug-in model includes a Distributed Computing Environment (DCE) Remote Procedure Call (RPC) based external authorization service, for backwards compatibility with previous versions of the Tivoli SecureWay Policy Director product. This plug-in transforms the parameters of the new plug-in interface into those used by the RPC interface in Policy Director Version 3.7 and earlier. Clients that want to consult an RPC-based external authorization service need to deploy a DCE client on the client machine.

Configuration of external authorization service plug-ins is performed in the same way as other authorization service plug-ins. Initialization settings are specified either through a configuration file or programmatically through the initialization attribute list of the azn_initialize() function. For legacy DCE-RPC external authorization servers, the configuration that was previously stored in the authorization database is now configured for each local mode authorization API client. This configuration information is passed to the plug-in as parameters that are specific to the individual plug-in.

Initialization settings consist of a service definition that specifies a policy trigger for which the external authorization service is invoked, a weighting that is assigned in the access decision process to the particular external authorization service, and the location of the dynamically-loadable library module that performs the...
authorization work specific to the external authorization service. The concepts of policy triggers and weightings are described later in this section.

Each external authorization service plug-in must expose three interfaces to the authorization service:

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

The external authorization service plug-ins follow the service plug-in model for returning error messages by combining major error codes and application-specific minor error codes to produce valid error codes that can be returned to the calling applications.

---

**Understanding the external authorization service**

This section consists of the following topics:

- "External authorization service architecture”
- “Policy triggers” on page 92
- “Weightings” on page 93

**External authorization service architecture**

The external authorization service architecture is derived directly from the general authorization service plug-in architecture.
The calling application is represented by the *authorization API application* object in the above diagram. The calling application sends access decision information (ADI) to the authorization engine to request an authorization decision. The authorization engine first makes an access decision based on the ADI that was passed-in and on the credentials of the requesting user. The decision is assigned an integer value called a *weighting*.

Next, the Access Decision Combinator is called. It returns a weighted access decision result, based on calls to all applicable external authorization service plug-ins. The Combinator is responsible for identifying and invoking each external authorization service that is applicable to the authorization decision request. The Combinator examines the policy actions contained in the Access Control List (ACL) that has requested the decision, and combines these with the protected object policy (POP) attributes from the applicable POP. These actions and attributes are combined to create a *policy trigger*.

The Combinator uses this policy trigger information to call the service dispatcher to identify a list of external authorization service plug-ins which must be called.
The Combinator then calls each external authorization service in turn. The permission value returned from each call is multiplied by the weighting for the specific EAS. The weighted result is then passed to the Tivoli Access Manager authorization engine.

The service dispatcher for the external authorization service plug-ins manages the location, configuration, and loading of the available EAS plug-ins. The service dispatcher handles EAS plug-ins in the same manner as other types of plug-ins, such as entitlement services, administration services, and credentials modification services.

**Policy triggers**

*Policy triggers* refer to the set of policy circumstances that trigger a particular EAS to be invoked for any particular access decision. In prior versions of Tivoli Access Manager, only a specific action or operation in an access control list (ACL) could serve as a policy trigger. For version 3.8 and later of Tivoli Access Manager, the policy triggers have been expanded to include more ACL permissions and POPs.

ACL based policy triggers have been expanded to include *action sets* in addition to single action flags. For example, you can set the trigger to `rx` to have the EAS called whenever this bit mask appears in the primary action group of the ACL under evaluation. Similarly, a trigger of `Printer[wx]` invokes an EAS when both the `w` and `x` permission bits of the user-defined `Printer` action group are present together in the ACL that is being evaluated. The condition is triggered when *all* actions in the set are present in the current ACL for the access decision.

POP-based policy triggers allow an EAS to be called for one or more specific protected objects within the protected object space. The POP trigger is defined as an extended attribute within the POP. Its value is set to the specific policy trigger string needed to trigger the target EAS. The policy trigger is simply the service ID of the target EAS.

When the extended attribute value is not NULL, the POP is evaluated as part of the authorization process in the Tivoli Access Manager authorization engine. The attribute name to be used must be specifically for the purpose of EAS registration. The attribute name can have more than one value. This means that multiple policy triggers can be configured for the one POP. Each policy trigger is called in turn. The EAS are called in the order in which they were first added to the attribute.

**Configuring a POP policy trigger**

POP objects reserve the extended attribute name `eas-trigger` for the purpose of registering EAS triggers. New POP objects have no entry for this name by default. User must explicitly register an policy trigger with the POP through `pdadmin` or the administration API. To be valid, the extended attribute must contains one or more values that must match the policy trigger field in the service definition of at least one of the EAS modules that are loaded when the authorization API is initialized.

The following examples show the pdadmin commands for adding and removing policy triggers from POP objects.

```
padmin> pop show <pop-name> attribute eas-trigger

padmin> pop modify <pop-name> set attribute eas-trigger trigger-string

padmin> pop modify <pop-name> delete attribute eas-trigger trigger-string
```
By passing the trigger string as a parameter to the `set` command you can add a specific trigger from the list of configured triggers. Likewise, by passing the trigger string as a parameter to the `delete` command you can remove a specific trigger from the list of configured triggers.

You can also use the Tivoli Access Manager administration API to add and delete policy triggers within the POP extended attributes. In each of the cases shown in the table below, the caller passes the string `eas-trigger` as the `attr_key` input string.

<table>
<thead>
<tr>
<th>Administration API Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ivadmin_pop_attrget()</code></td>
<td>Get the EAS policy triggers for a POP</td>
</tr>
<tr>
<td><code>ivadmin_pop_attrput()</code></td>
<td>Set an EAS policy trigger for a POP</td>
</tr>
<tr>
<td><code>ivadmin_pop_attrdelkey()</code></td>
<td>Delete all the EAS policy triggers for a POP</td>
</tr>
<tr>
<td><code>ivadmin_pop_attrdelval()</code></td>
<td>Delete a specific EAS policy trigger for a POP</td>
</tr>
</tbody>
</table>

### Weightings

The Tivoli Access Manager authorization service provides the core authorization engine responsible for making authorization decisions. You can add one or more external authorization service plug-ins to provide additional authorization decision-making decisions. When you have more than one authorization decision-making process, you must specify the relative rank or priority that each decision carries. You can use the concept of a `weighting` to specify this rank or priority. For each external authorization service, you specify the appropriate weighting in the service definition that is used to configure the external authorization service.

The weight parameter is an unsigned `size_t` value and is optional. The default Tivoli Access Manager Authorization engine has a weight of 100. The weight of each EAS is relative to other external authorization services, and to the core authorization engine. When the weighting for an external authorization service is not specified a default value of 101 is assigned.

When the Tivoli Access Manager authorization engine or an EAS returns an `access permitted` decision, the value of the overall access decision is increased by the integer amount of the applicable weighting. When the authorization engine or an EAS returns an `access denied` decision, the value of the overall access decision is decreased by the integer amount of the applicable weighting.

When the value of the overall decision is greater than zero (0), the access request is approved. When the value of the overall decision is less than zero (0), the access request is denied.

When the value equals zero, the access decision is made based on the value returned by the Tivoli Access Manager authorization engine. Thus, the authorization engine takes precedence over the external authorization services when the sum total of the supplemental external authorization services’ decisions is not strong enough to override the decisions made by the core engine.

For example, when the core engine is given a weight of 100 and it returns an `access permitted` decision, the value of the result is +100. When an external authorization service with a weighting of 60 is then called, and returns an `access denied` decision,
then this final result value is modified to +40. If another EAS with weighting of 60 is called, and it also returns an access denied decision, then the final result is -20. In this case, access is denied because the overall result is less than zero.

Each external authorization service has the option of not participating in a specific access decision. The EAS can return a permission value of AZN_C_INDIFFERENT. When this occurs, the Tivoli Access Manager authorization service does not factor the weighting for the EAS into the final result.

### Configuring an external authorization service plug-in

You can configure an external authorization service plug-in either manually or programmatically. Manual configuration is accomplished by setting values in a configuration file. Programmatic configuration is accomplished by passing attributes to the administration API at API initialization. These methods correspond directly to the registration steps used by all types of authorization service plug-ins, as described in “Constructing a service definition” on page 58.

Configuration of an external authorization service plug-in is described in the following sections:

- "Using a configuration file entry"
- “Configuring an external authorization service programmatically” on page 95

#### Using a configuration file entry

External authorization service plug-ins are typically configured in aznapi.conf, under the following stanza:

```
[aznapi-external-authzn-services]
```

The external authorization services definition syntax is as follows:

\<policy-trigger\> = \<plug-in location\> \[-weight \<N\>\] \[& \<plug-in parameters\>\]

For example:

```
Printer:rxT = eas_plugin -weight 60 & -server barney
```

Or

```
webseal_pop_trigger = eas_plugin_2 -weight 70 & -hostname fred
```

The policy-trigger can be any string that is recognized as a valid key name. Stanza key names cannot contain white space or the open bracket "[“ and close bracket “]” characters. The bracket characters are used to define new stanza names. The policy-trigger is case sensitive for action set definitions because the actions themselves are case sensitive. However, the policy-trigger is case insensitive if the trigger is a POP attribute.

The first example above shows an action set trigger with a user-defined action group of Printer and the actions rxT within that group. To specify the primary action group you would specify only :rxT. This entry has an empty action group name. Any policy-trigger that does not contain a colon “:” character is considered to be a protected object policy (POP) attribute.

The second example above is for a POP attribute trigger called webseal_pop_trigger. When a POP that contains a reference to this string is passed to the Combinator, the appropriate external authorization service is called to take
part in the access decision. Note that the POP configuration must have been completed previously by the secure domain administrator, using the pdadmin command.

The plug-in location is the path name to the shared library or DLL module that contains the implementation of the plug-in that matches the policy trigger. The path name can be in a truncated form if the external authorization service is to be loaded by clients on multiple platforms. In this case, the service dispatcher searches for the plug-in using platform-specific prefixes and suffixes to match DLL names.

The weight parameter is an unsigned size_t value and is optional. The value signifies the weight that any decision returned by this external authorization service should be given in the entire decision process.

Optionally, the external authorization service can be passed additional initialization information in the form of arguments. The arguments must be preceded by the ampersand “&”. The authorization service takes the remainder of the string following the ampersand &, breaks the string up into white space separated tokens, and passes the tokens directly to the administration service’s initialization interface, azn_svc_initialize(), in the argv array parameter. The number of strings in the argv array is indicated by the argc function parameter. The optional arguments in the first example above are & -server -barney

Configuring an external authorization service programmatically

The authorization API header file ogauthzn.h contains a service definition attribute:

```
azn_string_t azn_initExtern_authzn_svc
```

The value of the attribute is a string of the format:

```
policy-trigger = plug-in_location [-weight N] [& plug-in_parameters]
```

For example:

```
POP_EAS_A = my_eas_pop_1 -weight 50 & -server fred
```

Or

```
:rmx = my_acl_ops_pop_1 -weight 60 & -server barney
```

Complete the following steps to use this service definition attribute to configure an external authorization service plug-in.

1. Use the azn_attrlist_add_entry() API to assign to the init_data attribute list as many values to the azn_initExtern_authzn_svc attribute as the number of Administrative service plug-ins needed. These values are expressed as strings, and need to conform to the external authorization service definition syntax specified in “Using a configuration file entry” on page 94.

2. Pass the init_data attribute to the azn_initialize() call to ensure that the specified Administration service plug-ins are loaded.

Initializing and shutting down external authorization service plug-ins

Each external authorization 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 external authorization service plug-ins should provide the standard functions. See the following sections:

- "Initialization and configuration of service plug-ins" on page 58
- "Shut down" on page 66

When the authorization API is initialized, the configuration file is parsed and each external authorization service plug-in listed in the [aznapi-external-authzn-service] stanza is loaded and resolved by the service plug-in dispatcher. The authorization API also recognizes the external authorization service plug-ins specified by the azn_initExternAuthznSvc attribute in the init_data attribute list that is passed as input to azn_initialize(). Each of those plug-ins is also loaded and resolved by the service plug-in dispatcher.

If a plug-in is configured incorrectly, or the plug-in module can’t be found, the azn_initialize() function returns an appropriate error.

When each external authorization service plug-in has been successfully loaded, the authorization service initialization interface is called with the parameters specified after the ampersand “&” character in the service definition.

For more information on initialization and shutdown of service plug-ins, see the reference pages for "azn_service_initialize()” on page 179 and "azn_service_shutdown()” on page 182.

---

**Obtaining an authorization decision**

The service dispatcher calls this interface to request an access decision from the EAS plug-in. This interface is called by the azn_decision_access_allowed() and azn_decision_access_allowed_ext() API interfaces and is expected to return both permission codes and error codes consistent with that required by authorization API interface specification for the those interfaces. For example:

```c
azn_status_t
azn_decision_access_allowed_ext(
    const azn_creds_h_t creds, /* input */
    const azn_string_t protected_resource, /* input */
    const azn_string_t operation, /* input */
    const azn_attrlist_h_t app_context, /* input */
    int *permission, /* output */
    azn_attrlist_h_t *permission_info /* output */
);
```

The azn_decision_access_allowed() and azn_decision_access_allowed_ext() functions return the same permission code values as their counterpart calls do in the authorization API. The EAS can return one of three possible permission values to the authorization engine:

- AZN_C_PERMITTED 0
- AZN_C_NOT_PERMITTED 1
- AZN_C_INDIFFERENT -1

The EAS returns AZN_C_INDIFFERENT when it does not care about the access decision in question. For example, this can occur when a minimum requirement for the input conditions was not met. Alternatively, the EAS may decide it does not have jurisdiction for the decision. In these cases, the Combinator ignores the return value from this EAS when calculating the result of the access decision.
The `permission_info` parameter should contain any decision-related information attributes that the EAS wants to return to the calling application. If an initialized attribute list handle is passed back to the service dispatcher, then the attributes are added to the `permission_info` list that is passed back to the caller.

Ensure that your implementation of `azn_decision_access_allowed_ext()` is thread-safe. Input parameters must not be modified. You can assume that the calling application will free the output parameters that are returned. This interface returns AZN_S_COMPLETE when successful, or an error when it fails.

### Error codes

The authorization service plug-in interface, including the EAS plug-in interface, defines a number of major error codes and a minor error code mask that an EAS must use to construct error codes to return to the calling application. The error codes returned by the plug-in interfaces are not parsed or modified by the authorization API runtime functions. These codes are passed back unmodified to the calling application. Thus, the error code that is returned must be able to be parsed into its major and minor error code components by the calling application. The calling application uses `azn_error_major()` and `azn_error_minor()` to complete the parsing. To properly construct the error codes, the authorization API provides a utility function that enables developers of third-party extensions, (including service plugs-in such as the EAS) to the authorization API to construct valid error codes to return to application programmers. This function is `azn_util_errcode()`. This function takes a major and minor integer error code value and converts them into an `azn_status_t` value. The minor error code returned must be defined in accordance with the rules below.

### Major error codes

External authorization service plug-ins should return all applicable major error codes that are defined for returning status from the `azn_decision_access_allowed()` and `azn_decision_access_allowed_ext()` interfaces. Note that not all error codes will apply to all plug-in services.

The following table shows major error codes:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZN_S_INVALID_CREDS_HDL</td>
<td>The credentials handle supplied is invalid</td>
</tr>
<tr>
<td>AZN_S_INVALID_PROTECTED_RESOURCE</td>
<td>The <code>protected_resource</code> string supplied is invalid.</td>
</tr>
<tr>
<td>AZN_S_INVALID_OPERATION</td>
<td>The <code>operation</code> string supplied is invalid.</td>
</tr>
<tr>
<td>AZN_S_INVALID_EAS_ACL_TRIGGER</td>
<td>The EAS ACL policy trigger specified is invalid.</td>
</tr>
<tr>
<td>AZN_S_INVALID_EAS_POP_TRIGGER</td>
<td>The EAS POP policy trigger specified is invalid.</td>
</tr>
<tr>
<td>AZN_S_INVALID_EAS_WEIGHTING</td>
<td>The EAS weighting specified is invalid. An absolute <code>size_t</code> value is required.</td>
</tr>
<tr>
<td>AZN_S_UNKNOWN_EAS_SVC_PARAMETER</td>
<td></td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AZN_S_INVALID_APP_CONTEXT_HDL</td>
<td>The attribute list handle for the application context is invalid.</td>
</tr>
<tr>
<td>AZN_S_INVALID_PERMISSION_REF</td>
<td>The integer pointer for the permission parameter is invalid.</td>
</tr>
<tr>
<td>AZN_S_FAILURE</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 azn_error_minor().</td>
</tr>
</tbody>
</table>

The following additional major error codes are defined for authorization API service modules in ogauthzn.h. Some of the following error codes are returned only by the service dispatcher, and some are returned only by the service plug-in. The returning entity is noted in the description of each error code.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZN_S_SVC_DEFINITION_ERROR</td>
<td>Returned by the service dispatcher when an error has been found in the</td>
</tr>
<tr>
<td></td>
<td>service definition in the authorization API configuration file.</td>
</tr>
<tr>
<td>AZN_S_SVC_NOT_FOUND</td>
<td>Returned by the service dispatcher when an error occurs either while</td>
</tr>
<tr>
<td></td>
<td>locating or loading an authorization service plug-in.</td>
</tr>
<tr>
<td>AZN_S_SVC_INITIALIZER_NOT_FOUND</td>
<td>Returned by the service dispatcher when an error occurs either locating or</td>
</tr>
<tr>
<td>AZN_S_SVC_SHUTDOWN_NOT_FOUND</td>
<td>loading a service interface within a particular plug-in. For example, this</td>
</tr>
<tr>
<td>AZN_S_SVC_EAS_FUNC_NOT_FOUND</td>
<td>is returned by the dispatcher if the azn_service_initialize() interface was</td>
</tr>
<tr>
<td></td>
<td>not found in the loaded plug-in.</td>
</tr>
<tr>
<td>AZN_S_SVC_INIT_FAILED</td>
<td>Returned by the plug-in when an error occurs while it is initializing.</td>
</tr>
<tr>
<td>AZN_S_SVC_SHUTDOWN_FAILED</td>
<td>Returned by the plug-in when an error occurs while it is shutting down.</td>
</tr>
<tr>
<td>AZN_S_SVC_AUTHORIZATION_FAILED</td>
<td>The calling application does not possess the authority required to invoke</td>
</tr>
<tr>
<td></td>
<td>the services of this plug-in.</td>
</tr>
</tbody>
</table>

**Minor error codes**

Minor error codes are encoded by the authorization API using a table of known message catalogue prefixes. All Tivoli Access Manager messages are composed of a prefix value and the specific error code within that message set. When the authorization API encodes a minor error code, using azn_util_errcode(), the prefix is removed from the minor code and cross-referenced with a table of known prefixes. When the prefix is identified, it is converted into a mask, which is then
inserted into the major error portion of the azn_status_t code returned. The reverse operation is performed by the azn_error_minor() function to rebuild the original minor error code.

