HP OpenView Select Access SDK

For the Windows®, Unix®, and HP-UX® Operating Systems

Software Version: 6.2

Developer's Tutorial Guide

Document Release Date: September 2006
Software Release Date: September 2006

Legal Notices

Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notices

Trademark Notices

HP OpenView Select Access includes software developed by third parties. The software HP OpenView Select Access uses includes:

- Software developed by the Apache Software Foundation.
- Software developed by Claymore Systems, Inc.
- Cryptographic software written by Eric Young.
- Cryptographic software developed by The Cryptix Foundation Limited.
- cURL, Copyright 2000 Daniel Stenberg.
- JavaBeans Activation Framework version 1.0.1 Sun Microsystems, Inc.
- JavaMail, version 1.2 Sun Microsystems, Inc.
- JavaService software from Alexandria Software Consulting.
- JClass LiveTable, Copyright 2002 Sitraka Inc.
- The OpenSSL Project for use in the OpenSSL Toolkit.
- Protomatter Syslog, Copyright 1998-2000 Nate Sammons.
- SoapRMI, Copyright 2001 Extreme! Lab, Indiana University.

For expanded copyright notices, see HP OpenView Select Access `<install_path>/3rd_party_license` directory.

Documentation Updates

This manual's title page contains the following identifying information:

- Software version number, which indicates the software version
- Document release date, which changes each time the document is updated
- Software release date, which indicates the release date of this version of the software

To check for recent updates, or to verify that you are using the most recent edition of a document, go to:

http://ovweb.external.hp.com/lpe/doc_serv/

You will also receive updated or new editions if you subscribe to the appropriate product support service. Contact your HP sales representative for details.

Support

You can visit the HP OpenView Support web site at:

www.hp.com/managementsoftware/support

HP OpenView online support provides an efficient way to access interactive technical support tools. As a valued support customer, you can benefit by using the support site to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Download software patches
- Manage support contracts
- Look up HP support contacts
- Review information about available services
- Enter into discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and sign in. Many also require a support contract.

To find more information about access levels, go to:

www.hp.com/managementsoftware/access_level

To register for an HP Passport ID, go to:

www.managementsoftware.hp.com/passport-registration.html

Contents

1 Understanding Select Access APIs 13
Audience 13
The Select Access Documentation Set 13
Chapter Summary 14

2 Getting Started with Select Access APIs 17
Chapter Overview 17
Select Access Components 17
Customizing Select Access 19
Select Access' Programming APIs 19
Customizing with Select Access APIs 21
Select Access Source Files and Libraries 22
Building Examples in the SDK 23
To build C++ examples on UNIX 23
To build C++ examples on Windows 24
To build Java examples on Windows or UNIX 24
To build and install the Web Administration interface on Windows or UNIX 24
Understanding the Importance of XML 25
How Rules are Expressed and Stored 25
How Data is Encoded and Communicated 25
Scenario: Identity Requesting access to mycompany.com 26

3 Custom Configuration GUIs: the Policy Builder API 29
Chapter Overview 29
Understanding the Policy Builder 29
Understanding the Policy Builder API 30
Types of Policy Builder Plugins 30
Authentication plugins 30
To determine which Authentication plugins are installed 31
Authorization plugins 31
How the Policy Builder API Works 33
Policy Builder API Classes and Utilities 33
Creating a Policy Builder Plugin 34
To create a new Policy Builder plugin 34
Creating a Configuration Editor 35
To create a Configuration editor 35
Creating Icons for Authorization Plugins 37
Creating a component.xml File 37
Installing Your Policy Builder Plugin 39

To install your Policy Builder plugin . . . 39
Removing Policy Builder Plugins . . . 40
To remove the Policy Builder plugin . . . 40
4 Custom Authentication Methods and Rules: the Validator API . . . 43
Chapter Overview . . . 43
Understanding the Policy Validator . . . 43
Understanding the Validator API . . . 44
Types of Validator API Plugins . . . 45
Authentication plugins . . . 45
Decision Point plugins . . . 45
How the Validator API Works . . . 46
Validator API Classes and Utilities . . . 48
Creating a Policy Validator Plugin . . . 50
To create your plugin . . . 50
Overriding the authenticate() Method . . . 50
Overriding the decide() Method . . . 51
Installing Policy Validator Plugins . . . 51
Removing Policy Validator Plugins . . . 51
Creating an Authentication Plugin: File-based Authentication Example . . . 52
Including the Authentication Plugin Header Files . . . 52
FileAuthenticator Class Constants . . . 53
Registering an Authentication Plugin . . . 54
Creating Authentication Plugin Instances with the factory() Method . . . 54
Destroying Authentication Plugin Instances . . . 61
Authenticating Identities . . . 61
Creating a Decision Point Plugin: Directory Attributes Example . . . 66
Including the Decision Point Plugin Header Files . . . 67
Attributelogic Class Constants . . . 67
Registering the Decision Point Plugin . . . 68
Creating Decision Point Plugin Instances . . . 69
Destroying Decision Point Plugin Instances . . . 70
Evaluating a Policy Node . . . 70
LdapConnection, User, UserSource, and UserCache . . . 72
Building the AttributeLogic Example . . . 79
5 Custom Security Services: the Enforcer API . . . 81
Chapter Overview . . . 81
Understanding the Enforcer plugin . . . 81
Understanding the Enforcer API . . . 82
How the Enforcer API Works . . . 83
To create an Enforcer plugin . . . 83
Enforcer API Classes and Utilities . . . 84
Configuring the Enforcer API . . . 84
Development Considerations . . . 85
Determining if authorization is required . . . 86
Presenting login screens . . . 86

Maintaining session state . . . 86
Handling single sign-on nonces . . . 86
Performing multiple domain single sign-on . . . 86
Creating an Enforcer Plugin With the Java Enforcer API . . . 87
The ServletFilter Example . . . 87
Java Enforcer API Classes and Utilities . . . 88
Importing the Enforcer Classes . . . 88
The ServletFilter and ServletTransaction Classes . . . 88
Filtering servlet requests . . . 89
The ServletTransaction class . . . 90
Creating an Enforcer Object . . . 90
Instantiating new instances of an Enforcer object . . . 91
Extracting Authorization Data . . . 92
Determining if Authorization is Needed . . . 93
Building the Policy Validator Request . . . 94
Querying the Policy Validator . . . 95
Enforcing the Policy Validator Response . . . 96
Multidomain Single Sign-on . . . 99
Deploying the Sample Web Application with the Servlet Filter . . . 104
Creating an Enforcer Plugin With the C/C++ Enforcer API . . . 104
The Apache Example . . . 104
C/C++ Enforcer API Classes and Utilities . . . 105
Including the Enforcer Libraries . . . 106
Creating an Enforcer Object . . . 107
Determining if Authorization is Required . . . 108
Building the Policy Validator Request . . . 109
Querying the Policy Validator . . . 116
Enforcing the Response . . . 116
Multiple Domain Single Sign-on . . . 119
Special UNIX Considerations . . . 121
Registering Apache Plugins . . . 122
Compiling and Installing the Apache Enforcer Plugin . . . 122
Creating an Enforcer Plugin With the COM Enforcer API . . . 123
The WSE Enforcer Plugin Example . . . 123
The Helper class . . . 123
The InputFilter class . . . 124
The OutputFilter class . . . 125
The COM Enforcer API Classes and Utilities . . . 126
Importing the COM Libraries . . . 126
Creating an Enforcer Object . . . 127
Building an XML Request . . . 127
Querying the Policy Validator . . . 133
Enforcing the Policy Validator Response . . . 134
Building the WSE Enforcer Plugin . . . 142
Including Site-specific Data . . . 142
How Site Data is Injected into the Policy Validator Query . . . 143
To load the site_data plugin . . . 143

6 Using Personalization Attributes: the Personalization API . . . 145
Chapter Overview . . . 145
Understanding the Personalization API . . . 145
Supporting Personalization in Policy Validators . . . 146
Supporting Personalization in Enforcer Plugins . . . 146
The Servlet Enforcer Plugin Example . . . 147
Adding personalization data to the servlet HTTP headers . . . 147
The Apache Enforcer Plugin Example . . . 148
Exporting to the environment . . . 149
Accessing Personalization Data from Resources . . . 150
Displaying UNIX Environment Variables . . . 151
Displaying HTTP Headers from a Servlet . . . 151
Displaying Server Variables With Visual BASIC . . . 154
7 Querying for Multiple Resources: the Policy API . . . 155
Chapter Overview . . . 155
Understanding Access Control . . . 155
Understanding XML and the Policy API . . . 155
Using the Java Policy API . . . 157
Using the C/C++ Policy API . . . 158
Using the COM Policy API . . . 159
8 Transient Directory Profiles: the User API . . . 161
Chapter Overview . . . 161
Understanding the User API . . . 161
The File Authentication Plugin Example . . . 162
Installing the File Authenticator . . . 162
To build the plugin . . . 163
To install the Policy Builder plugin . . . 163
To install the Policy Validator plugin . . . 163
To install the online help . . . 163
Configuring the File Authenticator . . . 163
To create an authentication service . . . 163
How the File Authenticator Works . . . 165
Authenticating identities . . . 168
Searching for existing profiles . . . 170
Creating a new identity profile . . . 171
Creating attributes . . . 172
9 Administration API . . . 175
Chapter Overview . . . 175
Understanding the Administration Server . . . 175
Understanding the Administration API . . . 176
Exception Handling . . . 176
Logging . . . 176
Getting Started . . . 177
Data Type Overview . . . 177

Common Data Types . . . 178
resourcePath . . . 178
Network Resources Paths . . . 179
Administrative Resources Paths . . . 179
Network Management Resource paths . . . 179
Schema Management paths . . . 180
Function Management paths . . . 180
Identity Management paths . . . 180
LdapSearchCriteria . . . 181
LdapSearchCriteria parameters . . . 182
LdapSearchCriteriaFilter . . . 183
LdapSearchCriteriaFilter parameters . . . 183
AdminFault . . . 184
Resource-specific APIs . . . 184
getResource . . . 184
Input . . . 184
Output . . . 184
getResource parameters . . . 185
ResourceServer . . . 185
ResourceServer parameters . . . 186
Sample program . . . 186
setResource . . . 187
Input . . . 187
setResource input parameters . . . 188
setResource behavior . . . 188
Output . . . 188
setResource output parameters . . . 189
Sample program . . . 189
searchResources . . . 190
Input . . . 190
Output . . . 190
Sample program . . . 190
Policy-specific APIs . . . 191
getPolicies . . . 191
Input . . . 191
Output . . . 192
getPolicies parameters . . . 192
IdentityColumn . . . 192
IdentityColumn parameters . . . 193
Policy . . . 193
Policy parameters . . . 193
AccessPolicy . . . 194
AccessPolicy parameters . . . 194
Rule . . . 194
Rule parameters . . . 195
WorkflowPolicy . . . 195
WorkflowPolicy parameters . . . 196

