Contents

Preface..................................................... xi

Who Should Read This Book ................................ xi
What This Book Contains ....................................... xi
Publications ................................................ xii

Tivoli SecureWay Policy Director Library ...................... xii
Prerequisite Publications ................................... xii
Accessing Publications Online .............................. xiii
Ordering Publications ..................................... xiii
Providing Feedback about Publications ....................... xiii

Contacting Customer Support ................................. xiv

Conventions Used in This Book ................................. xiv

Typeface Conventions..................................... xiv

Chapter 1. Authorization C API ........................... 1

Introducing the Authorization API ............................. 2

The Open Group Authorization API standard .................... 3
The Policy Director Authorization Model ...................... 4

Locating the Authorization API Components .................... 5

Building Applications with the Authorization API ............. 7

Understanding the Authorization API Functions and Data Types ...... 9

API Functions ............................................ 9
Character Strings ......................................... 12
Buffers ................................................ 12
Protected Object Structures ................................. 13
Default User Registry Information Structure .................. 14
Unauthenticated User Information Structure .................. 15
Attribute Lists ........................................... 16
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_decision_access_allowed_ext()</td>
<td>128</td>
</tr>
<tr>
<td>azn_entitlement_get_entitlements()</td>
<td>131</td>
</tr>
<tr>
<td>azn_error_get_string()</td>
<td>133</td>
</tr>
<tr>
<td>azn_error_major()</td>
<td>135</td>
</tr>
<tr>
<td>azn_error_minor()</td>
<td>136</td>
</tr>
<tr>
<td>azn_error_minor_get_string()</td>
<td>137</td>
</tr>
<tr>
<td>azn_id_get_creds()</td>
<td>139</td>
</tr>
<tr>
<td>azn_initialize()</td>
<td>142</td>
</tr>
<tr>
<td>azn_pac_get_creds()</td>
<td>146</td>
</tr>
<tr>
<td>azn_release_buffer()</td>
<td>148</td>
</tr>
<tr>
<td>azn_release_pobj()</td>
<td>149</td>
</tr>
<tr>
<td>azn_release_string()</td>
<td>150</td>
</tr>
<tr>
<td>azn_release_strings()</td>
<td>151</td>
</tr>
<tr>
<td>azn_shutdown()</td>
<td>152</td>
</tr>
<tr>
<td>azn_util_errcode()</td>
<td>154</td>
</tr>
<tr>
<td>azn_util_handle_is_valid()</td>
<td>155</td>
</tr>
<tr>
<td>azn_util_password_authenticate()</td>
<td>156</td>
</tr>
<tr>
<td>sversslcfg</td>
<td>159</td>
</tr>
</tbody>
</table>

# Chapter 3. Authorization Service Plug-ins

Service Plug-in Architecture ........................................ 163
The Authorization Service Dispatcher ................................ 164
Authorization Service Plug-ins ........................................ 165
Calling Applications ................................................ 165
Supported Types of Service Plug-ins ............................... 167
Implementing a Service Plug-in ...................................... 169
Initialization and Configuration of Service Plug-ins .......... 169
Implementing Service Interfaces ................................. 174
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using Error Codes.</td>
<td>176</td>
</tr>
<tr>
<td>Shutdown</td>
<td>181</td>
</tr>
<tr>
<td>Example Service Source Code</td>
<td>181</td>
</tr>
<tr>
<td>Implementing an Entitlements Service</td>
<td>186</td>
</tr>
<tr>
<td>Understanding Entitlements</td>
<td>186</td>
</tr>
<tr>
<td>Initialization, Configuration, and Shutdown</td>
<td>188</td>
</tr>
<tr>
<td>Obtaining Entitlements for a Specified User</td>
<td>189</td>
</tr>
<tr>
<td>Authorizing a Caller to a Specific Entitlements Service Plug-In</td>
<td>190</td>
</tr>
<tr>
<td>Using Authorization API Interfaces</td>
<td>191</td>
</tr>
<tr>
<td>Entitlements Service Error Codes</td>
<td>191</td>
</tr>
<tr>
<td>Implementing an Administration Service Plug-in</td>
<td>193</td>
</tr>
<tr>
<td>Configuring an Administration Service Plug-in</td>
<td>195</td>
</tr>
<tr>
<td>Initializing and Shutting Down Administration Service Plug-Ins</td>
<td>197</td>
</tr>
<tr>
<td>Using an Administration Service Plug-in</td>
<td>198</td>
</tr>
<tr>
<td>Error Codes</td>
<td>200</td>
</tr>
<tr>
<td>Deploying an Administration Service Plug-in</td>
<td>203</td>
</tr>
<tr>
<td>Implementing External Authorization Service Plug-in</td>
<td>205</td>
</tr>
<tr>
<td>Introducing the External Authorization Service</td>
<td>205</td>
</tr>
<tr>
<td>Understanding the External Authorization Service</td>
<td>206</td>
</tr>
<tr>
<td>Configuring an External Authorization Service Plug-in</td>
<td>211</td>
</tr>
<tr>
<td>Initializing and Shutting Down External Authorization Service Plug-Ins</td>
<td>214</td>
</tr>
<tr>
<td>Obtaining an Authorization Decision</td>
<td>215</td>
</tr>
<tr>
<td>Error Codes</td>
<td>216</td>
</tr>
<tr>
<td>Supplied Implementations for Service Plug-ins</td>
<td>219</td>
</tr>
<tr>
<td>Policy Director Entitlements Services</td>
<td>220</td>
</tr>
<tr>
<td>Credentials Modification Service</td>
<td>221</td>
</tr>
<tr>
<td>Privilege Attribute Certificate Service</td>
<td>223</td>
</tr>
<tr>
<td>External Authorization Service</td>
<td>224</td>
</tr>
</tbody>
</table>
### Chapter 4. Authorization Service Plug-Ins: C API

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_service_initialize()</td>
<td>226</td>
</tr>
<tr>
<td>azn_service_shutdown()</td>
<td>230</td>
</tr>
<tr>
<td>azn_admin_get_object()</td>
<td>232</td>
</tr>
<tr>
<td>azn_admin_get_objectlist()</td>
<td>235</td>
</tr>
<tr>
<td>azn_admin_perform_task()</td>
<td>238</td>
</tr>
<tr>
<td>azn_admin_get_tasklist()</td>
<td>242</td>
</tr>
</tbody>
</table>

### Chapter 5. Authorization Java Classes

Using Java™ 2 Security with Policy Director ........................................ 248

Java Authentication and Authorization Services (JAAS) Model .................. 249
  Authenticating Users and Obtaining Credentials................................. 251
  Authorizing Access Requests .................................................... 252

Software Requirements for Installation .............................................. 254
  Policy Director Requirements .................................................... 254
  JAAS and JRE Requirements ....................................................... 254
  Java Archive Requirements ....................................................... 255

Java Classes Overview ........................................................................ 256
  com.tivoli.mts.PDLoginModule .................................................. 256
  com.tivoli.mts.PDPPrincipal ..................................................... 257
  com.tivoli.mts.PDPermission .................................................... 257
  com.tivoli.mts.PDAttrs .......................................................... 258
  com.tivoli.mts.PDAttrValue ...................................................... 259
  com.tivoli.mts.PDAttrValues ..................................................... 259
  com.tivoli.mts.PDStatics .......................................................... 260
  com.tivoli.mts.SvrSslCfg ......................................................... 260

Configuring a Java Application into Policy Director ................................ 260
  Establishing an Identity with Proper Credentials ................................ 261
  Identifying the Policy Director Server Machines .................................. 262
Tivoli® SecureWay® Policy Director Authorization ADK is an application development kit for Tivoli SecureWay Policy Director that enables application developers to add Policy Director authorization and security services to applications. The Tivoli SecureWay Policy Director Authorization ADK Developer Reference describes both a C implementation and a Java implementation of the Policy Director Authorization API.

Who Should Read This Book

The target audience for this administration guide includes:

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

What This Book Contains

This document contains the following chapters:

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

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

- Chapter 3, “Authorization Service Plug-ins”
  Describes the Authorization Service Plug-in model, including Entitlements Services, Administration Services, and External Authorization Services.

- Chapter 4, “Authorization Service Plug-ins: C API”
This chapter provides reference pages for each of the Service Plug-in function calls.

- Chapter 5, “Authorization Java Classes”
  This chapter describes how to use the Policy Director Java classes to add Policy Director authorization to an application.

- Chapter 6, “Authorization Java Classes Reference”
  This chapter provides reference pages for each of the Java classes and methods.

Publications

This section lists publications in the Tivoli SecureWay Policy Director library and any other related documents. It also describes how to access Tivoli publications online, how to order Tivoli publications, and how to make comments on Tivoli publications.

Tivoli SecureWay Policy Director Library

The following documents are available in the Tivoli SecureWay Policy Director library:

- **Tivoli SecureWay Policy Director Base Installation Guide**, GC32-0735
- **Tivoli SecureWay Policy Director Base Administration Guide**, GC32-0680
- **Tivoli SecureWay Policy Web Portal Manager Administration Guide**, GC32-0737
- **Tivoli SecureWay Policy Director Administration API Developer Reference**, GC32-0813

Prerequisite Publications

To be able to use the information in this book effectively, you must have some prerequisite knowledge, which you can get from the following books:

- **Tivoli SecureWay Policy Director Base Administration Guide**, GC32-0680
Accessing Publications Online
You can access many Tivoli publications online at the Tivoli Customer Support Web site:

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

These publications are available in PDF or HTML format, or both. Translated documents are also available for some products.

Ordering Publications
You can order many Tivoli publications online at the following Web site:

http://www.ibm.com/shop/publications/order

You can also order by telephone by calling one of these numbers:

- In the United States: 800-879-2755
- In Canada: 800-426-4968
- In other countries, for a list of telephone numbers, see the following Web site:
  http://www.tivoli.com/inside/store/lit_order.html

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

- Send an e-mail to pubs@tivoli.com.
- Complete our customer feedback survey at the following Web site:
  http://www.tivoli.com/support/survey/
Contacting Customer Support

If you have a problem with any Tivoli product, you can contact Tivoli Customer Support. See the *Tivoli Customer Support Handbook* at the following Web site:

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

The handbook provides information about how to contact Tivoli Customer Support, depending on the severity of your problem, and the following information:

- Registration and eligibility
- Telephone numbers and e-mail addresses, depending on the country you are in
- What information you should gather before contacting support

Conventions Used in This Book

This book uses several conventions for special terms and actions, operating system-dependent commands and paths, and margin graphics.

**Typeface Conventions**

The following typeface conventions are used in this book:

**Bold**  Lowercase and mixed-case commands, command options, and flags that appear within text appear like *this*, in **bold** type.

Graphical user interface elements (except for titles of windows and dialogs) and names of keys also appear like *this*, in **bold** type.

**Italic**  Variables, values you must provide, new terms, and words and phrases that are emphasized appear like *this*, in *italic* type.

**Monospace**  Commands, command options, and flags that appear on a separate line, code examples, output, and message text appear like *this*, in monospace type.
Names of files and directories, text strings you must type, when they appear within text, names of Java methods and classes, and HTML and XML tags also appear like this, in monospace type.
This chapter contains the following topics:

- “Introducing the Authorization API” on page 2
- “Locating the Authorization API Components” on page 5
- “Building Applications with the Authorization API” on page 7
- “Understanding the Authorization API Functions and Data Types” on page 9
- “Summarizing Authorization API Tasks” on page 21
- “Authenticating an API Application” on page 23
- “Initializing the Authorization Service” on page 23
- “Verifying the Identity of a User” on page 48
- “Obtaining User Authorization Credentials” on page 49
- “Obtaining an Authorization Decision” on page 54
- “Cleaning Up and Shutting Down” on page 57
- “Working with Credentials” on page 59
- “Deploying Applications with the Authorization API” on page 65
- “Deprecated API Elements” on page 67
Introducing the Authorization API

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

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

The Authorization API supports two implementation modes:

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

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

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

The Authorization API works independently from the underlying security infrastructure, the credential format, and the evaluating mechanism. The Authorization API makes it possible to request an authorization check and get a simple “yes” or “no” recommendation in return.
The Authorization API is a component of the Policy Director Application Development Kit (ADK).

**The Open Group Authorization API standard**

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

![Figure 1. The ISO 10181-3 Authorization Model](image)

Policy Director implements the ADF component of this model and provides the Authorization API as an interface to this function.
In the figure above, a browser (initiator) requests access to a file or other resource on a protected system (target). The browser submits the request to a Web application server (the resource manager incorporating the access enforcement function). The Web application server uses the Authorization API to submit the request to the Policy Director Authorization Service (the access decision function).

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

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

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

**The Policy Director Authorization Model**

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

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

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

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

1. Configure the Policy Director secure domain to recognize and support the objects, actions, and users that are relevant to your application.
   - For an introduction to the Policy Director authorization model, see the *Tivoli SecureWay Policy Director Base Administration Guide*.
   - For complete information on access control, see the *Tivoli SecureWay Policy Director Base Administration Guide*.

2. Use the Authorization API within your application to obtain the needed authorization decisions.
   - For an introduction to the Authorization API, including information on remote cache mode and local cache mode, see the *Tivoli SecureWay Policy Director Base Administration Guide*.

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

**Locating the Authorization API Components**

The Authorization API is included as an optional installation package in the Policy Director distribution. The Authorization API files are installed in several sub-directories under the Policy Director installation directory.
If you are installing the Authorization API portion of the ADK from the Tivoli SecureWay Toolbox, refer to the Toolbox README file for installation instructions.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>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. On Solaris systems, the library is libpdauthzn.so. On AIX systems, the library is libpdauthzn.a. On HP-UX systems, the library is libpdauthzn.sl. On Windows, the library to link is pdauthzn.lib</td>
</tr>
<tr>
<td>example</td>
<td>This directory contains an example program that demonstrates usage of the Authorization API. Source files and a MAKEFILE are provided.</td>
</tr>
</tbody>
</table>

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

- **Header Files**

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

<table>
<thead>
<tr>
<th>File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ogauthzn.h</td>
<td>The Authorization API standard functions</td>
</tr>
<tr>
<td>aznutils.h</td>
<td>Utility functions (extensions to The Authorization API)</td>
</tr>
<tr>
<td>azn_svc_protos.h</td>
<td>Prototypes for generic authorization service plug-in functions. Contains prototypes for the azn_service_initialize() and azn_service_shutdown() calls. This can optionally be included by a plug-in programmer to prototype the calls defined in the service.</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 Policy Director Authorization Service are found in a number of error message files. For example, pdbaclmsg.h</td>
</tr>
</tbody>
</table>

### Building Applications with the Authorization API

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

If you do not have a Policy Director secure domain installed, install one before beginning application development. The minimum installation consists of a single system with the following Policy Director Base components installed:
Policy Director Runtime Environment
Policy Director Management Server
Policy Director Application Development Kit

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

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

- Policy Director Runtime
- Policy Director Application Development Kit

Note: For Policy Director installation instructions refer to the Tivoli SecureWay Policy Director Base Installation Guide.

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

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

Demonstration Programs

The Policy Director Authorization API is provided with several example programs. The authzn_demo directory contains examples programs that demonstrate use of the Authorization API. Both a C example and a Java example are included. The C example contains a sample Makefile. See the sample Makefile for build instructions specific to each supported operating system platform. The Java example includes a javadoc directory with HTML pages that describe the Java classes.
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.

**Understanding the Authorization API Functions and Data Types**

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

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

- “API Functions” on page 9
- “Character Strings” on page 12
- “Buffers” on page 13
- “Protected Object Structures” on page 13
- “Default User Registry Information Structure” on page 14
- “Attribute Lists” on page 16
- “Credential Handles” on page 18
- “Status Codes and Error Handling” on page 19

**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.
## Attribute Lists

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_attrlist_add_entry()</code></td>
<td>on page 75</td>
</tr>
<tr>
<td><code>azn_attrlist_add_entry_buffer()</code></td>
<td>on page 77</td>
</tr>
<tr>
<td><code>azn_attrlist_add_entry_pobj()</code></td>
<td>on page 79</td>
</tr>
<tr>
<td><code>azn_attrlist_add_entry_ulong()</code></td>
<td>on page 80</td>
</tr>
<tr>
<td><code>azn_attrlist_create()</code></td>
<td>on page 81</td>
</tr>
<tr>
<td><code>azn_attrlist_copy()</code></td>
<td>on page 82</td>
</tr>
<tr>
<td><code>azn_attrlist_delete()</code></td>
<td>on page 83</td>
</tr>
<tr>
<td><code>azn_attrlist_delete_entry()</code></td>
<td>on page 84</td>
</tr>
<tr>
<td><code>azn_attrlist_get_entry_buffer_value()</code></td>
<td>on page 85</td>
</tr>
<tr>
<td><code>azn_attrlist_get_entry_ulong_value()</code></td>
<td>on page 89</td>
</tr>
<tr>
<td><code>azn_attrlist_get_entry_pobj_value()</code></td>
<td>on page 87</td>
</tr>
<tr>
<td><code>azn_attrlist_get_entry_string_value()</code></td>
<td>on page 91</td>
</tr>
<tr>
<td><code>azn_attrlist_get_names()</code></td>
<td>on page 93</td>
</tr>
<tr>
<td><code>azn_attrlist_name_get_num()</code></td>
<td>on page 95</td>
</tr>
<tr>
<td><code>azn_release_buffer()</code></td>
<td>on page 148</td>
</tr>
<tr>
<td><code>azn_release_pobj()</code></td>
<td>on page 149</td>
</tr>
<tr>
<td><code>azn_release_string()</code></td>
<td>on page 150</td>
</tr>
<tr>
<td><code>azn_release_strings()</code></td>
<td>on page 151</td>
</tr>
<tr>
<td><code>azn_util_handle_is_valid()</code></td>
<td>on page 155</td>
</tr>
</tbody>
</table>

## Credentials

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_creds_combine()</code></td>
<td>on page 97</td>
</tr>
<tr>
<td><code>azn_creds_copy()</code></td>
<td>on page 99</td>
</tr>
<tr>
<td><code>azn_creds_create()</code></td>
<td>on page 101</td>
</tr>
<tr>
<td><code>azn_creds_delete()</code></td>
<td>on page 103</td>
</tr>
<tr>
<td><code>azn_creds_equal()</code></td>
<td>on page 105</td>
</tr>
<tr>
<td><code>azn_creds_for_subject()</code></td>
<td>on page 107</td>
</tr>
<tr>
<td><code>azn_creds_get_attrlist_for_subject()</code></td>
<td>on page 112</td>
</tr>
</tbody>
</table>

© 1985-2018, BSD License.
10 Version 3.8
## Function Task

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_creds_set_attr_value_string()</code> on page 124</td>
<td>“Setting and Getting Attribute Values for a Credential” on page 64</td>
</tr>
<tr>
<td><code>azn_creds_get_attr_value_string()</code> on page 110</td>
<td></td>
</tr>
<tr>
<td><code>azn_creds_get_pac()</code> on page 115</td>
<td>“Converting Credentials to a Transportable Format” on page 59</td>
</tr>
<tr>
<td><code>azn_creds_modify()</code> on page 118</td>
<td>“Modifying the Contents of a Credential” on page 61</td>
</tr>
<tr>
<td><code>azn_creds_num_of_subjects()</code> on page 123</td>
<td>“Determining the Number of Credentials in a Credentials Chain” on page 61</td>
</tr>
<tr>
<td><code>azn_id_get_creds()</code> on page 139</td>
<td>“Obtaining User Authorization Credentials” on page 49</td>
</tr>
<tr>
<td><code>azn_pac_get_creds()</code> on page 146</td>
<td>“Converting Credentials to the Native Format” on page 60</td>
</tr>
<tr>
<td><code>azn_util_handle_is_valid()</code> on page 155</td>
<td>“Credential Handles” on page 18</td>
</tr>
</tbody>
</table>

### Authorization Decisions

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_decision_access_allowed()</code> on page 126</td>
<td>“Obtaining an Authorization Decision” on page 54</td>
</tr>
<tr>
<td><code>azn_decision_access_allowed_ext()</code> on page 128</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><code>azn_error_major()</code> on page 135</td>
<td>“Status Codes and Error Handling” on page 19</td>
</tr>
<tr>
<td><code>azn_error_minor()</code> on page 136</td>
<td></td>
</tr>
<tr>
<td><code>azn_error_minor_get_string()</code> on page 137</td>
<td></td>
</tr>
<tr>
<td><code>azn_error_get_string()</code> on page 133</td>
<td></td>
</tr>
</tbody>
</table>
### Function or Data Type Task

<table>
<thead>
<tr>
<th>Function or Data Type</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_initialize()</code> on page 142</td>
<td>“Initializing the Authorization Service” on page 23</td>
</tr>
<tr>
<td><code>azn_release_buffer()</code> on page 148</td>
<td>“Releasing Allocated Memory” on page 58</td>
</tr>
<tr>
<td><code>azn_release_pobj()</code> on page 149</td>
<td></td>
</tr>
<tr>
<td><code>azn_release_string()</code> on page 150</td>
<td></td>
</tr>
<tr>
<td><code>azn_release_strings()</code> on page 151</td>
<td></td>
</tr>
<tr>
<td><code>azn_shutdown()</code> on page 152</td>
<td>“Shutting Down the Authorization API” on page 58</td>
</tr>
<tr>
<td><code>azn_util_errcode()</code> on page 154</td>
<td>“Status Codes and Error Handling” on page 19</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 154</td>
<td>“Status Codes and Error Handling” on page 19</td>
</tr>
<tr>
<td><code>azn_util_handle_is_valid()</code> on page 155</td>
<td>“Attribute Lists” on page 16</td>
</tr>
<tr>
<td><code>azn_util_password_authenticate()</code> on page 156</td>
<td>“Credential Handles” on page 18</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:

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;
    void *value;
} azn_buffer_desc, *azn_buffer_t;
```

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

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

Use `azn_release_buffer` to release storage allocated for use by `azn_buffer_desc` objects.

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

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

**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 Policy Director credentials to the `azn_id_get_creds()` call. This

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 Policy Director 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.</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 Policy Director 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.