To enable minor error codes returned by authorization API services to be properly encoded by azn_util_errcode() an appropriate prefix must be defined for authorization API service plug-ins. The same prefix is shared by all services of the same type. The prefix for all external authorization services is defined in ogauthzn.h as follows:

extern unsigned int azn_eas_svc_err_prefix;

Define error messages for your external authorization service by using the azn_eas_svc_err_prefix. For example, minor error codes for an external authorization service plug-in are defined in the interface file as follows:

#define EAS_SVC_CORE_DUMP(azn_eas_svc_err_prefix | 0x1)
#define EAS_SVC_INVALID_ATTRIBUTE(azn_eas_svc_err_prefix | 0x2)
#define EAS_SVC_BAD_FILENAME(azn_eas_svc_err_prefix | 0x3)
## Appendix A. Authorization API reference

This section contains the following reference pages:

- `azn_attrlist_add_entry()`
- `azn_attrlist_add_entry_buffer()`
- “`azn_attrlist_add_entry_pobj()`” on page 105
- “`azn_attrlist_add_entry_ulong()`” on page 106
- “`azn_attrlist_copy()`” on page 107
- “`azn_attrlist_create()`” on page 108
- “`azn_attrlist_delete()`” on page 109
- “`azn_attrlist_delete_entry()`” on page 110
- “`azn_attrlist_get_entry_buffer_value()`” on page 111
- “`azn_attrlist_get_entry_pobj_value()`” on page 113
- “`azn_attrlist_get_entry_string_value()`” on page 114
- “`azn_attrlist_get_entry_ulong_value()`” on page 116
- “`azn_attrlist_get_names()`” on page 117
- “`azn_attrlist_name_get_num()`” on page 118
- “`azn_creds_combine()`” on page 119
- “`azn_creds_copy()`” on page 121
- “`azn_creds_create()`” on page 122
- “`azn_creds_delete()`” on page 123
- “`azn_creds_equal()`” on page 124
- “`azn_creds_for_subject()`” on page 125
- “`azn_creds_get_attr_value_string()`” on page 127
- “`azn_creds_get_attrlist_for_subject()`” on page 128
- “`azn_creds_get_pac()`” on page 130
- “`azn_creds_modify()`” on page 132
- “`azn_creds_num_of_subjects()`” on page 135
- “`azn_creds_set_attr_value_string()`” on page 137
- “`azn_decision_access_allowed()`” on page 139
- “`azn_decision_access_allowed_ext()`” on page 141
- “`azn_entitlement_get_entitlements()`” on page 144
- “`azn_error_get_string()`” on page 146
- “`azn_error_major()`” on page 147
- “`azn_error_minor()`” on page 148
- “`azn_error_minor_get_string()`” on page 149
- “`azn_id_get_creds()`” on page 150
- “`azn_initialize()`” on page 152
- “`azn_pac_get_creds()`” on page 155
- “`azn_release_buffer()`” on page 157
- “`azn_release_pobj()`” on page 158
- “`azn_release_string()`” on page 159
- “`azn_release_strings()`” on page 160
• "azn_shutdown()" on page 161
• "azn_util_strerror()" on page 162
• "azn_util_handle_is_valid()" on page 163
• "azn_util_password_authenticate()" on page 164
• "azn_util_password_change()" on page 166
**azn_attrlist_add_entry()**

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

**Syntax**

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

**Parameters**

**Input**

- **attr_list**
  Handle to an attribute list. The attribute list handle can be one of the following:
  - An attribute list handle for a new, empty attribute list, which has been initialized by a call to `azn_attrlist_create()`.
  - An attribute list handle for an existing, non-empty attribute list which contains valid data.

- **attr_name**
  Attribute name of the entry to be added.

- **string_value**
  String attribute value to be added.

**Description**

This function adds a string value to the attribute `attr_name` within the attribute list `attr_list`. If the attribute `attr_name` already exists in the list then the value is appended to the existing list of values for `attr_name`. The values under `attr_name` are stored in the order in which they were inserted into the list. There is no checking performed upon values in the list; therefore duplicate values are permitted.

**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(
    const azn_attrlist_h_t attr_list,
    const azn_string_t attr_name,
    const azn_buffer_t buffer_value
);
```

Parameters

Input

- `attr_list`: Handle to an attribute list. The attribute list handle can be one of the following:
  - An attribute list handle for a new, empty attribute list, which has been initialized by a call to `azn_attrlist_create()`.
  - An attribute list handle for an existing, non-empty attribute list which contains valid data.

- `attr_name`: Attribute name of the entry to be added.

- `buffer_value`: Buffer attribute value to be added.

Description

This function adds a buffer value to the attribute `attr_name` within the attribute list `attr_list`. If the attribute `attr_name` already exists in the list then the value is appended to the existing list of values for `attr_name`. The values under `attr_name` are stored in the order in which they were inserted into the list. There is no checking performed upon values in the list; therefore duplicate values are permitted.

Return Values

If successful, the function 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_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_add_entry_pobj()

Adds the name of a protected object entry to the attribute list.

Syntax

```c
azn_status_t azn_attrlist_add_entry_pobj(
    const azn_attrlist_h_t attr_list,
    const azn_string_t attr_name,
    const azn_pobj_t pobj_value
);
```

Parameters

Input

- `attr_list`  
  Handle to an attribute list. The attribute list handle can be one of the following:
  - An attribute list handle for a new, empty attribute list, which has been initialized by a call to `azn_attrlist_create()`.
  - An attribute list handle for an existing, non-empty attribute list which contains valid data.

- `attr_name`  
  Name of the attribute to which the value is to be added.

- `pobj_value`  
  Protected object attribute value to be added.

Description

This function adds a protected object value to the attribute `attr_name` within the attribute list `attr_list`. If the attribute `attr_name` already exists in the list then the value is appended to the existing list of values for `attr_name`. The values under `attr_name` are stored in the order in which they were inserted into the list. There is no checking performed upon values in the list; therefore duplicate values are permitted.

Return Values

When successful, the function returns AZN_S_COMPLETE.

When unsuccessful, the function returns one of the following major error codes:

- `AZN_S_INVALID_ATTRLIST_HDL`  
  The attribute list is invalid.

- `AZN_S_INVALID_ATTR_NAME`  
  The attribute list name is invalid.

- `AZN_S_INVALID_POBJ`  
  The protected object value is NULL.
**azn_attrlist_add_entry_ulong()**

Adds an unsigned long entry to the attribute list.

**Syntax**

```c
azn_status_t azn_attrlist_add_entry_ulong(
    const azn_attrlist_h_t attr_list,
    const azn_string_t attr_name,
    const azn_ulong_t ulong_value
);
```

**Parameters**

**Input**

- `attr_list`
  Handle to an attribute list. The attribute list handle can be one of the following:
  - An attribute list handle for a new, empty attribute list, which has been initialized by a call to `azn_attrlist_create()`.
  - An attribute list handle for an existing, non-empty attribute list which contains valid data.

- `attr_name`
  Iterate name to which the value is to be added.

- `ulong_value`
  Ulong attribute value to be added.

**Description**

This function adds a ulong value to the attribute `attr_name` within the attribute list `attr_list`. If the attribute `attr_name` already exists in the list then the value is appended to the existing list of values for `attr_name`. The values under `attr_name` are stored in the order in which they were inserted into the list. There is no checking performed upon values in the list; therefore duplicate values are permitted.

**Return Values**

When successful, the function returns AZN_S_COMPLETE.

When unsuccessful, the function returns one of the following major error codes:
- **AZN_S_INVALID_ATTRLIST_HDL**
  The attribute list is invalid.
- **AZN_S_INVALID_ATTR_NAME**
  The attribute list name is invalid.
azn_attrlist_copy()

Copies a valid attribute list to a new attribute list.

Syntax

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

Parameters

**Input**

`attr_list`

Handle to an attribute list. The attribute list handle can be one of the following:

- An attribute list handle for a new, empty attribute list, which has been initialized by a call to `azn_attrlist_create()`.
- An attribute list handle for an existing, non-empty attribute list which contains valid data.

Description

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

Return Values

If unsuccessful, the function will return AZN_S_INVALID_HANDLE.
**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 an `azn_attrlist_h_t` variable that will contain the handle to the new attribute list upon successful completion.

**Description**

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

**Note:** Alternatively, when you are declaring an attribute list handle to for use as an authorization API output parameter you can declare the handle and assign it the value `AZN_C_INVALID_HANDLE`, which indicates that it is uninitialized. In this case when the authorization API function returns data in the parameter, it will automatically create an attribute list and return the handle in the output parameter.

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

**Return Values**

If successful, the function will return `AZN_S_COMPLETE`.

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

- **AZN_S_COMPLETE**
  Successful completion.
- **AZN_S_INVALID_ATTRLIST_HDL**
  Attribute list handle is invalid.
- **AZN_S_FAILURE**
  An error or failure has occurred. Use `azn_error_minor()` to derive specific minor error codes from the returned status code.
azn_attrlist_delete()

Deletes the attribute list associated with the attribute list handle.

Syntax

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

Parameters

Input

old_attr_list
  Pointer to the attribute list handle for the attribute list to be deleted. The attribute list handle can be one of the following:
  • An attribute list handle for a new, empty attribute list, which has been initialized by a call to azn_attrlist_create().
  • An attribute list handle for an existing, non-empty attribute list which contains valid data.

Output

old_attr_list
  NULL pointer to an attribute list handle that is invalid upon return.

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_attrlist_delete_entry()

Deletes the specified attribute associated with the attribute list handle.

Syntax

```c
azn_status_t
azn_attrlist_delete_entry(
    const azn_attrlist_h_t attr_list,
    const azn_string_t attr_name
);
```

Parameters

Input

`attr_list`

Handle to an attribute list. The attribute list handle should be a handle for an existing, non-empty attribute list which contains valid data.

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

**Parameters**

**Input**

- `attr_list`
  Handle to an attribute list. The attribute list handle should be a handle for an existing, non-empty attribute list which contains valid data.

- `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 `azn_buffer_t` variable that will contain the address of the buffer data upon successful completion of the routine.

**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`

Appendix A. Authorization API reference 111
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_pobj_value()

Returns a single specified value attribute for a name attribute that has one or more values that contain protected object information.

Syntax

```c
azn_status_t
azn_attrlist_get_entry_pobj_value(
    const azn_attrlist_h_t attr_list,
    const azn_string_t attr_name,
    const unsigned int value_index,
    azn_pobj_t *pobj_value
);
```

Parameters

**Input**

- `attr_list`
  Handle to an attribute list. The attribute list handle should be a handle for an existing, non-empty attribute list which contains valid data.

- `attr_name`
  Name of the attribute from which the value needs to be returned.

- `value_index`
  Index within the entry of the value attribute to be returned

**Output**

- `pobj_value`
  Pointer to an `azn_pobj_t` variable that will contain the address of the protected object information upon successful completion of the routine.

Description

Returns a single specified value attribute for a name attribute that has one or more values that contain protected object information.

Return Values

When successful, the function returns `AZN_S_COMPLETE`

When unsuccessful, the function returns one of the following major error codes:
- `AZN_S_INVALID_ATTRLIST_HDL`
  The attribute list is invalid.
- `AZN_S_INVALID_ATTR_NAME`
  The attribute name is NULL, or the attribute name is not found.
- `AZN_S_INVALID_INDEX`
  The index `value_index` is invalid.
- `AZN_S_INVALID_POBJ_REF`
  The `pobj_value` pointer is NULL.
- `AZN_S_ATTR_VALUE_NOT_POBJ_TYPE`
  The value type at the specified index is not of type `pobj`.

Appendix A. Authorization API reference  113
azn_attrlist_get_entry_string_value()

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

Syntax

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

Parameters

Input

attr_list
Handle to an attribute list. The attribute list handle should be a handle for an existing, non-empty attribute list which contains valid data.

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 an azn_string_t variable that will contain the string data upon successful completion of the routine.

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_entry_ulong_value()

Returns a single specified value for a name attribute that has one or more values that are of type unsigned long.

Syntax

```
azn_status_t
azn_attrlist_get_entry_ulong_value(
    const azn_attrlist_h_t attr_list,
    const azn_string_t attr_name,
    const unsigned int value_index,
    azn_ulong_t *ulong_value
);
```

Parameters

**Input**

- **attr_list**
  Handle to an attribute list. The attribute list handle should be a handle for an existing, non-empty attribute list which contains valid data.

- **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**

- **ulong_value**
  Pointer to an unsigned long variable that holds the value of the returned attribute.

Description

Returns a single specified value for a name attribute that has one or more values that are of type unsigned long.

Return Values

When successful, the function returns AZN_S_COMPLETE.

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

- **AZN_S_INVALID_ATTRLIST_HDL**
  Attribute list handle is invalid.

- **AZN_S_INVALID_ATTR_NAME**
  Attribute name is invalid or the attribute name is not found.

- **AZN_S_INVALID_INDEX**
  The variable value_index is not valid.

- **AZN_S_ATTR_VALUE_NOT_ULONG_TYPE**
  The value attributes of this entry are not of type ulong.
**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(
    const azn_attrlist_h_t attr_list,
    azn_string_t *attr_names[]
);
```

**Parameters**

**Input**

- **attr_list**
  Handle to an attribute list. The attribute list handle should be a handle for an existing, non-empty attribute list which contains valid data.

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

Description

Input

- **attr_list**
  Handle to an attribute list. The attribute list handle should be a handle for an existing, non-empty attribute list which contains valid data.

- **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_creds_combine()**

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

**Syntax**

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

**Parameters**

**Input**

- **creds**
  Handle to an credentials chain whose first indexed entry is the credential of the initiator of the request. The credential handle should be a handle for an existing, non-empty credential which contains valid data.

- **creds_to_add**
  Handle to the credentials chain to be appended to creds. The credential handle should be a handle for an existing, non-empty credential which contains valid data.

**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.

  The credential handle pointer passed as the combined_creds parameter can point to one of the following:
  - A credential handle for a new, valid, empty credential.
  - A credential handle initialized to AZN_C_INVALID_HANDLE.

**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.

**Note:** This call uses the new credential handle combined_creds. When you declare a new credential handle, always initialize it to AZN_C_INVALID_HANDLE before using it.
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 pdbaclmsg.h for a complete list of minor error codes that describe access control problems.
**azn_creds_copy()**

Creates a reference copy of the target credential.

**Syntax**

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

**Parameters**

**Input**

`creds`

Handle to a credential. The handle can be one of the following:

- A credential handle for a new, empty credential, which has been initialized by a call to `azn_creds_create()`.
- A handle for an existing, non-empty credential which contains valid data.

**Description**

This function creates a reference copy of the target credential. This function does not make a new copy of the original credential. It creates a duplicate credentials handle to the original credential. Use `azn_id_get_creds()` to create an entirely new copy of a credential.

**Return Values**

If unsuccessful, the function will return AZN_S_INVALID_HANDLE. Otherwise it will return another handle to the credential `creds`. 
**azn_creds_create()**

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

**Syntax**

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

**Parameters**

**Output**

*creds*

A pointer to an `azn_creds_h_t` variable that will contain the handle to the new credential chain upon successful completion.

**Description**

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

**Note:** Alternatively, when you are declaring an attribute list handle to for use as an authorization API output parameter you can declare the handle and assign it the value `AZN_C_INVALID_HANDLE`, which indicates that it is uninitialized. In this case when the authorization API function returns data in the parameter, it will automatically create an attribute list and return the handle in the output parameter.

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 `pdbaclmsg.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 credential handle for the credential chain to be deleted. The handle should be a handle for an existing, non-empty credentials chain which contains valid data.

Output

creds

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 `pdbaclmsg.h` for a complete list of minor error codes that describe access control problems.
azn_creds_equal()

Compares the contents of two credentials.

Syntax

```c
azn_status_t azn_creds_equal(
    const azn_creds_h_t cred1,
    const azn_creds_h_t cred2,
    azn_boolean_t *is_equal
);
```

Parameters

**Input**

*cred1*

Handle to a credential. The credential handle should be a handle for an existing, non-empty credential which contains valid data.

*cred2*

Handle to a credential. The credential handle should be a handle for an existing, non-empty credential which contains valid data.

**Output**

*is_equal*

A pointer to an `azn_boolean_t` variable that will receive the boolean result of this call. `azn_creds_equal()` returns a boolean value of `true` or `false`, indicating if the two compared credentials are equal. This value is `true` when the credentials are equal, and `false` when the credentials are not equal.