AdminPolicy . . . 196
AdminPolicy parameters . . . 197
SelectAuthPolicy . . . 197
SelectAuthPolicy parameters . . . 198
SelectAuthPropertyType . . . 198
SelectAuthPropertyType parameters . . . 198
Sample SelectAuthProperty XML . . . 200
Sample program . . . 201
setPolicy . . . 203
Input . . . 203
setPolicy parameters . . . 203
Output . . . 204
Sample program . . . 204
Helper APIs . . . 207
refreshValidators . . . 207
getAuthServiceNames . . . 207
getRuleNames . . . 207
Input . . . 207
Output . . . 207
Modifying Passwords . . . 207
Changing User Passwords . . . 207
setUserPassword . . . 207
Input . . . 208
setUserPassword parameters . . . 208
Output . . . 208
Sample program . . . 208
Error Handling . . . 209
10 Administration Servlets: the Web Administration Interface . . . 211
Chapter Overview . . . 211
Understanding the Web Administration Interface . . . 211
Some of the Admin class methods . . . 212
Customizing Web Administration . . . 213
The Build Process and Angle Brackets (>< or >#<) . . . 213
JSP Files Used . . . 214
Providing a Menu . . . 215
HTML main menu . . . 216
Displaying Data: Directory Server Search Results . . . 217
Searching a directory server . . . 217
Displaying directory node information . . . 220
Displaying Data: Directory Entries as Identity Profiles . . . 221
Locating a directory server entry . . . 222
Displaying a directory server entry as a profile . . . 223
Modifying a Profile . . . 226
Modifying the profile's directory server entry . . . 226
About JSP Containers and JAR files . . . 227

11 Customizing Logging: the Logger API . . . 229
Chapter Overview . . . 229
Understanding the Logger API . . . 229
Using the Logger API . . . 230
To configure C++ log filters manually . . . 230
To configure Java log filters using XML . . . 231
The Logger API for C++ . . . 231
Logging Messages With the Default Destination . . . 231
Manually Configuring Log Filters . . . 232
Logging Messages . . . 233
The Logger API for Java . . . 234
Logging Messages With the Default Destination . . . 234
Using XML to Configure Log Filters . . . 234
The XML configuration . . . 235
Logging Messages . . . 236
Logging to custom channels . . . 237
12 Custom Property Editors: the Subject Editor API . . . 239
Chapter Overview . . . 239
Understanding the Subject Editor API . . . 239
Creating Subject Editor Plugins . . . 240
To create a new Subject Editor plugin . . . 240
Subject Editor plugin pverview . . . 240
Building the Screen Interface . . . 241
Overriding the getScreenName() method . . . 242
Overriding the createScreen() method . . . 242
Overriding the openScreen() method . . . 243
Overriding the loadScreenData() method . . . 243
Overriding the validateScreenData() method . . . 244
Overriding the saveScreenData() method . . . 245
Overriding the closeScreen() method . . . 245
Overriding the showHelp() method . . . 246
Displaying Help . . . 246
Installing and Removing Subject Editor Plugins . . . 247
To install your Subject Editor plugin . . . 247
To delete your Subject Editor plugin . . . 248
A Queries: Understanding and Evaluating Access . . . 249
Appendix Overview . . . 249
What is a Query? . . . 249
How is a Query Used? . . . 249
What is the Structure of an XML Query . . . 250
What Properties a Query Contains . . . 252
How the Policy Validator Replies to the XML Query . . . 253
The Query Utilities . . . 253
About the C++ Implementation . . . 254
Command line syntax . . . 254

Usage scenarios 256
About the Java Implementation 257
Program Options 258
Index 259

1 Understanding Select Access APIs

Select Access is constructed on a component-based, multi-tier architecture. Because of this unique approach, it is able to offer administrators and developers a versatile system that is both easy to configure and to upgrade. By making use of component-based architecture, Select Access can adapt to any existing network infrastructure and can be extended to meet the needs of future security requirements.

To achieve this extensibility, Select Access provides a number of API libraries that allow you to create new components that customize the functionality of Select Access for your network infrastructure and your business requirements. These new components, once created, can then be "plugged into" the Select Access software architecture without requiring you to reinstall the whole product.

Audience

This document is intended for developers looking to customize HP OpenView Select Access 6.2 to suit their business and industry environment. In particular, it is intended to support Select Access APIs, giving you the ability to find specific reference material for specific methods or functions in a given library.

This guide assumes an *advanced* knowledge of C/C++, Java, and COM programming languages. It also assumes advanced knowledge of web servers and how they work with Select Access.

The Select Access Documentation Set

This manual refers to the following Select Access documents. These documents are installed with Select Access and are available in the `<SDK_install_path>/docs` folder.

- *HP OpenView Select Access 6.2 Installation Guide*, © Copyright 2000-2006 Hewlett-Packard Development Company, L.P. (`installation_guide.pdf`)
- *HP OpenView Select Access 6.2 Policy Builder Guide*, Copyright 2000-2006 Hewlett-Packard Development Company, L.P. (`policy_builder_guide.pdf`)
- *HP OpenView Select Access 6.2 Network Integration Guide*, © Copyright 2002-2006 Hewlett-Packard Development Company, L.P. (`integration_guide.pdf`)
- *HP OpenView Select Access 6.2 Concepts Guide*, © Copyright 2005 - 2006 Hewlett-Packard Development Company, L.P. (`concepts_guide.pdf`)

Integration Papers for Select Access and vendor-specific technologies are available on the product CDs in the `docs/solutions` folder.

Online help is available with both the Setup Tool and the Policy Builder components.

As part of the Select Access SDK, two other documents are also available with this product:

- *HP OpenView Select Access 6.2 Developer's Tutorial Guide*, © Copyright 2004-2006 Hewlett-Packard Development Company, L.P. (`dev_tut_guide.pdf`)
- *HP OpenView Select Access 6.2 Developer's Reference Guide*, © Copyright 2004-2006 Hewlett-Packard Development Company, L.P. (`dev_ref_guide.pdf`)

For details on how to obtain this SDK, visit HP's Partner Care site (`http://support.openview.hp.com/partner_care.jsp`).

Chapter Summary

This guide includes the chapters listed in Table 1.

Table 1 Chapters in This Guide

Chapter	Description
Chapter 2, Getting Started with Select Access APIs	Select Access is constructed on a component-based, multi-tier architecture. This chapter describes Select Access components and how you can build your own plugins.
Chapter 3, Custom Configuration GUIs: the Policy Builder API	Select Access enables developers to provide additional Policy Builder configuration editors. This chapter describes how to use the Policy Builder API to create Configuration editors, called configurators, that integrate with the Select Access Policy Builder.
Chapter 4, Custom Authentication Methods and Rules: the Validator API	Select Access allows you to extend the Policy Validator to support custom authentication methods and authorization rules. This chapter describes how to use the Validator API to create authentication and authorization plugins.
Chapter 5, Custom Security Services: the Enforcer API	The Enforcer API allows you to integrate Select Access security services with most products and custom components. This chapter describes how to use the Enforcer API to provide security services for Java, C/C++, and COM objects.
Chapter 6, Using Personalization Attributes: the Personalization API	Select Access enables Enforcer plugins to provide identity personalization data. This chapter describes how to configure Select Access to support personalization, how Policy Validators provide personalization attributes to Enforcer plugins, and how Enforcer plugins can make this information available to applications.
Chapter 7, Querying for Multiple Resources: the Policy API	Select Access enables Enforcer plugins to provide to query the Policy Validator for access to multiple resources. This chapter describes how to use multiple resource requests.
Chapter 8, Transient Directory Profiles: the User API	Select Access provides an API to store persistent identity attributes for identitys that do not have a profile on the directory server. In this case, the Policy Validator cannot provide personalization attributes for them, unless the authentication plugin creates something known as a transient identity profile. This chapter describes how those profiles are created.

Table 1 Chapters in This Guide (cont'd)

Chapter	Description
Chapter 9, Administration API	The Administration API provides a programming interface for the Select Access Administration server that can be used to define resources, set policies, and modify user passwords. This chapter describes the functions of the Administration API.
Chapter 10, Administration Servlets: the Web Administration Interface	In addition to the Policy Builder applet, Select Access provides a Web Administration interface that allows administrators to perform Delegated Administration and Workflow tasks using a web browser. This chapter describes how you can customize the interface using servlets and the Web Administration interface.
Chapter 11, Customizing Logging: the Logger API	Select Access provides a Logger API that allows any Java or C++ application to report events in a secure, structured way. The Logger API utilizes the Select Access logging classes which provide named log channels, log severity levels, and log destinations. This chapter shows you how to integrate logging into your application.
Chapter 12, Custom Property Editors: the Subject Editor API	The Subject Editor API allows custom configuration screens that edit user, group, dynamic group, and folder properties. The default property panels were developed using this API. This chapter describes how to use the Subject Editor API to create custom property editor panels in the Policy Builder.
Appendix A, Queries: Understanding and Evaluating Access	Policy Validator queries are the backbones of the Select Access access control system. By using XML to encode and transmit data, there are no restrictions on the form or the types of data that can be contained in the query. This appendix helps you to better understand the format and implementation of these queries.

2 Getting Started with Select Access APIs

Select Access consists of several components, making for a sophisticated and consistent architecture. This chapter describes how to use of current programming languages, industry-adopted standards, and state-of-the-art security methods, to adapt Select Access adapts to any existing network infrastructure.

This flexibility allows you to extend Select Access to meet the needs of future security requirements. Because Select Access relies heavily on a component-based multi-tier architecture, it gives administrators and developers a system that is easier to configure and upgrade.

Chapter Overview

Topics in this chapter include subjects that describe Select Access components and how you can build your own plugins to customize Select Access functionality and features to suit your environment and your business needs:

- Select Access Components on page 17
- Customizing Select Access on page 19
- Select Access Source Files and Libraries on page 22
- Building Examples in the SDK on page 23
- Understanding the Importance of XML on page 25

Select Access Components

Before extending Select Access components, you should be familiar with how each of the components works. The four principle components discussed in this guide are the Policy Builder, the Policy Store, the Policy Validator, and the Enforcer plugin.

The Policy Builder is a graphical application used to configure security policies for network resources. The security policy is expressed as a set of rules in a decision tree. There are two types of rules: Select Auth rules and conditional access rules. Figure 1 shows the graphical interface for the Policy Builder and how to access the configuration dialogs for Select Auth rules and conditional rules.

Select Auth rules specify the authentication methods allowed for a network resource. A node in the rule represents an authentication service. The service is configured using Policy Builder authentication plugins, but the service itself is provided by Policy Validator plugin, called authenticators.