```c
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>auth_method</td>
<td>The authentication method.</td>
</tr>
<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>
<tr>
<td>authnmech_info</td>
<td>Additional authentication mechanism information.</td>
</tr>
<tr>
<td>reserved</td>
<td>Fields reserved for future use.</td>
</tr>
</tbody>
</table>
Field Description
browser_info The browser employed by the user. This field is optional.

Attribute Lists

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

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

Many Authorization API functions 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 Authorization API defines some names. You can also define additional names as needed by your application.

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

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create an attribute list</td>
<td>Use <code>azn_attrlist_create()</code> to complete the following tasks:</td>
</tr>
<tr>
<td></td>
<td>■ Allocate a new, empty attribute list.</td>
</tr>
<tr>
<td></td>
<td>■ Associate a handle with the attribute list.</td>
</tr>
<tr>
<td></td>
<td>■ Return the handle.</td>
</tr>
<tr>
<td>Task</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 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  | 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. |
|   attribute name                          | Copy an attribute list. Use `azn_attrlist_copy()` to make a copy of an attribute list.
<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get a value</td>
<td>Use <code>azn_attrlist_get_entry_string_value()</code> to get the value attribute of a string of type <code>azn_string_t</code> for a specified name in an attribute list. Use <code>azn_attrlist_get_entry_buffer_value()</code> to get the value attribute of a buffer of type <code>azn_buffer_t</code> for a specified name in an attribute list. Use <code>azn_attrlist_get_entry_ulong_value()</code> to get the value attribute of type <code>azn_ulong_t</code>. Use <code>azn_attrlist_get_entry_pobj_value()</code> to get the value attribute of type <code>azn_pobj_t</code>. 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.</td>
</tr>
<tr>
<td>Delete an attribute list</td>
<td>Use <code>azn_attrlist_delete()</code> to delete the attribute list associated with a specified attribute list handle.</td>
</tr>
<tr>
<td>Determine if the attribute list handle is valid</td>
<td>Use <code>azn_util_handle_is_valid()</code> to determine if an attribute list handle is associated with valid data.</td>
</tr>
</tbody>
</table>

**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 Policy Director components contain only a single Policy Director credential. The intermediary credentials are not referenced by Policy Director 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 Policy Director security framework.

Use the function `azn_creds_create()` to complete the following tasks:
- Allocate a new, empty credential structure.
- Associate a handle with the credential structure.
- Return a pointer to the handle.

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

To determine if a credentials handle is valid, use the Authorization API utility function `azn_util_handle_is_valid()`.

For more information on functions that use credentials handles to access credential information, see “Working with Credentials” on page 59.

### Status Codes and Error Handling

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

The returned status code includes both major and minor error codes. A major error code of `AZN_S_FAILURE` indicates that a minor error code contains the error status.
Use `azn_error_major()` to extract major error codes from the returned status. Major error codes are defined according to the The Open Group Authorization API Standard.

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

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

Use `azn_error_get_string()` to return the Policy Director 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 Policy Director 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>
Summarizing Authorization API Tasks

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

Use the Authorization API to present information about the user, operation, and requested resource to the Policy Director Authorization Service, 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:

- “Authenticating an API Application” on page 22
- “Initializing the Authorization Service” on page 23
- “Verifying the Identity of a User” on page 48
- “Obtaining User Authorization Credentials” on page 49
- “Obtaining an Authorization Decision” on page 54
- “Cleaning Up and Shutting Down” on page 57

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 59

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

- “Deploying Applications with the Authorization API” on page 65
Authenticating an API Application

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

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

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

- **ivacld-servers**
  This group membership is needed for applications using local cache mode.

- **remote-acl-users**
  This group membership is needed for applications using remote cache mode.

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

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

The `svrsslecf` utility creates a user identity for the application, and configures the SSL communication between the application and the Policy Director Management Server.

The `svrsslecf` 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 svrsslecf syntax, see the reference page for \texttt{svrsslecf} on page 159.

### Initializing the Authorization Service

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

There are two ways to specify the initialization data:

- Specify input arguments to \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 24
- “Specifying Cache Mode Settings” on page 25
- “Configuring SSL from the API Client to Policy Director” on page 26
Specifying an Authorization API Configuration File

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

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

Specify a configuration file by using the following attribute:

- `azn_init_cfg_file`

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

### Specifying Cache Mode Settings

The cache mode determines if the Authorization API talks to a Policy Director 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 *Tivoli SecureWay Policy Director Base Administration 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 reference page for "svrsslcfg" on page 159.

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 Policy Director Authorization Service runs in the same server process as the application using the Authorization API.</td>
</tr>
<tr>
<td>remote</td>
<td>The Policy Director Authorization Service runs as a different server process from the application using the Authorization API.</td>
</tr>
</tbody>
</table>

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><code>filename</code></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>
<tr>
<td>Values</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>number of seconds</td>
<td>Number of seconds between refreshes of the local authorization policy database. Set appropriate values to ensure that the replicated database is updated in a timely manner to reflect changes made to the master database.</td>
</tr>
</tbody>
</table>

**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 management 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 Policy Director**

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

**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>&lt;keyfile-path&gt;</td>
<td>This is the keyfile used to communicate with ivmgrd/ivacld. It will be created by the <code>svrsslcfg</code> utility.</td>
</tr>
<tr>
<td></td>
<td>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>&lt;keyfile-path&gt;</td>
<td>This the stash file for the keyfile. It is created by the <strong>svrsslcfg</strong> utility. It is used as an obfuscated password to the keyfile. This file should be appropriately secured.</td>
</tr>
<tr>
<td></td>
<td>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 svrsslcfg 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 Policy Director Authorization API client automatically create a new SSL session with new keys when a session expires.</td>
</tr>
<tr>
<td></td>
<td>This value only applies to the listening aspect of the authorization API’s (when the Policy Director Management Server is calling the application). When the application is calling the Policy Director Management Server or the Policy Director Authorization Server, the session timeout value is dictated by 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 Policy Director Authorization API client automatically refreshes the password or stash file before this expiry time, if automatic refresh is enabled.</td>
</tr>
</tbody>
</table>
Configuration File Stanza: ssl
This always has a value of certificate.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The type of authentication. Possible values are: certificate</td>
<td>The method that the Policy Director Management Server will use to authenticate the authorization API client. If the value is certificate the Policy Director Management Server will map the certificate provided by the authorization API client into an identity and authenticate against it. Note that even if password or none are specified, the client will still use SSL to communicate with the server, and as such will still require a keyring database file that has the Policy Director 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 none.</td>
</tr>
<tr>
<td>password</td>
<td></td>
</tr>
<tr>
<td>none</td>
<td></td>
</tr>
<tr>
<td>The default value is none.</td>
<td></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: authn-user
- Configuration File Entry: authn-password
- Configuration File Stanza: ssl
- Modified by svrsslcf?: 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 password. It may be unwise to store these in the configuration file, however they can be useful for testing communications.</td>
</tr>
</tbody>
</table>
Policy Director Configuration File Location

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

<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 Policy Director Runtime Environment (pd.conf). If it is specified, the values for master-host, master-port, and master-dn 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 Policy Director installation directory, under ./lib/pd.conf</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: `ssl`
- Modified by `svrsslefg`?: 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 ssl-keyfile-pwd and ssl-keyfile-stash are specified, the value in ssl-keyfile-pwd 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: ssl
- 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>Enable or disable automatic refresh of the SSL certificate and key database file password. When enabled, the certificate and password are regenerated when either is close to expiration. A value of yes enables automatic refresh. A value or no disabled 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>
</table>
| `<number of seconds>`        | 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. |

### Specifying Communications Attributes for the Management Server

Use the attributes described in this section to specify the location, port number, and distinguished name of the Policy Director Management Server. Authorization API Clients use this information to communicate with the management server.

#### Management Server Hostname

- Attribute List Entry: `azn_init_master_host`
- Configuration File Entry: `master-host`
- Configuration File Stanza: `manager`
- Modified by `svrsslefg`: Yes.

The value is taken from the pd.conf file. The pd.conf file is created when the Policy Director runtime component is configured on the machine.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| `<Management Server hostname>` | Specifies the hostname of the Policy Director Management Server.  

This entry and stanza can be in either the authorization API client’s configuration file (aznAPI.conf) or the 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. |
Management Server Port Number

- Attribute List Entry: `azn_init_master_port`
- Configuration File Entry: `master-port`
- Configuration File Stanza: `master`
- Modified by `svrsslcfg`: Yes.

The value is taken from the `pd.conf` file. The `pd.conf` file is created when the Policy Director runtime component is configured on the machine.

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

Management 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 Policy Director 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 Policy Director Management Server. The authorization API client can match this value against that provided by the Policy Director Management Server at runtime to prevent spoofing. If the value is empty, no checking will be performed. This entry and stanza can be in either the authorization API client’s configuration file (aznAPI.conf) or the 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>

**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 Policy Director 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=Policy Director,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</td>
<td>The LDAP distinguished name for the Authorization Server.</td>
</tr>
<tr>
<td>distinguished name&gt;</td>
<td></td>
</tr>
</tbody>
</table>

### Configuring the Authorization API for LDAP Access

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

#### 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 Policy Director 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 azn_init_ldap_port is not null.</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 `svrsslefg`? Yes.
  The value is taken from the pd.conf file. The pd.conf file is created when the Policy Director runtime component is configured on the machine.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>host name</code></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 `svrsslefg`? Yes.
  The value is taken from the pd.conf file. The pd.conf file is created when the Policy Director runtime component is configured on the machine.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>port number</code></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 `svrsslefg`? 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 Policy Director Authorization server and the LDAP server is over Secure Sockets Layer (SSL), you must specify the communication settings.

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

**SSL Communication with the LDAP Server**

- Attribute List Entry: None
- Configuration File Entry: `ssl-enabled`
- Configuration File Stanza: `ldap`
<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.</td>
</tr>
</tbody>
</table>

**SSL Keyfile Name**
- Attribute List Entry: **azn_init ldap_ssl_keyfile**
- Configuration File Entry: ssl-keyfile
- Configuration File Stanza: ldap

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</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: ldap

<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.</td>
</tr>
<tr>
<td></td>
<td>When the keyfile is specified but the password (ldap_ssl_keyfile_pwd) is not specified or is set to</td>
</tr>
<tr>
<td></td>
<td>NULL, the password string is set to NULL. When the keyfile is specified, and ldap_ssl_keyfile_pwd is</td>
</tr>
<tr>
<td></td>
<td>a non-NULL string, the password is set to the non-NULL string value.</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><code>&lt;search-size&gt;</code></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 Policy Director 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

Use can adds 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: ldap-replica
- Configuration File Stanza: ldap
Assign multiple values to `ldap-replica` by entering a list consisting of entries that are separated by commas. For example:

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

<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. 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_attr` attribute during initialization of the Authorization API. The `azn_init_set_perminfo_attr` 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_attr`. 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:
- Attribute List Entry: **azn_init_set_perminfo_attrs**
- Configuration File Entry: permission-info-returned
- Configuration File Stanza: aznapi-configuration

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_allattrs</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 acl-extended-attribute-name value above.</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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>filename</code></td>
<td>Path name to the persistent authorization policy database replica.</td>
</tr>
<tr>
<td><code>filename</code></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`

Version 3.8
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`

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;size in bytes&gt;</code></td>
<td>The maximum size in bytes for the auditing and debug files. When the log size reaches this value, the log is backed up and a new log file is started.</td>
</tr>
<tr>
<td></td>
<td>A value of 0 disables log rollover. A negative value performs rollover daily, regardless of size. The default value is 2000000.</td>
</tr>
</tbody>
</table>

#### 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>

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

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

When using this function on Windows NT, do not call it from `dllMain()`. For more information, see the reference page for "`azn_initialize()`" on page 142.

**Verifying the Identity of a User**

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

- **Authenticated**
  In this case, the user’s identity in the secure domain is registered in the LDAP User registry. The user is authenticated, and information about the user can be obtained. This information includes, for example, the LDAP Distinguished Name.

- **Unauthenticated**
  In this case, the user’s identity in the secure domain is not specifically registered in the LDAP user registry. The user is defined to be unauthenticated, and further information about the user’s identity is irrelevant to the authorization process.
Applications can obtain user identities through a variety of methods. These can include the use of a Credentials Acquisition Server, or a call to an application-specific method for querying user registries and establishing a security (login) context.

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

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

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

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

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

For more information, see the reference page for "`azn_util_password_authenticate()`" on page 156.

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

### Obtaining User Authorization Credentials

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

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

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

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

**Note:** Identity information can also be obtained from a privilege attribute certificate (PAC). See “Converting Credentials to the Native Format” on page 60.

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” on page 51
4. “Placing User Information into an API Buffer” on page 52
5. “Obtaining Authorization Credentials for the User” on page 52

**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 Policy Director 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 Policy Director Authorization Service does not use this information.

The application can store the additional user information in the data structures that the Authorization API provides for each type of authenticated identity. The data structures are: `azn_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>
<tr>
<td>browser_info</td>
<td>Information about the type of browser through which the user has submitted the request, if applicable. This string can contain any information that is useful to the application.</td>
</tr>
<tr>
<td>ipaddr</td>
<td>The IP address of the user. This is optional information for use by the application.</td>
</tr>
<tr>
<td>qop</td>
<td>The quality of protection required for requests made by this user.</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 50 and “Specifying Additional User Information” on page 51 into an Authorization API buffer.

Complete the following steps:

1. Declare a buffer of type `azn_buffer_t`:
   ```c
   typedef struct azn_buffer_desc_struct {
   size_t length;
   void *value;
   } azn_buffer_desc, *azn_buffer_t;
   ```

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

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

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

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 “Specifying the Authorization Authority” on page 50.</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” on page 50</td>
</tr>
<tr>
<td></td>
<td>&quot;Specifying Additional User Information” on page 51</td>
</tr>
<tr>
<td></td>
<td>&quot;Placing User Information into an API Buffer”</td>
</tr>
</tbody>
</table>
The **azn_id_get_creds()** function returns a handle to the authorization credentials for the user. The authorization credentials are contained in an **azn_creds_h_t** structure.