Description

This function compares the contents of two credentials. The function returns `true` when the credentials are identical and `false` when the credentials differ. The value is returned in the output parameter *is_equal*.

Return Values

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.
**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(
    const azn_creds_h_t creds,
    const 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.

  The credentials handle should be a handle for an existing, non-empty credentials chain which contains valid data.

- **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`.

**Note:** This call uses the new credential handle `new_creds`. When you declare a new credential handle, always initialize it to `AZN_C_INVALID_HANDLE` before using it.

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 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 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 `pdbaclmsg.h` for a complete list of minor error codes that describe access control problems.
**azn_creds_get_attr_value_string()**

Obtains the string value of the specified attribute in the specified credential

**Syntax**

```c
azn_status_t azn_creds_get_attr_value_string(
    const azn_creds_h_t creds,
    const unsigned int subject_index,
    const azn_string_t attr_name,
    azn_string_t *attr_value
);
```

**Parameters**

**Input**

- **creds**
  
  Handle to a credential list.

  The credentials handle should be a handle for an existing, non-empty credentials chain which contains valid data.

- **subject_index**

  The index of the specified credential with the credential chain for which the attribute value is to be retrieved.

- **attr_name**

  The attribute name.

**Output**

- **attr_value**

  Pointer to an azn_string_t variable that will contain the address of the attribute string value upon successful completion of the routine.

**Description**

This function retrieves attribute values from a user credential. This function accesses the attribute list of the specified credential directly and gets the string value of the specified attribute from it.

**Return Values**

When successful, the function returns AZN_S_COMPLETE.

When 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_STRING_REF**
  
  The specified attribute name is invalid.

- **AZN_S_INVALID_CREDS_HDL**

  The specified credentials handle is invalid.

- **AZN_S_INVALID_SUBJECT_INDEX**

  The index into the credentials chain is invalid.
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 (  
    const azn_creds_h_t creds,  
    const unsigned int subject_index,  
    azn_attrlist_h_t *creds_attrlist
);
```

Parameters

Input

`creds`
Handle to a credentials chain.

The credentials handle should be a handle for an existing, non-empty credentials chain which contains valid data.

`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. The attribute list handle pointer passed as the `creds_attrlist` parameter can point to one of the following:

- An attribute list handle that has been initialized to AZN_C_INVALID_HANDLE.
- An attribute list handle for a new, empty attribute list, which has been initialized by a call to `azn_attrlist_create()`.
- An attribute list handle for an existing, non-empty attribute list which contains valid data. In this case, attribute list data is appended to the existing attribute list.

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`.

**Note:** This function uses a new credentials handle, `creds`. When declaring a new credential handle or attribute list handle always initialize it to `AZN_C_INVALID_HANDLE` before using it.

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 pdbaclmsg.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(
    const azn_creds_h_t creds,
    const 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. The credentials handle should be a handle for an existing, non-empty credentials chain which contains valid data.

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

**Output**

*pac*  
Pointer to an `azn_buffer_t` variable that will contain the address of the buffer data upon successful completion of the routine.

**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`. 

---

**IBM Tivoli Access Manager: Authorization C API Developer’s Reference**
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 pdbaclmsg.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(
    const azn_creds_h_t creds,
    const azn_string_t mod_svc_id,
    const azn_attrlist_h_t mod_info,
    azn_creds_h_t *new_creds
);
```

**Parameters**

**Input**

- **creds**
  Handle to the authorization credentials chain to be modified.
  The credentials handle should be a handle for an existing, non-empty credentials chain which contains valid data.

- **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. The attribute list handle can be one of the following:
  - An attribute list handle that has been initialized to AZN_C_INVALID_HANDLE.
  - An attribute list handle for a new, empty attribute list, which has been initialized by a call to `azn_attrlist_create()`.
  - An attribute list handle for an existing, non-empty attribute list which contains valid data. In this case, attribute list data is appended to the existing attribute list.

**Output**

- **new_creds**
  Pointer to a handle to a credentials chain that contains the modified credentials chain upon return. The credentials handle pointer passed as the `new_creds` parameter can point to one of the following:
    - A credentials handle that has been initialized to AZN_C_INVALID_HANDLE. Note that this will result in an empty output credential.
    - A credentials handle for a new, empty credential, which has been initialized by a call to `azn_creds_create()`. Note that this will result in an empty output credential.

**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*.

The following credential attributes are considered readonly and can not be modified by azn_creds_modify(). If any of these attributes are specified in the *mod_info* input attribute list, they are ignored:

- *azn_cred_principal_uuid*
- *azn_cred_principal_name*
- *azn_cred_version*
- *azn_cred_mech_id*
- *azn_cred_group_uuids*
- *azn_cred_group_names*
- *azn_cred_authzn_id*

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.

**Note:** This function uses a new credential handle, *new_creds*. When declaring a new credential handle, always initialize it to AZN_C_INVALID_HANDLE before using it.

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**

Appendix A. Authorization API reference 133
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 pdbaclmsg.h for a complete list of minor error codes that describe access control problems.
azn_creds_num_of_subjects()

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

Syntax

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

Parameters

**Input**

*creds*

Handle to a credentials chain.

The credentials handle should be a handle for an existing, non-empty credentials chain which contains valid data.

**Output**

*num_of_subjects*

Pointer to an unsigned int variable that will contain the number of individual credentials within the credential chain upon successful completion of the routine. Note that if the credentials handle passed in as an input parameter is invalid (AZN_C_INVALID_HANDLE) or points to an empty credentials list, an error will be generated and `num_of_subjects` will be set to 0.

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 pdbaclmsg.h for a complete list of minor error codes that describe access control problems.
azn_creds_set_attr_value_string()

Sets the value of the specified attribute in the specified user credential.

**Syntax**

```c
azn_status_t azn_creds_set_attr_value_string
    const azn_creds_h_t creds,
    const unsigned int subject_index,
    const azn_string_t attr_name,
    const azn_string_t attr_value
);
```

**Parameters**

**Input**

*creds*

Handle to the credentials chain.
The credentials handle should be a handle for an existing, non-empty credentials chain which contains valid data.

*subject_index*

The index of the specified credential within the credential chain.

*attr_name*

The name of the attribute to be modified.

*attr_value*

The string value to be assigned to the attribute.

**Description**

Use this function to modify the attribute values in a user credential. This function edits the attribute list of the specified credential and sets the specified attribute to the specified string value.

The following credential attributes are considered *readonly* and must not be modified using this API.

- `azn_cred_principal_uuid`
- `azn_cred_principal_name`
- `azn_cred_version`
- `azn_cred_mech_id`
- `azn_cred_group_uuids`
- `azn_cred_group_names`
- `azn_cred_authzn_id`
- `azn_cred_ldap_dn`

**Return Values**

When successful, the function returns AZN_S_COMPLETE.

When 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_STRING_VALUE`
  The specified string value is invalid.
- `AZN_S_INVALID_CREDS_HDL`

The specified credentials handle is invalid.

- **AZN_S_INVALID_SUBJECT_INDEX**
  The index into the credentials chain is invalid.

- **AZN_S_ATTR_READONLY**
  Modification of the attribute is prohibited.
**azn_decision_access_allowed()**

Makes an access control decision.

**Syntax**

\[
\text{azn_status_t azn_decision_access_allowed(}
\text{\begin{tabular}{l}
const azn_creds_h_t \textit{creds}, \\
const azn_string_t \textit{protected_resource}, \\
const azn_string_t \textit{operation}, \\
const int *\textit{permission}
\end{tabular}}
\];
\]

**Parameters**

**Input**

\textit{creds}

Handle to the initiator's credential chain.

The credentials handle should be a handle for an existing, non-empty credentials chain which contains valid data.

\textit{protected_resource}

Name of the request's target.

\textit{operation}

Name of the requested operation.

**Output**

\textit{permission}

Pointer to an integer variable that will contain the appropriate permission code upon successful completion of the routine.

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

If additional information beyond a boolean result is needed, use \texttt{azn_decision_access_allowed_ext()}. 

**Description**

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

\texttt{azn_decision_access_allowed()} is semantically equivalent to \texttt{azn_decision_access_allowed_ext()} when app_context=NULL and permission_info=NULL.

**Return Values**

If successful, the function will return \texttt{AZN_S_COMPLETE}.

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

• \texttt{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 pdbaclmsg.h for a complete list of minor error codes that describe access control problems.
azn_decision_access_allowed_ext()

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

Syntax

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

Parameters

**Input**

*creds*
Handle to the initiator’s credentials chain.

The credentials handle should be a handle for an existing, non-empty credentials chain which contains valid data.

*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. The attribute list handle can be one of the following:

- An attribute list handle that has been initialized to AZN_C_INVALID_HANDLE.
- An attribute list handle for a new, empty attribute list, which has been initialized by a call to azn_attrlist_create().
- An attribute list handle for an existing, non-empty attribute list which contains valid data. In this case, attribute list data is appended to the existing attribute list.

*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*
Pointer to an integer variable that will contain the appropriate permission code upon successful completion of the routine.

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 information contained in the attribute list is added to the list when the attribute list handle is one of the following:

- An attribute list handle for a new, empty attribute list, which has been initialized by a call to `azn_attrlist_create()`.
- An attribute list handle for an existing, non-empty attribute list which contains valid data.

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.”

For more information, see “Enabling the return of permission information” on page 30.

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.

Note: This function uses a new attribute list handle, `permission_info`. When declaring a new attribute list handle, always initialize it to `AZN_C_INVALID_HANDLE` before using it.

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 pdbaclmsg.h for a complete list of minor error codes that describe access control problems.
**azn_entitlement_get_entitlements()**

Returns entitlements of an initiator

**Syntax**

```c
azn_status_t
azn_entitlement_get_entitlements(
    const azn_creds_h_t creds,
    const azn_string_t entitlements_svc_id,
    const azn_attrlist_h_t app_context,
    azn_attrlist_h_t *entitlements
);
```

**Parameters**

**Input**

**creds**
Handle to the credentials of the subject whose entitlements are to be returned.

The credentials handle should be a handle for an existing, non-empty credentials chain which contains valid data.

**entitlements_svc_id**
The id of the entitlements service to be used

**app_context**
Handle to an attribute list containing application-specific or entitlements-service-specific context information. A NULL value should be used if no application state is passed. The attribute list handle can be one of the following:

- An attribute list handle that has been initialized to AZN_C_INVALID_HANDLE.
- An attribute list handle for a new, empty attribute list, which has been initialized by a call to `azn_attrlist_create()`.
- An attribute list handle for an existing, non-empty attribute list which contains valid data. In this case, attribute list data is appended to the existing attribute list.

**Output**

**entitlements**
Pointer to a attribute list handle variable which will hold the entitlement information on return.

The attribute list handle pointer passed as the `entitlements` parameter can point to one of the following:

- A attribute list handle that has been initialized to AZN_C_INVALID_HANDLE.
- A attribute list handle for a new, empty attribute list, which has been initialized by a call to `azn_attrlist_create()`.
- A attribute list handle for an existing, non-empty attribute list which contains valid data.

**Description**

This uses an entitlements service identified by `entitlements_svc_id` to return the entitlements of an initiator identified by credentials `creds`. The calling application
may pass application-specific or entitlements-service-specific context data using an attribute list app_context. The entitlements are returned using the attribute list entitlements.

**Note:** This function declares a new attribute list handle, entitlements. When declaring a new attribute list handle, always initialize it to AZN_C_INVALID_HANDLE before using it.

The constants AZN_C_REQUEST_TIME, AZN_C_AUTHN_QUALITY, AZN_C_REQUESTER_LOC, and AZN_C_REQUEST_ROUTE_QOP, may be used as name attribute of entries in the app_context attribute list to communicate common types of context information.

**Return Values**

If successful, the function returns AZN_S_COMPLETE.

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

- **AZN_S_COMPLETE**
  Successful completion.
- **AZN_S_INVALID_CRED_HDL**
  The creds handle supplied is invalid.
- **AZN_S_INVALID_ENTITLEMENT_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_AUTHORIZATION_FAILURE**
  The caller does not possess the authority required to invoke this function.
- **AZN_S_UNIMPLEMENTED_FUNCTION**
  This function is not supported by the implementation.
- **AZN_S_FAILURE**
  An implementation specific error or failure has occurred. Implementation specific minor error codes can be derived from the returned status code with azn_error_minor().
azn_error_get_string()

Returns the Policy Director Serviceability message string for the specified data structure of type azn_status_t.

Syntax

azn_status_t azn_error_get_string(
    const azn_status_t aznstatus,
    azn_string_t *error_string,
);

Parameters

Input

aznstatus

Authorization API status

Output

error_string

Pointer to an azn_string_t variable that will contain the error string upon successful completion of the routine.

Description

This interface returns the Policy Director Serviceability (PDSVC) message string for the Authorization API status that is passed in. It returns the message string for the minor error code if one exists. When a minor error code does not exist, it returns the message string for the major error code.

Return Values

When successful, the function returns AZN_S_COMPLETE.

When unsuccessful, the function returns one of the following major error codes:

- AZN_S_INVALID_MAJOR_CODE
  The major error code is invalid.

- AZN_S_MAJOR_CODE_MESSAGE_NOT_FOUND
  There is no message in the message table for the major error code.

- AZN_S_MINOR_CODE_MESSAGE_NOT_FOUND
  The minor error code is invalid. Alternately, this error message can mean that there is no message in the message table for the minor error code.

- AZN_S_INVALID_STRING_REF
  The pointer of type error_string is NULL.
azn_error_major()

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

Syntax

    unsigned int
    azn_error_major(
        const 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(
    const 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 pdbaclmsg.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(
    const 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`

Pointer to an `azn_string_t` variable that will contain the minor error string upon successful completion of the routine.

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_INVALID_STRING_REF**
  
  The minor error string is NULL.
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(
    const azn_string_t authority,
    const azn_string_t mechanism_id,
    const 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. The only valid input value is the default value of NULL.

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

Pointer to the handle of a new, empty credentials chain that will hold the returned credentials.

The credentials handle pointer passed as the new_creds parameter can be one of the following:

- A credentials handle that has been initialized to AZN_C_INVALID_HANDLE.
- A credentials handle for a new, empty credentials list, which has been initialized by a call to azn_attrlist_create(). In this case, the handle is updated.
- A credentials handle for an existing, non-empty credentials list which contains valid data. In this case, the handle is updated.

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 causes the default authentication mechanism to be the installed user registry in the Policy Director secure domain. For more information, see “Default user registry information structure” on page 10.

**Note:** This function declares a new credential handle, *new_creds*. When declaring a new credential handle, always initialize it to AZN_C_INVALID_HANDLE before using it.

**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 pdbaclmsg.h for a complete list of minor error codes that describe access control problems.
**azn_initialize()**

Initializes the authorization service.

**Syntax**

```c
azn_status_t
azn_initialize(
    const 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.

The credentials handle should be a handle for an existing, non-empty credentials chain which contains valid data.

**Output**

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

The attribute list handle pointer passed as the *init_info* parameter can be one of the following:

- A attribute list handle that has been initialized to AZN_C_INVALID_HANDLE.
- A attribute list handle for a new, empty attribute list, which has been initialized by a call to `azn_attrlist_create()`.
- A credentials handle for an existing, non-empty attribute list which contains valid data.

**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_*`).

Upon return from this call, 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.

**Note:** This function declares a new attribute list handle, *init_info*. When declaring a new attribute list handle, always initialize it to AZN_C_INVALID_HANDLE before using it.

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_CONFIG_INVALID_LISTENING_PORT**
  An Administration Service plug-in was registered but no ssl-listening-port was specified.
- **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**

Appendix A. Authorization API reference 153
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_SVC_ADMIN_UNKNOWN_PARAMETER**
  The service definition for the Administration Service plug-in contained an invalid parameter.

- **AZN_S_SVC_ADMIN_POBJ_NOT_SPECIFIED**
  The Administration Service plug-in definition specified the -pboj parameter but failed to specify the name of the protected object hierarchy.

- **AZN_S_SVC_ADMIN_POBJ_ALREADY_REGISTERED**
  The protected object hierarchy name has already been registered by another Administration service definition within this Authorization API application.

- **AZN_S_SVC_ADMIN_TASK_FUNC_NOT_FOUND**
  The Administration Service plug-in implemented either azn_admin_get_tasklist() or azn_admin_perform_task() but did not implement both interfaces. Administration Service plug-ins must implement either both or none.

- **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 pdbaclmsg.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_getcreds(  
    const azn_buffer_t pac,  
    const 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.

The credentials handle pointer passed as the *new_creds* parameter can be one of the following:

- A credentials handle that has been initialized to AZN_C_INVALID_HANDLE.
- A credentials handle for a new, empty credential, which has been initialized by a call to azn_creds_create(). In this case, the handle is updated (overwritten) with new credentials.
- A credentials handle for an existing, non-empty credential chain which contains valid data. In this case, the handle is updated (overwritten) with new credentials.

**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.

**Note:** This function declares a new credentials handle, *new_creds*. When declaring a new credentials handle, always initialize it to AZN_C_INVALID_HANDLE before using it.

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

**Return Values**

If successful, the function will return AZN_S_COMPLETE.

If the returned status code is not equal to AZN_S_COMPLETE, the major error codes will be derived from the returned status code with azn_error_major().
• AZN_S_COMPLETE
  Successful completion.
• AZN_S_API_UNINITIALIZED
  This function has been called before azn_initialize().
• AZN_S_INVALID_PAC
  The PAC is invalid or could not be verified by the PAC service.
• AZN_S_INVALID_PAC_SVC
  The id of the PAC service is invalid.
• AZN_S_INVALID_NEW_CREDS_HDL
  The credentials handle supplied for new_creds is invalid.
• AZN_S_UNIMPLEMENTED_FUNCTION
  This function is not supported by the implementation.
• AZN_S_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 pdbaclmsg.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 upon successful completion.

**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_pobj()

Releases storage associated with a protected object data structure.

**Syntax**

```c
azn_status_t
azn_release_pobj(
    azn_pobj_t * pobj
);
```

**Parameters**

**Input**

*pobj*

Pointer to a protected object structure whose storage is to be released.

**Output**

*pobj*

Pointer to a protected object structure whose storage is released. The pointer is set to an invalid value upon successful completion.

**Description**

Releases storage associated with a protected object structure.

**Return Values**

When successful, the function 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_INVALID_POBJ_REF
  The pointer to the protected object is NULL.
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 upon successful completion.

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 upon successful completion.

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 `pdbaclmsg.h` for a complete list of minor error codes that describe access control problems.
azn_util_errcode()

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

Syntax

```
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 **true** if the handle is valid or **false** if the handle is invalid.
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.