Conditional rules specify an authorization policy for a resource. A conditional rule consists of a decision tree with policy nodes at each branch in the tree. The policy nodes are configured using the Rule Builder and Policy Builder authorization plugins. Policy Validator plugins, called evaluators, provide the service that evaluates the policy node when a resource is requested.

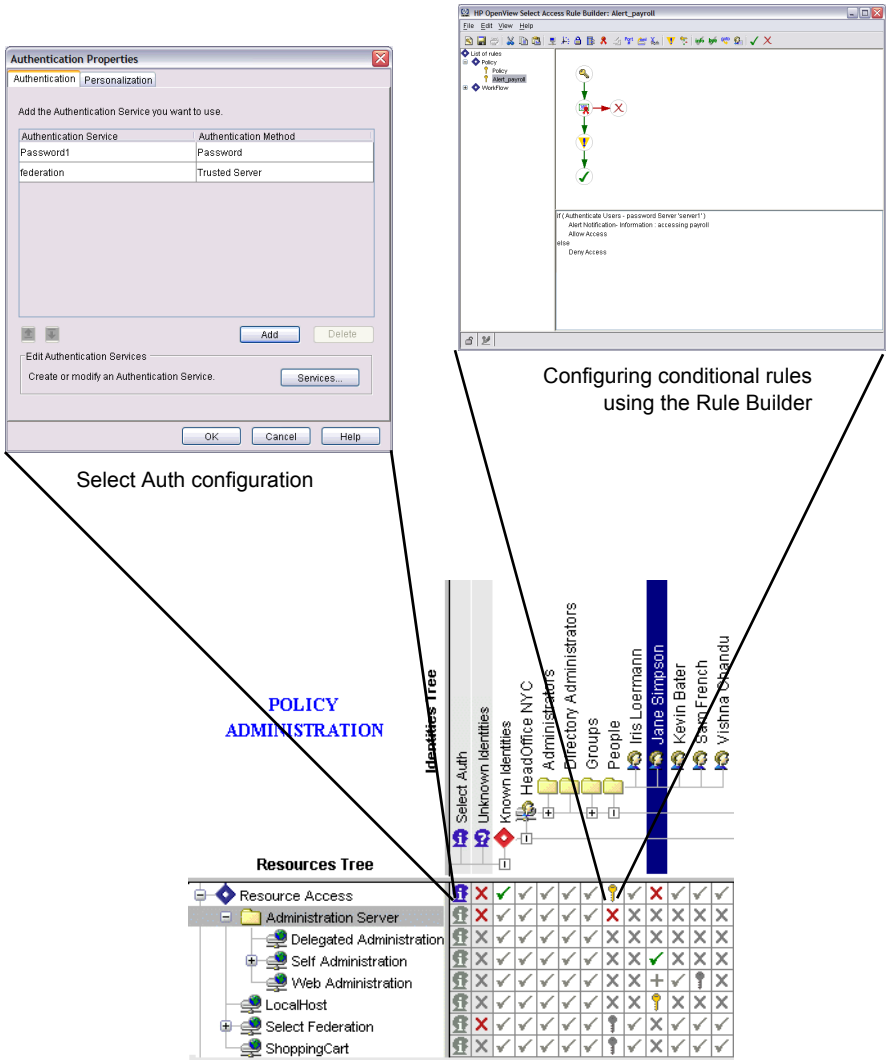

Figure 1 Configuring Select Auth and Conditional Rules

The Policy Builder stores the configured security policy, including the configuration of each of the policy nodes, in the Policy Store. The Policy Validator retrieves the configured security policy, and evaluates resource requests using the configured policy.

Enforcer plugins integrate Select Access security services with custom applications and servers. An Enforcer plugin intercepts a request for a resource, and queries the Policy Validator to determine if the request should be allowed or denied. The Enforcer plugin is responsible for enforcing the decision made by the Policy Validator.

Customizing Select Access

This guide explains how to tailor Select Access components for your network infrastructure and your business requirements. Select Access ships a series of APIs that allow you to customize the functionality of the product. Any additions or enhancements you make to a component, do not require that you reinstall the whole product. You simply need to create the component and plug it in. There are two ways to customize Select Access:

- With SelectAccess APIs
- With other APIs or customization methods

Select Access' Programming APIs

Table 2 describes all of the libraries included with Select Access' SDK. It lists all the APIs discussed in this guide, provides an overview on what you can develop with each of them, as well as describes any file requisites each API has.

Table 2 The Select Access APIs

API Name	What It Develops
Validator API	New authentication/authorization methods. Policy Validator plugins use the C/C++ programming language. A Policy Validator plugin is responsible for: 1 Processing authorization requests from Enforcer plugins. 2 Evaluating the request using the security policy configured by the Policy Builder. 3 Informing the Enforcer plugin what action to take based on the request evaluation. Note: If you are building a custom Policy Validator, you must provide a corresponding Policy Builder plugin to configure your Policy Validator plugin.
Policy Builder API	New configuration panels for Policy Validator decider plugins. Policy Builder plugins are responsible for: 1 Configuring new instances of a Policy Validator plugin instance. 2 Editing existing instances of a Policy Validator instance. 3 Saving the configured state of a Policy Validator plugin instance. Note: If you are building a Policy Builder plugin, you must provide a corresponding Policy Validator plugin before using the Policy Builder plugin in a security rule.

Table 2 The Select Access APIs

API Name	What It Develops
Personalization	Enforcer plugins that provide additional identity data used for personalization. The Personalization API is not a separate API, but rather a cooperation of the above APIs.
Enforcer API	New Enforcer plugins. An Enforcer plugin acts as an agent for Select Access on the web/application server. It is responsible for: 1 Passing details of a identity request to the Policy Validator in the form of an XML query. 2 Enforcing the outcome of the XML query once the Policy Validator has evaluated it. Enforcer plugin allow you to customize the backend application logic for resources. There are three Enforcer API libraries available: • Enforcer C++ API library • Enforcer Java API library • Enforcer COM Interface API library Note: If you are creating your own custom plugin, you need to create a template `enforcer.xml` file using the Setup Tool. This file is created when you run the setup wizard for any Enforcer plugin type. For details, see the *HP OpenView Select Access 6.2 Installation Guide*.
Adminstration API	Used to do the following tasks: • Add, modify or delete service and resource entries • Add, modify or delete policy settings for users and resources using Allow, Deny or existing rules • Allow users to modify their passwords The Administration API is an environment based on a published WSDL for developers working in languages with web service support for externally defining resources and setting policies. It is implemented via web services and is protected by the Axis Enforcer. The Setup tool automatically configures the Administration API environment and the Axis Enforcer.

Customizing with Select Access APIs

There are various components you can customize. To review which areas might pertain to your business situation, review Table 3.

Table 3 Customizing with Select Access APIs

Feature	API Needed	Details
A new method of authentication allows you to extend the authentication services in Select Access. To add a new authentication method requires two plugins: 1 Create a new Policy Builder authentication plugin to configure the authentication service in Select Auth rules. 2 Create an authenticator plugin for the Policy Validator to provide the authentication service.	• Policy Builder API • Validator API	Chapter 3, Custom Configuration GUIs: the Policy Builder API
A new authorization service to support new methods for evaluating access to resources. To add a new authorization service requires two plugins: 1 Create a new Policy Builder authorization plugin to configure the new feature in a conditional rule with the Rule Builder. 2 Create a decision point plugin for the Policy Validator to evaluate requests for resources.	Policy Builder API and the Validator API	Creating a Decision Point Plugin: Directory Attributes Example on page 66.
A new *Enforcer plugin* that allows backend applications and servers to integrate with the Select Access security infrastructure. The plugin must enforce the Policy Validator decisions. Note**:** If you are creating your own custom plugin, you need to create a template `enforcer.xml` file using the Setup Tool. This file is created when you run the Setup wizard for any Enforcer plugin type. For details, see the *HP OpenView Select Access 6.2 Installation Guide*.	Enforcer	Chapter 5, Custom Security Services: the Enforcer API
Personalization data allows backend applications to provide customized end user displays. Note: To make use of personalization data, you must use the Policy Builder to configure the Policy Validator to support personalization.	Validator API and Enforcer API	Chapter 6, Using Personalization Attributes: the Personalization API

If you intend to build a Java Enforcer plugin, you need to find the JAR files you require: for unsigned JAR files, look in `<install_path>/shared/` for signed JAR files, look in `<install_path>/jetty/protected/`.

- If you are upgrading a custom Java plugin from a previous version, you need to migrate it to use libraries from this release. For details, see the *HP OpenView Select Access 6.2 Release Notes*.
- If you intend to build a Java Enforcer plugin, note that a DLL required to build the plugin may be missing from your host computer, depending on which components you have installed on it. `NTEventLogAppender.dll` is only installed with a full installation of Select Access. Copy this file to your local `system32` folder to ensure your custom plugin is built properly.

Select Access Source Files and Libraries

Select Access includes source files and libraries that will help you create your own customized plugins.

You can find the files you'll need to help you create your own customized plugins on the Select Access SDK CD, in the `SelectAccess/source` folder. The SDK also contains a `local_tools` folder. This folder contains the directories and files needed to localize Select Access. You can ignore this folder and its contents.

By default, `SelectAccess/source` folder includes the child folders shown in the table below.

Table 4 Select Access Default Directory Structure

Folder	Description of Contents
`enforcer`	The C/C++ Enforcer API header files.
`include`	Third-party header files, such as ACE, Expat, cURL, OpenSSL etc.
`Java`	The Java source files for three Policy Builder decision point examples (toggle, AttributeLogic, identity filter, and protocol filter), as well as Java query test and Web Administration.
`server_plugins`	The C/C++ source files for the Apache 2 Enforcer plugin and the TCP Enforcer plugin. There are also `site_data` examples for Apache 2, IIS and iPlanet.
`validator/plugins`	The C/C++ source files needed to rebuild the example Policy Validator decision point plugins (toggle, AttributeLogic, identity filter, and protocol filter).
`linux, solaris, and hpux, win32, 64bits/hpux-ia64, 64bits/linux-ia64`	Platform-specific header files and place-holder directories so you can build examples for these platforms.
`query`	Source code for the C++ query program.

Building Examples in the SDK

This section documents the procedures you need to consider to make this SDK release fully functional. Please follow the instructions of these topics closely, to ensure you can build custom plugins using Select Access' libraries:

- To build C++ examples on UNIX on page 23
- To build C++ examples on Windows on page 24
- To build Java examples on Windows or UNIX on page 24
- To build and install the Web Administration interface on Windows or UNIX on page 24

To build C++ examples on UNIX

1. Run the `unixify.sh` script to convert Windows line endings to UNIX line endings.
2. Ensure that your build environment contains the files listed in Table 5. These files must be available from your install path.