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

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

azn_creds_create(&creds);

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

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

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

/* Obtain an authorization credential. */status = azn_id_get_creds(NULL, mech, &buf, &creds);
if (status != AZN_S_COMPLETE) {
    fprintf(stderr, "Could not get creds.
    continue;
}
```

For more information, see the reference page for **azn_id_get_creds()** on page 139. Refer also to the Authorization API demonstration program.

The application is now ready to submit the authorization request. See **"Obtaining an Authorization Decision"** on page 54.
Obtaining an Authorization Decision

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

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

- “Mapping the User Operation to a Policy Director Permission”
- “Mapping the Requested Resource to a Protected Object” on page 55
- “Assigning the User Credentials to a Credentials Handle” on page 55
- “Building an Attribute List for Additional Application Information” on page 55
- “Obtaining an Authorization Decision” on page 56

Mapping the User Operation to a Policy Director Permission

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

Note: For a complete list of 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 WebSEAL protected resource, such as a file in the Web space. Alternatively, the resource can be a custom protected object.

Complete the following step:

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

Assigning the User Credentials to a Credentials Handle

The authorization credentials for a user obtained in “Obtaining User Authorization Credentials” on page 49 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 60.

Building an Attribute List for Additional Application Information

The Policy Director Authorization API provides the extended function `azn_decision_access_allowed_ext()` for obtaining an access
decision. This function extends \texttt{azn\_decision\_access\_allowed()} by providing an additional input parameter and an additional output parameter.

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

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

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

1. Use \texttt{azn\_attrlist\_create()} to create a new, empty attribute list.
2. Use \texttt{azn\_attrlist\_add\_entry()} or \texttt{azn\_attrlist\_add\_entry\_buffer()} to add attributes.
3. When all attributes have been added, assign the input parameter \texttt{app\_context} to point to the attribute list.

For more information, see the reference page for "\texttt{azn\_decision\_access\_allowed\_ext()}" on page 128.

**Obtaining an Authorization Decision**

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

- \texttt{azn\_decision\_access\_allowed()}
- \texttt{azn\_decision\_access\_allowed\_ext()}

If the API is operating in remote cache mode, the authorization request will be forwarded to the Policy Director Authorization Server. The Authorization Server makes the decision and returns the result.
If the API is operating in local cache mode, the API uses the local authorization policy database replica to make the authorization decision.

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>permission</td>
<td>The result of the access request. Consists of one of the following constants:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AZN_C_PERMITTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AZN_C_NOT_PERMITTED</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>azn_attrlist_h_t</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 126
- “azn_decision_access_allowed_ext()” on page 128

### Cleaning Up and Shutting Down

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

- “Releasing Allocated Memory” on page 58
- “Shutting Down the Authorization API” on page 58
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 60
- "Creating a Chain of Credentials" on page 60
- "Determining the Number of Credentials in a Credentials Chain" on page 61
- "Obtaining a Credential from a Chain of Credentials" on page 61
- "Modifying the Contents of a Credential" on page 61
- "Obtaining an Attribute List from a Credential" on page 62

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()` to modify a credential by placing additional information, contained in an attribute list, into the credentials structure. Use this function when you need to add application-specific information to a user’s credentials.
Complete the following steps:

1. Use the attribute list functions to create an attribute list containing the information to be added. Assign the attribute list handle `mod_info` to the new attribute list.
   For more information on attribute lists, see the section “Attribute Lists” on page 16.

2. Set the credential modification service `mod_svc_id` to NULL.

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

4. Call `azn_creds_create()` to create a new, empty credentials structure.

5. Call `azn_creds_modify()`.

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

**Obtaining an Attribute List from a Credential**

Use the function `azn_creds_for_subject()` to obtain information, in the form of an attribute list, from a credential. Attribute lists are added to credentials structures by calls to `azn_creds_modify()`.

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

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

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>azn_creds_version</code></td>
<td>The credential version</td>
</tr>
<tr>
<td><code>azn_creds_mech_id</code></td>
<td>The mechanism ID for this credential</td>
</tr>
<tr>
<td><code>azn_creds_principal_uuid</code></td>
<td>The UUID for the entity</td>
</tr>
<tr>
<td><code>azn_creds_principal_name</code></td>
<td>The string name of the entity</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>azn_cred_group_uuids</td>
<td>The string group UUID memberships of this entity</td>
</tr>
<tr>
<td>azn_cred_group_names</td>
<td>The string group name memberships of this entity</td>
</tr>
<tr>
<td>azn_cred_ldap_dn</td>
<td>The LDAP DN used to build these authorization credentials</td>
</tr>
<tr>
<td>azn_cred_ufaf_name</td>
<td>The URAF name used to build these authorization credentials</td>
</tr>
<tr>
<td>azn_cred_user_info</td>
<td>Any user information that was passed in the mechinfo structure</td>
</tr>
<tr>
<td>azn_cred_auth_method</td>
<td>Any authentication method information that was passed in the mechinfo structure.</td>
</tr>
<tr>
<td>azn_cred_authmech_info</td>
<td>Any authentication mechanism information that was passed in the mechinfo structure.</td>
</tr>
<tr>
<td>azn_cred_ip_address</td>
<td>The IP address information passed in the mechinfo structure</td>
</tr>
<tr>
<td>azn_cred_browser_info</td>
<td>The browser information passed in the mechinfo structure</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 Attribute Values for 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 access the attribute list for a specified credential, and returns the string value of the specified attribute. For more information, see the reference page for "`azn_creds_get_attr_value_string()`" on page 110.

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 there are a number of attributes that are considered read-only and must not be modified by functions such as `azn_creds_set_attr_value_string()` or `azn_creds_modify()`. These attributes are read-only:

- `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`

For more information, see the reference page for "`azn_creds_set_attr_value_string()`" on page 124.
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 function `azn_creds_copy()` to copy the contents of one credentials to another, new credentials.

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

2. Pass the credentials handle as the input parameter to `azn_creds_copy()`.

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

   When the application is finished using the new credential, use `azn_creds_delete()` to release the memory that was allocated.

   **Note:** To add a credential to an existing credential, use `azn_creds_combine()`.

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

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

- Policy Director Runtime

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

### Demonstration Programs

The Authorization ADK provides the following demonstration programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>authzn_demo</td>
<td>Authorization API demonstration program</td>
</tr>
<tr>
<td>azn_admin_svc_demo</td>
<td>Administration Service demonstration program</td>
</tr>
<tr>
<td>azn_eas_demo</td>
<td>External Authorization Service demonstration program</td>
</tr>
</tbody>
</table>

### Backwards Compatibility and Application Migration

Tivoli SecureWay Policy Director Base Version 3.8 is binary backwards compatible to existing Tivoli SecureWay Policy Director Version 3.7 Authorization API clients.

Existing Authorization API clients that are being migrated to use the Tivoli SecureWay Policy Director Version 3.8 Authorization ADK must run **svrsslcfg** 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 may receive errors for some interfaces when compiling existing Authorization API clients with the Policy Director 3.8 Authorization ADK. This is because some of the Authorization API interfaces from Version 3.7 have been deprecated for Version 3.8.

The deprecated interfaces are located in the azn_deprecated.h header file. You can look in azn_deprecated.h 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 azn_deprecated.h to continue to use the deprecated Version 3.7 interfaces. However, you should switch as soon as possible to the new interfaces, to avoid further problems.

## Deprecated API Elements

This section describes elements of the Authorization API that have been deprecated for release 3.8. The deprecated elements are supported for backwards compatibility in Policy Director 3.8. Application developers should not use the deprecated elements for any new applications.

### Permission Info Attribute Values

The following values for the azn_init_set_perminfo_attr 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`.

For Policy Director 3.8, 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 Policy Director 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
1. Authorization C API

- 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
This section contains reference pages for the following Authorization API:

- `azn_attrlist_add_entry()`
- `azn_attrlist_add_entry_buffer()`
- `azn_attrlist_copy()
- `azn_attrlist_create()
- `azn_attrlist_delete()
- `azn_attrlist_delete_entry()
- `azn_attrlist_get_entry_buffer_value()
- `azn_attrlist_get_entry_string_value()
- `azn_attrlist_get_names()
- `azn_attrlist_name_get_num()
- `azn_creds_combine()
- `azn_creds_copy()
- `azn_creds_create()
- `azn_creds_delete()
- `azn_creds_for_subject()
- `azn_creds_get_attrlist_for_subject()
- `azn_creds_get_pac()`
This section includes reference pages for the following utilities:

- azn_creds_get_attr_value_string()
- azn_creds_modify()
- azn_creds_num_of_subjects()
- azn_creds_set_attr_value_string()
- azn_decision_access_allowed()
- azn_decision_access_allowed_ext()
- azn_entitlement_get_entitlements()
- azn_error_major()
- azn_error_minor()
- azn_error_minor_get_string()
- azn_id_get_creds()
- azn_initialize()
- azn_pac_get_creds()
- azn_release_buffer()
- azn_release_string()
- azn_release_strings()
- azn_shutdown()
- azn_util_errcode()
- azn_util_password_authenticate()
- azn_util_handle_is_valid()
### azn_attrlist_add_entry()

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

**Syntax**

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

**Parameters**

**Input**

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

**Description**

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

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

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

**Return Values**

If successful, the function will return AZN_S_COMPLETE.

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

- **AZN_S_COMPLETE**
  - Successful completion.
- **AZN_S_INVALID_ATTRLIST_HDL**
Attribute list handle is invalid.

- **AZN_S_INVALID_ATTR_NAME**
  Attribute name is invalid.

- **AZN_S_INVALID_STRING_VALUE**
  Attribute value is invalid.

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

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

Syntax

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

Parameters

Input

    attr_list
    Handle to an attribute list.

    attr_name
    Name attribute of the entry to be added.

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

Description

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

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

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

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( azn_attrlist_h_t attr_list,
                        azn_string_t attr_name, azn_pobj_t pobj_value);
```

**Parameters**

**Input**

- **attr_list**
  Handle to an attribute list

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

- **pobj_value**
  The value (protected object) of the attribute to be added

**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

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

Parameters

Input

attr_list
Handle to an attribute list.

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

ulong_value
Unsigned long value.

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_create()

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

**Syntax**

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

**Parameters**

*Output*

*new_attr_list*

Pointer to the new attribute list handle that is returned.

**Description**

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

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

**Return Values**

If successful, the function will return AZN_S_COMPLETE.

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

- AZN_S_COMPLETE
  
  Successful completion.

- AZN_S_INVALID_ATTRLIST_HDL

  Attribute list handle is invalid.

- AZN_S_FAILURE

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

Copies a valid attribute list to a new attribute list.

**Syntax**

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

**Parameters**

**Input**

*attr_list*

The handle of the attribute list to be copied.

**Description**

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

**Return Values**

If successful, the function will return AZN_S_COMPLETE.

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

- **AZN_S_COMPLETE**
  Successful completion.

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

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

Deletes the attribute list associated with the attribute list handle.

Syntax

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

Parameters

Input

*old_attr_list*

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

Output

*old_attr_list*

On output, a NULL pointer containing an invalid value.

Description

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

Return Values

If successful, the function will return AZN_S_COMPLETE.

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

- **AZN_S_COMPLETE**
  
  Successful completion.

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

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

Deletes the attribute list associated with the attribute list handle.

**Syntax**

```c
azn_status_t azn_attrlist_delete( azn_attrlist_h_t attr_list, azn_string_t attr_name);
```

**Parameters**

**Input**

- `attr_list`
  An existing attribute list handle.

- `attr_name`
  The name of an attribute contained within the attribute list that is referenced by `attr_list`.

**Description**

This function deletes the specified attribute `attr_name` from the attribute list associated with the handle `attr_list`.

**Return Values**

If successful, the function will return AZN_S_COMPLETE.

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

- **AZN_S_COMPLETE**
  Successful completion.

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

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

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

Syntax

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

Parameters

**Input**

*attr_list*
Handle to an attribute list.

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

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

**Output**

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

Description

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

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

Return Values

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

- **AZN_S_COMPLETE**
  Successful completion.

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

- **AZN_S_INVALID_ATTR_NAME**
  Attribute name is invalid.

- **AZN_S_INVALID_BUFFER_REF**
  Buffer reference is not valid.

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

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

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

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

Syntax

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

Parameters

Input

attr_list
Handle to an attribute list

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 a protected object structure that holds the value of the attribute to be returned.

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`.
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

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

Parameters

**Input**

* attr_list
  Handle to an attribute list.

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

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

**Output**

* 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_entry_string_value()

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

Syntax

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

Parameters

    Input

    attr_list
        Handle to an existing attribute list.

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

    value_index

    Output

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

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

Description

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

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

Return Values

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

- **AZN_S_COMPLETE**
  Successful completion.

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

- **AZN_S_INVALID_ATTR_NAME**
  Attribute name is invalid.

- **AZN_S_INVALID_STRING_REF**
  String reference is invalid.

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

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

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

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

Syntax

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

Parameters

**Input**

- `attr_list`
  Handle to an existing attribute list

**Output**

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

Description

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

Return Values

If successful, the function will return `AZN_S_COMPLETE`.

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

- `AZN_S_COMPLETE`
  Successful completion.

- `AZN_S_INVALID_ATTRLIST_HDL`
  Attribute list handle is invalid.

- `AZN_S_INVALID_STRING_REF`
String reference is invalid.

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

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

Syntax

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

Description

Input

- `attr_list`
  Handle to an existing attribute list.

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

Output

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

Description

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

Return Values

If successful, the function will return AZN_S_COMPLETE.

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

- AZN_S_COMPLETE
  Successful completion.
- AZN_S_INVALID_ATTRLIST_HDL
Attribute list handle is invalid.

- **AZN_S_INVALID_ATTR_NAME**
  Attribute name is invalid.

- **AZN_S_INVALID_INTEGER_REF**
  Integer reference is invalid.

- **AZN_S_FAILURE**
  An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.
azn_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( azn_creds_h_t creds, azn_creds_h_t creds_to_add, azn_creds_h_t *combined_creds);
```

Parameters

**Input**

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

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

**Output**

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

Description

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

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

If successful, the function will return AZN_S_COMPLETE.

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

- **AZN_S_COMPLETE**
  Successful completion.

- **AZN_S_API_UNINITIALIZED**
  This function has been called before azn_initialize().

- **AZN_S_INVALID_CREDSD_HDL**
  Handle passed as creds is invalid.

- **AZN_S_INVALID_ADDED_CREDSD_HDL**
  Credentials handle passed as creds_to_add is invalid.

- **AZN_S_INVALID_COMB_CREDSD_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

azn_creds_h_t azn_creds_copy( const azn_creds_h_t creds);

Parameters

Input
cred
Handle to a credential.

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 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_CRED_HDL**
  Handle passed as cred is invalid.

- **AZN_S_FAILURE**
  An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code. The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for
the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism. See pdbaclmsg.h for a complete list of minor error codes that describe access control problems.
azn_creds_create()

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

Syntax

    azn_status_t azn_creds_create( azn_creds_h_t *creds );

Parameters

    Output

    creds

            Pointer to the new credentials handle that is returned.

Description

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

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

Return Values

If successful, the function will return AZN_S_COMPLETE.

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

- AZN_S_COMPLETE
  Successful completion.

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

- AZN_S_INVALID_CREDS_HDL
  The credentials handle supplied is invalid.

- AZN_S_FAILURE
  An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code.
The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See 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 handle of the credentials chain to be deleted.

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

Description

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

Return Values

If successful, the function will return AZN_S_COMPLETE.

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

- **AZN_S_COMPLETE**
  Successful completion.

- **AZN_S_API_UNINITIALIZED**
  This function has been called before azn_initialize().

- **AZN_S_INVALID_CREDS_HDL**
  The credentials handle supplied is invalid.

- **AZN_S_FAILURE**


An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code. The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.

See 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

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.

cred2
Handle to a credential.

Output

is_equal
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 cred 2 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( azn_creds_h_t creds, unsigned int subject_index, azn_creds_h_t *new_creds);
```

Parameters

**Input**

- `creds`
  - Handle to a credentials structure representing the combined credentials chain of several subjects. The combined credentials chain contains a list of 1 or more individual credentials chains. When this function returns, the structure referred to by `creds` is unchanged.

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

**Output**

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

Description

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

This function does not modify the input handle `creds` and the credentials chain to which it refers. Later changes to this structure, including the release of its storage, have no effect on `new_creds`.
Combined credentials chains are created by `azn_creds_combine()`. The first credential chain in a combined credentials chain is that of the initiator, and its index is zero (0). Callers can retrieve the credentials of the initiator by passing the constant `AZN_C_INITIATOR_INDEX` as the value of `subject_index`.

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

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

**Return Values**

If successful, the function will returns `AZN_S_COMPLETE`.

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

- **AZN_S_COMPLETE**
  Successful completion.

- **AZN_S_API_UNINITIALIZED**
  This function has been called before `azn_initialize()`.

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

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

- **AZN_S_INVALID_SUBJECT_INDEX**
  The supplied index is invalid.

- **AZN_S_UNIMPLEMENTED_FUNCTION**
  This function is not supported by the implementation.

- **AZN_S_FAILURE**
  An error or failure has occurred. Use `azn_error_minor()` to derive specific minor error codes from the returned status code.
The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism. See 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

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

Parameters

**Input**

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

- **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**
  The value of the specified attribute.

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().

Version 3.8
- **AZN_S_INVALID_STRING_REF**
  The specified attribute name is invalid.

- **AZN_S_INVALID_CRED_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 ( azn_creds_h_t creds,
unsigned int subject_index, azn_attrlist_h_t *creds_attrlist);
```

Parameters

**Input**

- `creds`
  Handle to a credentials chain.

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

**Output**

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

Description

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

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

This function does not modify the input handle `creds` and the credentials chain to which it refers. Later changes to `creds`, including releasing its storage, will have no effect on `creds_attrlist`.
Use the `azn_attrlist*` functions to retrieve individual attribute values from `creds_attrlist`. See ogauthzn.h for a list of attribute names.

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

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

**Return Values**

If successful, the function will return `AZN_S_COMPLETE`.

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

- **AZN_S_COMPLETE**
  Successful completion.

- **AZN_S_API_UNINITIALIZED**
  This function has been called before `azn_initialize()`.

- **AZN_S_INVALID_CREDS_HDL**
  The credentials handle supplied is invalid.

- **AZN_S_INVALID_SUBJECT_INDEX**
  The supplied index is invalid.

- **AZN_S_INVALID_ATTRLIST_HDL**
  The attribute list handle supplied is invalid.

- **AZN_S_UNIMPLEMENTED_FUNCTION**
  This function is not supported by the implementation.

- **AZN_S_FAILURE**
  An error or failure has occurred. Use `azn_error_minor()` to derive specific minor error codes from the returned status code.
  The minor error code `ivacl_s_unauthorized` is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for
the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism.
See 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( azn_creds_h_t creds, azn_string_t pac_svc_id, azn_buffer_t *pac);
```

**Parameters**

**Input**

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

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

**Output**

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

**Description**

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

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

This function takes as an input parameter a handle to an existing credentials structure, and returns a pointer to the output PAC in an Authorization API buffer.
This function does not modify the input handle creds and the credentials chain to which it refers. Later changes to creds, including releasing its storage, will have no effect on pac.

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

**Return Values**

If successful, the function will return AZN_S_COMPLETE.

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

- **AZN_S_COMPLETE**
  Successful completion.

- **AZN_S_API_UNINITIALIZED**
  This function has been called before azn_initialize().

- **AZN_S_INVALID_CRED_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( azn_creds_h_t creds, azn_string_t mod_svc_id, azn_attrlist_h_t mod_info, azn_creds_h_t *new_creds);
```

Parameters

**Input**

*creds*
Handle to the authorization credentials chain to be modified.

*mod_svc_id*
Identification (id) of the credential modification service.

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

**Output**

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

Description

This function uses the specified modification service *mod_svc_id*, and optionally an attribute list *mod_info* which contains modification information provided by the caller, to modify a copy of the supplied credentials chain *creds*. The function returns a pointer to a handle to a new credentials chain *new_creds* containing the requested modifications. The supplied credentials chain is unchanged.
When `mod_svc_id` is NULL, this function modifies an existing credential chain `creds` by adding the attribute list `mod_info` to the credentials chain, and returning the modified credential in `new_creds`.

The following credential attributes are considered `readonly` and must not be modified:

- `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.

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_CREDSD_HDL
  The credentials handle supplied is invalid.

- AZN_S_INVALID_MOD_FUNCTION
  The supplied modification service identifier is invalid.

- AZN_S_INVALID_ATTRLIST_HDL
  The attribute list handle is invalid.

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

- AZN_S_UNIMPLEMENTED_FUNCTION
  This function is not supported by the implementation.

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

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

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

- AZN_S_ATTR_READONLY
  Modification of the attribute is prohibited.

- 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

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

Parameters

Input

creds
Handle to a credentials chain.

Output

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

Description

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

Return Values

If successful, the function will return AZN_S_COMPLETE.

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

- **AZN_S_COMPLETE**
  Successful completion.

- **AZN_S_API_UNINITIALIZED**
  This function has been called before azn_initialize().

- **AZN_S_INVALID_CRED_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
    azn_creds_h_t creds,
    unsigned int subject_index,
    azn_string_t attr_name,
    azn_string_t attr_value
);
```

Parameters

Input

- **creds**
  Handle to the credentials chain.

- **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

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

Parameters

**Input**

- **creds**
  Handle to the initiator’s credential chain.

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

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

**Output**

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

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

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

**Description**

This function decides whether the initiator specified by credentials `creds` is authorized to perform the operation `operation` on the target `protected_resource`. The decision is returned through `permission`. 
azn_decision_access_allowed() is semantically equivalent to azn_decision_access_allowed_ext() when app_context=NULL and permission_info=NULL.

**Return Values**

If successful, the function will return AZN_S_COMPLETE.

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

- **AZN_S_COMPLETE**
  
  Successful completion.

- **AZN_S_API_UNINITIALIZED**
    
    This function has been called before azn_initialize().