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_getcreds()` 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.
**Note:** When used with Domino or Active Directory, additional error codes can be returned, in cases where there is more than one error in the input parameters.

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

When unsuccessful, the following major error codes are returned:

- **AZN_S_FAILURE**
  An error has been encountered.
- **AZN_S_U_INVALID_PASSWORD**
  The password is not valid.
- **AZN_S_U_INVALID_PRINC_NAME**
  The principal name is unknown.
- **AZN_S_U_LDAP_AUTHEN_FAILED**
  Authentication to the LDAP user registry failed.
- **AZN_S_U_URAF_AUTHEN_FAILED**
  URAF authentication failed
- **AZN_S_U_PASSWORD_EXPIRED**
  The user authenticated correctly but the password has expired and must be changed. *Mechanism_id* and *authinfo* are not returned.
- **AZN_S_U_ACCOUNT_DISABLED**
  The account has been disabled by the administrator.
- **AZN_S_U_TOD_ACCESS_DENIED**
  The login failed due to policy restrictions based on Time of Day.
- **AZN_S_U_ACCOUNT_LOCKEDOUT**
  The account has been locked because too many invalid login attempts have been made. The number of invalid attempts and the logout period can be configured through policy.
- **AZN_S_U_INSUFFICIENT_ACCESS**
  The caller of this function has insufficient privileges to perform the requested operation.
- **AZN_S_API_UNINITIALIZED**
  This function has been called before *azn_initialize()*.
azn_util_password_change()

Changes the password for the specified Tivoli Access Manager principal in the registry.

Syntax

```c
azn_status_t
azn_util_password_change(
    const azn_string_t principal_name,
    const azn_string_t password
);
```

Parameters

Input

- `principal_name` - Name of the principal who’s password is to change. When using an LDAP registry for authentication, this may also be a Distinguished Name (DN) string.

- `password` - New password for the principal.

Description

Changes the password for a username pair. The method of changing the password is selected automatically according to the underlying authentication mechanism with which the authorization API was installed. Examples of authentication mechanisms are LDAP, Lotus Domino Server, and Microsoft Active Directory.

Limitations: This call can succeed even though the calling application server lacks sufficient access, when used with either Lotus Domino Server or Microsoft Active Directory authentication. On Lotus Domino Server, this is because the server uses the local Lotus Notes ID file, and hence always has the authority to change the password.

On Active Directory, the authorization to make this call is taken from the Active Directory system group membership, not from a Tivoli Access Manager group. Use the following command, entered all on one line, to add the application to the appropriate system group membership:

```
uraftool.exe -p Admin_Password
    -c "rgyid_addmem group \"cn=Domain Admin,cn=users\" application_server_ID"
```

On LDAP-based user registries, use the `pdadmin` command to add the application server to the appropriate Tivoli Access Manager group:

```
padmin group modify SecurityGroup add application_server_ID
```

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 pdbac1msg.h for a complete list of minor error codes that describe access control problems.

**Note:** When used with Lotus Domino Server or Microsoft Active Directory, additional error codes can be returned, in cases where there is more than one error in the input parameters.

When unsuccessful, the following major error codes are returned:

- **AZN_S_FAILURE**
- **AZN_S_U_ACCOUNT_DISABLED**
  The user account is disabled by the administrator.
- **AZN_S_U_ACCOUNT_LOCKEDOUT**
  The account has been locked because too many invalid login attempts have been made. The number of invalid attempts and the logout period can be configured through policy.
- **AZN_S_U_INSUFFICIENT_ACCESS**
  The caller of this function has insufficient privileges to perform the requested operation.
- **AZN_S_U_INVALID_PASSWORD**
  The password is not valid.
- **AZN_S_U_INVALID_PRINC_NAME**
  The principal name is unknown.
- **AZN_S_U_LDAP_AUTHEN_FAILED**
  Authentication to the LDAP user registry failed.
- **AZN_S_U_PASSWORD_EXPIRED**
  The user password has expired.
- **AZN_S_U_PASSWORD_HAS_SPACES**
  The user password contains spaces.
- **AZN_S_U_PASSWORD_TOO_FEW_ALPHA**
  The user password must have more alphanumeric characters.
- **AZN_S_U_PASSWORD_TOO_FEW_NONALPHA**
  The user password must have more non-alphanumeric characters.
- **AZN_S_U_PASSWORD_TOO_MANY_REPEATED**
  The user password has one or more characters that are repeated too many times.
- **AZN_S_U_PASSWORD_TOO_SHORT**
  The user password must contain more characters.
- **AZN_S_U_UARAF_AUTHEN_FAILED**
  UARAF authentication failed
- **AZN_S_U_TOD_ACCESS_DENIED**
  Access denied because time or day restrictions prevent access at the current time.
- **AZN_S_API_UNINITIALIZED**
  This function has been called before azn_initialize().
Appendix B. Authorization service plug-in API reference

This chapter contains the following reference pages:

- “azn_admin_get_object()” on page 170
- “azn_admin_get_objectlist()” on page 172
- “azn_admin_get_tasklist()” on page 174
- “azn_admin_perform_task()” on page 176
- “azn_service_initialize()” on page 179
- “azn_service_shutdown()” on page 182
azn_admin_get_object()

Retrieves a potential protected object. Tests for the existence of a protected object.

**Syntax**

```c
azn_status_t
azn_admin_get_object(
    azn_creds_h_t creds,
    azn_string_t locale,
    azn_string_t path,
    azn_attrlist_h_t indata,
    azn_attrlist_h_t outdata
);
```

**Parameters**

**Input**

*creds*  
Credentials of the identity requesting the object. The administration service must verify that the credentials have sufficient authorization to perform the requested operation. The Tivoli Access Manager policy server forwards the credentials of the administrator when performing this function.

*locale*  
Locale of the caller at the client machine. The locale is the string equal to LC_MESSAGES returned from the client machine.

**Note:** This parameter is not used in Tivoli Access Manager Version 4.1., but is maintained for compatibility with Tivoli SecureWay Policy Director Version 3.8.

*path*  
The complete path name of the protected object.

*indata*  
An undefined (free form) attribute list that can be used for agreed upon communication between the administration service implementation and a custom user interface.

**Output**

*outdata*  
An undefined (free form) attribute list. This attribute list is allocated by the administration service implementation. See the Description section immediately below for a discussion of how to use this output parameter.

**Description**

This function returns information about an object in the Tivoli Access Manager protected object namespace. The credentials of the identity requesting the object are forwarded by the Tivoli Access Manager policy server when this function is called.

When the caller has sufficient permissions, the protected object information is placed into a authorization API attribute list. This list is returned as the output parameter *outdata*.

The administration service must pass the protected object information in the azn_admin_svc_pobj attribute. Use azn_attrlist_add_entry_pobj() to add this attribute to the attribute list.
The administration service must pass the result strings information in the azn_admin_svc_results attribute. Use azn_attrlist_add_entry_pobj() to add this attribute to the attribute list. Typically, these strings include error, warning, and information messages.

The administration service and the custom user interface can exchange information of their choice by setting other attributes. Use the azn_attrlist_add_entry_* APIs to set attributes. The Tivoli Access Manager policy server passes these other attributes in the outdata parameter of the corresponding administration API function call.

Multiple authorization API applications can register to service the protected object hierarchy name for the protected object. You can use this feature to provide failover support in the event that a particular application server fails. If the authorization API implementation fails to connect to a registered service implementation, it attempts to contact other service implementations until a connection is successful or until the list of appropriate service implementations is exhausted.

Return Values

When successful, the function returns AZN_S_COMPLETE.

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

When this function calls another authorization API function, and the called function returns an error, this function should return the error to the calling application.

- **AZN_S_SVC_ADMIN_OUT_OF_MEMORY**
  A memory allocation error has occurred.

- **AZN_S_SVC_ADMIN_INVALID_SVCINFO_HDL**
  The svc_info passed to the administration service plug-in shared library is invalid.

- **AZN_S_SVC_ADMIN_INVALID_ARG_COUNT**
  The argument count argc passed to the administration service plug-in shared library is invalid.

- **AZN_S_SVC_ADMIN_INVALID_ARG_ARRAY**
  The argument array passed to the administration service plug-in shared library is invalid.

- **AZN_S_SVC_ADMIN_INVALID_ARGUMENT**
  One or more of the arguments passed in to initialize the administration service plug-in is invalid.

- **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_admin_get_objectlist()

Accesses the administration service to provide a list of all potential protected objects that are children of the specified parent object.

Syntax

```c
azn_status_t
azn_admin_get_object(
    azn_creds_h_t creds,
    azn_string_t locale,
    azn_string_t path,
    azn_attrlist_h_t indata,
    azn_attrlist_h_t outdata
);
```

Parameters

Input

creds
Credentials of the identity requesting the object. The administration service must verify that the credentials have sufficient authorization to perform the requested operation. The Tivoli Access Manager policy server forwards the credentials of the administrator when performing this function.

locale
Locale of the caller at the client machine. The locale is the string equal to LC_MESSAGES returned from the client machine.

Note: This parameter is not used in Tivoli Access Manager Version 4.1.

path
The complete path name of the protected object.

indata
An undefined (free form) attribute list that can be used for agreed upon communication between the administration service implementation and a custom user interface.

Output

outdata
An undefined (free form) attribute list. This attribute list is allocated by the administration service implementation. See the Description section immediately below for a discussion of how to use this output parameter.

Description

This function returns information about objects in the Tivoli Access Manager protected object namespace. The credentials of the identity requesting the objects are forwarded by the Tivoli Access Manager policy server when this function is called.

When the caller has sufficient permissions, the protected object information is placed into a authorization API attribute list. This list is returned as the output parameter outdata.
The administration service plug-in must pass the protected object information in the `azn_admin_svc_pobj` attribute. Use `azn_attrlist_add_entry_pobj()` to add this attribute to the attribute list.

The administration service plug-in must pass the result strings information in the `azn_admin_svc_results` attribute. Use `azn_attrlist_add_entry_pobj()` to add this attribute to the attribute list. Typically, these strings include error, warning, and information messages.

The administration service and the custom user interface can exchange information of their choice by setting other attributes. Use the `azn_attrlist_add_entry_` APIs to set attributes. The Tivoli Access Manager policy server passes these other attributes in the `outdata` parameter of the corresponding administration API function call.

Multiple authorization API applications can register to service the protected object hierarchy name for the protected object. You can use this feature to provide failover support in the event that a particular application server fails. If the authorization API implementation fails to connect to a registered service implementation, it attempts to contact other service implementations until a connection is successful or until the list of appropriate service implementations is exhausted.

**Return Values**

When successful, the function returns `AZN_S_COMPLETE`.

When the returned status code is not equal to `AZN_S_COMPLETE`, the major error codes are derived from the returned status code with `azn_error_major()`.

When this function calls another authorization API function, and the called function returns an error, this function should return the error to the calling application.

- `AZN_S_SVC_ADMIN_OUT_OF_MEMORY`
  A memory allocation error has occurred.

- `AZN_S_SVC_ADMIN_INVALID_SVCINFO_HDL`
  The `svc_info` passed to the administration service plug-in shared library is invalid.

- `AZN_S_SVC_ADMIN_INVALID_ARG_COUNT`
  The argument count `argc` passed to the administration service plug-in shared library is invalid.

- `AZN_S_SVC_ADMIN_INVALID_ARG_ARRAY`
  The argument array passed to the administration service plug-in shared library is invalid.

- `AZN_S_SVC_ADMIN_INVALID_ARGUMENT`
  One or more of the arguments passed in to initialize the administration service plug-in is invalid.

- `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_admin_get_tasklist()

Returns a list of all the supported administration tasks.

Syntax

```c
azn_status_t
azn_admin_get_tasklist(
    azn_creds_h_t creds,
    azn_string_t locale,
    azn_attrlist_h_t indata,
    azn_attrlist_h_t outdata
);
```

Parameters

Input

creds
Credentials of the identity requesting the object. The administration service must verify that the credentials have sufficient authorization to perform the requested operation. The Tivoli Access Manager policy server forwards the credentials of the administrator when performing this function.

locale
Locale of the caller at the client machine. The locale is the string equal to LC_MESSAGES returned from the client machine. Note: This parameter is not used in Tivoli Access Manager Version 4.1.

indata
An undefined (free form) attribute list that can be used for agreed upon communication between the administration service implementation and a custom user interface.

Output

outdata
An undefined (free form) attribute list. This attribute list is allocated by the administration service implementation.

Description

Returns a list of all the supported administration tasks. The list is obtained from the administration service.

The administration service plug-in must pass the task information in the azn_admin_svc_task attribute. Use the azn_attrlist_add_entry() API to build the list. Information about each specific task is a value of this attribute.

The administration service plug-in must pass the result strings information in the azn_admin_svc_results attribute. Use azn_attrlist_add_entry_pobj() to add this attribute to the attribute list. Typically, these strings include error, warning, and information messages.

Each value for the azn_admin_svc_results attribute indicates any result information for the azn_admin_get_tasklist() call. These results are displayed by pdadmin in response to typing the command server listtasks server name at the pdadmin command line interface. For example, if this call was successful, the implementor may choose not to return anything in the azn_admin_svc_results
attribute. If this call was unsuccessful, the implementor can return implementation-specific error messages in the adm_admin_svc_results attribute.

The administration service and the custom user interface can exchange information of their choice by setting other attributes. Use the azn_attrlist_add_entry_* APIs to set attributes. The Tivoli Access Manager policy server passes these other attributes in the outdata parameter of the corresponding administration API function call.

The authorization API invokes this function on all registered administration service plug-ins and returns the output back from all of them. If two plug-ins return the same task names, both are returned. If any of the plug-ins return an error, the remaining plug-ins do not get invoked. The results obtained thus far are returned along with the error that occurred. If a plug-in does not support any tasks, it should return an AZN_S_COMPLETE major code so that this function can be invoked on the remaining plug-ins (if any).

Each value for the azn_admin_svc_task attribute needs to be the complete specification of how to invoke the task. These values are displayed in response to typing the command server listtasks server name at the pdadmin command line interface. These value could also potentially be used by a graphical user interface, so they must to be complete.

Return Values

When successful, the function returns AZN_S_COMPLETE.

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

When this function calls another authorization API function, and the called function returns an error, this function should return the error to the calling application.

- AZN_S_SVC_ADMIN_OUT_OF_MEMORY
  A memory allocation error has occurred.
- AZN_S_SVC_ADMIN_INVALID_SVCINFO_HDL
  The svc_info passed to the administration service plug-in shared library is invalid.
- AZN_S_SVC_ADMIN_INVALID_ARG_COUNT
  The argument count argc passed to the administration service plug-in shared library is invalid.
- AZN_S_SVC_ADMIN_INVALID_ARG_ARRAY
  The argument array passed to the administration service plug-in shared library is invalid.
- AZN_S_SVC_ADMIN_INVALID_ARGUMENT
  One or more of the arguments passed in to initialize the administration service plug-in is invalid.
- 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_admin_perform_task()**

Instructs the service to perform an administration task. The service returns the results of the task.

**Syntax**

```c
azn_status_t 
azn_admin_perform_task( 
    azn_creds_h_t creds, 
    azn_string_t locale, 
    azn_string_t command, 
    azn_attrlist_h_t indata, 
    azn_attrlist_h_t outdata
);
```

**Parameters**

**Input**

*creds*

Credentials of the identity requesting the object. The administration service must verify that the credentials have sufficient authorization to perform the requested operation. The Tivoli Access Manager policy server forwards the credentials of the administrator when performing this function.

*locale*

Locale of the caller at the client machine. The locale is the string equal to LC_MESSAGES returned from the client machine. Note: This parameter is not used in Tivoli Access Manager Version 4.1.

*command*

Command string that identifies the task to be performed. The string must conforms to the **pdadmin** command line interface syntax rules.

*indata*

An undefined (free form) attribute list that can be used for agreed upon communication between the administration service implementation and a custom user interface.

**Output**

*outdata*

An undefined (free form) attribute list. This attribute list is allocated by the administration service implementation.

**Description**

Instructs the service to perform an administration task. The caller must have credentials with sufficient permission to perform the task. The service returns the results of the task in the **outdata** attribute list.

The administration service must pass the result strings information in the **azn_admin_svc_results** attribute. Use **azn_attrlist_add_entry_pobj()** to add this attribute to the attribute list. Typically, these strings include error, warning, and information messages.

The administration service and the custom user interface can exchange information of their choice by setting other attributes. Use the **azn_attrlist_add_entry_*** APIs
to set attributes. The Tivoli Access Manager policy server passes these other attributes in the `outdata` parameter of the corresponding administration API function call.