Table 5 Build Environment Requirements

File	Required For
`GNU make`[a], for example `gmake <fileName>` Note: For Red Hat AS 4.0, use `make CXX=g++32`, for example `make <fileName>`	The Select Access Makefile system
`gcc v3.3.2 & g++`	Solaris
`aCC: HP aC++/ANSI C B3910B A.05.55 [Dec 04 2003]`	HP-UX (both PA-RISC and IA64)
`gcc version 3.2.3`	RedHat Linux AS 3.0 and 4.0 (both x86 and IA64)
`Visual C++ 6.0`	Windows

a. Place `gmake` before any other version of `make` in your path.

3. Do one of the following:
   - For Apache examples: from the root folder, run `make` to compile that plugin.

     Ignore any "undefined reference" warnings. These can safely be ignored, as they will be resolved at runtime when Apache is started.
   - For Red Hat Enterprise Linux 4: Run `make CXX=g++32` for the plugin.

     Run `make CXX=g++32` to build any of the C++ components in the SDK. This is required to force the use of `gcc 3.2`, making the build consistent with Red Hat Enterprise Linux 3.
   - For all other examples: from the corresponding folder for that plugin, run `make`.

To build C++ examples on Windows

1. Because the Select Access SDK does not include the project files needed to build with Visual Studio .NET, ensure your build environment contains Visual Studio 6.0, Service Pack 5.
2. Load the `SelectAccess.dsw` workspace in the main directory.
3. Choose `_allsdk` as the project.
4. Click **Build** → **Set Active Configuration**.
5. Set **Configuration Release**.
6. Click **Build** → **Rebuild All** to recompile all of the examples.

   Ignore any symbolic name warning messages when compiling the File Authenticator example. These may safely be ignored.

7. To rebuild a particular example only, right-click the corresponding project in Visual Studio and click **Build** (selection only).

To build Java examples on Windows or UNIX

1. Make sure that your build environment contains:
   - Both `javac` (the Java compiler) and `java` (the Java runtime compiler). Java SDK version 1.4.2_10 includes both.
   - The Ant build tool. 1.6.1, which is available from `http://ant.apache.org`.
2. Build the examples with the `build.xml` file in the Java directory. For details, see To build and install the Web Administration interface on Windows or UNIX below.
3. To rebuild a particular example, give the example's name to Ant on the command line. For example, at a Command Prompt type `ant toggle` or `ant FileAuthenticator`.

To build and install the Web Administration interface on Windows or UNIX

1. Modify the Java source files.
2. Open a DOS command or UNIX shell window and do the following:
   a. cd to the `<SDK_install_path>/SelectAccess/source/Java` folder.
   b. At the prompt, type **`ant`** to compile the Java source files.

      After a successful build, the entire customized Web Administration is packaged in the `deploy` folder.

      Because you will be overwriting files installed by the Select Access installer in the next step, HP recommends that you first backup all original files in the `<SA_install_path>/shared/jetty/policy_builder/webadmin` folder. This allows you to revert files if you need to. For example, you could rename the `webadmin` folder to `webadmin.orig`.

   c. Copy and replace all the contents of the `<SA_install_path>/shared/jetty/policy_builder/webadmin` folder with the customized ones you have packaged in the `webadmin` folder of the `deploy` staging directory.
3. Restart the Jetty web server/Administration server for the servlet deployment to take effect.

Understanding the Importance of XML

SelectAccess uses XML for both data storage and communication. Therefore it is important you understand how XML is used by the software before you can customize Select Access components. XML is instinctively readable, which makes it easy to understand and extend the system, especially when XML documents are used with an XML editor.

How Rules are Expressed and Stored

Because XML is inherently tree structured, it integrates well with Select Access' binary decision tree rules. Rules, and the authenticators and decision points that comprise them, are expressed as XML objects and stored in the directory server. Each Policy Validator plugin has a corresponding Policy Builder plugin that provides a property editor. The property editor translates the properties to XML. This XML is nested inside the XML for the rule as a whole, which is then written back to the directory server's LDAP database. There are no restrictions on the form or the types of data that may be used. This extends the authentication and authorization engine to meet the tailored needs of individual organizations.

How Data is Encoded and Communicated

Data is encoded in XML and communicated between the Enforcer plugin and a Policy Validator (queries and responses). This allows for complete extensibility with respect to adding new information.

- Queries are simply a set of attribute-value pairs, new attributes with values can be arbitrarily added to a query. For example, `<PROPERTY NAME="service">http://mycompany.com:8080</PROPERTY>`.
- Responses are also XML-based and come with information embedded. This information can be in the form of:
  - Cues to tell the Enforcer plugin to gather more data upon which the response may be reconsidered.
  - Data returned to the application about the identity making the query. This kind of information can be used to provide a personalized experience for that identity, for that resource.

The flexibility of the XML-based plugin architecture allows Select Access to enforce any kind of decision. For example, read only access can be enforced by making decisions based on the commands entered by the identity on a given service (for example, `HTTP method=GET` versus `POST` or `FTP` command `GET` versus `PUT`).

The XML-based plugin architecture can best be illustrated via Figure 2.

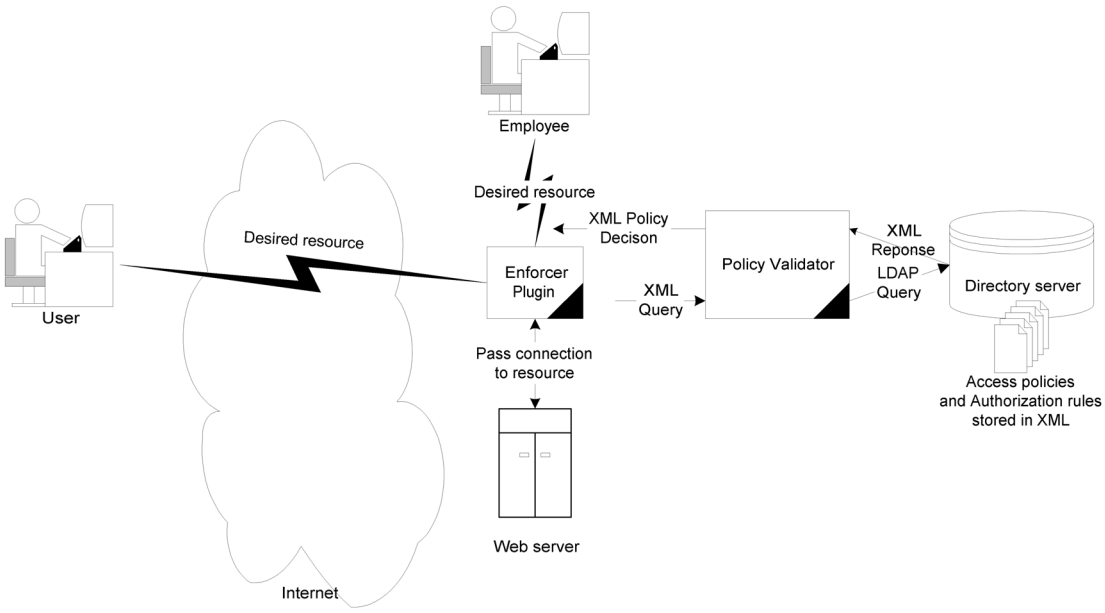

Figure 2 Select Access Plugin Architecture

Scenario: Identity Requesting access to mycompany.com

The power of XML can be seen in the following scenario: imagine you have an identity who is unknown and he is requested access to `HTTP://mycompany.com`. The Enforcer plugin sends a query to the Policy Validator to determine the identity's access entitlements for `HTTP://mycompany.com`. The Policy Validator initially sends a conditional access response requesting that the unknown identity registers. The unknown identity registers and all the identity information collected is encoded in the XML query and is mapped directly as attributes in the identity's new directory entry/identity profile.

Table 6 summarizes what the subsequent code examples demonstrate.

Table 6 Query/Response Examples

This code example ...	Demonstrates this ...
Sample query to the Policy Validator on page 26	The first query from the Enforcer plugin to the Policy Validator requesting access to the resource.
Second sample query to the Policy Validator on page 27	The second query from the Enforcer plugin to the Policy Validator with the registration details included.
Example of a simplified Policy Validator reply on page 28	The simplified Policy Validator reply that allows known identity Steve Smith access.

Sample query to the Policy Validator

The following code sample is an example of a query to the Policy Validator:

```
<PolicyValidatorQuery>
<PROPERTYLIST>
   <PROPERTY NAME="service">http://mycompany.com:8080</PROPERTY>
   <PROPERTY NAME="path">/secure/form</PROPERTY>
   <PROPERTY NAME="srcIP">10.10.10.6</PROPERTY>
    <PROPERTY NAME="srcPort">1107</PROPERTY>
    <PROPERTY NAME="dstIP">10.10.10.7</PROPERTY>
    <PROPERTY NAME="dstPort">8000</PROPERTY>
    <PROPERTY NAME="native_auth">native_password</PROPERTY>
    <PROPERTY NAME="native_auth">native_register</PROPERTY>
   <PROPERTY NAME="site_data">time: TueJul1013:14:402001</PROPERTY>
   <PROPERTY NAME="owner">gandalf</PROPERTY>
   <PROPERTY NAME="size">268</PROPERTY>
   <PROPERTY NAME="method">GET</PROPERTY>
      </PROPERTYLIST>
   <PROPERTY NAME="queryID">1</PROPERTY>
 </PolicyValidatorQuery>
```

Second sample query to the Policy Validator

The following code sample is an example of a Policy Validator query:

```
<PolicyValidatorQuery>
    <PROPERTY NAME="queryID">10.10.10.6:1107</PROPERTY>
    <PROPERTY NAME="service">http://mycompany.com:8080</PROPERTY>
    <PROPERTY NAME="path">/secure/form</PROPERTY>
    <PROPERTY NAME="srcIP">10.10.10.6</PROPERTY>
    <PROPERTY NAME="srcPort">1107</PROPERTY>
    <PROPERTY NAME="dstIP">10.10.10.7</PROPERTY>
    <PROPERTY NAME="dstPort">8000</PROPERTY>
    <PROPERTY NAME="native_auth">native_password</PROPERTY>
    <PROPERTY NAME="native_auth">native_register</PROPERTY>
    <PROPERTYLIST NAME="registration">
        <PROPERTY NAME="givenName">Steve</PROPERTY>
        <PROPERTY NAME="sn">Smith</PROPERTY>
        <PROPERTY NAME="o">HP Inc.</PROPERTY>
        <PROPERTY NAME="mail">Ssmith@mycompany.com</PROPERTY>
        <PROPERTY NAME="telephonenumber">555-555-555 ext 555</PROPERTY>
        <PROPERTY NAME="facsimileTelephoneNumber">416-555-2399
        </PROPERTY>
        <PROPERTY NAME="userPassword">letMeIn4now</PROPERTY>
        <PROPERTY NAME="preferredLanguage">Security, LDAP, XML, SSL
        </PROPERTY>
        <PROPERTY NAME="cn">Steve Smith</PROPERTY>
    </PROPERTYLIST>
    <PROPERTY NAME="method">POST</PROPERTY>
</PolicyValidatorQuery>
```