- **AZN_S_INVALID_CREDS_HDL**
    
    The credentials handle supplied is invalid.

- **AZN_S_INVALID_PROTECTED_RESOURCE**
    
    The target name is invalid.

- **AZN_S_INVALID_OPERATION**
    
    The operation has no meaning for the specified target.

- **AZN_S_INVALID_PERMISSION_REF**
    
    The integer reference to return the permission is invalid.

- **AZN_S_FAILURE**
    
    An error or failure has occurred. Use azn_error_minor() to derive specific minor error codes from the returned status code. The minor error code ivacl_s_unauthorized is returned when the caller is not authorized to use this function. Authorization might fail because the caller does not belong to the correct group for the Authorization API mode (remote or local), or because of issues specific to the authentication mechanism. See 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( azn_creds_h_t creds,
    azn_string_t protected_resource, azn_string_t operation,
    azn_attrlist_h_t app_context, int *permission,
    azn_attrlist_h_t *permission_info);
```

Parameters

**Input**

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

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

- `operation`
  Name of the requested operation.

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

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

**Output**

- `permission`
  Value of the returned permission.

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

**permission_info**

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

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

For more information, see “Enabling the Return of Permission Information” on page 44.

**Description**

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

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

**Return Values**

If successful, the function will return AZN_S_COMPLETE.

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

Ref: Enabling the Return of Permission Information on page 44.
- **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_PROTECTEDRESOURCE**
  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()

Obtains entitlements from the specified entitlements service plug-in.

**Syntax**

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

**Description**

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

This function returns an attribute list containing the entitlements.

If a NULL service ID is passed in then, this function invokes the Policy Director protected object entitlements service. For more information on this service, and its expected inputs, see “Implementing an Entitlements Service” on page 186.

**Parameters**

**Input**

`creds`

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

`svc_id`

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

`app_context`

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

**Output**
A pointer to an attributes list that contains the user’s entitlements. The entitlements are presented as a series of name-value pairs.

**Return Values**

If successful, the function will return AZN_S_COMPLETE.

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

- **AZN_S_INVALID_CREDS_HDL**
  The credentials handle supplied is invalid.

- **AZN_S_INVALID_ENTITLEMENTS_SVC**
  The Entitlements Service identifier is invalid.

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

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

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

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

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

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

Parameters

Input

    aznstatus

        Authorization API status

Output

    error_string

        Pointer that holds the message string to be returned.

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**

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

**Parameters**

**Input**

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

**Description**

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

**Return Values**

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

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

Syntax

unsigned int azn_error_minor( azn_status_t status_code);

Parameters

Input

status_code

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

Description

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

Return Values

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

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

Syntax

azn_status_t azn_error_minor_get_string( unsigned int minor_error
azn_string_t *minor_error_string);

Parameters

Input

minor_error
Minor error code previously returned by azn_error_minor().

Output

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

Description

This function returns a string that describes the error corresponding to a previously returned minor error status code. When minor_error_string is no longer needed, use azn_release_string() to release its storage.

Return Values

If successful, the function will return AZN_S_COMPLETE.

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

- AZN_S_COMPLETE
  Successful completion.

- AZN_S_MINOR_CODE_MESSAGE_NOT_FOUND
  The specified minor code has not corresponding error message in the catalog. This message also appears when a minor code of 0 (success) is passed.
- **AZN_S_FAILURE**
  The specified *minor_error* code is invalid, or no string describing the specified *minor_error* can be returned.
azn_id_get_creds()

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

Syntax

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

Parameters

Input

authority
Identification (id) of the authorization authority to be used to build the credential. 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
Handle to a new, empty credentials chain that will hold the returned credentials.

Description

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

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

Specifying NULL values for *mechanism_id* and 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 14.

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

**Parameters**

**Input**

*init_data*

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

**Output**

*init_info*

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

**Description**

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

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.

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**
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 
decribe access control problems.
azn_pac_get_creds()

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

Syntax

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

Parameters

**Input**

* pac
  Buffer structure that holds the supplied PAC.

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

**Output**

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

Description

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

This function decodes PACs that are built by azncreds_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.

**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

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.

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

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

Parameters

Input

(string) Pointer to the string to be released.

Output

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

Description

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

Return Values

If successful, the function will return AZN_S_COMPLETE.

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

- AZN_S_COMPLETE
  Successful completion.

- AZN_S_INVALID_STRING_REF
  The pointer to the string is invalid.

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

Frees storage that is associated with an array of strings.

Syntax

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

Parameters

Input

*strings*

Pointer to the array of azn_string_t structures to be released.

Output

*string*

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

Description

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

Return Values

If successful, the function will return AZN_S_COMPLETE.

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

- **AZN_S_COMPLETE**
  Successful completion.

- **AZN_S_INVALID_STRING_REF**
  Pointer to the array of strings is invalid.

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

Cleans up internal authorization service state in preparation for shutdown.

Syntax

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**

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

**Description**

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

**Parameters**

**Input**

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

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

**Return Values**

Returns complete `azn_status_t` error code.
azn_util_handle_is_valid()

Determine if the handle references a valid data structure.

Syntax

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

Parameters

- **Input**
  - `handle`
    - A handle to an attribute list.

Description

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

Return Values

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

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

**Syntax**

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

**Parameters**

**Input**

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

*password*
   Password for the user.

**Output**

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

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

**Description**

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

The authentication mechanism used depends upon the underlying authentication mechanism that was configured when the Authorization API was installed. Policy Director supports DCE and LDAP authentication. For LDAP Authorization API authentication, the `azn_initialize()` function must have completed successfully.
This function does not establish a security context for the application.

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

**Return Values**

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

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

See 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_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.
svrsslcfg

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

Syntax

svrsslcfg -config -f <cfg_file> -d kdb_dir -n server_name -s
server_type -r port -P admin_pwd [-S server_password] [-A
listening_mode] [-a refresh_mode]

svrsslcfg -unconfig -f <cfg_file> -n server_name -P admin_pwd[-A
admin_id]

Description

Configuration

The svrsslcfg program performs the following configuration tasks:

- Creates a user with a name by combining the specified
  <server_name> with the local TCP/IP host name.
  Note that if the <server_name> supplied with the -n option
  includes the local TCP/IP host name, then the <server_name>
  will equate to the user name. For example, svrsslcfg will
  combine a <server_name> of MyApp with a local host name of
  host1.domain.com to form a user account name of
  myApp/host1.domain.com. When the <server_name> is
  specified as myApp/host1.domain.com, svrsslcfg does not
determine the TCP/IP host name, but instead uses the supplied
string as the user name.

- Creates an SSL key file for that user. For example,
demo_user.key and demo_user.sth.

- Adds the user to the ivacld-servers group when the
  <server_type> is local, or to the remote-acl-users group when
  the <server_type> is remote.
Administrators use the `-config` option to configure an application that has been written with the Authorization API. This option enables the application to communicate with either the Management Server or the Authorization Server. The Policy Director Runtime Environment must be installed before using this utility.

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

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

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

```
svrsslcfg -config
```

### Unconfiguration

You can also use `svrsslcfg` to unconfigure the communication between the Authorization API application and the Policy Director servers. The command syntax is:

```
svrsslcfg -unconfig -f cfg_file -n server_name -P admin_pwd [-A admin_id]
```

To view the usage message for unconfiguration, type:

```
svrsslcfg -unconfig
```

### Options

- `-d kdb_dir`
  
  The directory that is to contain the keyring database files for the server. This directory must already exist, it will not be created by `svrsslcfg`. This is a required parameter.

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

-s server_type
The type of server being configured. The value must be either local or remote. This is a required parameter.

-P admin_pwd
The Policy Director Administrator password. When the User Registry type is LDAP, then this must be the Security Master principal’s password. This is a required parameter. If this parameter is not specified, the password will be read from stdin.

-S server_pwd
The server’s password. This parameter is required. However, you can request that a password be created by the system by specifying a dash (-) for the password. If this option is used, the configuration file will not be updated with the password created by the system. If the User Registry type is LDAP and a password is specified, it will be saved in the configuration file. If this parameter is absent, the server password will be read from stdin.

-A admin_id
The Policy Director Administrator name. When the User Registry type is LDAP, then this parameter is ignored, and the value “sec_master” is used instead.

-e pwd_life
The keyring file’s password expiration time in days. This parameter is optional. If not specified during initial configuration, a default of 183 days is used.

-r port_num
Set the listening port number for the server. This is a required parameter. A value of 0 may be specified only if the [aznapi-admin-services] stanza in the configuration file is empty.

-t ssl_timeout
Specify the SSL session timeout in seconds. The value must be
in the range 1-86400. This parameter is optional. If it is not specified during initial configuration, a default value of 7200 is used.

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

-l listen_mode
Sets the listening-enabled flag in the configuration file. The value of this parameter must be yes or no. If not specified, the default is no. When used with the -config action, a value of yes requires that the -r parameter must have a non-zero value. When used with the -modify action, a value of yes requires that the listening port number in the configuration file be non-zero.

-a refresh_mode
Sets the certificate and keyring file password auto-refresh enabled flag in the configuration file. The value of this parameter must be yes or no. If not specified, the default is yes.

Example
For example, the demonstration program that is distributed with the Authorization ADK invokes svrsslcfg as follows:

```
svrsslcfg -config -f /opt/PolicyDirector/example/authzn_demo/local.conf -d /opt/PolicyDirector/example/authzn_demo -n authzn_local -S <svr-password> -s local -P <admin-password> -r 7777
```

Note that the demonstration program uses a configuration file named local.conf. For more information, see the Readme file that accompanies the demonstration program software.
The Authorization API supports a Service Plug-in model. This model enables developers to write plug-in modules that extend the capabilities of the Policy Director Authorization Service. Developers of third party applications can use Authorization API functions that access the Service Plug-in interface to perform authorization operations that are specific to the Policy Director secure domain.

This chapter contains the following sections:
- “Service Plug-in Architecture”
- “Implementing a Service Plug-in” on page 169
- “Implementing an Entitlements Service” on page 186
- “Implementing an Administration Service Plug-in” on page 193
- “Implementing External Authorization Service Plug-in” on page 205
- “Supplied Implementations for Service Plug-ins” on page 219

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 194.

**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 Policy Director Authorization Server. The data is passed unchanged through the Authorization Service Dispatcher to the Authorization Service Plug-ins.

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

**Calling Applications**

Applications may choose to use information received from Service Plug-ins to make authorization decisions without requiring access to
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 
\texttt{azn\_entitlement\_get\_entitlements()}. 

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

The following table lists the handles that must be valid:

<table>
<thead>
<tr>
<th>Service</th>
<th>API Function Invoked By the Calling Application</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entitlements</td>
<td>\texttt{azn_entitlements_get_entitlements()}</td>
<td>entitlements</td>
</tr>
<tr>
<td>Credentials Modification</td>
<td>\texttt{azn_creds_modify()}</td>
<td>creds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>new_creds</td>
</tr>
<tr>
<td>Privilege Attribute Certificate</td>
<td>\texttt{azn_pac_get_creds()}</td>
<td>new_pac</td>
</tr>
<tr>
<td></td>
<td>\texttt{azn_creds_get_pac()}</td>
<td>pac</td>
</tr>
<tr>
<td>Administration Service</td>
<td>\texttt{ivadmin_protobj_get2()}</td>
<td>creds indata</td>
</tr>
<tr>
<td></td>
<td>\texttt{ivadmin_protobj_list3()}</td>
<td>outdata</td>
</tr>
<tr>
<td></td>
<td>\texttt{ivadmin_server_gettasklist()}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\texttt{ivadmin_server_performtask()}</td>
<td></td>
</tr>
<tr>
<td>External Authorization Service</td>
<td>\texttt{azn_decision_access_allowed_ext()}</td>
<td>creds app_context permission_info</td>
</tr>
</tbody>
</table>

Applications are responsible for initializing and shutting down the Authorization API. To do this, applications call the Authorization API functions \texttt{azn\_initialize()} and \texttt{azn\_shutdown()}. These functions call the appropriate functions for initializing and shutting down the Service Plug-ins.
For some services, Policy Director provides a default service. For all other services, the application must specify a Service ID as a parameter to `azn_service_initialize()`.

**Supported Types of Service Plug-ins**

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

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

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

**Entitlements Services**

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

- An application can allow or deny a user request for access to a protected action or protected resource, based on the user’s entitlements.

- A graphical user interface application can use entitlements information to construct a graphical view of the Policy Director secure domain 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 Policy Director credential. The Credentials Modification Service can then return this modified credential for use by the calling application. Applications can use this service to add additional information to a
user’s credential. For example, this additional information could include the user’s credit card number and the user’s credit limit.

**Privilege Attribute Certificate Service**

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

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

**Administration Service**

An Administration Service plug-in enables applications to perform application-specific administration tasks on protected object resources that are secured in the Policy Director 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 Policy Director Administration API calls. The calling application can be either a administrative utility such as the Policy Director `pdadmin` command or the Policy Director 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 Policy Director 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 make based on application-specific criteria that are not known to the Policy Director 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 174
- “Using Error Codes” on page 176
- “Shutdown” on page 181
- “Example Service Source Code” on page 181

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 attributes defined for the `svc_init` parameter.

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

For more information, see the reference page for `azn_service_initialize()` on page 226.

**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 Policy Director platform.

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

For example, the Authorization API will search for a library with the short form name of azn_ent_user by looking for each of the following files:

<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. 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 172
- “Configuring a Service Programmatically” on page 173

**Configuring A Service By Using a Configuration File**

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

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

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

<table>
<thead>
<tr>
<th>Entry</th>
<th>Service Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[aznapi-entitlement-services]</td>
<td>Entitlements Service Plug-ins</td>
</tr>
<tr>
<td>[aznapi-pac-services]</td>
<td>Privilege Attribute Certificate Service Plug-ins</td>
</tr>
</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 `aznapi.conf`.

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

```c
/* Entitlements service definition attribute */
AZN_DECLSPEC azn_string_t azn_init_ent_svc;
/* PAC service definition attribute */
AZN_DECLSPEC azn_string_t azn_init_pac_svc;
/* Credential modification service definition attribute */
AZN_DECLSPEC azn_string_t azn_init_cred_mod_svc;
/* Administration service definition attribute */
AZN_EXTSPEC azn_string_t azn_init_admin_svc;
/* External Authorization Service definition attribute */
AZN_EXTSPEC azn_string_t azn_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 170. 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.
azn_status_t status;
azn_attrlist_h_t init_data, init_info;
azn_string_t service_entry;

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

/*
 * Load an Entitlements Service programmatically. The service
 * entry will load the DLL "mysvc" and associate it with the
 * service ID "MYSVC". The dispatcher will automatically search
 * the library search path for the platform specific DLL name
 * variations of this name. 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>Svc</th>
<th>Authorization API Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Invoked By the Calling Application</td>
</tr>
<tr>
<td>Entitlements Service</td>
<td>azn_entitlement_get_entitlements()</td>
</tr>
<tr>
<td>Credentials Modification Service</td>
<td>azn_creds_modify()</td>
</tr>
<tr>
<td>Privilege Attribute Certificate</td>
<td>azn_pac_get_creds()</td>
</tr>
<tr>
<td></td>
<td>azn_creds_get_pac()</td>
</tr>
<tr>
<td>Administration Service</td>
<td>ivadmin_protobj_get2()</td>
</tr>
<tr>
<td></td>
<td>ivadmin_protobj_list3()</td>
</tr>
<tr>
<td></td>
<td>ivadmin_server_gettasklist()</td>
</tr>
<tr>
<td></td>
<td>ivadmin_server_performtask()</td>
</tr>
<tr>
<td>External Authorization Service</td>
<td>azn_decision_access_allowed()</td>
</tr>
<tr>
<td></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
\texttt{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
\texttt{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 \texttt{azn\_decision\_access\_allowed\_ext()} that can make authorization decisions based on conditions or rules
that are potentially unknown to the core Policy Director
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.

\section*{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 the using the
\texttt{azn\_error\_major()} and \texttt{azn\_error\_minor()} function calls.

Developers of Service Plug-ins should use the Authorization API
utility function \texttt{azn\_util\_errcode()} to construct valid error codes to
return to the calling application. The function `azn_util_errcode()` takes major and minor integer error code values and converts them in to an `azn_status_t` value. The minor error code must be defined as described in "Minor Error Codes" on page 180.

**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>
<tbody>
<tr>
<td>AZN_S_SVC_DEFINITION_ERROR</td>
<td>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 <code>init_data</code> attribute list of <code>azn_initialize()</code>.</td>
</tr>
<tr>
<td>AZN_S_SVC_SERVICE_NOT_FOUND</td>
<td>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.</td>
</tr>
<tr>
<td>AZN_S_SVC_INITIALIZE_NOT_FOUND</td>
<td>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 <code>azn_service_initialize()</code> interface could not be found in the loaded Service Plug-in.</td>
</tr>
<tr>
<td>Error</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AZN_S_INVALID_PAC_SVC</td>
<td>The identifier (id) of the PAC service is invalid.</td>
</tr>
<tr>
<td>AZN_S_SVC_DISPATCHER_FAILURE</td>
<td>The service dispatcher failed. This can be caused by incorrect initialization of the Authorization API.</td>
</tr>
<tr>
<td>AZN_S_SVC_DLL_LOAD_FAILED</td>
<td>The DLL for a service plug-in failed to load correctly</td>
</tr>
<tr>
<td>AZN_S_SVC_SERVICE_IS_REGISTERED</td>
<td>The service ID cannot be registered because it is already listed as registered by the Service Dispatcher.</td>
</tr>
<tr>
<td>AZN_S_SVC_INITIALIZE_NOT_FOUND</td>
<td>The azn_service_initialize() function was not found.</td>
</tr>
<tr>
<td>AZN_S_SVC_SHUTDOWN_NOT_FOUND</td>
<td>The azn_service_shutdown() function was not found.</td>
</tr>
<tr>
<td>AZN_S_SVC_ENT_FUNC_NOT_FOUND</td>
<td>The azn_entitlement_get_entitlements() function was not found.</td>
</tr>
<tr>
<td>AZN_S_SVC_PAC_FUNC_NOT_FOUND</td>
<td>One of either azn_pac_get_creds() or azn_creds_get_pac() was not found.</td>
</tr>
<tr>
<td>AZN_S_SVC_CRED_MOD_FUNC_NOT_FOUND</td>
<td>The azn_creds_modify() function was not found.</td>
</tr>
<tr>
<td>AZN_S_SVC_EAS_FUNC_NOT_FOUND</td>
<td>One of either azn_decision_access_allowed() or azn_decision_access_allowed_ext() was not found.</td>
</tr>
<tr>
<td>AZN_S_SVC_ADMIN_POBJ_FUNC_NOT_FOUND</td>
<td>One of either azn_admin_get_object() or azn_admin_get_objectlist() was not found.</td>
</tr>
<tr>
<td>AZN_S_SVC_ADMIN_TASK_FUNC_NOT_FOUND</td>
<td>One of either azn_admin_perform_task() or azn_admin_get_tasklist() was not found.</td>
</tr>
</tbody>
</table>
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_INIT_FAILED</td>
<td>The Service Plug-in returns this error when it encounters an error during initialization.</td>
</tr>
<tr>
<td>AZN_S_SVC_SHUTDOWN_FAILED</td>
<td>The Service Plug-in returns this error when it encounters an error during shutdown.</td>
</tr>
<tr>
<td>AZN_S_SVC_AUTHORIZATION_FAILURE</td>
<td>The Service Plug-in returns this error when the calling application lacks authorization to invoke the Service.</td>
</tr>
</tbody>
</table>

In 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.