The authorization API implementation invokes this function on all registered administration service plug-ins until it finds one that implements this interface. If more than one service plug-in supports this task, the first plug-in is selected to execute the task. The service plug-in should return an `AZN_S_SVC_ADMIN_INVALID_TASK` major code if it is passed on a task that it cannot perform.

The `ssl-v3-timeout` parameter specified in the `[ssl]` stanza of the authorization configuration file determines the timeout for the communication between the authorization AP application and Tivoli Access Manager policy server, and for the communication between the Tivoli Access Manager policy server and the `pdadmin` utility command line interface.

A value of zero specifies an infinite timeout. Other values represent the actual amount of time within which a response is expected for a request. If the task being performed by `azn_admin_perform_task()` takes a long time, this communication could time out and cause an SSL error. Therefore, this timeout value needs to be set properly in the `[ssl]` stanza in the various configuration files. The relevant configuration files include `ivmgrd.conf`, the authorization API configuration file, `ivacld.conf`, and `pd.conf`.

**Return Values**

When successful, the function returns `AZN_S_COMPLETE`.

When the returned status code is not equal to `AZN_S_COMPLETE`, the major error codes are derived from the returned status code with `azn_error_major()`.

When this function calls another authorization API function, and the called function returns an error, this function should return the error to the calling application.

- `AZN_S_SVC_ADMIN_OUT_OF_MEMORY`
  A memory allocation error has occurred.

- `AZN_S_SVC_ADMIN_INVALID_TASK`
  The passed-in task is not implemented by this administration service plug-in.

Use the `pdadmin server listtasks authorization_API_application_name` command to determine the list of tasks supported by this authorization API application. If the task in question is not shown on the list, make sure that the authorization administration service plug-in that implements this task is registered by the authorization API application.

- `AZN_S_SVC_ADMIN_INVALID_SVCINFO_HDL`
  The `svc_info` passed to the administration service plug-in shared library is invalid.

- `AZN_S_SVC_ADMIN_INVALID_ARG_COUNT`
  The argument count `argc` passed to the administration service plug-in shared library is invalid.

- `AZN_S_SVC_ADMIN_INVALID_ARG_ARRAY`
  The argument array passed to the administration service plug-in shared library is invalid.
• AZN_S_SVC_ADMIN_INVALID_ARGUMENT
  One or more of the arguments passed in to initialize the administration service plug-in is invalid.
• 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_initialize()

Initialize the specified authorization service plug-in.

Syntax

```c
azn_status_t azn_service_initialize(
    int argc,
    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 functions that are specific to each plug-in. For example, this function is called before calling `azn_entitlement_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:
```plaintext
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:
```plaintext
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`:
```plaintext
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.

The prototype for this function is included in the file `azn_svc_protos.h`, in the Tivoli Access Manager include directory.

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. An example of the information listed in the attributes is the version of 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.

- **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_SVC_ADMIN_INVALID_SVCINFO_HDL**
  The `svc_info` passed to the administration service plug-in shared library is invalid.

- **AZN_S_SVC_ADMIN_INVALID_ARG_COUNT**
  The argument count `argc` passed to the administration service plug-in shared library is invalid.

- **AZN_S_SVC_ADMIN_INVALID_ARG_ARRAY**
  The argument array passed to the administration service plug-in shared library is invalid.

- **AZN_S_SVC_ADMIN_INVALID_ARGUMENT**
  One or more of the arguments passed in to initialize the administration service plug-in is invalid.
• 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,
    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:
```
tensvc = /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:

```
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.

The prototype for this function is included in the file `azn_svc_protos.h`, in the Tivoli Access Manager include directory.

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().
Appendix C. Authorization API client configuration file

The initialization and operation of an authorization API client application can be controlled through the use of a client configuration file. The example client configuration file is aznAPI.conf. The name of an authorization API client configuration file is usually changed to match the name of the application.

The file contains sections called stanzas.

Stanza labels appear within brackets, such as: [stanza_name]. For example, the [ssl] stanza defines the SSL configuration settings for use by the authorization API application.

Each stanza in a Tivoli Access Manager configuration file contains one or more stanza entries. A stanza entry consists of a key value pair, which contains information that is expressed as a paired set of parameters. Each stanza entry has the following format:

key = value

The initial configuration of an authorization API client application establishes some default values. Some values are static and will never change; other values can be modified to customize server functionality and performance.

Note: Authorization API client applications have the option of using initialization parameters instead of using a configuration file. The initialization parameters will override the configuration file settings. For more information, see Chapter 3, “Initializing the authorization API”, on page 17.

Guidelines for configuring stanzas

These guidelines are provided to help you make changes to the authorization API application configuration file. The guidelines are divided into these types:

- General
- Default settings
- Strings
- Defined strings
- File names
- Integers
- Boolean values

For instructions, see “Change configuration settings” on page 188

General guidelines

Use the following general guidelines when making changes to the configuration settings:

- There is no order dependency or location dependency for stanzas in any configuration file.
- Stanza entries are marked as required or optional. When an entry is required, the entry must contain a valid key and value.

© Copyright IBM Corp. 2000,2002
• Do not change the names of the keys in the configuration files. Changing the name of the key might cause unpredictable results for the servers.
• Capitalization of the keys is not important. For example, you can use UseSSL or useSSL.
• Spaces are not allowed for names of keys.
• For the key value pair format of key = value, the spaces surrounding the equal sign (=) are not required, but they are recommended.
• Nonprintable characters (such as tabs, carriage returns, and line feeds) that occur at the end of a stanza entry are ignored. Nonprintable characters are ASCII characters with a decimal value less than 32.

Default values
Use the following guidelines when changing default configuration settings:
• Many values are created or modified only by using configuration programs. Do not manually edit these stanzas or values.
• Some values are filled in automatically during configuration. These values are needed for the initialization of the application after the configuration.
• The default values for a stanza entry might be different, depending on the application configuration. Some key value pairs are not applicable to certain applications.

Strings
Some values accept a string value. When you manually edit the configuration file, use the following guidelines to change configuration settings that require a string:
• String values must be part of the local codeset.
• String values can be specified using any case.
• Valid string characters are the letters a-Z, the numbers 0-9, a period (.), a comma (,), an equal sign (=), a forward slash (/), an underscore (_), a plus sign (+), a hyphen (-), an at sign (@), an ampersand (&), and an asterisk (*).

Some strings impose additional or different restrictions on the set of allowable string characters. These restrictions are listed under the appropriate stanza entry discussion later in this chapter.
• Double quotation marks are sometimes, but not always, required when you use spaces or more than one word for values. Refer to the descriptions or examples for each stanza entry when in doubt.
• The minimum and maximum lengths of user registry-related string values, if there are limits, are imposed by the underlying registry. For example, for Active Directory the maximum length is 256 alphanumeric characters.

Defined strings
Some values accept a string value, but the value must be one of a set of defined strings. When you manually edit the configuration file, make sure that the string value you type matches one of the valid defined strings values.

For example, the [aznapi-configuration] stanza contains the following entry:
auditcfg = {azn|authn|mgmt}

The value for auditcfg is expected to be either azn or authn or mgmt. Any other value is invalid and results in an error.
**File names**

Some values are file names. For each stanza entry that expects a file name as a value, the description of the stanza entry specifies which of the following constructs are valid:

- **Filename**
  No directory path included.

- **Relative filename**
  A directory path is allowed but not mandatory.
  These files typically are expected to be located relative to the location of a standard Tivoli Access Manager directory. The stanza entry for each relative path name lists the root directory to which the file name is relative.

- **Fully qualified (absolute) path**
  An absolute directory path is required.

**Note:** Some stanza entries allow more than one of the above choices.

The set of characters permitted in a file name can be determined by the file system and by the local codeset. For Windows, file names cannot have these characters: a backward slash (\), a colon (:), a question mark (?), or double quotation marks.

**Integers**

Many stanza entries expect the value for the entry to be expressed as an integer.

- **Stanza entries that take an integer value expect integer values within a valid range.** The range is described in terms of a *minimum* value and a *maximum* value.

  For example, in the `[logging]` stanza, the `logflush` stanza entry has a minimum value of 1 second and a maximum value of 600 seconds.

- **For some entries, the integer value must be positive, and the minimum value is 1.** For other entries, a minimum integer value of 0 is allowed.

  Use caution when setting an integer value to 0. For example, an integer value of 0 might the function that is controlled by that stanza entry. For example, in the `[aznapi-configuration]` stanza, the entry `logsize = 0` disables the creation of a rollover log file. A port number. Or, an integer value of 0 might indicate that the number is unlimited. For example, in the `[ssl]` stanza, the entry `ssl-io-inactivity-timeout = 0` means there is no inactivity timeout.

- **For some entries requiring integer values, Tivoli Access Manager does not impose an upper limit.** for the maximum number allowed. For example, there is typically no maximum for port numbers in the `[ldap]` stanza.

  For this type of entry, the maximum number is limited only by the size of memory allocated for an integer data type. This number can vary, based on the type of operating system. For systems that allocate 4 bytes for an integer, this value is 2147483647.

  However, an administrator should use a number that represents the value that is most logical for the value you are trying to set.

**Boolean values**

Many stanza entries represent a boolean value. Tivoli Access Manager recognizes the boolean values `yes` and `no`. 

Appendix C. Authorization API client configuration file 187
Some of the entries in the configuration files are read by other servers and utilities. For example, many entries in the [ldap] stanza are read by the LDAP client. Some of these other programs recognize additional boolean characters:

- *yes* or *true*
- *no* or *false*

Anything other than *yes*[true] will be interpreted as *no*[false], including a blank value.

The recognized boolean entries are listed for each stanza entry. Refer to the individual descriptions to determine when *true* or *false* are also recognized.

### Change configuration settings

To change a configuration setting, do the following:

1. Make a backup copy of the configuration file you plan to modify.
   This allows you to return the configuration file to a known working state, should you encounter an error later.

2. Stop the authorization API application that is affected.

3. Make the changes by doing one of the following:
   - Use an ASCII text editor to edit the configuration file and make any necessary changes. Save your changes.
   - For some settings in the authorization API client configuration file, use the *svrsslcfg* utility.
     See the stanza entry descriptions to determine which entries are modified by *svrsslcfg*.

Many stanzas or values are created or modified only by using configuration programs. Some values are filled in automatically after the configuration is completed. Do not manually edit these values.

4. Restart the authorization API application affected.
### [aznapi-configuration] stanza

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Values</th>
<th>Default Value</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>logsize = 0</td>
<td>Log file rollover threshold (in bytes) for audit logs. If the audit log file reaches this threshold, the original audit log file will be renamed and a new log file with the original name will be created.</td>
<td></td>
<td>0</td>
<td>logsize = 2000000.</td>
</tr>
<tr>
<td>neg_number_bytes</td>
<td>Any negative number indicates that the logs are rolled over daily, regardless of the size.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>number_bytes</td>
<td>The maximum size (in bytes) of the audit log file before the rollover occurs. The allowable range is from 1 byte to 2 megabytes.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>logflush = 600 seconds</td>
<td>Time interval (in seconds) between log flushes of log file buffers for audit logs.</td>
<td></td>
<td>600 seconds</td>
<td></td>
</tr>
<tr>
<td>logaudit = no</td>
<td>Indication of whether to audit by turning the auditing parameter on or off.</td>
<td>yes</td>
<td>true</td>
<td>Auditing is enabled or turned on.</td>
</tr>
<tr>
<td>auditlog = C:\pd\audit\pdacld.log</td>
<td>Location of the audit trail file for the local client. If no location and name are supplied, auditing will not be performed. Each server provides its own audit log setting in its corresponding configuration file.</td>
<td></td>
<td></td>
<td>Example for Windows: auditlog = C:\pd\audit\pdacld.log</td>
</tr>
</tbody>
</table>
auditcfg = {azn|authn|mgmt}

Capture of audit logging configuration events.

To enable component specific audit records, add the appropriate definition. Each server provides its own value in its configuration file. For example, default behavior for aznAPI.conf is azn; default behavior for pdacld.conf is also azn.

The auditcfg values are mutually exclusive so only one of these values can be used at a time; the other values must be commented out. To comment out a stanza entry, start the entry with a pound sign (#). For example:

```
auditcfg = azn
#auditcfg = authn
#auditcfg = mgmt
```

Valid values:
- **azn** Use to capture authorization events.
- **authn** Use to capture authentication events.
- **mgmt** Use to capture management events.

This stanza entry is required if logaudit = yes.

There is no default value.

Example: auditcfg = azn

db-file = fully_qualified_path

Name and location of the pdacld database cache file for capturing authentication events. This value must be specified, and each server provides its own value.

The fully_qualified_path value represents an alphanumeric string.

This stanza entry is required if logaudit = yes.

There is no default value.

Example for Windows: \C:\pd\db\ivacll.db

cache-refresh-interval = {disable|default|number_seconds}

Poll interval between checks for updates to the master authorization server. **Note:** The local cache is rebuilt only if an update is detected.

Valid values:
- **disable** The interval value in seconds is not set.
- **default** The default value of 600 seconds is used.
- **number_seconds** The exact time interval that you set by specifying the number of seconds. The minimum value is 0 seconds. Tivoli Access Manager does not impose a maximum other that the maximum value allowed by the memory allocated for an unsigned integer. This number is platform-specific, but typically is approximately 136 years.

This stanza entry is optional.

Default value: default

Example: cache-refresh-interval = 500

listen-flags = {enable|disable}
### listen-flags

Indication of whether to turn on or off the reception of policy cache update notifications. This parameter is set by the `svrsslcfg` utility.

<table>
<thead>
<tr>
<th>Valid values:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>enable</strong></td>
<td>Activates the notification listener.</td>
</tr>
<tr>
<td><strong>disable</strong></td>
<td>Deactivates the notification listener.</td>
</tr>
</tbody>
</table>

This stanza entry is optional.

Default value: `disable`

Example: `listen-flags = enable`

---

### azn-app-host = *other_hostname*

Attribute that is used to customize the host on which the authorization API application is listening.

For *other_hostname*, you can provide any valid internet host name. If this attribute is not specified, the default host name will be used.

By default, this attribute is disabled. When disabled, the stanza entry is commented out by using a pound sign (#) at the beginning of the stanza entry in the configuration file. For example:

```
#azn-app-host = libra
```

To enable this value, uncomment the entry in the configuration file by removing the pound sign (#). Be sure to include a host name value for the host on which the authorization API application is listening.

This stanza entry is optional.

There is no default value.

Example: `azn-app-host = libra.dallas.ibm.com`

---

### mode = {local|remote}

Specifies the authorization API client mode. This stanza entry is set by `svrsslcfg` during configuration and should not be changed by the administrator.

The value local means that the authorization API client uses a local policy cache. The value remote means that the authorization API client uses a remote Tivoli Access Manager authorization server.

This stanza entry is required.

Default value: `local`

Example: `mode = local`
The set of attributes that the caller wants to receive from the `azn_decision_access_allowed_ext()` function in the permission info attribute list. When an attribute name is added to this list, the attribute is allowed to be returned as permission information if it is applicable to the current decision call. Attributes can be defined by users, for example, by setting an attribute on an ACL.

The following string constants are recognized by the authorization engine and equate to their corresponding permission information constants in `ogauthzn.h`. Consult this file for more information on permission information attributes.

| Attribute Name | Description
|---------------|-------------|
| `azn_perminfo_all_attr` | Return all attributes.
| `azn_perminfo_al` | Audit level (unsigned integer)
| `azn_perminfo_qop` | Quality of protection (string)
| `azn_perminfo_qop_ulong` | Quality of protection (unsigned long)
| `azn_perminfo_wm` | Warning mode (boolean)
| `azn_perminfo_wm_ulong` | Warning mode (unsigned long)
| `azn_perminfo_wm_permitted` | Access permitted by warning mode (boolean)
| `azn_perminfo_wm_permitted_ulong` | Access permitted by warning mode (unsigned long)

This stanza entry is optional.

There are no default values (attributes).

Example:
```
permission-info-returned = azn_perminfo_qop azn_perminfo_qop_ulong
```

Example with a user defined attribute `my_attribute`:
```
permission-info-returned = azn_perminfo_qop_ulong my_attribute
```
**[ssl] stanza**

<table>
<thead>
<tr>
<th><strong>[ssl] stanza</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ssl-keyfile = ssl-key-path</strong></td>
</tr>
<tr>
<td>Path location and file name of the local system of the SSL key file. If the key value pair does not exist in the configuration file, the application will fail. The file extension can be anything but it is usually .kdb.</td>
</tr>
<tr>
<td>This file is created and the value is set by <code>svrsslcfg</code>.</td>
</tr>
<tr>
<td>This stanza entry is required when SSL is enabled.</td>
</tr>
</tbody>
</table>
| Default value for UNIX: 
  `opt/PolicyDirector/keytab/ivmgrd.kdb` |
| Default value for Windows: 
  `C:\pd\keytab\ivmgrd.kdb` |
| Example for UNIX: `ssl-keyfile = opt/PolicyDirector/keytab/ivmgrd.kdb` |

| **ssl-keyfile-stash = ssl-stash-path** |
| Path location and file name of the SSL password stash file. The file extension can be anything but it is usually .sth. |
| The password is used to protect private keys in the key file. The password might be stored encrypted in a stash file. If both `ssl-keyfile-pwd` and `ssl-keyfile-stash` are specified, then the `ssl-keyfile-pwd` value will be used. |
| This file is created and the value is set by `svrsslcfg`. The path is defined by the `-d` option to the `svrsslcfg` utility. The name is defined by the `-n` option to the `svrsslcfg` utility. |
| This stanza entry is required when SSL is enabled. |
| Default value for UNIX: 
  `opt/PolicyDirector/keytab/ivmgrd.sth` |
| Default value for Windows: 
  `C:\pd\keytab\ivmgrd.sth` |
| Example for Windows: `ssl-keyfile-stash = C:\pd\keytab\ivmgrd.sth` |

| **ssl-v3-timeout = number_seconds** |
Session timeout (in seconds) for SSL v3 connections between clients and servers. This timeout value controls how often a full SSL handshake is completed between Tivoli Access Manager clients and servers.