Example of a simplified Policy Validator reply

The following code sample is of a simplified reply:

```
<PolicyValidatorReply>
<PROPERTYLIST>
   <PROPERTY NAME="queryID">1</PROPERTY>
   <PROPERTY  NAME="authenticated_dn">uid=ssmith,ou=users,ou=steve,
   o=ca.mycompany.com</PROPERTY>
   <PROPERTY NAME="devo_groups">users</PROPERTY>
   </PROPERTYLIST>
   <PROPERTY
NAME="nonce">bWFub3dhci5jYS5iYWx0aW1vcmUuY29tOjk5ODh8dWlkPW
   dhbmRhbGYsb3U9dXNlcnMsb3U9c3RldmUsbz1jYS5iYWx0aW1vcmUuY29tf
   Ex8cGFzc192YW
   xpZGF0b3J8O0zd4DuLSR2cw0K7Yp5pTb04TSWWxJJqqc7rUszF16atYOqTLnVqt
   +O4a6OFr
   WQegqu89L17Kwzv4n3XWIwFpsP+wJ8V1eEw4KG9c+REkJFs67bLKZuWZ6xNz2Xpgs7
   FLb5s2
   O3iwswvuDvcBNOZqF2pbOproktsiuaC36SqxmKLSzY/</PROPERTY>
   <PROPERTY NAME="action">ALLOW</PROPERTY>
 </PolicyValidatorReply>
```

3 Custom Configuration GUIs: the Policy Builder API

Select Access enables developers to provide additional Policy Builder configuration editors. This enables administrators to configure custom authentication methods and authorization rules. This chapter describes how to use the Policy Builder API to create configuration editors, called configurators, that integrate with the Select Access Policy Builder.

Chapter Overview

Topics in this chapter include subjects that describe the Policy Builder, its API, and the plugins you can build with this API to customize the Policy Builder's behaviour:

- Understanding the Policy Builder on page 29
- Understanding the Policy Builder API on page 30
- Creating a Policy Builder Plugin on page 34
- Installing Your Policy Builder Plugin on page 39
- Removing Policy Builder Plugins on page 40

Understanding the Policy Builder

The Policy Builder is a graphical application that allows administrators to manage access to network resources. Administrators use the Policy Builder to manage a resource's authentication methods and authorization rules. The Policy Builder configures a security policy. The Policy Validator validates resource requests against the configured policy using Policy Validator plugins. Every Policy Validator plugin needs a Policy Builder plugin to configure it. Typically you will develop both a Policy Builder plugin and a Policy Validator plugin to work together.

Do not use your Policy Builder plugin to protect a resource until you have created the corresponding Policy Validator plugin. This will cause the Policy Validator to fail the first time the access rule is evaluated. For details on creating Policy Validator plugins, see Chapter 4, Custom Authentication Methods and Rules: the Validator API.

Figure 3 on page 30 illustrates how the Policy Builder works with the Policy Validator to extend authentication and authorization services. The Policy Validator plugins provide additional authentication and authorization services. The Policy Builder is used to configure those services in a security policy. The Policy Builder API allows developers to create customized configuration panels to manage the Policy Validator plugin instances. When the Policy Builder needs to configure a Policy Validator plugin instance, the Policy Builder API displays the appropriate configuration panel. After the Policy Validator plugin instance is configured, the configured state is stored in the Policy Store by the Policy Builder API.

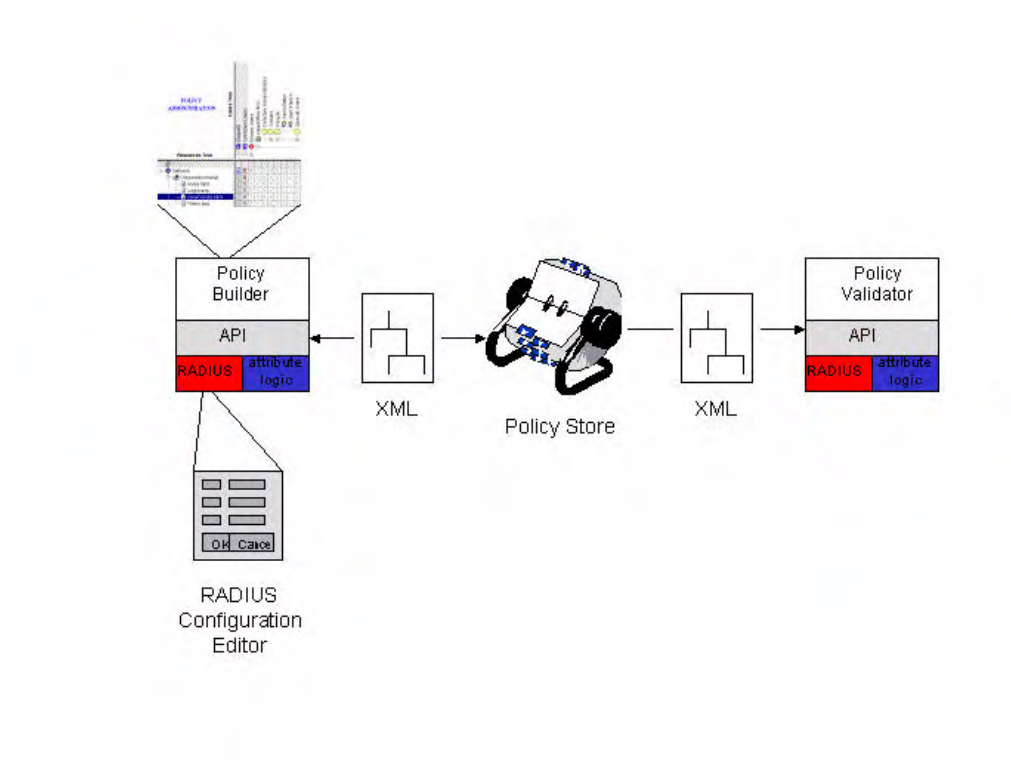

Figure 3 How Policy Builder Plugins Work with Select Access Components

Understanding the Policy Builder API

The Policy Builder API enables developers to create Java-based graphical components to configure custom authentication methods and custom authorization policy nodes. The same API is used for both authentication and authorization plugins. In theory, a single Policy Builder plugin could function as both an authentication and an authorization configuration editor. In practice however, most authentication and authorization plugins require very different configuration properties.

Types of Policy Builder Plugins

There are two types of Policy Builder plugins, authentication plugins and authorization plugins. Authentication plugins configure instances of Policy Validator authenticators. Authorization plugins configure instances of Policy Validator decision points. For details on decision points, see Creating a Decision Point Plugin: Directory Attributes Example on page 66.

Authentication plugins

Authentication plugins enable administrators to configure authentication services used by Select Auths.

To determine which Authentication plugins are installed

1. From the Policy Builder, click **Tools** → **Authentication Services**. The **Authentication Services** dialog box appears.
2. Click **Add** to display the **Authentication Method** dialog box, as shown in Figure 4 on page 31. This dialog displays a radio button for each authentication plugin already uploaded.

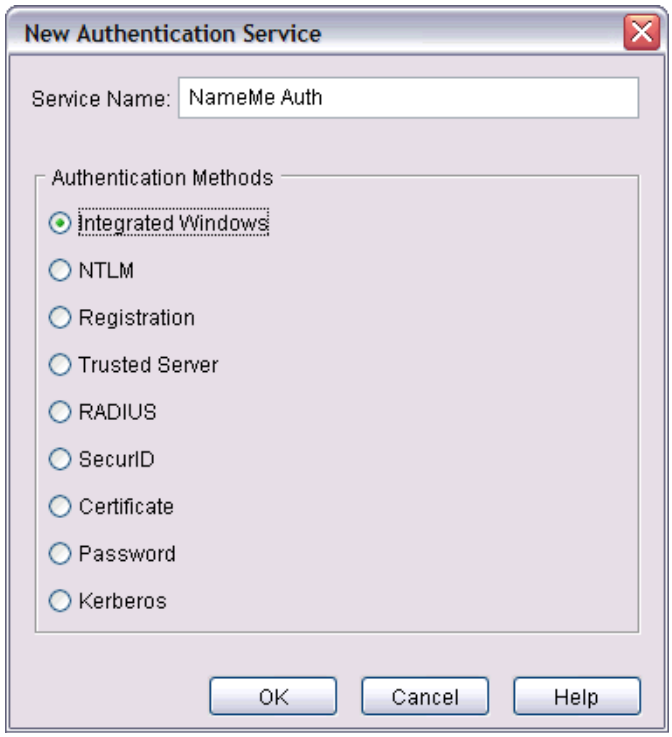

Figure 4 Authentication Methods

Authorization plugins

Authorization plugins enable administrators to configure rule nodes in an authorization rule. To determine which authorization plugins are available, start the Rule Builder by selecting **Tools** → **Rule Builder** from the Policy Builder. The Rule Builder toolbar displays an icon for every authorization plugin installed, as shown in Figure 5.

Authorization plugins also have a large, Rule Builder icon that is displayed when the decision point is used in a rule. To determine which authorization plugins are used in an authorization rule, select the rule from the **List of Rules**.

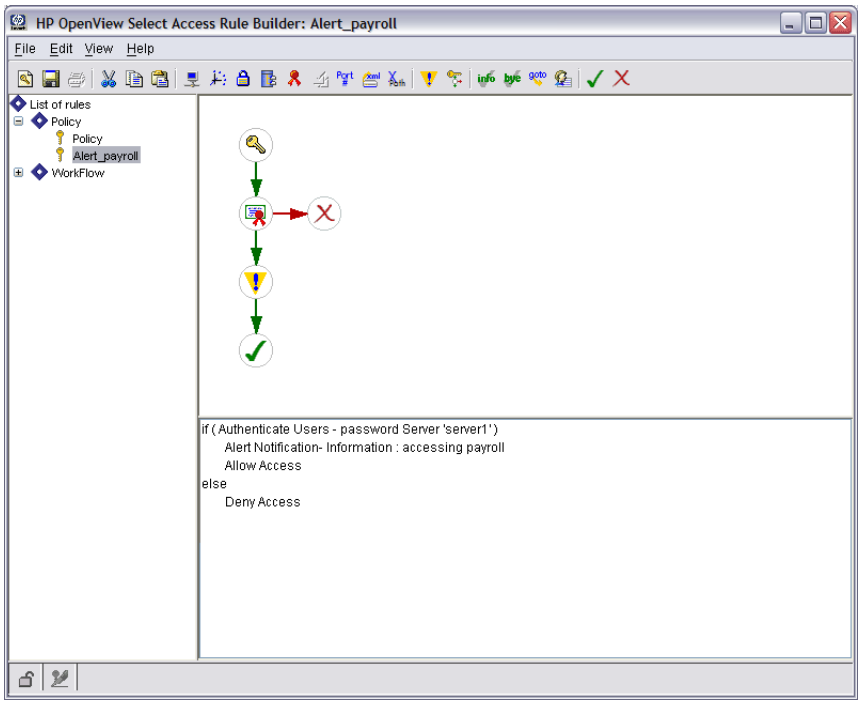

Figure 5 Rule Builder

Notice that in Figure 5, each rule node has zero, one, or two connector branches, indicated by red and green arrows. A connector branch is used to connect rule nodes together to construct an authorization rule. The type of configuration editor determines the number of connector branches that are displayed.