<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>
<tr>
<td>AZN_S_SVC_PAC_*</td>
<td>PAC Services plug-in major error codes.</td>
</tr>
</tbody>
</table>

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

```
extern unsigned int azn_ent_svc_err_prefix;
```

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

For example, minor error codes for Authorization API Entitlement Service plug-ins are defined in the interface file as follows:
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 __attribute__((visibility(visibility)))(azn_ent_svc_err_prefix | (MY_ENTSVC_ERR | 0x1))
```

**Shutdown**

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

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

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

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

For more information, see the reference page for "`azn_service_shutdown()`" on page 230.

**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.

/**
 * 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 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.
**********************************************************************/

/*
 * FUNCTION NAME
 * azn_service_initialize

*/

#endif /* C++ */

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

/*
 * FUNCTION NAME
 * azn_service_initialize

*/
**DESCRIPTION**
*init the entitlements service*

**ARGUMENTS**
*\[in\] \texttt{argc}\ The count of arguments to the service.*
*\[in\] \texttt{argv}\ The array of argument strings.*
*\[in\] \texttt{svc\_init}\ List of initialization attributes for the service.*
*\[out\] \texttt{svc\_info}\ attr list ptr for attributes returned by the service.*

**RETURN VALUE**
*AZN S\_COMPLETE on success, error code on failure*

```c
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));
    }

    /* */
    /* */
    /* 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_DECLS
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

Tivoli SecureWay Policy Director Authorization ADK Developer Reference
Implementing an Entitlements Service

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

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

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 Policy Director Authorization Service to obtain and return the entitlements.

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

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

The Authorization API uses attributes within an attribute list to represent individual entitlements. Each attribute within the list
consists of an attribute name and a multi-valued list of values for that attribute. Attributes can consist of values represented by data of type `azn_string_t`, 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 "Understanding the Authorization API Functions and Data Types" on page 9.

### Entitlements of type `azn_string_t`

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

```c
extern azn_string_t time_of_day_restriction;
```

The value of the data returned is specific to the implementation and so may be returned as multiple strings identifying times of the day in which the resource may be accessed.

For example:

“mon-fri: 9am-12pm”

“mon-fri: 1pm-5pm”

“sat: 9am-12pm”

These string values for `time_of_day_restriction` define valid access times.

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

### The Policy Director Protected Object Entitlements Service

Another example of an entitlement is the data returned by the Policy Director Protected Object Entitlement Service.
This service returns a list of objects within the protected object space for which the given user credential has the specified access privileges.

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

**Entitlements of type azn_buffer_t**

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

For more information on data type `azn_buffer_t`, see “Buffers” on page 12.

**Initialization, Configuration, and Shutdown**

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

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

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

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

The Policy Director Authorization Service also recognizes and registers Entitlements Service Plug-ins through arguments passed to the `init_data` parameter of the `azn_initialize()` function.
For complete configuration specifications, see "Initialization and Configuration of Service Plug-ins" on page 169.

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 169
- "Shutdown" on page 181

See also the reference pages for "azn_service_initialize()" on page 226 and "azn_service_shutdown()" on page 230.

Obtaining Entitlements for a Specified User

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

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

The Service Plug-in also takes the above parameters as input, and returns the requested entitlements in the attribute list entitlements. The Service Plug-in is not required to use the input parameters, but it is required to return valid data in the entitlements output parameter.

Entitlements Service Plug-ins share the address space of both the calling application and the Authorization API shared library. Service Plug-ins can assume that pointers passed in through the app_context parameter are valid within the address space of the Service Plug-in. Credential and attribute list handles are also valid to use within the Service Plug-in.

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_entitlements_get_entitlements()`

For information on error codes for `azn_service_initialize()` and `azn_service_shutdown()`, see “Using Error Codes” on page 176.

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

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>The attribute list handle for the entitlements is invalid.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 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()`.

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 180.
Implementing an Administration Service Plug-in

The Administration Service makes available to applications several administrative functions that operate on objects in the Policy Director 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 Policy Director 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 Policy Director Web Portal Manager graphical console, or a third-party application that has been written to use the Policy Director 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 Policy Director Administration API to the Policy Director management server. The Policy Director management server uses an internal interface to forward the call to the Administration Service interface. The Administration Service interface maps the incoming call to the appropriate Administration Service function and passes it to the Authorization Service Plug-in Dispatcher. The Dispatcher routes the request to the application that registered an Administration Service Plug-in to handle this function. The function call within the registered library is made and the results are returned to the Policy Director management server. The Policy Director management server then returns the results to the calling application.

Figure 4. The Administration Service Plug-in to the Authorization API
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 Policy Director Authorization Administration Service with the Policy Director Administration API. The Policy Director Administration API provides a series of programmatic interfaces that a calling application can use to send requests to the Policy Director management 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 198.

Most of the Policy Director 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 Policy Director Administration API, see the Tivoli SecureWay Policy Director Administration API Developer Reference.

## Configuring an Administration Service Plug-in

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 171.

Configuration of an Administration Service Plug-in is described in the following sections:

- “Using a Configuration File Entry” on page 196
Using a Configuration File Entry

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 an refer to either the name of a protected object space (hierarchy) or simply to a protected object. The Authorization Service does not process the characters after the ampersand &. It passes these characters directly to the Administration Service Plug-in.

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
```

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 Policy Director management server during the `azn_initialize()` call. This enables the Policy Director management 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. 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:

```
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 169.

   Ensure that protected object hierarchies that are specified as part of the service definition attribute are unique. See the discussion just above in "Using a Configuration File Entry" on page 196.

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:
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>azn_admin_get_object()</td>
<td>Retrieves a potential protected object. Tests for the existence of a protected object.</td>
</tr>
<tr>
<td>azn_admin_get_objectlist()</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>azn_admin_perform_task()</td>
<td>Instructs the service to perform an administration task. The service returns the results of the task.</td>
</tr>
<tr>
<td>azn_admin_get_tasklist()</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 Function</th>
<th>Administration API function</th>
<th>pdadmin command line interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_admin_get_object()</td>
<td>ivadmin_protobj_get2()</td>
<td>pdadmin object show &lt;object-name&gt;</td>
</tr>
<tr>
<td>azn_admin_get_objectlist()</td>
<td>ivadmin_protobj_list3()</td>
<td>pdadmin object list &lt;object-name&gt; pdadmin object listandshow &lt;object-name&gt;</td>
</tr>
</tbody>
</table>
The Policy Director management 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 Policy Director management server just forwards them to the corresponding Authorization API client application.

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 they return.

<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:

<table>
<thead>
<tr>
<th>Authorizations API Administration Function</th>
<th>Administration API function</th>
<th>pdadmin command line interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>azn_admin_perform_task()</td>
<td>ivadmin_server_performtask()</td>
<td>pdadmin server task &lt;server-name&gt; &lt;task&gt;</td>
</tr>
<tr>
<td>azn_admin_get_tasklist()</td>
<td>ivadmin_server_gettasklist()</td>
<td>pdadmin server listtasks &lt;server-name&gt;</td>
</tr>
</tbody>
</table>
Error Codes

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 <code>azn_admin_get_objectlist()</code> and the <code>azn_admin_get_object()</code> 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>
</tbody>
</table>
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.

### AZN_S_INVALID_LISTENING_PORT

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.

---

### Errors When Registering Administration Definitions

The Authorization API registers the Administration Service definitions with the Policy Director management server during the `azn_initialize()` call. The management 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 Policy Director management 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 Policy Director management server can occur within one of the following contexts:

- The Administration Services have already been registered with the management server.

  In this case, when the management 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 management 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 management server succeeds, the Administration API functions and pdadmin commands will succeed.

- The Authorization API has already registered the Administration Services with the management 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 management server starts. The Authorization API needs to be restarted. This causes the Authorization API to be re-initialized, which results in the management 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 management 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 management 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 
\texttt{azn\_util\_errorcode()} to return plug-in specific error return codes within the 16-bit minor error code portion of \texttt{azn\_status\_t}. The returned minor error code must be a valid Policy Director minor error code, in order for the Policy Director management server to generate a message for it. If the returned minor code is invalid, an error message of “unknown error” is returned to the \texttt{pdadmin} command or Administration API function that issued the request.

For more information on Authorization Service minor error codes, see \texttt{[Minor Error Codes] on page 186}.

\section*{Error Codes Specific to an Authorization Services Plug-in}

Administration Service plug-ins can also return implementation-specific return codes in the \texttt{outdata} output parameter for the \texttt{azn\_admin\_*} calls. For example, the functions can return status codes of type \texttt{unsigned long} by using the \texttt{azn\_attrlist\_add\_entry\_ulong()} API to add unsigned long values to a pre-defined attribute in the \texttt{outdata} attribute list.

The Administration Service functions can return results strings (which are displayed at the \texttt{pdadmin} CLI) as values for the \texttt{azn\_admin\_svc\_results} attribute of the \texttt{outdata} attribute list. These results strings are forwarded by the management server to the \texttt{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 \texttt{outdata} attribute list and directly forwarded to the caller of the Administration API.

\section*{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 \texttt{svrsslcfg} for the Authorization API application. Specify the appropriate input parameters.

For more information, see the reference page for \texttt{svrsslcfg} on page 159.
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 an Administration Service Plug-in” on page 195.
Implementing External Authorization Service Plug-in

This sections consists of the following topics:

- “Introducing the External Authorization Service”
- “Understanding the External Authorization Service” on page 206
- “Configuring an External Authorization Service Plug-in” on page 211
- “Initializing and Shutting Down External Authorization Service Plug-Ins” on page 214
- “Obtaining an Authorization Decision” on page 215
- “Error Codes” on page 216

Introducing the External Authorization Service

The External Authorization Service is a modular Authorization Service plug-in that allows system designers to supplement Policy Director 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 Policy Director. 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" on page 207
- "Policy Triggers" on page 208
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*. 

Figure 5. The External Authorization Service Architecture
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 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 entitlements 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 Policy Director, only a specific action or operation in an access control list (ACL) could serve as a policy trigger. For this version of Policy Director, 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 \( \text{Printer}[wx] \) invokes an EAS when both the \( w \) and \( x \) permission bits of the user-defined \( \text{Printer} \) 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 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.

```
pdadmin> pop show <pop-name> eas-trigger
pdadmin> pop modify <pop-name> set eas-trigger <trigger-string>
pdadmin> pop modify <pop-name> delete eas-trigger
pdadmin> pop modify <pop-name> delete 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 Policy Director 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>ivadmin_pop_attrget()</td>
<td>Get the EAS policy triggers for a POP</td>
</tr>
<tr>
<td>ivadmin_pop_attrput()</td>
<td>Set an EAS policy trigger for a POP</td>
</tr>
<tr>
<td>ivadmin_pop_attrdelpolicy()</td>
<td>Delete all the EAS policy triggers for a POP</td>
</tr>
<tr>
<td>ivadmin_pop_attrdelval()</td>
<td>Delete a specific EAS policy trigger for a POP</td>
</tr>
</tbody>
</table>

**Weightings**

The Policy Director 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 Policy Director 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 Policy Director 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 Policy Director 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 Policy Director 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 170.
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 213

**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 name.

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
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 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:

```c
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 212.

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 169
- "Shutdown" on page 181

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_initExtern_authzn_svc` 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 226 and `azn_service_shutdown()` on page 230.

### 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 those interfaces. For example:

```c
azn_status_t azn_decision_access_allowed_ext(
    azn_creds_h_t creds, /* input */
    azn_string_t protected_resource, /* input */
    azn_string_t operation, /* input */
    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 

```c
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

```c
azn_error_major()
```

and

```c
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

```c
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 protected_resource string supplied is invalid.</td>
</tr>
<tr>
<td>AZN_S_INVALID_OPERATION</td>
<td>The operation 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 size_t value is required.</td>
</tr>
<tr>
<td>AZN_S_UNKNOWN_EAS_SVC_PARAMETER</td>
<td>A parameter other than -weight was specified in an External Authorization Service definition.</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 specific minor error code should be returned in the status code for the 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 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 locating or loading an authorization service plug-in.</td>
</tr>
<tr>
<td>AZN_S_SVC_INITIALIZE_NOT_FOUND</td>
<td>Returned by the service dispatcher when an error occurs either locating or loading a service interface within a particular plug-in. For example, this is returned by the dispatcher if the azn_service_initialize() interface was not found in the loaded plug-in.</td>
</tr>
<tr>
<td>AZN_S_SVC_SHUTDOWN_NOT_FOUND</td>
<td></td>
</tr>
<tr>
<td>AZN_S_SVC_EAS_FUNC_NOT_FOUND</td>
<td></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 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 Policy Director 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)
```

### Supplied Implementations for Service Plug-ins

The 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 170.

The following sections describe the supplied service implementations:
Policy Director Entitlements Services

Policy Director Version 3.7 provides a default entitlement service specific to the Policy Director environment. This entitlement service is called the Policy Director Protected Objects Entitlements Service, and is provided in the Authorization API distribution package.

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

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

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

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

- Entitlements data is returned as a multi-valued attribute list of protected objects for each search tree root node passed in to the
azn_entitlement_get_entitlements() call. The list contains only objects underneath the specified root node.

<table>
<thead>
<tr>
<th>Service Plug-in Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Service ID</td>
</tr>
<tr>
<td>Plug-in Short Name</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>app_context</td>
</tr>
<tr>
<td>entitlements</td>
</tr>
<tr>
<td>Description</td>
</tr>
</tbody>
</table>

**Credentials Modification Service**

The 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>Category</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Service ID</td>
</tr>
<tr>
<td>Plug-in Short Name</td>
</tr>
</tbody>
</table>
The service described in the table below is not built-in to the Authorization API.

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>This service will replace the attribute list in credential &lt;creds&gt; with the attribute list &lt;list&gt;. Read-only attributes, such as group UUIDs, cannot be changed by this service. The original credential is left unchanged and a new credential containing &lt;list&gt; is returned in &lt;outcred&gt;.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>creds</td>
<td>The credentials to modify</td>
</tr>
<tr>
<td>list</td>
<td>The attribute list to replace.</td>
</tr>
<tr>
<td>outcred</td>
<td>The credential resulting from the operation.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Credentials Group Membership Modification Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service ID</td>
<td>User specified in the service definition.</td>
</tr>
<tr>
<td>Plug-in Short Name</td>
<td>azn_mod_rad</td>
</tr>
<tr>
<td>Parameters</td>
<td>creds The credentials to modify</td>
</tr>
<tr>
<td></td>
<td>mod_info AZN_MOD_RAD_GROUP_NAMES Contains the names of the groups to which the resulting credential &lt;creds&gt; will effectively be added.</td>
</tr>
<tr>
<td></td>
<td>newcreds A new credential based on &lt;creds&gt; and built to include the groups in mod_info.</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>Gives a credential membership in a specified set of groups. The new credential is returned in newcreds. The original credential is unaffected and the results are localized to authorization checks made with the returned credential &lt;newcreds&gt;. The actual user registry and the user’s entry within the registry are not affected by this modification service.</td>
</tr>
</tbody>
</table>

Privilege Attribute Certificate Service

This service is built in to the Authorization API.

Service Plug-in Summary

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Policy Director 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>
<tr>
<td>Parameters:</td>
<td>azn_creds_get_pac()</td>
</tr>
<tr>
<td>cred</td>
<td>Input parameter. The credentials to be encoded.</td>
</tr>
<tr>
<td>pac</td>
<td>An azn_buffer_t that will contain the encoded credentials PAC for &lt;creds&gt; on completion.</td>
</tr>
<tr>
<td>Parameters:</td>
<td>azn_pac_get_creds()</td>
</tr>
<tr>
<td>pac</td>
<td>An encoded PAC to be decoded into credentials.</td>
</tr>
<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 &lt;pac&gt;.</td>
</tr>
<tr>
<td>Description</td>
<td>Encodes and decodes a Policy Director credential to or from a format that is transmissible in a text only environment. The format is a combination of ASN1 and MIME encoding.</td>
</tr>
</tbody>
</table>
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>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>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>Description</td>
</tr>
</tbody>
</table>
This chapter contains the following reference pages:

- “azn_service_initialize()” on page 226
- “azn_service_shutdown()” on page 230
- “azn_admin_get_object()” on page 232
- “azn_admin_get_objectlist()” on page 235
- “azn_admin_perform_task()” on page 238
- “azn_admin_get_tasklist()” on page 242
**azn_service_initialize()**

Initialize the specified Authorization Service Plug-in.

**Syntax**

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

**Description**

Use this initialization function to initialize a Service Plug-in. The Service Dispatcher calls this function prior to calling the 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:

```
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:

```c
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`:

```c
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 Policy Director `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.

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

**Return Values**

If successful, the function will return `AZN_S_COMPLETE`.

If the returned status code is not equal to `AZN_S_COMPLETE`, the major error codes will be derived from the returned status code with `azn_error_major()`.
Each of the following error messages also returns an implementation-specific minor error code. Use `azn_error_minor()` to derive specific minor error codes from the returned status code.

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

- **AZN_S_SVC_NOT_FOUND**
  Returned by the Service Dispatcher when an error occurs either while locating or loading an Authorization API service plug-in.

- **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, const char **argv,
    azn_attrlist_h_t svc_init, azn_attrlist_h_t *svc_info);
```

**Description**

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

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

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

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

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

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

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

Information that may be requested of the service during shutdown is not currently defined but may be defined by the interface in future.

The prototype for this function is included in the file `azn_svc_protos.h`, in the Policy Director `include` directory.

**Parameters**

- **Input**
The number of arguments in the `argv` array.

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

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.

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.

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()`.

---

Tivoli SecureWay Policy Director Authorization ADK Developer Reference 231
azn_admin_get_object()

Retrieves a potential protected object. Tests for the existence of a protected object.

Syntax

```
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 Policy Director management 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 Policy Director 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 Policy Director protected object namespace. The credentials of the identity requesting the object are forwarded by the Policy Director management 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 Policy Director Management 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 Policy Director management 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 Policy Director 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 objects in the Policy Director protected object namespace. The credentials of the identity requesting the objects are forwarded by the Policy Director management 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 Policy Director Management 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_SVCIINFO_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 Policy Director management 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 Policy Director 3.8.

`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 Policy Director Management 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.

À 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 Policy Director management server, and for the communication between the Policy Director management 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_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 Policy Director management 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 Policy Director 3.8.

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.

commands[]
A NULL terminated list of administration commands supported by this service implementation.

Commands should conform to pdadmin command line interface syntax rules.

The command list may be state sensitive. Only commands that can be successful in the current state can be returned. Thus the returned list does not include all possible commands.
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 Policy Director Management 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().
Authorization Java Classes

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

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

- “Using Java™ 2 Security with Policy Director” on page 248
- “Java Authentication and Authorization Services (JAAS) Model” on page 249
- “Software Requirements for Installation” on page 254
- “Java Classes Overview” on page 256
- “Configuring a Java Application into Policy Director” on page 260
- “Configuring Java Authentication and Authorization Service” on page 263
- “Developing a Resource Manager” on page 265
- “Making Authorization Decisions Outside of Java 2” on page 266
Using Java™ 2 Security with Policy Director

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

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

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

The azn_decision_access_allowed() function requires the following information:

- Authentication information
- Resource name
- Access mode

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

Policy Director functions as a back-end for normal Java 2 permission checks by providing:
A custom JAAS LoginModule that manufactures authentication credentials.