Valid range of values for `number_seconds` is from 10-86400 seconds, where 86400 seconds is equal to 1 day. If you specify a number outside this range, the default number will be used.

This file is created and the value is set by `svrsslcfg`. The path is defined by the `-d` option to the `svrsslcfg` utility. The name is defined by the `-n` option to the `svrsslcfg` utility.

**Note**: Tivoli Access Manager components might not function with small timeout values in some network environments.

This stanza entry is required when SSL is enabled.

Default value: 7200

Example: `ssl-v3-timeout = 9800`.

```
ssl-listening-port = (0|port_number)
```

TCP port on which the application listens for incoming requests and policy cache update notifications.

Valid values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disables listening. The value is specified during configuration by using the <code>svrsslcfg</code> utility.</td>
</tr>
<tr>
<td><code>port_number</code></td>
<td>Enables listening at the specified port number. The valid range for <code>port_number</code> is any positive number that is allowed by TCP/IP and is not currently being used by another application.</td>
</tr>
</tbody>
</table>

This stanza entry is valid only with the following `mode` scenarios:

- Local mode, when `listen-flags` is set to enable.
- Local mode, when `[aznapi-admin-services]` are registered.
- Remote mode, when `[aznapi-admin-services]` are registered.

This stanza entry is required when SSL is enabled.

Default value: 0

Example: `ssl-listening-port = 7137`.

```
ssl-io-inactivity-timeout = (0|number_seconds)
```

Duration (in seconds) that an SSL connection waits for a response before timing out. There is no one default value because the default value is set by the configuration program of each server.

Valid values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No timeout.</td>
</tr>
<tr>
<td><code>number_seconds</code></td>
<td>The timeout specified in number of seconds. There is no range limitation for timeout values.</td>
</tr>
</tbody>
</table>

The number of seconds value is set by `svrsslcfg`.

This stanza entry is required when SSL is enabled.

Default value is server-dependent.

Example: `ssl-io-inactivity-timeout = 90`.

```
ssl-maximum-worker-threads = number_threads
```
<table>
<thead>
<tr>
<th><strong>Number of threads that can be created by the authorization API's internal server to handle incoming requests.</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Valid values:</strong></td>
</tr>
<tr>
<td>number_threads</td>
</tr>
<tr>
<td>Number of threads that can be specified. The valid range must be equal to 1, or greater than 1. The maximum number varies because it is dependent on available system resources.</td>
</tr>
<tr>
<td>This stanza entry is required when SSL is enabled.</td>
</tr>
<tr>
<td>Default value: 50</td>
</tr>
<tr>
<td>Example: ssl-maximum-worker-threads = 80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ssl-pwd-life = number_days</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Password lifetime for the key database file, specified in the number of days. For automatic password renewal, the value for the lifetime of a password is controlled by the number_days value when the server is started. Valid values for the number_days is from 1 to 7,299 days.</td>
</tr>
<tr>
<td>For manual password renewal, the value is dictated by the value supplied to the svrsslcfg —chpwd command. This value is also written into the appropriate configuration file.</td>
</tr>
<tr>
<td><strong>Note:</strong> If a certificate and the password to the keyring database file containing that certificate are both expired, then the password must be refreshed first.</td>
</tr>
<tr>
<td>The number of days value is created and the value is set by svrsslcfg.</td>
</tr>
<tr>
<td>This stanza entry is required when SSL is enabled.</td>
</tr>
<tr>
<td>Default value: 183</td>
</tr>
<tr>
<td>Example: ssl-pwd-life = 105</td>
</tr>
</tbody>
</table>

| **ssl-auto-refresh = {yes|no}** |
| --- |
| Indication of whether automatic refresh of the SSL certificate and the key database file password occur. |
| **Valid values:** |
| yes Enables automatic refresh. When enabled, the certificate and password are regenerated if either is in danger of expiration (less than half the time left). |
| no Turns off automatic certificate and password refresh. |
| This value is created and the value is set by svrsslcfg. |
| This stanza entry is required when SSL is enabled. |
| Default value: yes |
| Example: ssl-auto-refresh = no |

<table>
<thead>
<tr>
<th><strong>ssl-authn-type = certificate</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of authentication.</td>
</tr>
<tr>
<td>This value is created and the value is set by svrsslcfg.</td>
</tr>
<tr>
<td>This stanza entry is required when SSL is enabled.</td>
</tr>
<tr>
<td>Default value: certificate</td>
</tr>
<tr>
<td>Example: ssl-authn-type = certificate</td>
</tr>
<tr>
<td><strong>ssl-authn-user = user_name</strong></td>
</tr>
<tr>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>The user name that is used when the authentication type is <strong>password</strong>.</td>
</tr>
<tr>
<td>This stanza entry is optional.</td>
</tr>
<tr>
<td>There is no default value.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ssl-authn-password = password</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The password for the user name that is used if the authentication type is <strong>password</strong>.</td>
</tr>
<tr>
<td>This stanza entry is optional, but is required when <strong>ssl-authn-user</strong> is specified.</td>
</tr>
<tr>
<td>There is no default value.</td>
</tr>
<tr>
<td>Example: <strong>ssl-authn-password = mypassw0rd</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ssl-mgr-config = relative_pathname</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Location of the configuration file used to configure the Tivoli Access Manager runtime environment (pd.conf). When specified, values for master-host, master-port, and master-dn from pd.conf will override any values specified in the application configuration file (this file).</td>
</tr>
<tr>
<td>This value can be either a fully qualified path or can be a relative pathname. When the relative pathname is used, the configuration file is expected to be located under the Tivoli Access Manager installation directory.</td>
</tr>
<tr>
<td>This stanza entry is optional.</td>
</tr>
<tr>
<td>There is no default value.</td>
</tr>
<tr>
<td>Example: <strong>ssl-mgr-config = ./lib/pd.conf</strong></td>
</tr>
</tbody>
</table>
### [ldap] stanza

| enabled = {yes|true|no|false} |
|--------------------------------|
| Indication of whether LDAP is being used as the user registry. Only one user registry can be specified at a time. |
| Valid values: yes|true Enables LDAP user registry support. no|false Disables LDAP user registry support and indicates that LDAP is not the user registry being used. Anything other than yes|true is interpreted as no|false, including a blank value. |
| This stanza entry is required when LDAP is the user registry. The default value can be different, depending on how the server is configured. |
| Example: enabled = yes |

<table>
<thead>
<tr>
<th>host = host_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host name of the LDAP server. Valid values for host_name include any valid Internet Protocol (IP) host name.</td>
</tr>
<tr>
<td>The host_name value is taken from the pd.conf file. The pd.conf file is created when the Tivoli Access Manager runtime component is configured on the machine.</td>
</tr>
<tr>
<td>Use the svrsslcfg utility to set the host_name value when the configured Policy Director user registry is LDAP.</td>
</tr>
<tr>
<td>This stanza entry is required. There is no default value. The value is taken from the pd.conf file.</td>
</tr>
<tr>
<td>Examples of host names: host = libra host = libra.dallas.ibm.com</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>port = port_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-SSL IP port number that is used for communicating with the LDAP server. For port_number, use any valid port number. A valid port number is any positive number that is allowed by TCP/IP and that is not currently being used by another application.</td>
</tr>
<tr>
<td>This stanza entry is required. Default value: 389</td>
</tr>
<tr>
<td>Example: port = 389</td>
</tr>
</tbody>
</table>

| bind-dn = LDAP_dn |
### bind-dn = LDAP dn

LDAP user distinguished name (DN) that is used when binding (signing on) to the LDAP server. The `LDAP_dn` value is created, based on the server name that was specified with the `-n server_name` flag and the local host of the machine.

Use the `svrsslcfg` utility to set the `LDAP_dn` value.

This stanza entry is required when the configured user registry is LDAP.

There is no default value.

Example:

```
bind-dn = cn=ivacld/libra,cn=SecurityDaemons,secAuthority=Default
```

### bind-pwd = LDAP_password

Password for the LDAP user distinguished name identified in `bind-dn = dn` key value pair. The `LDAP_password` value is created based on the password specified by using the `-S password` flag.

Use the `svrsslcfg` utility to set the `LDAP_password` value.

This stanza entry is required when the configured user registry is LDAP.

There is no default value.

Example:

```
bind-pwd = zs77WVoLSZn1rKrL
```

### cache-enabled = {yes|true|no|false}

Indication of whether LDAP client-side caching is used to improve performance for similar LDAP queries.

Valid values:

- `yes | true` - Enables LDAP client-side caching.
- `no | false` - Disables LDAP client-side caching. This value is the default value. Anything other than `yes | true` will be interpreted as `no | false`, including a blank value.

This stanza entry is optional.

Default value: `no`

Example: `cache-enabled = no`

### prefer-readwrite-server = {yes|true|no|false}

Indication of whether the client can question the Read/Write LDAP server before querying any replica Read-only servers that are configured in the domain.

The default value can be different. For example, the default value for `ivmgrd.conf` is `yes` while the default value for `ivacld.conf` is `no`.

Valid values:

- `yes | true` - Enables the client to be able to question the Read/Write LDAP server.
- `no | false` - Disables the client. Anything other than `yes | true` will be interpreted as `no | false`, including a blank value.

This stanza entry is optional.

The default value is server dependent.

Example: `prefer-readwrite-server = no`
ssl-enabled = {yes|true|no|false}

Indication of whether to enable SSL communication with the LDAP server. The value for each server can be different, depending on how the server was configured.

Valid values:
- **yes|true**: Enables SSL communication.
- **no|false**: Disables SSL communication. Anything other than **yes|true** will be interpreted as **no|false**, including a blank value, and SSL will be automatically configured.

This stanza entry is optional.

The default value is server dependent.

Example: ssl-enabled = yes

ssl-keyfile = ldap-ssl-key-filename

SSL key file name and location containing the LDAP server certificate. Use the SSL key file to handle certificates that are used in LDAP communication. The file type can be anything but the extension is usually .kdb.

This stanza entry is required only when ssl-enabled = yes.

Default value is server dependent.

Example for UNIX: ssl-keyfile = /opt/PolicyDirector/keytab/ivmgrd.kdb

ssl-keyfile-dn = keyLabel

Key label of the client personal certificate within the SSL key file. This key label is used to identify the client certificate that is presented to the LDAP server.

This stanza entry is required when the LDAP server is configured to perform client authentication.

There is no default value.

Example: ssl-keyfile-dn = "PD_LDAP"

ssl-keyfile-pwd = ldap-ssl-keyfile-password

Password to access the SSL key file.

**Note:** The password associated with the default SSL keyfile is gsk4ikm.

This stanza entry is required only if ssl-enabled = yes.

There is no default value.

Example: ssl-keyfile-pwd = mysslpwd

ssl-port = port_number

SSL IP port that is used to connect to the LDAP server. For **port_number**, use any valid port number. A valid port number is any positive number that is allowed by TCP/IP and that is not currently being used by another application.

This stanza entry is required only if ssl-enabled = yes.

Default value: 636

Example: ssl-port = 636

max-search-size = [0|number_entries]
Limit for the maximum search size, specified as the number of entries, that can be returned from the LDAP server. The value for each server can be different, depending on how the server was configured.  
**Note:** This value can also be limited by the LDAP server itself.

Valid values:

- 0: The number is unlimited; there is no limit to the maximum search size.

- `number_entries`: The maximum number of entries for search, specified as an integer whole number. WebSEAL does not impose a limit on the value of this integer. See the discussion on maximum integer values in "Guidelines for configuring stanzas" on page 185.

This stanza entry is optional.

Default value: 2048

Example: `max-search-size = 2048`

**auth-using-compare = {yes|true|no|false}**

Choice of whether `ldap_compare()` will be used instead of the `ldap_bind()` call to verify the password and authenticate the user. For those LDAP servers that allow it, a compare operation might perform faster than a bind operation. The value for each server can be different, depending on how the server was configured.

This option changes the method used by these authorization API calls:

- `azn_util_client_authenticate()`
- `azn_util_password_authenticate()`

Valid values:

- `yes|true`: A compare operation will be used to authenticate LDAP users instead of a bind operation.

- `no|false`: A bind operation will be used to authenticate LDAP users instead of a compare operation. Anything other than `yes|true` will be interpreted as `no|false`, including a blank value.

This stanza entry is optional.

Default value: `yes|true`

Example: `auth-using-compare = true`
Definition of the LDAP user registry replicas in the domain where:

- `ldap_server` is the network name of the server
- `port` is the port number for the LDAP server. A valid port number is any positive number that is allowed by TCP/IP and that is not currently being used by another application.
- `type` is one of `readonly` or `readwrite`
- `preference` is a number from 1 to 10 (10 is the highest preference)

This stanza entry is optional.

Default value is that no replicas are specified.

Example of one replica specified and two replicas commented out:

```
replica = freddy,390,readonly,1
#replica = barney,391,readwrite,2
#replica = benny,392,readwrite,3
```
### [uraf-ad] stanza

<table>
<thead>
<tr>
<th><strong>ad-server-config = fully_qualified_path</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Location and file name of the Active Directory registry activedir.conf configuration file. This value is generated, but it can be changed.</td>
</tr>
<tr>
<td>The <em>fully_qualified_path</em> value represents an alphanumeric, non-case sensitive string. The maximum string length for the Active Directory user registry is 256 alphanumeric characters.</td>
</tr>
<tr>
<td>This stanza entry is required when your user registry is Microsoft Active Directory and is required only for configuration files other than activedir.conf.</td>
</tr>
<tr>
<td>Default value for UNIX: /opt/PolicyDirector/etc/activedir.conf</td>
</tr>
<tr>
<td>Default value for Windows: c:\Program files\tivoli\Policy Director\etc\activedir.conf</td>
</tr>
<tr>
<td>Example for UNIX: ad-server-config = /opt/PolicyDirector/etc/activedir.conf</td>
</tr>
</tbody>
</table>

| **enabled = {yes|no}** |
|----------------------|
| Indication of whether Active Directory is being used as the user registry. |
| Valid values: |
| yes Indicates that Active Directory is the user registry. |
| no Indicates that Active Directory is not the user registry. Anything other than yes, including a blank, is interpreted as no. |
| This stanza entry is required when your user registry is Microsoft Active Directory. |
| Default value: no |
| Example: enabled = yes |

| **multi-domain = {true|false}** |
|-----------------------------|
| Indication of whether the domain is a single-domain or multi-domain configuration. You select the value at the time the runtime is configured for Tivoli Access Manager. The value is filled in automatically, based on information supplied during the runtime configuration. |
| Valid values: |
| true For multiple Active Directory domains. |
| false For a single Active Directory domain. |
| This stanza entry is required when your user registry is Microsoft Active Directory. |
| There is no default value. |
| Example: multi-domain = true |
### hostname = hostname

*Active Directory domain name system (DNS) host name. The value is filled in automatically, based on information supplied during the runtime configuration. The *hostname* is an alphanumeric, non-case sensitive string. The dot (.) cannot be the last character of the host name. The maximum string length for the Active Directory user registry is 256 alphanumeric characters.*

This stanza entry is required when your user registry is Microsoft Active Directory.

There is no default value.

Example: `hostname = adserver.tivoli.com`

### domain = root_domain_name

*Active Directory root (primary) domain. The value is filled in automatically, based on information supplied during the runtime configuration. The *root_domain_name* is an alphanumeric, non-case sensitive string. The maximum length for the domain name is user registry dependent. For Active Directory that maximum length is 256 alphanumeric characters.*

This stanza entry is required when `multi-domain = true`.

There is no default behavior.

Example: `domain = dc=tivoli,dc=com`

### useEncryption = {true|false}

*Indication of whether encryption communication to Active Directory is being used. This value is filled in automatically, based on information supplied during the runtime configuration.*

**Valid values:**
- **true** — Enables encryption communication.
- **false** — Disables encryption communication.

This stanza entry is required when your user registry is Microsoft Active Directory.

There is no default behavior.

Example: `useEncryption = false`

### bind-id = ad_id

*Active Directory administrator or user log-in identity that is used to bind (sign on) to the registry server. If the ID belongs to a user rather than an administrator, the Active Directory user must have enough privileges to update and modify data in the user registry.*

The *ad_id* value is an alphanumeric, non-case sensitive string. The minimum and maximum lengths of the ID, if there are limits, are imposed by the underlying registry. For Active Directory the maximum length is 256 alphanumeric characters.

This value is filled in automatically, based on information supplied during server configuration. Whenever you change this value after the configuration is completed, a conflict might occur.

This stanza entry is required when your user registry is Microsoft Active Directory.

The default value is generated; do not change it.

Example: `bind-id = adpdadmin`

### bind-pwd = admin_password
Encoded administrator log-in password that is used to bind (or sign on) to the Active Directory registry server. Additional password requirements can be specified for Tivoli Access Manager after the product is installed. However, the initial password does not necessarily conform to those requirements.

The `admin_password` value is filled in automatically, based on information that is supplied during server configuration. The password is an encrypted string.

This stanza entry is required when your user registry is Microsoft Active Directory. The default value is generated; do not change it.