The authorization rule shown in Figure 6 illustrates the four different types of authorization plugins:

- Subrules: Subrules are used to connect one rule to another. This configurator type is reserved. You should not develop subrule configurators. Subrules do not have connectors.
- Filters: Filters perform some action, but does not alter the flow of the authorization rule. Filters have only one connector, the `true` branch. The Alert Notification configurator is an example of a filter.
- Decision points: Decision points configure an decider plugin that performs conditional access. A decision point represents a conditional branch in the authorization rule. Each decision point has two connectors, `true` and `false`. The Evaluate Directory Attributes and Authenticate Identities configurators are examples of decision points.
- Terminal points: Terminal points end the processing of an authorization rule. Terminal point configurators are further classified into two categories, allow and deny. Terminal points have no connectors. The Logout and Redirect configurators are examples of Terminal points.

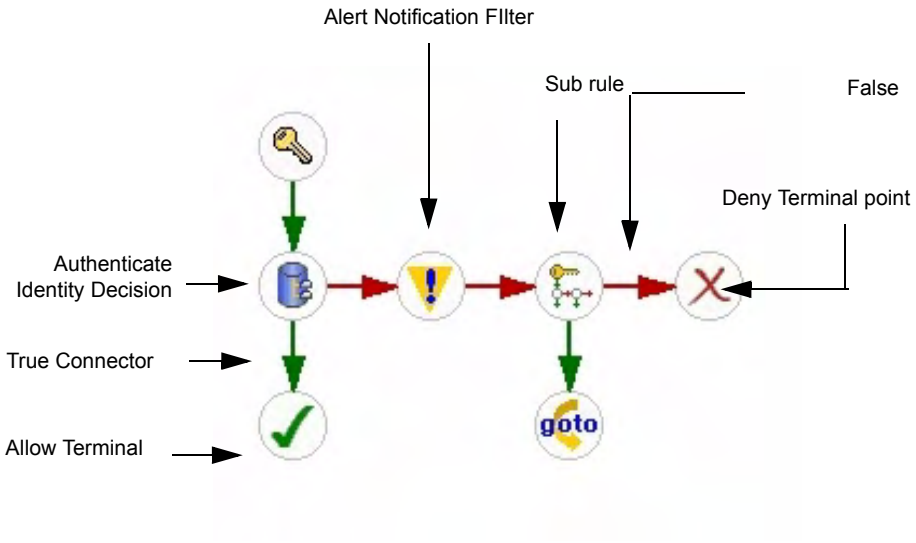

Figure 6 Authorization Rule

Creating customized filters, Terminal points, and subrules is a topic beyond the scope of this document.

How the Policy Builder API Works

When you develop to the Policy Builder API, you are essentially creating a `JPanel` that will be displayed by the Policy Builder. This provides developers the greatest flexibility when designing configuration editors.

The Policy Builder displays configuration editors in a `JDialog` that contains the standard buttons: **OK**, **Cancel**, and **Help**. Details on how to override callback methods are provided in section Creating a Configuration Editor on page 35.

The Policy Builder API retrieves the current configuration state from the Policy Store. The XML properties are available in the class field `m_properties`. A configuration editor should reflect the state configured in this field. When the **OK** button is pressed, the configuration editor should set the `m_properties` field to reflect the new configuration. The Policy Builder automatically stores the XML configuration in the Policy Store when the configuration editor exits normally.

Policy Builder API Classes and Utilities

The core of the Policy Builder API is the `RuleComponentPanel` class. This class extends the standard Java `JPanel` and encapsulates communication with the Policy Store. All configuration editors must extend the `RuleComponentPanel` class.

The `RuleComponentPanel` class extracts the appropriate XML configuration for your plugin and stores it in the inherited field, `m_properties`. This field is of type `Property`, described below. The `m_properties` field stores the configured state of the plugin instance. When your configuration editor exits normally, your plugin should update the `m_properties` variable to reflect the new configured state. The Policy Builder API automatically stores the contents of the `m_properties` field. The XML document contained in this field is stored back in the Policy Store.

The Policy Builder API uses XML properties to store plugin configurations. To access, manipulate, and save configurations, you must use the Policy Builder API XML classes located in the `com.hp.selectaccess.util` package:

- `Property`: The `Property` interface is implemented by `PropertyElement` and `PropertyListElement`.
- `PropertyElement`: The `PropertyElement` class represents an XML element that stores a name-value pair. The value must be a `String`.
- `PropertyListElement`: The `PropertyListElement` class represents a list of `PropertyElements` and nested `PropertyListElements`.

For more information on XML properties, see the *HP OpenView Select Access 6.2 Developer's Reference Guide*.

The Policy Builder API also includes several other classes in the `com.hp.selectaccess.util` package that many plugins will need to use:

- `CDN`: The `CDN` class represents a canonical Distinguished Name (DN). Since the syntax of a DN allows the same name to be defined in multiple ways, the `CDN` class ensures that the DN is normalized. The `CDN` class provides methods to obtain a normalized DN and check whether one DN is located within the name space of another DN. This class is used to configure the directory location for identity objects and the cache location for transient profiles. Details on transient profiles are provided in section LdapConnection, User, UserSource, and UserCache on page 72.
- `TableSorter` and `TableUtil` are useful if you plan to use a table model like the one used to support RADIUS server configurations.

Creating a Policy Builder Plugin

Creating a Policy Builder plugin is easy. Building a configuration editor is not much more complicated than creating a standard `JPanel`.

To create a new Policy Builder plugin

1. Create a configuration editor to configure a Policy Validator plugin. See Creating a Configuration Editor on page 35.
2. If developing an authorization plugin, create the plugin icon and toolbar icon. See Creating Icons for Authorization Plugins on page 37.
3. Create a `component.xml` file to configure your plugin. See Creating a component.xml File on page 37.
4. Load your plugin. See Installing Your Policy Builder Plugin on page 39.
5. Test your plugin. Verify that the graphical components operate properly and that the configured state is properly stored and retrieved.

Creating a Configuration Editor

Before you create a configuration editor, you should be familiar with the configuration requirements of the corresponding Policy Validator plugin. Your configuration editor will have to provide a graphical interface to configure each field. The Policy Builder will display your configuration editor in a `JDialog` when an administrator configures or adds your custom component to a rule.

Guidelines for creating a custom editor

Your configuration editor must:

- Display an administrative graphical interface to configure a Policy Validator plugin.
- Retrieve the current configuration and initialize the graphical interface to reflect the configuration.
- Process end user interaction. Depending on the configuration editor you create, you may need to process buttons, update fields, pop-up windows, etc.
- Validate and save the configuration state when the **OK** button is pressed.

To create a Configuration editor

1. Add the `java archives` *`<SDK_install_path>`*`/shared/PolicyBuilder.jar` and *`<SDK_install_path>`*`/shared/jetty/policy_builder/protected/shared.jar` to the `CLASSPATH` setting for your development environment.
2. Create a class that extends `RuleComponentPanel`. Add text fields, browsers, buttons, and other graphical components necessary to configure the Policy Validator plugin.
3. Override the `RuleComponentPanel` methods. These methods are described below.
4. Compile the class.
5. Package all the class files in a `jar` file named `component.jar`.

When the Policy Builder displays your configuration editor, it places your panel into a base window that contains **OK**, **Cancel**, and **Help** buttons. The `RuleComponentPanel` class loads the XML configuration from the Policy Store and places the XML document in a class member variable, `m_properties`. To ensure proper interaction with the `RuleComponentPanel` class, you should override the following methods:

- `initialize()`: The Policy Builder API calls the `initialize()` method after initializing `m_properties` with the XML configuration. You should override this method to process the XML `configuration` in `m_properties`, validate the configuration data, and set the state of the graphical interface to reflect the configuration.
- `okClicked()`: The `okClicked()` method is called by the Policy Builder API when the administrator clicks the **OK** button after configuring the configuration editor. Your plugin should override this method to validate the configured state of the graphical interface and store the properties back into the `m_properties` variable. When this method returns `true`, the `RuleComponentPanel` class stores the `m_properties` variable in the Policy Store and exits. When this method returns `false`, the `RuleComponentPanel` class does not exit. This allows the configuration editor to force the administrator to correct any improperly configured fields.

- `helpClicked()`: When the **Help** button is pressed, the Policy Builder API calls the `helpClicked()` method. You may optionally override the `helpClicked()` method to provide administrators help with using your configuration editor. You may provide a pop-up window with help text, or call an external class to provide a help window. Using a language index with an external help class allows you support internationalization.
- `cancelClicked()`: The **Cancel** button is automatically processed by the `RuleComponentPanel`. The Policy Builder reverts to the original configuration and exits. You do not need to override this method unless there is some special processing your plugin requires before the Policy Builder processes the **Cancel** request.

The following methods are useful when initializing or sizing certain types of graphical components that can not be initialized before the screen is displayed. For this reason, some plugins may choose to override the following methods in place of the `initialize()` method:

- `handleWindowOpened()`: This method is called only when the screen is displayed for the first time.
- `handleComponentShown()`: This method is called by the Policy Builder API when the screen becomes visible.

Sample Policy Builder plugin framework on page 36 shows a sample framework for a Policy Builder plugin.

Sample Policy Builder plugin framework

The following code sample gives you an example of how to build a plugin framework.

```
import com.hp.selectaccess.rulebuilder.RuleComponentPanel;
import com.hp.selectaccess.util.CDN;
import com.hp.selectaccess.util.Property;
import com.hp.selectaccess.util.PropertyListElement;
import com.hp.selectaccess.util.PropertyElement;

... other imports ...

public class SamplePropertyEditor extends RuleComponentPanel implements
ActionListener
{
    ... constructor with graphical widgets ...

    // Overload Policy Builder API methods

    // Initialize the GUI
    public boolean initialize(){ ... }

    // Save the GUI state to XML
    public boolean okClicked() { ... }

    // Provide Help
    public boolean helpClicked() { ...}

    // Do not need to overload the Cancel button
```

```
    ... GUI methods to handle user interaction ...
}
```

Creating Icons for Authorization Plugins

For Rule Builder plugins, you will need to provide two GIF icons for your plugin, a tool bar icon and a rule pane icon. Tool bar icons are 16x16 pixels and rule pane icons are 32x32 pixels. The icons may use 256 colors.