A custom permission class that knows how to locate and call Policy Director.

Java Authentication and Authorization Services (JAAS) Model

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

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

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

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

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

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

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

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

- “Authenticating Users and Obtaining Credentials” on page 251
“Authorizing Access Requests” on page 252
Authenticating Users and Obtaining Credentials

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

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

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

**Authenticating with a Username and Password**

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

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

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

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

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

**Retrieving Credentials without Authenticating**

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

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

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

This configuration takes the form of an optional keyword, `nameOnly=true`.

If `nameOnly` is omitted or specified to be `false`, both the user name and the password are required.

Authorizing Access Requests

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

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

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

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

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

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

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

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

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

2. If the Subject contains a Principal of type
   com.tivoli.mts.PDPrincipal, then the appropriate credentials are
secured for the call to Policy Director.

The example below illustrates one way a resource manager
(WebServer, EJB container, etc.) could place the Subject on the
current thread of execution.

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

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

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

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

SecurityManager.checkPermission(perm);
Software Requirements for Installation

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

- “Policy Director Requirements” on page 254
- “JAAS and JRE Requirements” on page 254
- “Java Archive Requirements” on page 255

Policy Director Requirements

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

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

JAAS and JRE Requirements

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

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

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

- Java 1.3 with the JAAS standard extension
- An IBM JVM (Java 1.2.2 or higher) that includes the JAAS
The security technologies in this Policy Director package that are outside of the JAAS standard require Java 2. They have been tested on both Java 1.2.2 and Java 1.3.

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

This JAR file must be copied to the JRE’s extension directory. Typically this is:
<java.home>/jre/lib/ext

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

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

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

Java Classes Overview

This section discusses the Policy Director Authorization Java Classes:

- `com.tivoli.mts.PDLoginModule`
- `com.tivoli.mts.PDPrincipal` on page 257
- `com.tivoli.mts.PDPermission` on page 257
- `com.tivoli.mts.SvrSslCfg` on page 260
- `com.tivoli.mts.PDAttrs` on page 258
- `com.tivoli.mts.PDAttrValue` on page 259
- `com.tivoli.mts.PDAttrValues` on page 259
- `com.tivoli.mts.PDStatics` on page 260

See “Authorization Java Classes Reference” on page 267 for detailed information about each of these classes.

`com.tivoli.mts.PDLoginModule`

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

```java
public class PDLoginModule implements javax.security.auth.spi.LoginModule{

    public PDLoginModule()
    public login()
    public logout()
    public abort()
    public commit()
    public initialize(javax.security.auth.Subject subject,
                     javax.security.auth.callback.CallbackHandler
                     callbackHandler,
                     java.util.Map sharedState,
                     java.util.Map options)
}
```
com.tivoli.mts.PDPrincipal

This class represents the identity of a Policy Director user.

```java
public class PDPrincipal implements Principal, PrincipalComparator, java.io.Serializable {

    public PDPrincipal(String name)
    public PDPrincipal(String name, char[] password)
    public PDPrincipal addGroupMemberships(
            java.lang.String service ID,
            java.lang.String[] groups)

    public String getName()
    public String toString()
    public boolean equals(Object o)
    public int hashCode()
    public boolean implies(javax.security.auth.Subject subject)
    public boolean implies(PDPermission perm)
    public boolean implies(PDPermission perm,
            PDAttrs attrsIn,
            PDAttrs, attrsOut)
}
```

For more information, see the reference page for Class PDPrincipal on page 298.

com.tivoli.mts.PDPermission

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

```java
public class PDPermission {

    public PDPermission(java.lang.String rname,
            java.lang.String actions)
    public boolean implies(java.security.Permission p)
    public boolean implies(PDPrincipal princ)
    public boolean implies(PDPrincipal princ,
            PDAttrs inputList,
            PDAttrs, outputList)
```
public boolean equals(Object obj)
public String getActions()
public int hashCode()
}

For more information, see the reference page for “Class PDPermission” on page 292.

**com.tivoli.mts.PDAattrs**

This class represents an attribute list. Attribute lists are data types used by the Policy Director C API. Each attribute consists of entries that have a *name* and one or more *values*. The names are Strings, and the values can be either Strings or byte arrays.

public class PDAattrs extends java.lang.Object
implements java.lang.Cloneable, java.io.Serializable {

    public PDAattrs()
    public PDAattrs(int initialCapacity)
    public PDAattrs(PDattrs int initialCapacity, float loadFactor)
    public PDAattrs(PDattrs that)
    public add(java.lang.String name, PDAttrValues vals)
    public java.util.Collection add(java.lang.String name,
                                    java.lang.String value)
    public java.util.Collection add(java.lang.String name,
                                    byte[] value)
    public void addAll(PD attrs attr)
    public void clear()
    public boolean delete(java.lang.String key)
    public java.lang.Object clone()
    public java.util.Set entrySet()
    public boolean equals(java.lang.Object obj)
    public PDAttrValues get(java.lang.String key)
    public int getQop()
    public int hashCode()
    public java.util.Set keySet()
    public void setQop(int qop)
    public int size()
    public java.lang.String toString()
}

For more information, see the reference page “Class PDAattrs” on page 268.
com.tivoli.mts.PDAttrValue
This class represents the value of a Policy Director attribute. A value may be either a String or a byte array.

```java
public class PDAttrValue extends java.lang.Object implements java.lang.Cloneable, java.io.Serializable{
    public PDAttrValue(byte[] bytes)
    public PDAttrValue(java.lang.String string)
    public boolean equals(java.lang.Object obj)
    public java.lang.Object getValue()
    public int hashCode()
    public java.lang.Object clone()
    public java.lang.String toString()
}
```

For more information, see the reference page for "Class PDAttrValue" on page 277.

com.tivoli.mts.PDAttrValues
This class represents a collection of values for a particular PDAtrr. This implementation is a Set, so duplicates are not allowed in a particular PDAttrValues object.

```java
public class PDAttrValues extends java.util.HashSet implements java.lang.Cloneable, java.io.Serializable{
    public PDAttrValues()
    public PDAttrValues(int initialCapacity)
    public PDAttrValues(int initialCapacity, float loadFactor)
    public PDAttrValues(java.util.Collection c)
    public boolean add(PDAttrValue value)
    public boolean add(java.lang.Object obj)
    public boolean addAll(java.util.Collection c)
    public java.lang.Object clone()
    public boolean equals(java.lang.Object obj)
    public int hashCode()
    public java.lang.String toString()
}
```

For more information, see the reference page for "Class PDAttrValues" on page 281.
com.tivoli.mts.PDStatics

This is a class for various constants used in PDPermission and associated classes.

```java
public class PDStatics extends java.lang.Object {

    public static final java.lang.String AZN_MOD_SVC_RAD_2AB
    public static final java.lang.String AZN_MOD_RAD_GROUP_NAMES
    public static final java.lang.String AZN_ENT_SVC_PD_POBJ
    public static final java.lang.String AZN_ENT_SVC_PD_POBJ_PATH
    public static final java.lang.String AZN_ENT_SVC_PD_POBJ_REQD_OPS
    public static final java.lang.String AZN_ENT_SVC_PD_POBJ_MATCHES
    public static final int QOP_NONE
    public static final int QOP_INTEGRITY
    public static final int QOP_PRIVACY
    public static final int AZN_VALTYPE_BUFFER
    public static final int AZN_VALTYPE_UTF8STRING
}
```

For more information, see the reference page for “Class PDStatics” on page 304.

com.tivoli.mts.SvrSslCfg

This class configures SSL communication with remote Policy Director servers.

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

For more information, see the reference page for “Class SvrSslCfg” on page 308.

Configuring a Java Application into Policy Director

To configure a Java application to run within the Policy Director secure domain, you must establish an identify for the application, and configure the location of the Policy Director Management Server and Policy Director Authorization Server.
Policy Director uses a self-generated and self-signed certificate to authenticate its Secure Sockets Layer (SSL) communications. Therefore, the Java SSL support contained in the Policy Director Authorization Java Classes must determine which certificate Policy Director is using. This is accomplished by using the SvrSslCfg class.

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

- “Establishing an Identity with Proper Credentials” on page 261
- “Identifying the Policy Director Server Machines” on page 262
- “Using the SvrSslCfg Class” on page 262

### Establishing an Identity with Proper Credentials

Complete the following instructions:

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

2. Use the `pdadmin` command to create an account for this machine:
   
   ```
   pdadmin> user create PdPermission9999 \
   cn=PdPermission/scalawag,o=ibm,c=us PdPermission/scalawag \
   PdPermission secretpw9999
   ```
   
   The above command illustrates creation of a user on a machine named `scalawag`. The machine `scalawag` will use Policy Director for permission checks.

   An account should be set up for each machine that will be using the PDPermission class. The name must be unique, since a certificate is created to represent this account, and the certificate will be registered with Policy Director.

3. Use `pdadmin` to make the user’s account valid.
   
   ```
   pdadmin> user modify PdPermission9999 account-valid yes
   ```
4. Use `pdadmin` to add the new user to the group necessary for performing remote permissions checks.

   `pdadmin> group modify remote-acl-users add PdPermission9999`

**Identifying the Policy Director Server Machines**

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

For example:

- `flyer.austin.lab.tivoli.com`
  
  For this example, the machine running the Policy Director Management Server

- `timon.austin.lab.tivoli.com`
  
  For this example, the machine running the Policy Director Authorization Server.

**Using the SvrSslCfg Class**

Based on the example shown above in "Establishing an Identity with Proper Credentials" on page 261 and "Identifying the Policy Director Server Machines", use the SvrSslCfg Class as follows:

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

**SvrSslCfg Usage Syntax**

The use of SvrSslCfg can be summarized as follows:

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

The above command includes the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userDN</td>
<td>Distinguished name of a Policy Director user. This account must already exist</td>
</tr>
<tr>
<td>pw</td>
<td>The password for sec_master</td>
</tr>
<tr>
<td>mgrd</td>
<td>The machine running the Policy Director Management Server.</td>
</tr>
</tbody>
</table>
The result of this configuration is that a `PdPerm.properties` file ends up in the `java.home` directory and a `pdperm.ks` file ends up in `java.home/jre/lib/security`.

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

The `SvrSslCfg` class supports additional parameters. For more information, see the reference page for "Class `SvrSslCfg`" on page 308.

### Configuring Java Authentication and Authorization Service

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

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

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

Complete the instructions in the following sections:

- “Creating a Login Configuration File” on page 264
- “Specify the Login File Location” on page 264
Creating a Login Configuration File

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

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

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

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

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

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

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

Note that the last stanza allows applications that use `pd-nopass` in their LoginContext constructor to simply supply usernames but not passwords. For more information, see the reference page for “Class PDLoginModule” on page 287 or review the javadoc HTML page for com.tivoli.mts.PDLoginModule.

Specify the Login File Location

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

- Point to the login configuration file from the file `java.home\jre\lib\security\java.security`

Following is a sample entry from the java.security file:

```bash
login.config.url.1=file:d:/Java/j122ibm/jre/lib/security/config.pd
```

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

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

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

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

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

5. Authorization Java Classes
Making Authorization Decisions Outside of Java 2

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

The usage is simple.

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

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

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

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

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

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

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

- “Class PDAttrs” on page 268
- “Class PDAttrValue” on page 277
- “Class PDAttrValues” on page 281
- “Class PDLoginModule” on page 287
- “Class PDPermission” on page 293
- “Class PDPrincipal” on page 298
- “Class PDStatics” on page 304
- “Class SvrSslCfg” on page 308
Class PDAttrs

    com.tivoli.mts.PDAttrs
    public class PDAttrs
    extends java.lang.Object
    implements java.lang.Cloneable, java.io.Serializable

Description
This class represents an attribute list. An attribute list is a data type used in the Policy Director Authorization C API. The individual attributes have String names. Values for an attribute can be either Strings or byte arrays. PDAttrs allows an attribute to have more than one value. When you add a value to a name that already has been added to a PDAttrs, the name is assigned multiple values.

Constructor Summary

- PDAttrs()
  Constructs a new, empty map with default capacity and load factor, which is 0.75.

- PDAttrs(int initialCapacity)
  Constructs a new, empty map with the specified initial capacity and default load factor, which is 0.75.

- PDAttrs(int initialCapacity, float loadFactor)
  Constructs a new, empty map with the specified initial capacity and the specified load factor.

- PDAttrs(PDAttrs that)
  Constructs a new PDAttrs containing the elements in the specified PDAttrs.

Method Summary

- java.util.Collection add(java.lang.String name, PDArrtValues vals)
  Associates the specified values with the specified name in this PDAttrs.

- java.util.Collection add(java.lang.String name, java.lang.String value)
Associates the specified value with the specified name in this PDAttrs.

- **java.util.Collection add(java.lang.String name, byte[] value)**
  Associates the specified value with the specified name in this PDAttrs.

- **void addAll(PDAttrs attr)**
  Adds all of the elements in the specified PDAttrs to this PDAttrs.

- **void clear()**
  Clear the current PDAttrs.

- **boolean delete(java.lang.String key)**
  Remove the named attribute from the PDAttrs.

- **java.lang.Object clone()**
  Clone the current PDAttrs.

- **java.util.Set entrySet()**
  Return a Set view of the entries in the PDAttrs.

- **boolean equals(java.lang.Object obj)**
  Indicates whether some other Object is equal to this one.

- **PDAttrValue get(java.lang.String key)**
  Returns the value(s) to which this PDAttrs maps the specified key.

- **int getQoP()**
  Returns the current value of QoP.

- **int hashCode()**
  Returns a hashcode for the current object.

- **java.util.Set keySet()**
  Returns a set view of the keys contained in this PDAttrs.

- **void setQop(int Qop)**
  Sets the current value of QoP.
- **int size()**
  Returns the number of key-values mappings in the current PDAttrs.

- **java.lang.String toString()**
  Return a string version of this object.

- Methods inherited from class java.lang.Object
  - finalize
  - getClass
  - notify
  - notifyAll
  - wait
  - wait
  - wait

### Constructor Detail

#### PDAttrs

```java
public PDAttrs()
```

Constructs a new, empty map with default capacity and load factor, which is 0.75.

#### PDAttrs

```java
public PDAttrs(int initialCapacity)
```

Constructs a new, empty map with the specified initial capacity and default load factor, which is 0.75.

#### PDAttrs

```java
public PDAttrs(PDAttrs int initialCapacity, float loadFactor)
```

Constructs a new, empty map with the specified initial capacity and the specified load factor.
PDAttrs

public PDAttrs(PDAttrs that)
throws java.lang.NullPointerException,
       java.lang.ClassCastException

Constructs a new PDAttrs containing the elements in the specified PDAttrs.

Parameters:

that
   The PDAttrs whose elements are to be placed into this PDAttrs

java.lang.NullPointerException
   No PDAttrs was passed in

java.lang.ClassCastException
   This is thrown when one or more of the elements in the PDAttrs to be added are not PDAttr objects.

Method Detail

add

public java.util.Collection add(java.lang.String name,
                                      PDAttrValues vals)

Associates the specified values with the specified name in this PDAttrs. If this PDAttrs previously contained values for this name, then the input values are added to the values already known for that name.

Parameters

name
   The name of the attribute

vals
   The new values to be associated with the specified name.

Returns

Returns the previous values associated with the name, or null if no values were previously known for that attribute.

Throws
java.lang.NullPointerException
This error is thrown on parameter errors.

add

public java.util.Collection add(java.lang.String name,
java.lang.String value)

Associates the specified values with the specified name in this
PDAttrs. If this PDAttrs previously contained values for this name,
then the input values are added to the values already known for that
name.

Parameters

  name
  The name of the attribute

  value
  The new value to be associated with the specified name.

Returns

Returns the previous values associated with the name, or null if
no values were previously known for that attribute.

Throws

java.lang.NullPointerException
This error is thrown on parameter errors.

add

public java.util.Collection add(java.lang.String name,
byte[] value)

Associates the specified values with the specified name in this
PDAttrs. If this PDAttrs previously contained values for this name,
then the input value is added to the values already known for that
name. Note that if the value is mutable, it is cloned before being
entered in the PDAttrs.

Parameters

  name
  The name of the attribute.
value
The new value to be associated with the specified name.

Returns
Returns the previous values associated with the name, or null if no values were previously known for that attribute.

Throws
java.lang.NullPointerException
This error is thrown on parameter errors.

addAll
public void addAll (PDAttrs attrs)
throws java.lang.NullPointerException
java.lang.ClassCastException

Adds all of the elements in the specified PDAttrs to this PDAttrs. The behavior of this operation is undefined if the specified PDAttrs is modified while the operation is in progress. (This implies that the behavior of this call is undefined if the specified PDAttrs is this PDAttrs, and this PDAttrs is not empty.) This implementation iterates over the specified PDAttrs, and adds to this map, in turn, each PDAttr returned by the iterator.

Throws:
java.lang.NullPointerException
This exception is thrown if no PDAttrs are passed.

clear
public void clear()

Clears the current PDAttrs. Removes all mappings in the current PDAttrs. Resets QoP to zero, and resets the version number to the current supported level.

delete
public boolean delete(java.lang.String key)

Removes the named attribute from the PDAttrs.

Parameters
key
The name of the attribute to be removed.

Returns
true
Returns true when the attribute was in the PDAttrs.

close
public java.lang.Object clone()
Clone the current PDAttrs. This is implemented as a deep copy

Overrides
clone
Overrides clone in class java.lang.Object

entrySet
public java.util.Set entrySet()
Return a Set view of the entries in the PDAttrs. Modifications to the Set result in modifications to the underlying PDAttrs.

equals
public boolean equals(java.lang.Object obj)
Indicates whether some other Object is equal to this one. In particular, this implementation checks that the QoP and versions are the same, and then defers to the HashMap.equals check, which will devolve into a containsAll check on the current set of PDAtrr.

Overrides
equals
Override equals in class java.lang.Object

Parameters

obj
The object to be compared to this one.

Returns
true
Returns true when the PDAttrs are identical.

false
Returns false when the PDAttrs are not identical.

get
public PDAttrValues get(java.lang.String key)

Returns the value(s) to which this PDAttrs maps the specified key.
Returns null if the PDAttrs contains no mapping for the specified
key.

Parameters
key
The key whose associated value(s) are to be returned.

Returns
The value(s) to which this PDAttrs maps the specified key.

getQoP
public int getQop()

Returns the current value of QoP.

Returns
The current value of QoP.

hashCode
public int hashCode()

Returns a hashcode for the current object.

Overrides
hashcode in the class java.lang.Object

Returns
A hashcode for the current object.

keySet
public java.util.Set keySet()
Returns a set view of the keys contained in this PDAttrs. The set is backed by the map, so changes to the PDAttrs are reflected in the set, and vice-versa. The set supports element removal, which removes the corresponding mapping from this PDAttrs, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.

Returns

A set view of the keys contained in this PDAttrs.

**setQop**

```java
public void setQop(int qop)
```

Sets the current value of QoP (quality of protection).

Throws

*java.lang.IllegalArgumentException*

This is thrown when the QoP is unknown.

**size**

```java
public int size()
```

Returns the number of key-values mappings in the current PDAttrs. Note that is equivalent to the number of keys in the PDAttrs, not the number of values, as a multivalue key counts as only one key.

Returns

Returns the number of key-values mappings in the current PDAttrs

**toString**

```java
public java.lang.String toString()
```

Returns a string version of this object.