Example: `bind-pwd = MyADbindPwd`

<table>
<thead>
<tr>
<th><code>dnforpd = ad_dn</code></th>
</tr>
</thead>
</table>
| Distinguished name that is used by Active Directory to store Tivoli Access Manager data. The `ad_dn` value is an alphanumeric, non-case sensitive string. The minimum and maximum lengths of the distinguished name, if there are limits, are imposed by the underlying registry. For Active Directory the maximum length is 256 alphanumeric characters.

This stanza entry is required when your user registry is Microsoft Active Directory. The default value is generated; do not change it.

Example: `dnforpd = dc=child2,dc=com` |
### [uraf-domino] stanza

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Values</th>
<th>Default Value</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>domino-server-config = fully_qualified_path</code></td>
<td>Location and file name of the Active Directory registry <code>activedir.conf</code> configuration file. Name and location of the Lotus Domino registry <code>domino.conf</code> configuration file. The <code>fully_qualified_path</code> represents an alphanumeric string.</td>
<td></td>
<td>Default value for UNIX: <code>/opt/PolicyDirector/etc/domino.conf</code> Default value for Windows: <code>c:\Program files\tivoli\Policy Director\etc\domino.conf</code></td>
<td>Example for Windows: <code>domino-server-config = c:\Program files\tivoli\Policy Director\etc\domino.conf</code></td>
</tr>
<tr>
<td>`enabled = {yes</td>
<td>no}`</td>
<td>Indication of whether Domino is being used as the user registry. Valid values: yes Indicates Domino is the user registry. no Indicates Domino is not the user registry. Anything other than yes, including a blank, is interpreted as no.</td>
<td>yes, no</td>
<td>Default value: no</td>
</tr>
<tr>
<td><code>server = server_name</code></td>
<td>Name of the Lotus Domino™ server. The <code>server_name</code> value represents an alphanumeric, non-case sensitive string. The minimum and maximum lengths of the name are imposed by the underlying registry.</td>
<td></td>
<td>There is no default value.</td>
<td>Example: <code>server = grizzly/Austin/IBM</code> Where grizzly is the Domino server machine host name and the remainder is the Domino domain name.</td>
</tr>
<tr>
<td><code>hostname = hostname</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Lotus Domino server TCP/IP host name.</strong> The <em>hostname</em> value is manually input during configuration. The <em>hostname</em> value must be an alphanumeric, non-case sensitive string. The format is the same as a typical TCP/IP host name.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This stanza entry is required when your user registry is Lotus Domino.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>There is no default value.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Example: <code>hostname = myhost.austin.ibm.com</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>LDAPPort = <em>port_number</em></strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAP port number for the Lotus Domino server. The <em>port_number</em> value is manually input during configuration. A valid port number is any positive number that is allowed by TCP/IP and that is not currently being used by another application.</td>
</tr>
<tr>
<td>This stanza entry is required.</td>
</tr>
<tr>
<td>Default value: 389</td>
</tr>
<tr>
<td>Example: <code>LDAPPort = 389</code></td>
</tr>
</tbody>
</table>

| **UseSSL = {yes|no}** |
|---|
| Indication of whether to use SSL. |
| Value values: |
| yes | Specifies that you want to use SSL. |
| no | Specifies that you do not want to use SSL. |
| This stanza entry is required. |
| Default value: yes |
| Example: `UseSSL = no` |

<table>
<thead>
<tr>
<th><strong>keyfile = <em>filename</em></strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>SSL key file name and location. Use the SSL key file to handle certificates that are used in LDAP communication. The <em>filename</em> is an alphanumeric, non-case sensitive string that must conform to the underlying (Windows) file system naming convention. The file must be an existing file on the client machine. The file type can be anything but the extension is usually .kdb.</td>
</tr>
<tr>
<td>This stanza entry is required when <code>UseSSL = yes</code></td>
</tr>
<tr>
<td>The default value is server-dependent.</td>
</tr>
<tr>
<td>Example for Windows: <code>keyfile = C:/pd/keytab/ivacld.kdb</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>KeyFile_PW = <em>password</em></strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Password for the SSL key file.</td>
</tr>
<tr>
<td>This stanza entry is required when <code>UseSSL = yes</code></td>
</tr>
<tr>
<td>The default value is server-dependent.</td>
</tr>
<tr>
<td>Example: <code>KeyFile_PW = mykeyfilepw</code></td>
</tr>
</tbody>
</table>

| **KeyFile_DN = *cert_label*** |
| Distinguished name (DN) of the SSL key file private key. The `cert_label` value is an alphanumeric string that represents the certificate label of the client personal certificate within the SSL key file. This key label is used to identify the client certificate that is presented to the Domino server.  
This stanza entry is required when `UseSSL = yes`  
There is no default value. |
|---|
| password = password  
Domino server password that is used to bind (or sign on) to the Domino registry server. The password is an encrypted string.  
This stanza entry is required when `enabled = yes`  
The value is generated; do not change it.  
Example: `password = myEncryptedSrvrPwd` |
| NAB = names.nsf  
Lotus Domino Name and Address Book (NAB) database. The `names.nsf` database is set at configuration time and cannot be changed.  
This stanza entry is required when `enabled = yes`  
Default value: `names.nsf`  
Example: `NAB = names.nsf` |
| PDM = nsf_filename  
Tivoli Access Manager meta-data database. The `nsf_filename` represents a Domino database file name. The file name conforms to the underlying operating system file naming conventions of the Domino server. The recommended file extension is `.nsf`. This file is created on the Domino server during configuration.  
This stanza entry is required when `enabled = yes`  
Default value: `PDMdata.nsf`  
Example: `PDM = PDMdata.nsf` |
[aznapi-admin-services] stanza

<table>
<thead>
<tr>
<th>[aznapi-admin-services] stanza</th>
</tr>
</thead>
<tbody>
<tr>
<td>service-id = {short_name</td>
</tr>
</tbody>
</table>

Defines the authorization API service for functions that enable a plug-in to obtain the contents of a defined portion of the protected object hierarchy, or to enable a plug-in to define application-specific administration tasks that also return commands that perform those tasks. Each stanza entry defines different types of authorization API services, and each entry is the same format where:

**service-id**

Developer-specified identification (ID) of the administration service. An authorization API application can register more than one administration service plug-in, but each must have a unique service ID.

{short_name|path_to_dll}

The path to the dynamic link library (DLL) that contains the service executable code.

If the DLL resides in a directory that is normally searched by the system for DLLs (for example, /usr/lib on UNIX platforms and %PATH% on Windows NT), you do not need to specify the full path to the DLL, only the DLL name. If you want a platform-independent DLL name, so it can be loaded on any supported Tivoli Access Manager platform, provide a short form library name. The short name is prepended and appended with known library prefixes and suffixes for each platform and each possibility is searched for in turn. For example, using a short form library name of azn_ent_user, the following names are automatically searched for on each platform:

- **NT:** azn_ent_user.dll
- **AIX:** libazn_ent_user.so, libazn_ent_user.a
- **Solaris:** libazn_ent_user.so
- **HP/UX:** libazn_ent_user.sl

**protected_object_hierarchy_name**

The protected object hierarchy name is an optional parameter. This parameter refers to either the name of a protected object space (hierarchy) or simply to a protected object. Protected object hierarchy names must be unique for each administration service plug-in within the scope of an authorization API application. Multiple authorization API application instances, however, can register to service the same protected object hierarchy names, which provides failover support for administration of an object space in the event that a particular authorization API application server fails.

**params**

Optionally, the external authorization service can be passed additional initialization information in the form of arguments. The arguments must be preceded by the ampersand (for example, &-server fred). The authorization service does not process the characters after the ampersand &. It passes these characters directly to the administration service plug-in. The service definition is discussed in more detail in the "Implementing a service plug-in" on page 58.

This stanza entry is optional.

There is no default value.

Example: AZN_ADMIN_SVC_TRACE = pdtraceadmin
[aznapi-cred-modification-services] stanza

<table>
<thead>
<tr>
<th>[aznapi-cred-modification-services] stanza</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>service-id</strong> = short_name</td>
</tr>
</tbody>
</table>

Defines the authorization API service for the credentials attribute list modification service. Each stanza entry defines different types of authorization API service, and each entry is in the same format where:

**service-id**

Developer-specified identification (ID) of the credential modification service. The service ID string must be unique.

**[short_name] path_to_dll**

The path to the dynamic link library (DLL) that contains the service executable code.

If the DLL resides in a directory that is normally searched by the system for DLLs (for example, /usr/lib on UNIX platforms and %PATH% on Windows NT), you do not need to specify the full path to the DLL, only the DLL name. If you want a platform-independent DLL name, so it can be loaded on any supported Tivoli Access Manager platform, provide a short form library name. The short name is prepended and appended with known library prefixes and suffixes for each platform and each possibility is searched for in turn. For example, using a short form library name of azn_ent_user, the following names are automatically searched for on each platform:

- NT: azn_ent_user.dll
- AIX: libazn_ent_user.so, libazn_ent_user.a
- Solaris: libazn_ent_user.so
- HP/UX: libazn_ent_user.sl

**params**

Optionally, you can specify parameters to pass to the service when it is initialized by the authorization API. Parameters are considered to be all data following the ampersand (&) symbol in the string. The service definition is discussed in more detail in "Implementing a service plug-in" on page 58.

This stanza entry is optional.

There is no default value.

Example: AZN_MOD_SVC_RAD_2AB = azn_mod_rad

---

Appendix C. Authorization API client configuration file 209
[aznapi-entitlement-services] stanza

| service-id = [short_name|path_to_dll] [ & params ... ] |
|----------------------------------------------------------|

Defines the authorization API service for the protected objects entitlements service. Each stanza entry defines different types of authorization API service, and each entry is of the same format where:

**service-id**

Developer-specified identification (ID) by which the service can be identified by the authorization API client. The service ID string must be unique.

**{short_name|path_to_dll}**

The path to the dynamic link library (DLL) that contains the service executable code.

If the DLL resides in a directory that is normally searched by the system for DLLs (for example, /usr/lib on UNIX platforms and %PATH% on Windows NT), you do not need to specify the full path to the DLL, only the DLL name. If you want a platform-independent DLL name, so it can be loaded on any supported Tivoli Access Manager platform, provide a short form library name. The short name is prepended and appended with known library prefixes and suffixes for each platform and each possibility is searched for in turn. For example, using a short form library name of azn_ent_user, the following names are automatically searched for on each platform:

- **NT:** azn_ent_user.dll
- **AIX:** libazn_ent_user.so, libazn_ent_user.a
- **Solaris:** libazn_ent_user.so
- **HP/UX:** libazn_ent_user.sl

**params**

Optionally, you can specify one or more parameters to pass to the service when it is initialized by the authorization API. Parameters are considered to be all data following the ampersand (&) symbol in the string. The service definition is discussed in more detail in [[Implementing a service plug-in” on page 58]](https://www.ibm.com). This stanza entry is optional.

There is no default value.

Example: `AZN_ENT_EXT_ATTR = azn_ent_ext_attr`
**[aznapi-external-authzn-services] stanza**

<table>
<thead>
<tr>
<th><strong>[aznapi-external-authzn-services] stanza</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>**policy-trigger = {short_name</td>
</tr>
</tbody>
</table>

 Defines the authorization API service for external authorization service definitions that force authorization decisions to be made based on application-specific criteria. Each stanza entry defines different types of authorization API service, and each entry is the same format where:

- **policy-trigger**
  
  The policy trigger is the means by which the external authorization service is invoked by the authorization engine. It is one of either a service ID or an access control list (ACL) action string. For example, it can be `my_service_1` or `Trx`. If the service is defined with a service ID, the service ID will be used as an extended attribute on a POP policy that triggers the external authorization service whenever an object has this POP attached to it. If the service is defined using an ACL action string, the service will be invoked whenever this ACL action mask is requested as part of an authorization decision.

  The policy-trigger can be any string that is recognized as a valid key name. The `policy-trigger` string is case sensitive for action set definitions because the actions themselves are case sensitive. However, the `policy-trigger` is case insensitive if the trigger is a POP attribute.

- **{short_name|path_to_dll}**
  
  The path to the dynamic link library (DLL) that contains the service executable code.

  If the DLL resides in a directory that is normally searched by the system for DLLs (for example, `/usr/lib` on UNIX platforms and `%PATH% on Windows NT), you do not need to specify the full path to the DLL, only the DLL name. If you want a platform-independent DLL name, so it can be loaded on any supported Tivoli Access Manager platform, provide a short form library name. The short name is prepended and appended with known library prefixes and suffixes for each platform and each possibility is searched for in turn. For example, using a short form library name of `azn_ent_user`, the following names are automatically searched for on each platform:

  - **NT:** `azn_ent_user.dll`
  - **AIX:** `libazn_ent_user.so, libazn_ent_user.a`
  - **Solaris:** `libazn_ent_user.so`
  - **HP/UX:** `libazn_ent_user.sl`

- **[-weight number]**

  A weighting that is assigned in the access decision process to the particular external authorization service. The weight parameter is an unsigned `size_t` value and is optional. The value signifies the weight that any decision returned by this external authorization service should be given in the entire decision process. The default value is 101.

- **params**

  Optionally, the external authorization service can be passed additional initialization information in the form of arguments. The arguments must be preceded by the ampersand (for example, `& -server fred`). The service definition is discussed in more detail in "Implementing a service plug-in" on page 58.

This stanza entry is optional. There is no default value.
**[aznapi-pac-services] stanza**

| service-id = [short_name|path_to_dll] [ & params ... ] |
|-------------------------------------------------------------|
| Defines the authorization API service for the Tivoli Access Manager privilege attribute certificate (PAC) encoding service. Each stanza entry defines different types of authorization API service, and each entry is the same format where: |
| **service-id** |
| Developer-specified identification (ID) of the PAC service that produces the PAC. The service ID string must be unique. |
| **{short_name|path_to_dll}** |
| The path to the dynamic link library (DLL) that contains the service executable code. |
| If the DLL resides in a directory that is normally searched by the system for DLLs (for example, /usr/lib on UNIX platforms and %PATH% on Windows NT), you do not need to specify the full path to the DLL, only the DLL name. If you want a platform-independent DLL name, so it can be loaded on any supported Tivoli Access Manager platform, provide a short form library name. The short name is prepended and appended with known library prefixes and suffixes for each platform and each possibility is searched for in turn. For example, using a short form library name of azn_ent_user, the following names are automatically searched for on each platform: |
| **NT:** azn_ent_user.dll |
| **AIX:** libazn_ent_user.so, libazn_ent_user.a |
| **Solaris:** libazn_ent_user.so |
| **HP/UX:** libazn_ent_user.sl |
| **params** |
| Optionally, you can specify parameters to pass to the service when it is initialized by the authorization API. Parameters are considered to be all data following the ampersand (&) symbol in the string. The service definition is discussed in more detail in [Implementing a service plug-in](#) on page 58 |

This stanza entry is optional.

There is no default value.
[authentication-mechanisms] stanza

This stanza defines the library that is to be used for password and certificate authentication.

The configuration entries in this stanza are required by the server to communicate with a user registry. You can use either a User Registry Adapter Framework (URAF) registry (either Active Directory or Domino) or an LDAP registry library, depending on the type of user registry.

Because you can specify only one type of user registry, certain key value pairs in the [authentication-mechanisms] stanza are mutually exclusive. For example:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>passwd-uraf</td>
<td>/opt/PolicyDirector/lib/liburafauthn.a</td>
</tr>
<tr>
<td>cert-uraf</td>
<td>/opt/PolicyDirector/lib/liburafcertauthn.a</td>
</tr>
<tr>
<td>passwd-ldap</td>
<td>C:\pd\bin\ldapauthn.dll &amp; -cfgfile [C:/pd/etc/ivacld.conf]</td>
</tr>
<tr>
<td>cert-ldap</td>
<td>C:\pd\bin\certauthn.dll &amp; -cfgfile [C:/pd/etc/ivacld.conf]</td>
</tr>
</tbody>
</table>

In this example, the URAF registry items are commented out by using the pound sign (#); the LDAP-oriented stanza entries are not commented out.

You can manually edit these values; no configuration utility is required.

<table>
<thead>
<tr>
<th>[authentication-mechanisms] stanza</th>
</tr>
</thead>
<tbody>
<tr>
<td>passwd-uraf =  fully_qualified_path</td>
</tr>
<tr>
<td>Location of the library to use for password authentication.</td>
</tr>
</tbody>
</table>

This stanza entry is required when you use a URAF registry as your user registry.

Default values:
- AIX: /opt/PolicyDirector/lib/liburafauthn.a
- HP: /opt/PolicyDirector/lib/liburafauthn.sl
- Sun: /opt/PolicyDirector/lib/liburafauthn.so
- Linux: /opt/PolicyDirector/lib/liburafauthn.so
- Windows: install_dir\bin\urafauthn.dll

Example for Linux: passwd-uraf = ../opt/PolicyDirector/lib/liburafauthn.so

| cert-uraf =  fully_qualified_path |
| Location of the library to use for certificate authentication. |

This stanza entry is required when you use a URAF registry as the user registry.

Default values:
- AIX: /opt/PolicyDirector/lib/liburafcertauthn.a
- HP: /opt/PolicyDirector/lib/liburafcertauthn.sl
- Solaris: /opt/PolicyDirector/lib/liburafauthn.so
- Linux: /opt/PolicyDirector/lib/liburafcertauthn.so
- Windows: install_dir\bin\urafcertauthn.dll

Example for Solaris: cert-uraf = ../opt/PolicyDirector/lib/liburafauthn.so
### passwd-ldap = fully_qualified_path

Location of the library to use for LDAP password authentication.

This stanza entry is required when you use LDAP as the user registry.