Creating a component.xml File

The `component.xml` file serves two purposes. It tells the Policy Builder how to configure your plugin, and it contains a property list that is passed to new instances of your configuration editor. The properties contained in the property list may be used to configure default values for your configuration editor. The `component.xml` file serves as a template for the plugin configuration property values.

You do not need to create a document type definition (DTD) for the `component.xml` file.

Guidelines for creating a component.xml file

To create a `component.xml` file you must include three parts:

- A line specifying compliance with XML version 1.0.
- A component element using attributes defined in Table 7.
- A property list, with optional properties, used to configure default values for new instances of your configuration editor.

The following table describes the attributes of the `COMPONENT` element in the `component.xml` file.

Table 7 Component Tags Attributes

Attribute Name	Description
`NAME`	Required. This string uniquely identifies the plugin for use in the Policy Builder. Each plugin must have a unique name.
`DESCRIPTION`	Required. This string description is the label displayed to administrators. For decision point plugins, the description is displayed as a tooltip in the Rule Builder toolbar. For authentication plugins, the description is displayed as the radio button option in the **Authentication Method** panel.
`TYPE`	Required. Indicates the type of plugin. Supported values include: • `authenticate` • `decision` • `filter` • `allow` • `deny` • `subrule` For more information on Policy Builder types, see Types of Policy Builder Plugins on page 30.

Table 7 Component Tags Attributes (cont'd)

Attribute Name	Description
CONDITION	Required. Use the value "any". This attribute is used internally when the plugin is incorporated into a Policy Builder rule.
CONFIGURATOR	Required. This string tells the Rule Builder what configuration editor to use for the plugin. This value must contain the full name of the Java class. For example, "com.hp.selectaccess.rulebuilder.screens. RadiusPanel".
EVALUATOR	Required for authorization plugins. This string tells the Policy Validator which decision point plugin to use when validating this authorization rule. For more information on Policy Validator plugins, see Chapter 4, Custom Authentication Methods and Rules: the Validator API.
AUTHENTICATOR	Required for authentication plugins. This string tells the Policy Validator which authenticator plugin to use. For more information on Policy Validator plugins, see Chapter 4, Custom Authentication Methods and Rules: the Validator API.

Sample RADIUS component.xml below shows the component.xml file for the RADIUS plugin. The property values are used to provide default values when adding a new RADIUS server to the plugin configuration.

Sample RADIUS component.xml

```
<?xml version='1.0'?>
<COMPONENT TYPE="authenticate"
    CONDITION="any"
    NAME="RADIUS"
    DESCRIPTION="RADIUS"
    AUTHENTICATOR="radius"
    CONFIGURATOR="com.hp.selectaccess.rulebuilder.screens.RadiusPanel">
    <PROPERTYLIST NAME="radius">
        <PROPERTY NAME="defaultLoginForm">login_form.html</PROPERTY>
        <PROPERTY NAME="defaultChallengeForm">radius_form.html
        </PROPERTY>
        <PROPERTY NAME="defaultIP">127.0.0.1</PROPERTY>
        <PROPERTY NAME="defaultPort">1821</PROPERTY>
        <PROPERTY NAME="defaultTimeout">3</PROPERTY>
        <PROPERTY NAME="defaultRetry">3</PROPERTY>
    </PROPERTYLIST>
</COMPONENT>
```

Installing Your Policy Builder Plugin

Once you have completed your Policy Builder plugin, you will need to install it into the Policy Builder.

To install your Policy Builder plugin

1. Package your configuration editor classes in a Java Archive (JAR) file. The JAR file must be named `component.jar`. You can create the archive file by running `jar` on UNIX systems, or `jar.exe` on Windows systems. For more information on how to create Java archives, please see the documentation for your Java programming system.
2. Place your `component.xml` and `component.jar` files in the same directory. You may use a temporary directory. If your plugin is an authorization plugin also place the `toolbaricon.gif` and `icon.gif` files in the directory.
3. Load your plugin into the Policy Builder by clicking **Tools** → **Configure Policy Plugins**. The Policy Builder will prompt you for the plugin location as shown in Figure 7. After you upload your plugin, administrators will be able to use your configuration editor to configure authorization rules or authentication services. If you upload a directory, all the plugins contained in the directory will be installed.

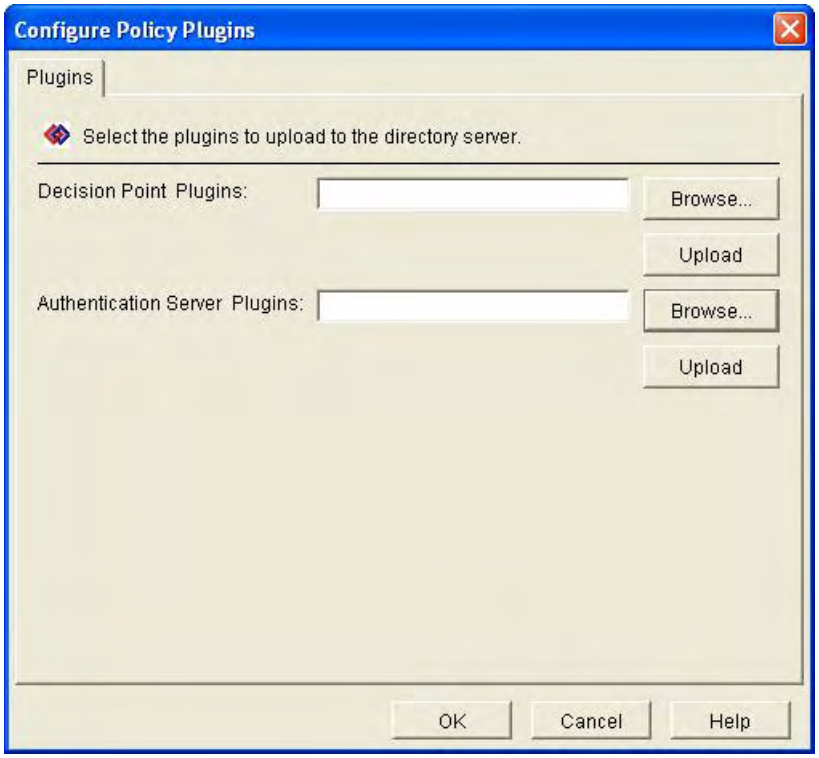

Figure 7 Uploading New Policy Builder Plugins

4. Once the plugin is stored in LDAP, you may delete the plugin files and directory.

The Policy Validator cannot evaluate identity access until you have created and installed the corresponding Policy Validator plugin.

Removing Policy Builder Plugins

Under normal conditions, you do not need to remove Policy Builder plugins. You can replace the old version with the new simply by uploading the new version. For rare cases, you may need to remove old instances of the plugin. Select Access does not provide a mechanism for removing Policy Builder plugins; uninstalling a plugin is a manual process.

The steps to remove a plugin require directly manipulating LDAP data, and assume the reader is familiar with modifying LDAP data. Removing plugins should not be performed on a production system unless absolutely necessary.

Hewlett-Packard does not recommend removing Policy Builder plugins from production systems unless absolutely necessary. Failure to follow the steps below could result in LDAP data integrity issues, which may result in inconsistent Policy Validator behavior.

To remove the Policy Builder plugin

1. Backup the directory server database.
2. Review all access rules and Select Auth rules. Remove the plugin from all rules. If you are removing a authorization plugin, remove the component from all access control rules using the Rule Builder. If you are removing an authentication plugin, remove the component from all Select Auth rules using the Policy Builder.
3. Locate your plugin entry in the directory server. The plugin will be located relative to the LDAP branch where you installed Select Access. Figure 8 illustrates the LDAP location of the Select Access components. The product was installed in the LDAP branch "`ou=SelectAccess_5.1, ou=Applications, dc=baltimore, dc=com`" authentication plugins are located in the "`ou=authenticationmethods, ou=securitypolicy`" subtree and authorization plugins are located in the "`ou=rulecomponents, ou=securitypolicy`" subtree. If you do not have a graphical LDAP browser, you can locate the RADIUS plugin, use the following LDAP URL: "`ldap://localhost:389/ou=authenticationmethods, ou=securitypolicy, ou=SelectAccess_5.1, ou=Applications, dc=hp, dc=com?sub?(nxPolicyComponent=RADIUS)`".

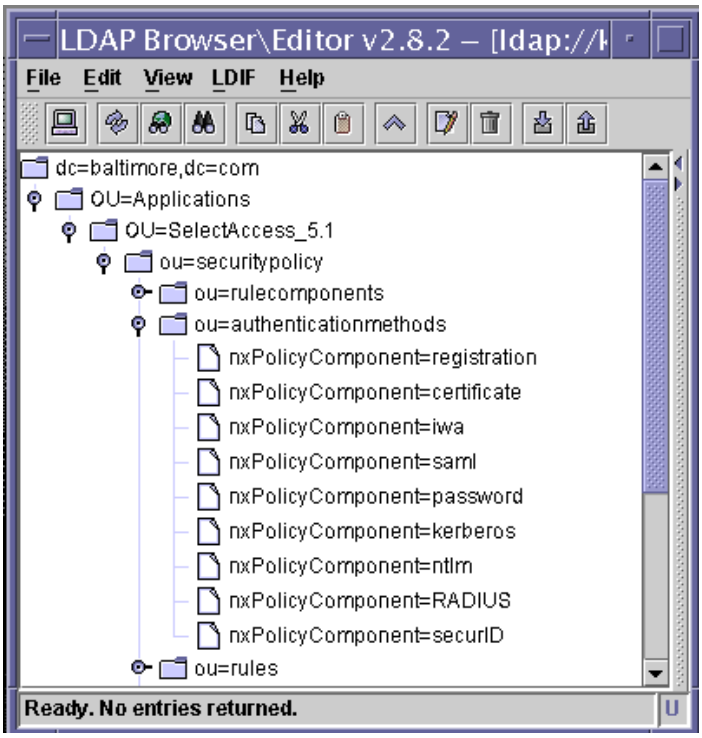

Figure 8 Locating the LDAP Entry for a Policy Builder Plugin

4 Browse the directory object's properties to make sure you have selected the correct entry. The `nxPolicyComponent` attribute should be set to the name of the plugin. You will also see the `config.xml` file for your plugin stored in the `nxXml` attribute. If you do not have a graphical browser, you may have to decode the attribute value before seeing the XML. Figure 8 shows a typical LDAP entry for an authentication plugin.