Overrides

toString in class java.lang.Object

Returns

A String representing this object.
Class PDAttrValue

```
com.tivoli.mts
public class PDAttrValue
extends java.lang.Object
implements java.lang.Cloneable, java.io.Serializable
```

**Description**

Represents the value of a Policy Director attribute. A value may be either a String or a byte array.

**Constructor Summary**

- `PDAttrValue(byte[] bytes)`
  - Construct a PDAttrValue from a clone of the input byte array.

- `PDAttrValue(java.lang.String string)`
  - Construct a PDAttrValue from a String.

**Method Summary**

- `java.lang.object clone()`
  - Return a clone of the object.

- `boolean equals(java.lang.Object obj)`
  - Indicates whether some other object is equal to this one.

- `java.lang.Object getValue()`
  - Returns the value of the current object, which can then be examined to see what it is.

- `int hashCode()`
  - Returns a hashcode for the current object.

- `java.lang.string toString()`
  - Returns a string version of the object.

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

PDAttrValue
public PDAttrValue(byte[] bytes)
throws java.lang.NullPointerException

Construct a PDAttrValue from a clone of the input byte array.

Parameters

bytes
  The input byte array

PDAttrValue
public PDAttrValue(java.lang.String string)
throws java.lang.NullPointerException

Construct a PDAttrValue from String.

Parameters

string
  The input String.

Method Detail

equals
public boolean equals(java.lang.Object obj)

Indicates whether some other Object is equal to this one. In particular, this implementation checks that the types are the same (byte array or String), and then checks the type appropriately.

Overrides

equals
  Overrides equals in class java.lang.Object
Parameters

**obj**
The object to be compared to this one.

Returns

**true**
Returns true when the PDAttrValues are identical.

**false**
Returns false when the PDAttrValues are not identical.

**getValue**

```java
public java.lang.Object getValue()
```

Returns the value of the current object, which can then be examined to see what it is.

**Returns**
The value of this PDAttrValue.

**hashCode**

```java
public int hashcode()
```

Returns a hashcode for the current object.

**Overrides**

hashcode in the class `java.lang.Object`

**Returns**
A hashcode for the current object.

**clone**

```java
public java.lang.Object clone()
```

Return a clone of the object. This is implemented as a deep copy.

**Overrides**

clone

**Overrides clone in class java.lang.Object**

**Returns**
A clone of the object.
**toString**

```java
public java.lang.String toString()
```

Returns a string version of this object.

**Overrides**

toString in class java.lang.Object

**Returns**

A String representing this object.
Class PDAttrValues

    com.tivoli.mts
public class PDAttrValues
extends java.util.HashSet
implements java.lang.Cloneable, java.io.Serializable

All Implemented Interfaces:
    java.lang.Cloneable, java.util.Collection, java.io.Serializable, java.util.Set

Description
This class represents a collection of values for a particular PDAttr. This particular implementation is a Set, so duplicates are not allowed a particular PDAttrValues object.

Constructor Summary

- **PDAttrValues()**
  Constructs a new, empty set. The backing HashMap instance has default capacity and load factor of 0.75.

- **PDAttrValues(java.util.Collection c)**
  Constructs a new Set containing the elements in the specified collection.

- **PDAttrValues(int initialCapacity)**
  Constructs a new, empty set. The backing HashMap instance has the specified initial capacity and the default load factor of 0.75.

- **PDAttrValues(int initialCapacity, float loadFactor)**
  Constructs a new, empty set. The backing HashMap instance has the specified initial capacity and the specified load factor.

Method Summary

- **boolean add(java.lang.Object obj)**
  Overrides the generic method in HashSet that allows objects of any type to be added to a Set.

- **boolean add(PDAttrValue value)**
Adds the input PDAttrValue to this PDAttrValues.

- boolean `addAll(java.util.Collection c)`
  Adds all of the elements in the specified collection to the collection.

- java.lang.Object `clone()`
  Return a clone of this object.

- boolean `equals(java.lang.Object obj)`
  Indicates whether some other Object is equal to this one.

- int `hashCode()`
  Returns a hashcode for the current object.

- java.lang.String `toString()`
  Return a string version of this object.

### Constructor Detail

**PDAttrValues**

```java
public PDAttrValues()
```

Constructs a new, empty set. The backing HashMap instance has default capacity and load factor of 0.75.
PDAttrValues
   public PDAttrValues(int initialCapacity)

   Constructs a new, empty set. The backing HashMap instance has the
   specified initial capacity and the default load factor of 0.75.

PDAttrValues
   public PDAttrValues(int initialCapacity,
                        float loadFactor)

   Constructs a new, empty set. The backing HashMap instance has the
   specified initial capacity and the specified load factor.

PDAttrValues
   public PDAttrValues(java.util.Collection c)
                      throws java.lang.IllegalArgumentException)

   Constructs a new Set containing the elements in the specified
   collection.

   Parameters
      c   The collection whose elements are to be placed in this set.

   Throws
       java.lang.NullPointerException
           This is thrown when no Collection is passed in.

       java.lang.ClassCastException
           This is thrown if one or more of the elements in the
           Collection to be added are not PDAttrValue objects.

Method Detail

add
   public boolean add(PDAttrValue value)
                      throws java.lang.NullPointerException

   Adds the input PDAttrValue to this PDAttrValues. If a PDAttrValue
   with the same value is already in the Set, the input PDAttrValue will
   replace it.
Parameters:

value
The value to be added.

Returns

true
Returns true if this Collection changed as a result of this call.

Throws

java.lang.NullPointerException
This is thrown when no PDAttrValue is supplied.

add

public boolean add(java.lang.Object obj)
throws java.lang.IllegalArgumentException

Overrides the generic method in HashSet that allows objects of any type to be added to a Set. This method will throw an IllegalArgumentException if called with an object other than a PDAttrValue.

Overrides

add in class java.util.HashSet

Throws

java.lang.IllegalArgumentException
This is thrown if the method is called with an object other than a PDAttrValue.

addAll

public boolean addAll(java.util.Collection c)
throws java.lang.NullPointerException,
java.lang.ClassCastException

Adds all of the elements in the specified collection to this collection. The behavior of this operation is undefined if the specified collection is modified while the operation is in progress. (This implies that the behavior of this call is undefined if the specified collection is this collection, and this collection is nonempty.) This implementation
iterates over the specified collection, and adds each PDAttrValue returned by the iterator to this collection, in turn. If any objects are returned by the iterator that are not PDAttrValue, a ClassCastException is thrown.

Overrides
addAll in class java.util.AbstractCollection.

Returns:
true
Returns true if this Collection changed as a result of the call.

Throws:
java.lang.NullPointerException
This is thrown if no Collection is passed.

java.lang.ClassCastException
This is thrown if the input Collection contains an object other than a PDAttrValue.

close
public java.lang.Object clone()

Return a clone of the object. This is implemented as a deep copy.

Overrides
clone
clone
Overrides clone in class java.lang.HashSet.

Returns
A clone of this object.

equals
public boolean equals(java.lang.Object obj)

Indicates whether some other Object is equal to this one. In particular, this implementation checks that the object is a PDAttrValues, and then defers to the HashSet.equals check, which will devolve into a containsAll check on the current set of PDAttrValues.
Overrides

equals

Overrides equals in class java.util.AbstractSet.

Parameters

obj

The object to be compared to this one.

Returns

true

Returns true when the PDAtrValues are identical.

false

Returns false when the PDAtrValues are not identical.

hashCode

public int hashCode()

Returns a hashcode for the current object.

Overrides

hashcode in the class java.lang.AbstractSet.

Returns

A hashcode for the current object.

toString

public java.lang.String toString()

Returns a string version of this object.

Overrides

toString in class java.lang.AbstractCollection.

Returns

A String representing this object.
Class PDLoginModule

```
com.tivoli.mts.PDLoginModule
public class PDLoginModule extends java.lang.Object
    implements javax.security.auth.spi.LoginModule
```

**Description**

This PDLoginModule represents the JAAS authentication mechanism for Policy Director.

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

The objectives of the PDLoginModule class are to:

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

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

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

**Constructor Summary**

- `PDLoginModule()`

**Method Summary**

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

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

<table>
<thead>
<tr>
<th>Methods inherited from class java.lang.Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone equals finalize getClass hashcode notify notifyAll toString wait wait wait</td>
</tr>
</tbody>
</table>

**Constructor Detail**

**PDLoginModule**

```java
public PDLoginModule()
```

**Method Detail**

**initialize**

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

Initialize this LoginModule.

**Specified by:**

`initialize` in interface `javax.security.auth.spi.LoginModule`

**Parameters:**

- `subject`
The Subject to be authenticated.
callbackHandler
A CallbackHandler for communicating with the end user. For example, prompting for user names and passwords.

sharedState
Shared LoginModule state.

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

login
public boolean login(
    throws
    javax.security.auth.login.AccountExpiredException,
    javax.security.auth.login.CredentialExpiredException,
    javax.security.auth.login.FailedLoginException,
    javax.security.auth.login.LoginException
)

Authenticate the user (first phase).

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

Specified by:
login in interface javax.security.auth.spi>LoginModule

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

Throws
javax.security.auth.login.AccountExpiredException
Thrown when the account has been disabled.

javax.security.auth.login.CredentialExpiredException
Thrown when the password is rejected.

javax.security.auth.login.LoginException
Thrown when environmental or setup errors occur when trying to login.

javax.security.auth.login.FailedLoginException
Thrown when any other difficulties occur when trying to login.
commit

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

Commit the authentication (second phase).

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

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

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

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

Throws:
    javax.security.auth.login.LoginException

When the commit() fails.

abort

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

Abort the authentication (second phase).

This method is called when the LoginContext’s overall authentication failed. (The relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules did not succeed).

This method cleans up any state that was originally saved as part of the authentication attempt from the login and commit methods.
Specified by:
abort in interface javax.security.auth.spi.LoginModule

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

Throws:
javax.security.auth.login.LoginException

When the abort fails.

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

Logout the user.

This method removes the Principals associated with the Subject.

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

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

Throws:
javax.security.auth.login.LoginException

When the logout fails.
Class PDPermission

com.tivoli.mts.PDPermission
public class PDPermission
extends java.security.Permission

Description

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

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

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

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

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

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

The requested resource to query for must correspond to a resource that has been defined as a protected object within Policy Director’s protected object namespace.
Constructor Summary

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

Method Summary

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

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

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

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

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

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

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

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

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

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

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

Method Detail

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

Checks to see if Policy Director grants the specified permission.

More specifically, this method returns true if:

  - p is an instance of PDPermission,
  - p’s actions are a proper subset of this object’s action map, and
  - p is a permission granted permission by the Policy Director Authorization Server for the specified principal.
Parameters:

p The permission to check.

Returns:

true if the specified permission is granted by Policy Director,
false if not.

implies

public boolean implies(PDPrincipal princ)

Determines if Policy Director grants the specified permissions to the
PDPrincipal.

This method returns true if:

- princ is non-null.
- This PDPermission’s actions are a proper subset of this object’s
  action map, and
- The specified actions are allowed by the Policy Director
  Authorization Server for the specified principal.

Parameters:

princ

The PDPrincipal whose permissions will be checked.

Returns:

true if the specified permission is implied by this object, false
if not.

implies

public boolean implies(PDPrincipal princ,
PDAattrs inputlist,
PDAattrs outputlist)

Determines if Policy Director grants the specified permissions to the
PDPrincipal.

This method returns true when:

- princ is non-null.
This PDPermission’s actions are a proper subset of this object’s action map, and

- The specified actions are allowed by the Policy Director for the specified principal.

**Parameters:**

- **princ**
  - The PDPrincipal whose permissions will be checked.

- **inputlist**
  - The input attribute list.

- **outputlist**
  - The output attribute list.

**Returns:**

- true if the specified permission is implied by this object, false if not.

**equals**

```java
public boolean equals(java.lang.Object obj)
```

Checks two PDPermission objects for equality. Checks that obj is a PDPermission, and has the same name and actions as this object.

**Parameters:**

- **obj**
  - The object that is being tested for equality with this object.

**Returns:**

- true if obj is a PDPermission, and has the same name and actions as this PDPermission object.

**getActions**

```java
public java.lang.String getActions()
```

Returns the canonical string representation of the actions.

**Returns:**

- The canonical string representation of the actions.
hashCode
public int hashCode()

Returns the hash code value for this object.

Returns:
   A hash code value for this object.
Class PDPrincipal

com.tivoli.mts.PDPrincipal

public class PDPrincipal
extends java.lang.Object
implements java.security.Principal,
com.ibm.security.auth.PrincipalComparator, java.io.Serializable

Description

This class implements the Principal interface and represents general information about a Policy Director user.

Principals such as this PDPrincipal may be associated with a particular Subject to augment that Subject with an additional identity. Refer to the Subject class for more information on how to achieve this. Authorization decisions can then be based upon the Principals associated with a Subject.

Constructor Summary

- PDPrincipal(java.lang.String name, java.lang.String char[] password)
  Create a PDPrincipal with an identifying name.

- PDPrincipal(java.lang.String name)
  Create a PDPrincipal with an identifying name.

Method Summary

- boolean equals(java.lang.Object o)
  Compare the specified Object with this PDPrincipal for equality.

- java.lang.String getName()
  Return a string name of this PDPrincipal.

- int hashCode()
  Return a hash code for this PDPrincipal.

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

- boolean implies(javax.security.auth.Subject subject)
Check if the specified Subject is implied by this object.

- `java.lang.String toString()`
  Return a string representation of this PDPrincipal.

- Methods inherited from class java.lang.Object
  - clone
  - finalize
  - getClass
  - notify
  - notifyAll
  - wait
  - wait
  - wait

### Constructor Detail

**PDPrincipal**

```java
public PDPrincipal(java.lang.String name,
                   char[] password)
```

Create a PDPrincipal with an identifying name.

**Parameters:**

- `name`  
  The identifying name.

- `password`  
  The password for the identifying name.

**Throws:**

- `java.lang.NullPointerException`
  When the name or the password is null.

- `java.lang.IllegalArgumentException`
  When the name has zero length.

- `java.lang.SecurityException`
  When the caller does not have permission to create PDPrincipals.

- `java.lang.IllegalStateException`
  When Policy Director could not obtain this user’s credentials.
PDPrincipal

public PDPrincipal(java.lang.String name)

Create a PDPrincipal with an identifying name.

Parameters:

name

The identifying name.

Throws:

java.lang.NullPointerException

When the name is null.

java.lang.IllegalArgumentException

When the name has zero length.

java.lang.SecurityException

When the caller does not have permission to create PDPrincipals.

java.lang.IllegalStateException

When Policy Director could not obtain this user’s credentials.

Method Detail

addGroupMemberships

public PDPrincipal addGroupMemberships(
    java.lang.String serviceID,
    java.lang.String[] groups)

throws

java.lang.NullPointerException

Returns a new PDPrincipal that adds these group memberships to the current PDPrincipal.

Parameters

serviceID

The credential modification service identifier. If null, the default credential modification service is employed.

groups

The groups to be added to the new PDPrincipal’s credentials.
Returns
A new PDPrincipal that reflects the additional group memberships.

Throws

java.lang.NullPointerException
When no groups are passed in.

java.lang.IllegalStateException
When remote errors occur.

implies

public boolean implies(PDPermission perm)

Determine if the current PDPrincipal has the specified PDPermission.

Parameters:
perm
The specified PDPermission.

Returns:
true if the current PDPrincipal has the specified PDPermission and false otherwise.

implies

public boolean implies(PDPermission perm,
    PDArrs attrsIn,
    PDArrs attrsOut)

Determine if the current PDPrincipal has the specified PDPermission. Report any resulting attributes from the server.

Parameters:
perm
The specified PDPermission.
attrsIn
The input attribute list.
attrsOut
The output attribute list.
Returns:

true
When the current PDPrincipal has the specified PDPermission.

false
When the current PDPrincipal does not have the specified PDPermission.

getName

public java.lang.String getName()

Return a string name of this PDPrincipal.

Specified by:
getName in interface java.security.Principal.

Returns:
A string name of this PDPrincipal.

toString

public java.lang.String toString()

Return a string representation of this PDPrincipal.

Specified by:
toString in interface java.security.Principal

Overrides:
toString in class java.lang.Object

Returns:
A string representation of this PDPrincipal.

equals

public boolean equals(java.lang.Object o)

Compares the specified Object with this PDPrincipal for equality. Returns true if the given object is also a PDPrincipal and the two PDPrincipals have the same String representation.
Specified by:
equals in interface java.security.Principal

Overrides:
equals in class java.lang.Object

Parameters:
o Object to be compared for equality with this PDPrincipal.

Returns:
true if the specified Object is equal to this PDPrincipal.

hashCode
public int hashCode()

Return a hash code for this PDPrincipal.

Specified by:
hashCode in interface java.security.Principal

Overrides:
hashCode in class java.lang.Object

Returns:
A hash code for this PDPrincipal.

implies
public boolean implies(javax.security.auth.Subject subject)

Check if the specified Subject is implied by this object.

Specified by:
implies in interface com.ibm.security.auth.PrincipalComparator

Parameters:
subject
A Subject to check for implication.

Returns:
true if the specified Subject is implied by this object, or false otherwise.
Class PDStatics

com.tivoli.mts
public class PDStatics
extends java.lang.Object

Description
A class for various constants used in PDPermission and associated classes.

Field Summary

- static java.lang.String AZN_ENT_SVC_PD_POBJ
  Entitlements Service ID: Policy Director Protected Object Entitlements

- static java.lang.String AZN_ENT_SVC_PD_POBJ_MATCHES
  Output attribute: the list of objects which match the input criteria.

- static java.lang.String AZN_ENT_SVC_PD_POBJ_REQD_OPS
  Input attribute: the requested set of operations for the credential.

- static java.lang.String AZN_MOD_RAD_GROUP_NAMES
  Input attribute; the groups to be added to the credential.

- static java.lang.String AZN_MOD_SVC_RAD_2AP
  The default Credential Modification service.

- static int AZN_VALTYPE_BUFFER
  The attribute is a byte array.

- static int AZN_VALTYPE_UTF8STRING
  The attribute is a UTF8 string.

- static int QOP_INTEGRITY

Version 3.8
The quality of protection required between the caller and the domain before access to the protected object can be granted.

- static int QOP_NONE
  The quality of protection required between the caller and the domain before access to the protected object can be granted.

- static int QOP_PRIVACY
  The quality of protection required between the caller and the domain before access to the protected object can be granted.

<table>
<thead>
<tr>
<th>Methods inherited from class java.lang.Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone</td>
</tr>
<tr>
<td>equals</td>
</tr>
<tr>
<td>finalize</td>
</tr>
<tr>
<td>getClass</td>
</tr>
<tr>
<td>hashCode</td>
</tr>
<tr>
<td>notifyAll</td>
</tr>
<tr>
<td>toString</td>
</tr>
<tr>
<td>wait</td>
</tr>
<tr>
<td>wait wait</td>
</tr>
</tbody>
</table>

### Field Detail

**AZN_MOD_SVC_RAD_2AB**

```java
public static final java.lang.String AZN_MOD_SVC_RAD_2AB
```

The default Credential Modification service.

**AZN_MOD_RAD_GROUP_NAMES**

```java
public static final java.lang.String AZN_MOD_RAD_GROUP_NAMES
```

Input attribute: the groups to be added to the credential.