Default values:

- AIX: /opt/PolicyDirector/lib/libldapauthn.a
- HP: /opt/PolicyDirector/lib/libldapauthn.sl
- Solaris: /opt/PolicyDirector/lib/libldapauthn.so
- Linux: /opt/PolicyDirector/lib/libldapauthn.so
- Windows: `<install_dir>/bin/ldapauthn.dll`

Example for AIX: `passwd-ldap = ../opt/PolicyDirector/lib/libldapauthn.a`

### cert-ldap = fully_qualified_path

Location of the library to use for LDAP certificate authentication.

This stanza entry is required when you use LDAP as the user registry.

Default values:

- AIX: /opt/PolicyDirector/lib/libcertauthn.a
- HP: /opt/PolicyDirector/lib/libcertauthn.sl
- Solaris: /opt/PolicyDirector/lib/libcertauthn.so
- Linux: /opt/PolicyDirector/lib/libcertauthn.so
- Windows: `<install_dir>/bin/certauthn.dll`

Example for Windows: `passwd-ldap = `C:\pd\bin\certauthn.dll` & -cfgfile [C:\pd/etc/ivacld.conf]`
**[manager] stanza**

<table>
<thead>
<tr>
<th><strong>[manager] stanza</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>master-host = server_hostname</strong></td>
</tr>
<tr>
<td>Host name of the Tivoli Access Manager server.</td>
</tr>
<tr>
<td>Examples of valid host names:</td>
</tr>
<tr>
<td>• mycomputer.city.company.com</td>
</tr>
<tr>
<td>• mycomputer</td>
</tr>
<tr>
<td>This stanza entry is used for both remote and local mode.</td>
</tr>
<tr>
<td>This stanza entry is required.</td>
</tr>
<tr>
<td>There is no default value.</td>
</tr>
<tr>
<td>Example: master-host = libra</td>
</tr>
</tbody>
</table>

| **master-port = port_number** |
| TCP port on which the server is listening for requests. This value is created and set by `svrsslcfg` |
| For `port_number`, use any valid port number. A valid port number is any positive number that is allowed by TCP/IP and that is not currently being used by another application. It is recommended that you use the default port number value, or use a port number over 1000 that is currently not being used. |
| This stanza entry is used for both remote and local mode. |
| This stanza entry is required. |
| Default value: 7135 |
| Example: master-port = 7135 |
replica = hostname:port_number:preference:replica_certificate_dn

Configuration settings for policy server replicas. This stanza entry is used for remote mode only.

The values for this stanza entry are set by using the `svrslcfg` utility with the `–add_replica` option.

The values are:
• **hostname**
  The hostname of the policy server replica.
• **port_number**
  The port on which to communicate with the policy server replica
• **preference**
  Integer value between 1 and 10, specifying the priority for using this particular replica to provide authorization decisions. This is useful when the domain contains multiple replicas. Replicas are accessed in order of their preference integer. The replica with the largest integer value is accessed first.
• **replica_certificate_DN**
  The distinguished name used to access the replica’s certificate.

This stanza entry is optional.

There is no default value.

Example (entered as one continuous line):
replica = repl.acme.com:7135:5:cn=ivmgrd/repl.acme.com,o=Policy Director,C=US
Appendix D. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
2Z4A/101
11400 Burnet Road
Austin, TX 78758
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM’s application programming interfaces.
Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

**Trademarks**

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX  
DB2  
IBM  
IBM logo  
SecureWay  
Tivoli  
Tivoli logo  
WebSphere  

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.
Index

A
access control decisions
making 139
making and extending 141
access decision function (ADF) 1
access enforcement function (AEF) 2
adding
additional application-specific context 40
attributes for LDAP access 26
authorization to an application 3
credentials and handle 119
name or buffer value to attribute list 104
name or value to attribute list 103
additional user information 37
address of data structure 38
ADF (access decision function) 1
ADK 1
administration service
configuring 83
error codes 86
implementing 81
initializing 84
shutdown 84
supported functions 85
administrator’s distinguished name 27
AEF (access enforcement function) 2
AIX
libivauthzn.a library file 3
allocated memory 41
API
attribute lists 7
authorization decisions 8
credentials 8
error handling 8
extensions 9
functions 7
API functions
azn_attrlist_add_entry_buffer() 104
azn_attrlist_add_entry() 103
azn_attrlist_create() 107, 108
azn_attrlist_delete() 109, 110
azn_attrlist_get_entry_buffer_value() 116
azn_attrlist_get_entry_string_value() 114
azn_attrlist_get_names() 117
azn_attrlist_name_get_num() 118
azn_creds_combine() 119
azn_creds_create() 122
azn_creds_delete() 123
azn_creds_for_subject() 125
azn_creds_get_attrlist_for_subject() 128
azn_creds_get_pac() 130
azn_creds_modify() 132
azn_creds_num_of_subjects() 135
azn_decision_access_allowed_ext() 141
azn_decision_access_allowed() 139
azn_error_major() 147
azn_error_minor_get_string() 149
azn_error_minor() 148
azn_id_get_creds() 150
azn_initialize() 152
azn_pac_get_creds() 155

API functions (continued)
azn_release_buffer() 157
azn_release_strings() 160
azn_shutdown() 161
azn_util_password_authenticator() 164
Application Development Kit (ADK) 1
applications
building 5
building an attribute list 40
deploying with the Authorization API 6
determining user’s authorization credentials 36
array of strings
memory, releasing 42
storage, freeing 160
assigning
handle for an empty attribute list 107, 108
handle to empty credentials structure 122
user credentials to a credentials handle 40
attribute list functions 7
attribute lists
adding name or buffer value 104
adding name or value 103
building for additional application information 40
creating 13
creating and assigning a handle 107, 108
deleting 13, 109, 110
getting an attribute name 13
getting the number of values 13
getting values 13
obtaining a credential 43
releasing memory 41
setting an entry 13
audit
identifier 129
user information user_info 37
authenticated user identity 36
authentication
identity, user 37
information 37, 164
mechanism 36
authentication-mechanisms stanza 213
authority, authorization 37
authorization
authority 37
credentials 36, 38, 39
decisions 8, 39, 41
authorization API
changing the credential’s contents 44
Authorization API
buffer 38
building applications 5
character strings 9
converting credentials to a transportable format 42
converting credentials to the native format 43
creating a chain of credentials 43
deploying applications 6
determining the number of credentials in a chain 43
handling credentials 42
initializing 152
introducing 1
obtaining a credential from a chain 43

© Copyright IBM Corp. 2000,2002
Authorization API (continued)

obtaining credential from a chain 43
shutting down 41
specifying cache mode type 18
tasks 6

Authorization server
specifying cache mode type 18
authorization service
initializing 152
minor error codes 4
minor errors 15
starting 33
submitting requests to 2

authorization service dispatcher 54

azn_admin_get_object() 170
azn_admin_get_objectlist() 172
azn_admin_get_tasklist() 174
azn_attrlist_add_entry_buffer() 40
azn_attrlist_add_entry() 40, 46
azn_attrlist_delete() 109
azn_attrlist_get_entry_buffer_value() 111, 116
azn_attrlist_get_entry_string_value() 114
azn_attrlist_get_names() 117
azn_attrlist_h_ 12
azn_attrlist_h_t 41
azn_attrlist_name_get_num() 118
azn_buffer_desc 10
azn_buffer_t 10
AZN_C_AUDIT_ID 129
AZN_C_EMPTY_BUFFER 10
AZN_C_INITIATOR_INDEX 43, 46, 125, 128
AZN_C_NO_BUFFER 10
AZN_C_NOT_PERMITTED 41, 139, 141
AZN_C_PERMITTED 41, 139, 141
AZN_C_VERSION 34
azu_creds_combine() 43, 119
azu_creds_create() 41, 43, 122
azu_creds_delete() 123
azu_creds_for_subject() 44, 125
azu_creds_get_attrlist_for_subject() 46, 128
azu_creds_get_pac() 42, 130
azu_creds_h_t 14, 41, 43
azu_creds_modify() 44, 132
azu_creds_num_of_subjects 44
azu_creds_num_of_subjects() 43, 135
azu_decision_access_allowed_ext() 41, 141
azu_decision_access_allowed() 39, 41, 139
azu_error_get_string() 15
azu_error_major() 147
azu_error_minor_get_string() 149
azu_error_minor() 148
azu_id_get_creds() 38, 40, 42, 150
azu_initialize() 152
azu_operation_read 39
azu_operation_traverse 39
azu_pac_get_creds() 155
azu_release_buffer() 157
azu_release_pobj() 42
azu_release_string 9
azu_release_strings() 9, 159
azu_release_strings() 9, 160
AZN_S_ATTR_VALUE_NOT_BUFFER_TYPE 112, 116

B
backwards compatibility
Solaris limitations 49
Version 3.8 49
Version 3.9 49

backwards compatibility
deprecated API elements 49

browser information 38
buffer 9
declaration 9
empty 10
buffer attribute value 111
buffers
declaration 38
introduction to 9
none 10
release of memory 42
storage, freeing 157
building
applications 5
attribute lists 40

C
cache modes 18
chain of credentials 43, 135
changing
contents of a credential 44
existing credential 132
character strings 9
cleaning up 41, 161
combining credentials and handle 119
components of
ADK 3
Tivoli Access Manager 3
configuration file 60
configuration files
ldap.conf 197
configuring
Authorization API 18
Policy Director secure domain 3
contents of the credential 44
converting
credentials to a transportable format 42
creating
attribute lists 13
chain of credentials 43
empty credentials structure 122
new attribute lists 18, 40
privilege attribute certificates 130
valid empty attribute list 107, 108
credentials 8
changing 132
changing the credential’s contents 44
combining with a handle 119
converting to a transportable format 42
converting to the native format 43
creating a chain of credentials 43
creating and assigning a handle 122
deleting 123
determining number of credentials 43
extracting individual credentials 125
handle 40
invoking a privilege attribute certificate service 130
making access control decisions 139
making extended access control decisions 141
obtaining attribute list from a credential 44
obtaining for user authorization 36, 38
obtaining from a chain of credentials 43
returning handle to new PAC credentials 155
returning in a chain 135
returning information from 128
user authorization 38
creds_attrlist 46
custom-protected object 40

data type structure
azn_attrlist_h_t 12
azn_buffer_t 9
azn_status_t 14
azn_string_t 9
decision
authorization 41
decision, authorization 41
decisions
access control 139, 141
authorization 8
defining
security policy 3
deleting
attribute list 13, 109, 110
credentials 123
deploying
applications 6
applications into secure domain 3
deprecated API elements 49
deprecated DCE authentication 51
determining
authorization credentials for a user 38
identity for a user 37
number of credentials in a credentials chain 43
disabling
notification listener 20
refreshes of local authorization policy database 20
distinguished name 27

e
empty credentials chain 122
enabling
notification listener 20
entitlements service
authorizing a caller 78
configuring 76
error codes 78
initializing 76
implementing 75
shut down 76
entitlements service plug-in
sample code 66
derror codes
aznutils.h 4, 15
ogauthzn.h 4, 15
pdb*msg.h 4, 15
derror handling 8, 14
dexample of
assigning user identity information 38
attribute list initialization data 20
declaring a buffer 38
extending
API function standard 9
function for obtaining an access decision 41
extensions, API 9
external authorization server (see Authorization server) 40
external authorization service
architecture 90
authorization decision 96
configuring 94
terror codes 97
implementing 89
initializing 95
policy triggers 92
shutdown 95
weightings 93
extracting individual credentials 125

f
files
aznutils.h 4, 15
ogauthzn.h 4, 15
pdb*msg.h 15
format
credentials, transportable 42
freeing
array of strings storage 160
buffer storage 157
string storage 159

G
getting
attribute list name 13
getting (continued)
- entry string value  114
- handle for a specified identity  150
- name attributes  117
- number of attribute entries  118
- number of values for attribute list name  13
- value attributes  13

H
- handle  119, 122, 123, 150
  - credentials  14, 40
- handling credentials  42
- header files
  - aznutils.h  4
  - ogauthzn.h  4
- host name, LDAP server  27

I
- IBM Directory user registry
  - See LDAP server
- identities, user  36, 37
- implementation modes  1
- initialization  8
- initializing
  - authorization service  152
  - data  152
  - invalid data  153
- initiator  2
- installing
  - Policy Director  6
- interfaces
  - Authorization API manual pages  101
- International Organization for Standardization (ISO)  1
- introduction to
  - Authorization API  1
- IP address  38
- ISO (International Organization for Standardization)  1
- ivacld-servers  35

K
- key file, SSL  28
- key label, SSL  28

L
- LDAP
  - administrator’s distinguished name  27
  - administrator’s password  28
  - configuration file ldap.conf  197
  - key file password  29
  - port number  27
  - server host name  27
  - server key label  28
  - SSL key file  28
- ldap stanza  189, 197
- ldap.conf configuration file  197
- length
  - data structure  38
  - local cache mode  1, 18
- logging in
  - using username and password pair  164

M
- major errors  4, 14, 147
- making
  - access control decisions  139
  - extended access control decisions  141
- manager stanza  215
- manual pages
  - summary  101
- mapping
  - requested resource to protected object  40
- user operation to a permission  39
- memory
  - credential structure  14
  - release  41
- minor errors  4, 15, 148
- mod_info  44
- mod_svc_id  44
  - mode
    - local cache  1
    - remote cache  1
- modes, specifying  18
- modifying
  - contents of a credential  44
  - existing credential  132

N
- name value  117
- notification listener  19
- number of
  - bytes in the data  9
  - individual credentials in a chain  135
  - seconds before refreshing  20
  - value attributes in the entry  118
  - values for an attribute name  13
- number of port, LDAP server  27

O
- obtaining
  - authorization decision  41
  - credential from a chain of credentials  43
  - user authorization credentials  37, 38
- ogauthzn.h  4, 15
- Open Group  1
- output parameters
  - authorization decision  41
  - extended authorization decision  41

P
- PAC (privilege attribute certificate)  37, 42, 130, 155
- pac_svc_id  43
- password
  - accessing the SSL key file  29
  - authenticating  164
  - LDAP administrator  28
- pdb*msg.h  15
- permissions  39
- persistent authorization policy database  19
- placing the data structure into a buffer  38
- policy database replica  19
- port number
  - for a TCP port  20, 27
- privilege attribute certificate (PAC)  37, 42, 130, 155
protected object 40
protected object namespace 40
providing
  additional parameters 40

R
refeshing
  local authorization database 20
registry, user 5, 36
related publications xii
releasong
  allocated memory 41
  memory allocated 14
remote cache mode 1, 19
remote-acl-users 35
removing
  attribute list 109, 110
  credentials 123
requested resource 40
returning
  access control decision information 142
  entry string value 114
  handle 132
  handle for a specified identity 150
  handle to credentials structure 125
  handle to new PAC credentials 155
  individual credentials in a chain 135
  information from a credentials structure 128
  major error code 147
  minor error code 148
  name attributes 117
  number of buffer attribute entries 118
  number of value attributes 118
  privilege attribute certificate 130

S
secure domain 5
Secure Socket Layer (SSL) 28
security policy 3
server
  host name, LDAP 27
service definition 58
service plug-in 58, 60
  administration service 57
  architecture 53
  configuring 58
  credentials modification service 57, 71
  entitlements service 57, 69
  error codes 62
  external authorization service 57, 72
  implementing 58
  initializing 58
  privilege attribute certificate service 57, 72
  shut down 66
  supplied implementations 69
service plug-in interfaces
  implementing 61
setting an attribute list entry 13
shutdown 8, 41
shutting down 8, 41, 42, 161
Solaris Operating Environment
  libivauthzn.so library file 3
specifing
  additional user information 37
  authorization (continued)
  authorization authority 37
  type of cache mode 18
  user authentication identity 37
SSL
  communications 28
  key file password 29
  key label 28
  ssl stanza 193
standard, The Open Group 1
stanzas
  authentication-mechanisms 213
  ldap 189, 197
  manager 215
  ssl 193
  uraf-ad in activedir.conf 202
  uraf-domino 205
starting
  authorization service 33
status codes 14, 147, 148
storage
  array of strings, freeing 160
  buffer, freeing 157
  string, freeing 159
  strings
    freeing of storage 159
    release of memory 42
    see also array of strings 42
    value 114
successful login 164
summary of
  API extensions 9
  API functions 7
  attribute list functions 7
  attribute list tasks 12
  authentication parameters 38
  Authorization API manual pages 101
  Authorization API tasks 6
  authorization decision functions 8
  authorization decision output parameters 41
  buffer names and values 9
  cache modes 18
  credentials functions 8
  error code files 14, 15
  initialization, shutdown, and error handling functions 8
  local cache mode attributes and values 19
  SSL attributes for LDAP access 28
  user identity types 36
T
  tasks, Authorization API 6
  TCP port number 20
types of
  additional user information 37
  authentication parameters 38
  cache modes 18
  user identities 36
U
  unauthenticated user 36
  unauthenticated users 11
  uraf-ad stanza in activedir.conf 202
  uraf-domino 205
additional auditing information 37
assigning credentials to a credentials handle 40
authentication identity 37
authorization credentials 37, 38
mapping the user operation 39
obtaining an identity 36
specifying additional information 37
unauthenticated 37

user registry
  LDAP configuration file 197
  specifying LDAP 5
  specifying the user authentication identity 37
User Registry Adapter Framework (URAF) 202
username and password 164
using
  as input of credentials information 11
  username and password to log in 164
utility function error codes
  major errors 4
  minor errors 4

V
value attributes
  buffer 111, 116
  entry number 118
  name 117
  string 114
  values 9
  version number 34

W
weightings 93
Windows NT
  ivauthzn.dll library file 3