**AZN_ENT_SVC_PD_POBJ**

```java
public static final java.lang.String AZN_ENT_SVC_PD_POBJ
```

Entitlements Service ID: Policy Director Protected Object Entitlements. This service takes a credential, a directory within the protected object tree and a set of operations and returns the list of protected objects in the given directory and its subdirectories for which the given aznAPI credential has permission to perform the requested set of operations. The output is returned as a multi-string valued attribute.
AZN_ENT_SVC_PD_POBJ_PATH
public static final java.lang.String AZN_ENT_SVC_PD_POBJ_PATH

Input attribute: the protected object directory path to be searched.

AZN_ENT_SVC_PD_POBJ_REQD_OPS
public static final java.lang.String AZN_ENT_SVC_PD_POBJ_REQD_OPS

Input attribute: the requested set of operations for the credential.

AZN_ENT_SVC_PD_POBJ_MATCHES
public static final java.lang.String AZN_ENT_SVC_PD_POBJ_MATCHES

Output attribute: the list of objects which match the input criteria.

QOP_NONE
public static final int QOP_NONE

The quality of protection required between caller and the domain before
access to the protected object can be granted. In this case, no
requirements have been placed.

QOP_INTEGRITY
public static final int QOP_INTEGRITY

The quality of protection required between caller and the domain before
access to the protected object can be granted. In this case, data integrity
must be maintained.

QOP_PRIVACY
public static final int QOP_PRIVACY

The quality of protection required between caller and the domain before
access to the protected object can be granted. In this case, data privacy
must be maintained.
AZN_VALTYPE_BUFFER
public static final int AZN_VALTYPE_BUFFER

The attribute is a byte array.

AZN_VALTYPE_UTF8STRING
public static final int AZN_VALTYPE_UTF8STRING

The attribute is a UTF8 String.
Class SvrSslCfg

com.tivoli.mts.SvrSslCfg
public class SvrSslCfg
extends java.lang.Object

Description

This program configures this machine for participation with Policy Director in resolving Java security checks that involve the PDPermission class or subclasses of PDPermission.

The output of a successful run of this program is a set of properties in a file named PdPerm.properties and a Java KeyStore that has been initialized to both recognize the certificate that Policy Director is currently using and to securely store a client certificate. The client certificate allows PDPermission to make authenticated usage of Policy Director services to check authorization.

Constructor Summary

- SvrSslCfg()

Method Summary

- static void main(java.lang.String[] argv)
  This routine configures this machine for use with Policy Director.

<table>
<thead>
<tr>
<th>Methods inherited from class java.lang.Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals getClass</td>
</tr>
<tr>
<td>hashCode</td>
</tr>
<tr>
<td>notify notifyAll</td>
</tr>
<tr>
<td>toString</td>
</tr>
<tr>
<td>wait wait wait</td>
</tr>
</tbody>
</table>

Constructor Detail

SvrSslCfg

public SvrSslCfg()
Method Detail

main

public static void main(java.lang.String[] argv)

This routine configures this machine for use with Policy Director.

To illustrate how this configuration might be done, consider the following example. The administrator will have run the following set of commands on the Policy Director Management Server. In this example, the Policy Director Management Server is on the host timon.i.com:

dadmin -a sec_master -p xxxpassword

user create PdPermission9993 \ cn=PdPermission/gordo,o=myCompany PdPermission/gordo \ PdPermission somePW9993

user modify PdPermission9993 account-valid yes

group modify remote-acl-users add PdPermission9993

(This example assumes that the LDAP root for the users is anchored at o=myCompany, and uses the convention that the “distinguishing” part of the distinguished name will in fact be the hostname of the machine, which in this example is “gordo”.)

The command that could be issued now on gordo is:

java com.tivoli.mts.SvrSslCfg cn=PdPermission/gordo,o=myCompany xxxpassword timon.i.com rafiki.i.com

This example command takes the default SSL port numbers for the Policy Director Authorization Server and the Policy Director Management Server, and configures gordo to use the Policy Director Authorization Server on the host rafiki.i.com.
Parameters:

`dn`
The distinguished name of the principal that this machine is to use.

`pw`
The password for sec_master.

`ivmgrd-hostname`
The name of the machine that is running the Policy Director Management Server.

`ivacld-hostname`
The name of the machine that is running the instance of the Policy Director Authorization Server with which this machine is to communicate.

`ivmgrd-port`
The SSL port number for the Policy Director Management Server, if other than the default (7135).

`ivacld-port`
The SSL port number for Policy Director Authorization Server, if other than the default (7136).

Throws:

`java.lang.IllegalArgumentException`
When it appears that invalid data was passed in.

`java.lang.IllegalStateException`
When this request could not be processed.
Index

A
access control decisions 126
making 126
making and extending 128
access decision function (ADF) 3
access enforcement function (AEF) 3
adding
additional application-specific context 55
attributes for LDAP access 38
authorization to an application 4
credentials and handle 97
name or buffer value to attribute list 77
name or value to attribute list 75
additional user information 51
address of data structure 52
ADF (access decision function) 3
ADK 3
administrator’s distinguished name 40
AEF (access enforcement function) 3
AIX
libivauthzn.a library file 6
allocated memory 58
API
attribute lists 10
authorization decisions 11
credentials 10
error handling 11
extensions 12
functions 9
API functions
azn_attrlist_add_entry_buffer() 77
azn_attrlist_add_entry() 75
azn_attrlist_create() 81, 82
azn_attrlist_delete() 83, 84
azn_attrlist_get_entry_buffer_value() 85, 89
azn_attrlist_get_entry_string_value() 91
azn_attrlist_get_names() 93
azn_attrlist_name_get_num() 95
azn_creds_combine() 97
API functions (continued)
azn_creds_create() 101
azn_creds_delete() 103
azn_creds_for_subject() 107
azn_creds_get_attrlist_for_subject() 112
azn_creds_get_pac() 115
azn_creds_modify() 118
azn_creds_num_of_subjects() 122
azn_decision_access_allowed_ext() 128
azn_decision_access_allowed() 126
azn_error_major() 135
azn_error_minor_get_string() 137
azn_error_minor() 136
azn_id_get_creds() 139
azn_initialize() 142
azn_pac_get_creds() 146
azn_release_buffer() 148
azn_release_string() 150
azn_release_strings() 151
azn_shutdown() 152
azn_util_password_authenticate() 156
Application Development Kit (ADK) 3
applications
building 7
building an attribute list 55
deploying with the Authorization API 65
determining user’s authorization credentials 49
array of strings
memory, releasing 58
storage, freeing 151
assigning
handle for an empty attribute list 81, 82
handle to empty credentials structure 101
user credentials to a credentials handle 55
attribute list functions 10
attribute lists
adding name or buffer value 77
adding name or value 75
attribute lists (continued)
building for additional application information 55
creating 16
creating and assigning a handle 81, 82
deleting 18, 83, 84
getting an attribute name 17
getting the number of values 17
getting values 18
obtaining a credential 61
releasing memory 58
setting an entry 17
audit
identifier 113
user information user_info 51
authenticated user identity 48
authentication
identity, user 50
information 51, 156
mechanism 49
authority, authorization 50
authorization
authority 50
credentials 49, 52, 53
decisions 11, 54, 56
Authorization API
buffer 52
building applications 7
changing the credential’s contents 61
character strings 12
converting credentials to a transportable format 59
converting credentials to the native format 60
creating a chain of credentials 60
deploying applications 65
determining the number of credentials in a chain 61
functions and data types 9
handling credentials 59
initializing 23, 142
introducing 2
obtaining a credential from a chain 61
obtaining credential from a chain 61
shutting down 57
specifying cache mode type 25
Authorization API (continued)
tasks 21
Authorization server
specifying cache mode type 25
authorization service
initializing 23, 142
minor error codes 7
minor errors 20
starting 47
submitting requests to 3
azn_attrlist_add_entry_buffer() 56
azn_attrlist_add_entry() 56
azn_attrlist_create() 56, 63
azn_attrlist_delete() 83
azn_attrlist_get_entry_buffer_value() 85, 89
azn_attrlist_get_entry_string_value() 91
azn_attrlist_get_names() 93
azn_attrlist_h__ 16
azn_attrlist_h_t 57
azn_attrlist_name_get_num() 95
azn_buffer_desc 13
azn_buffer_t 13
AZN_C_AUDIT_ID 113
AZN_C_EMPTY_BUFFER 13
AZN_C_INITIATOR_INDEX 61, 63, 108, 112
AZN_C_NO_BUFFER 13
AZN_C_NOT_PERMITTED 57, 126, 129
AZN_C_PERMITTED 57, 126, 128
AZN_C_VERSION 48
azn_creds_combine() 60, 97
azn_creds_create() 58, 60, 62, 101
azn_creds_delete() 103
azn_creds_for_subject() 61, 107
azn_creds_get_attrlist_for_subject() 64, 112
azn_creds_get_pac() 59, 115
azn_creds_h_t 19, 58, 60
azn_creds_modify() 62, 118
azn_creds_num_of_subjects 61
azn_creds_num_of_subjects() 61, 122
azn_decision_access_allowed_ext() 56, 128
azn_decision_access_allowed() 54, 56, 126
azn_error_get_string() 20
azn_error_major() 135
azn_error_minor_get_string() 137
azn_error_minor() 136
azn_id_get_creds() 52, 55, 59, 139
azn_initialize() 142
azn_operation_read 54
azn_operation_traverse 54
azn_pac_get_creds() 146
azn_release_buffer() 148
azn_release_pobj() 58
azn_release_string 12
azn_release_string() 12, 150
azn_release_strings() 12, 151
AZN_S_INVALID_MOD_FUNCTION 120
AZN_S_ATTR_INVALID_INDEX 86
AZN_S_ATTR_INVALID_INTEGER_REF 123
AZN_S_ATTR_VALUE_NOT_BUFFER_TYPE 86, 90
AZN_S_ATTR_VALUE_NOT_STRING_TYPE 92
AZN_S_COMPLETE 19
AZN_S_FAILURE 19
AZN_S_INVALID_ADDED_CREDS_HDL 98
AZN_S_INVALID_APP_CONTEXT_HDL 130
AZN_S_INVALID_ATTR_NAME 76, 86, 90
AZN_S_INVALID_ATTRLIST_HDL 75, 86, 89
AZN_S_INVALID_AUTHORITY 140
AZN_S_INVALID_BUFFER 78
AZN_S_INVALID_BUFFER_REF 86, 90, 148
AZN_S_INVALID_COMB_CREDS_HDL 98
AZN_S_INVALID_INTEGER_REF 96
AZN_S_INVALID_MECCHANISM 140
AZN_S_INVALID_MECHANISM_INFO 140
AZN_S_INVALID_NEW_CREDS_HDL 108, 140, 147
AZN_S_INVALID_OPERATION 127
AZN_S_INVALID_PAC 147
AZN_S_INVALID_PAC_SVC 116, 147
AZN_S_INVALID_PERMISSION_REF 127
AZN_S_INVALID_PROTECTED_Resource
AZN_S_INVALID_STRING_REF 150, 151
AZN_S_INVALID_STRING_VALUE 76
AZN_S_INVALID_SUBJECT_INDEX 108
AZN_S_UNIMPLEMENTED_FUNCTION 108, 147
azn_shutdown() 48, 152
azn_status_t 19
azn_string_t 12, 17, 50
azn_util_passwordAuthenticate() 49, 156
aznutils.h 6, 20

B
books
feedback xii
online xii
ordering xii
browser information 51
buffer 12
declaration 13
empty 13
buffer attribute value 85
buffers
declaration 52
introduction to 12
none 13
release of memory 58
storage, freeing 148
building
applications 7
attribute lists 55

C
cache modes 25
chain of credentials 60, 122
changing
contents of a credential 61
existing credential 118
character strings 12
cleaning up 57, 152
combining credentials and handle 97
components of
ADK 5
Policy Director 5
configuring
Authorization API 25
network environment 21
Policy Director secure domain 5
contents of the credential 61
converting
credentials to a transportable format 59
creating
attribute lists 16
chain of credentials 60
empty credentials structure 101
new attribute lists 24, 56
privilege attribute certificates 115
valid empty attribute list 81, 82
credentials 10
changing 118
changing the credential’s contents 61
combining with a handle 97
converting to a transportable format 59
converting to the native format 60
creating a chain of credentials 60
creating and assigning a handle 101
deleting 103
determining number of credentials 61
extracting individual credentials 107
handle 55
invoking a privilege attribute certificate
service 115
making access control decisions 126
making extended access control
decisions 128
obtaining attribute list from a credentia 62
obtaining for user authorization 50, 52
obtaining from a chain of credentials 61
returning handle to new PAC
credentials 146
returning in a chain 122
returning information from 112
user authorization 52
creds_attrlist 64
custom-protected object 55
Customer Support xiv

D
data type structure
azn_attrlist_h_t 16
azn_buffer_t 12

314

data type structure (continued)
azn_status_t 19
azn_string_t 12
data types 9
dceaclmsg.h 20
decision
authorization 56
decision, authorization 56
decisions
access control 126, 128
authorization 11
defining
security policy 4
deleting
attribute list 18, 83, 84
credentials 103
deploying
applications 65
applications into secure domain 5
determining
authorization credentials for a user 52
identity for a user 50
number of credentials in a credentials
chain 61
disabling
notification listener 28
refreshes of local authorization policy
database 27
distinguished name 40

E
e-mail contact xiii
empty credentials chain 101
enabling
notification listener 28
error codes
aznutils.h 7, 20
dceaclmsg.h 7, 20
ogauthzn.h 7, 20
error handling 11, 19
example of
assigning user identity information 53
attribute list initialization data 29

Version 3.8


example of (continued)
declaring a buffer 52
extending
  API function standard 12
  function for obtaining an access decision 56
extensions, API 12
external authorization server (see Authorization server) 56
extracting individual credentials 107

F
feedback about publications xiii
files
  aznutils.h 6, 20
dceaclmsg.h 20
ogauthzn.h 7, 20
format
  credentials, transportable 59
freeing
  array of strings storage 151
  buffer storage 148
  string storage 150

G
getting
  attribute list name 17
  entry string value 91
  handle for a specified identity 139
  name attributes 93
  number of attribute entries 95
  number of values for attribute list name 17
  value attributes 18

H
handle 97, 101, 103, 139
  credentials 18, 55, 101
handling credentials 59
header files
  aznutils.h 6
  ogauthzn.h 6
host name, LDAP server 39

I
IBM SecureWay
  Toolbox 6
identities, user 50
implementation modes 2
initialization 11
initializing
  authorization service 23, 142
  data 142
  invalid data 143
  initiator 4
  installing
    Policy Director 66
  interfaces
    Authorization API manual pages 73
  International Organization for Standardization (ISO) 3
  introduction to
    Authorization API 2
IP address 51
ISO (International Organization for Standardization) 3
ivacld-servers 22

K
  key file, SSL 41
  key label, SSL 41
**LDAP**
- administrator’s distinguished name 40
- administrator’s password 40
- key file password 42
- port number 39
- server host name 39
- server key label 41
- SSL key file 41
- length
  - data structure 52
  - local cache mode 2, 25
- logging in
  - using username and password pair 156

**N**
- name value 93
- notification listener 26
- number of
  - bytes in the data 13
  - individual credentials in a chain 122
  - seconds before refreshing 28
  - value attributes in the entry 95
  - values for an attribute name 17
- number of port, LDAP server 39

**M**
- major errors 7, 20, 135
- making
  - access control decisions 126
  - extended access control decisions 128
- manual pages
  - summary 73
- manuals
  - feedback xii
  - online xii
  - ordering xii
- mapping
  - requested resource to protected object 55
  - user operation to a permission 54
- memory
  - credential structure 19
  - release 58
- minor errors 7, 20, 136
- mod_info 62
- mod_svc_id 62
- mode
  - local cache 2
  - remote cache 2
- modes, specifying 25
- modifying
  - contents of a credential 61
  - existing credential 118

**O**
- obtaining
  - authorization decision 56
  - credential from a chain of credentials 61
  - user authorization credentials 50, 52
- ogauthzn.h 6, 20
- online publications xiii
- Open Group 3
- ordering publications xiii
- output parameters
  - authorization decision 57
  - extended authorization decision 57

**P**
- PAC (privilege attribute certificate) 50, 59, 115, 146
- pac_svc_id 60
- password
  - accessing the SSL key file 42
  - authenticating 156
  - LDAP administrator 40
- permissions 54
- persistent authorization policy database 27, 46
- placing the data structure into a buffer 52
- policy database replica 27, 46
- port number
  - for a TCP port 29, 39
Privilege attribute certificate (PAC) 50, 59, 115, 146
protected object 55
protected object namespace 55
providing additional parameters 55
publications feedback xii
  online xii
  ordering xii

R
refreshing
  local authorization database 27
registry, user 8, 48
releasing
  allocated memory 58
  memory allocated 19
remote cache mode 2, 26
remote-acl-users 22
removing
  attribute list 83, 84
  credentials 103
requested resource 55
returning
  access control decision information 129
  entry string value 91
  handle 118
  handle for a specified identity 139
  handle to credentials structure 107
  handle to new PAC credentials 146
  individual credentials in a chain 122
  information from a credentials structure 112
  major error code 135
  minor error code 136
  name attributes 93
  number of buffer attribute entries 95
  number of value attributes 95
  privilege attribute certificate 115

S
secure domain 7
Secure Socket Layer (SSL) 40
security policy 4
server
  host name, LDAP 39
  setting an attribute list entry 17
shut down 11
shut down 57
shutting down 11, 57, 58, 152
Solaris
  libivauthzn.so library file 6
  library linking 9
specifying
  additional user information 51
  authorization authority 50
  type of cache mode 25
  user authentication identity 50
SSL
  communications 40
  key file password 42
  key label 41
standard, The Open Group 3
starting
  authorization service 47
status codes 19, 135, 136
storage
  array of strings, freeing 151
  buffer, freeing 148
  string, freeing 150
strings
  freeing of storage 150
  release of memory 58
  see also array of strings 58
  value 91
successful login 156
summary of
  API extensions 12
  API functions 9
  attribute list functions 10
  attribute list tasks 16
  authentication parameters 52
Authorization API manual pages 73
Authorization API tasks 21
authorization decision functions 11
summary of (continued)
authorization decision output parameters  57
buffer names and values  12
cache modes  25
credentials functions  10
data types  9
error code files  19, 20
initialization, shutdown, and error handling functions  11
local cache mode attributes and values  27
SSL attributes for LDAP access  40
user identity types  48

user registry
  specifying LDAP  8
  specifying the user authentication identity  50
username and password  156
using
  as input of credentials information  15
  username and password to log in  156
utility function error codes
  major errors  7
  minor errors  7

V
  value attributes
    buffer  85, 89
    entry number  95
    name  93
    string  91
  values  13
  version number  48

W
  Windows NT
    ivauthzn.dll library file  6

T
  tasks, Authorization API  21
  TCP port number  29
  Tivoli Customer Support  xiv
  Toolbox, IBM SecureWay  6
  tools
    IBM SecureWay Toolbox (Toolbox)  6
  types of
    additional user information  51
    authentication parameters  52
    cache modes  25
    user identities  48

U
  unauthenticated user  48
  unauthenticated users  15
  user
    additional auditing information  51
    assigning credentials to a credentials handle  55
    authentication identity  50
    authorization credentials  50, 52
    mapping the user operation  54
    obtaining an identity  48
    specifying additional information  51
    unauthenticated  51