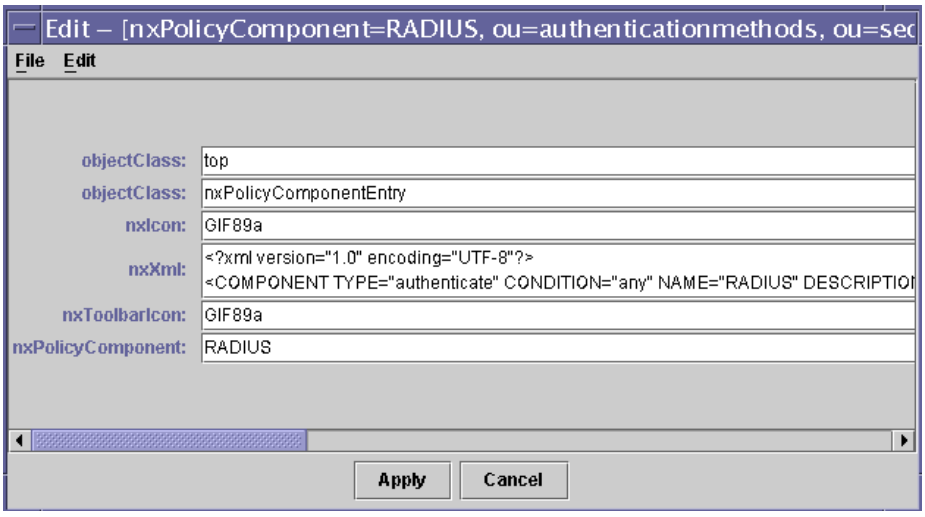

Figure 9 LDAP Entry for a Policy Builder Plugin

5 Delete the LDAP entry.

6 Refresh the Policy Validator caches. From the Policy Builder, select **Tools → Clear Validator Cache**.

7 Restart the Policy Builder and verify that the plugin no longer appears in the Rule Builder menu bar or in the list of available authentication services. Your plugin and the associated icons, if any, should now be removed from the system.

Verify that the remaining policies are still working by accessing protected resources.

8 Remove any equivalent Policy Validator. For details, see Removing Policy Validator Plugins on page 51.

4 Custom Authentication Methods and Rules: the Validator API

Select Access allows you to extend the Policy Validator to support custom authentication methods and authorization rules. This chapter describes how to use the Validator API to create authentication and authorization plugins.

Chapter Overview

Topics in this chapter include subjects that describe the Policy Validator, its API, and the plugins you can build with this API to customize the Policy Validator's behaviour:

- Understanding the Policy Validator on page 43
- Understanding the Validator API on page 44
- Creating a Policy Validator Plugin on page 50
- Installing Policy Validator Plugins on page 51
- Removing Policy Validator Plugins on page 51
- Creating an Authentication Plugin: File-based Authentication Example on page 52
- Creating a Decision Point Plugin: Directory Attributes Example on page 66

Understanding the Policy Validator

The Policy Validator is a service that authenticates identities and authorizes access to resources. Enforcer plugins, such as the Apache 2 Enforcer plugin, query the Policy Validator to authenticate an identity and determine whether access is allowed to the requested resource. The Policy Validator evaluates this request using the security policy configured by the Policy Builder.

You will not be able to configure Select Access to use your Policy Validator plugin until you create a Policy Builder configuration editor for your plugin. For more information on building Policy Builder plugins, see Chapter 3, Custom Configuration GUIs: the Policy Builder API.

Figure 10 on page 44 illustrates how the Policy Validator works with the Policy Builder and Enforcer plugins to extend authentication and authorization services. Policy Validator plugins provide additional authentication and authorization services. The Policy Builder is used to configure those services in a security policy, which is stored in the Policy Store. When the Policy Validator creates plugin instances to evaluate policy nodes, the Policy Validator initializes the plugin using the configured state stored in the Policy Store.

After the plugins have been initialized, the Policy Validator can evaluate resource requests from Enforcer plugins. The Policy Validator may instruct the Enforcer plugin to allow access, deny access, prompt the identity to authenticate, or take some other action. The Policy Validator and Enforcer plugins communicate using XML.

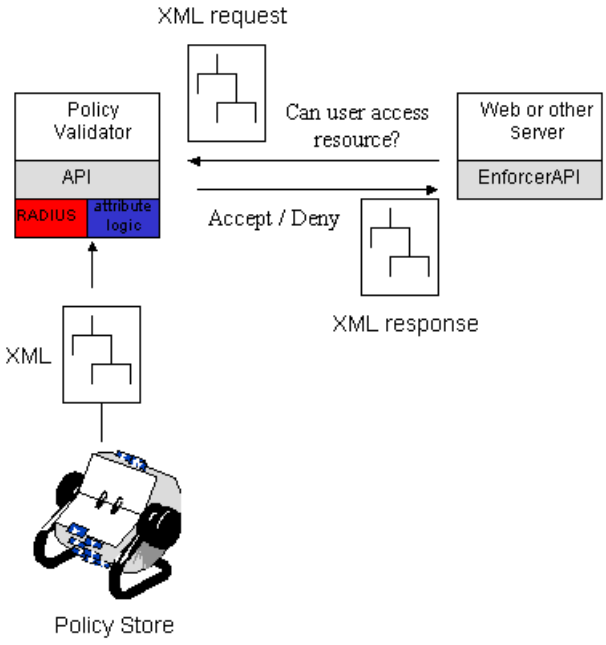

Figure 10 How Policy Validator Plugins Work with Select Access Components

Understanding the Validator API

The Policy Validator evaluates resource access requests using the security policy configured by the Policy Builder. The security policy is a decision tree consisting of policy nodes. Each policy node is implemented by a Policy Validator plugin. Policy Validator plugins provide authentication and authorization services. As the Policy Validator evaluates the decision tree, it determines which Policy Validator plugins to call. The Policy Validator compares the plugin responses to the decision tree configured for the resource. In this way, the Policy Validator controls the flow of policy evaluation.

Policy Validator plugins must handle a high volume of authentication and authorization requests. To provide maximum throughput, the Validator API uses the C++ programming language.

The Validator API leverages XML for communication with other Select Access components. The Policy Validator uses XML to store plugin configurations and to communicate with Enforcer plugins. Using XML provides developers with the greatest flexibility in designing plugins; any type of data may be included in the transmitted XML.

Select Access ships with classes that store and process XML documents, nodes, PropertyLists and Properties. These XML classes can provide higher performance than standard XML toolkits. You must use the Select Access classes to manage XML PropertyLists and Properties. For more information on these classes, see Validator API Classes and Utilities on page 48.

Types of Validator API Plugins

There are two types of Validator plugins, Authentication plugins and Decision Point plugins.

Authentication plugins

Authentication plugins allow you to create new ways to verify identities. Using the Policy Builder, you can configure SelectID to use one or more authentication services. When the Policy Validator receives a request to access a protected resource, the Policy Validator uses the Policy Matrix to determine which Select Auth authentication services are allowed for the resource. Each authentication method in a Select Auth rule corresponds to a Policy Validator Authentication plugin.

For example, the SelectID in Figure 11 on page 45 may be configured to use the RADIUS authentication method. When the Policy Validator receives a request to access the protected resource, the RADIUS plugin will be used to authenticate identities.

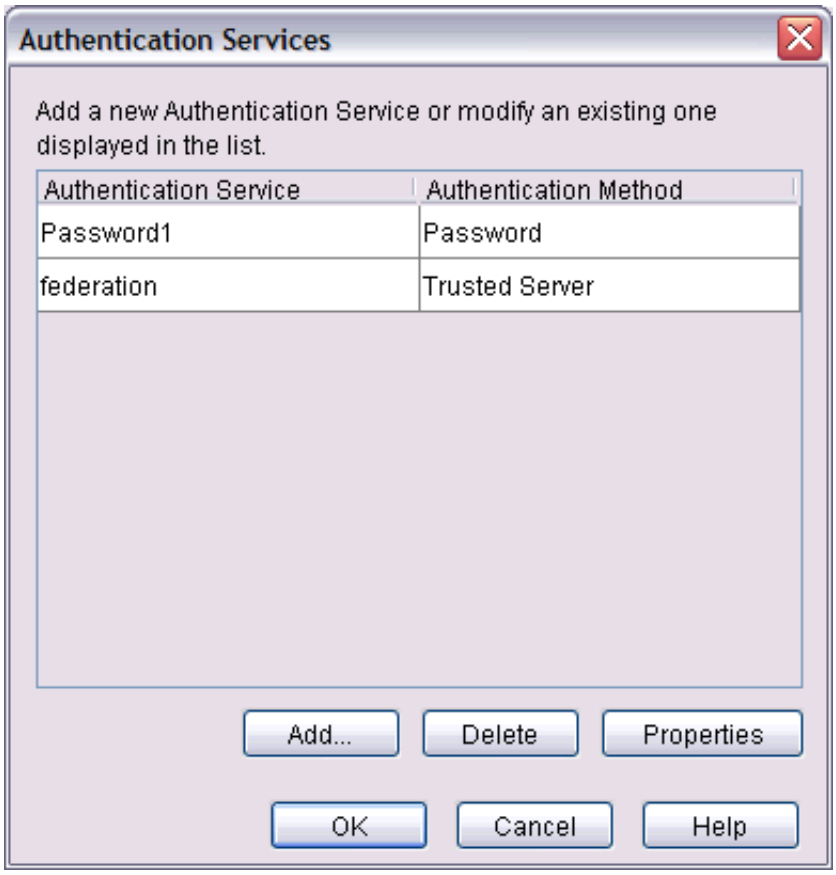

Figure 11 Configuring Select Access to use Policy Validator plugins

Decision Point plugins

Decision Point plugins allow you to create new methods of controlling access to resources. Some security policies may simply allow or deny access based on the identity or group attempting to access the resource. However, many security policies require more complex

access control decisions. Select Access supports complex enterprise security policies with conditional rules. There are three types of access rules in the Policy Builder: allow access, deny access, and conditional access.

A conditional rule, such as the one in Figure 12 on page 46, is identified by the key icon. Conditional rules are configured using the Rule Builder and allow the Policy Validator to support complex security policies. A conditional rule defines a portion of a security policy decision tree. The decision tree nodes correspond to evaluation plugin instances.

Figure 12 on page 46 illustrates the various types of Decision Point plugins. Of the plugin types listed below, only decision point plugins will be discussed in this document.

- Subrule plugins are the root of the decision tree. This plugin type is reserved for Select Access internal use only. You should not develop subrule Decision Point plugins. Subrule plugins have a single tree connector, used to identify the first plugin in the tree.
- Filter plugins perform some action, such as logging, but do not alter the flow of the security policy. Filters have a single tree connector to identify the next plugin in the tree.
- Decision Point plugins correspond to a branch in the decision tree. All decision point Decision Point plugins have both `true` and a `false` connectors in the decision tree. Decision Point plugins evaluate a request and instruct the Policy Validator to follow either the `true` or `false` decision tree branch.
- Terminal points correspond to the decision tree leaves. Terminal points signal the Policy Validator to allow or deny the access request.

> Creating customized filters, terminal points, and subrule Decision Point plugins is a topic beyond the scope of this document.

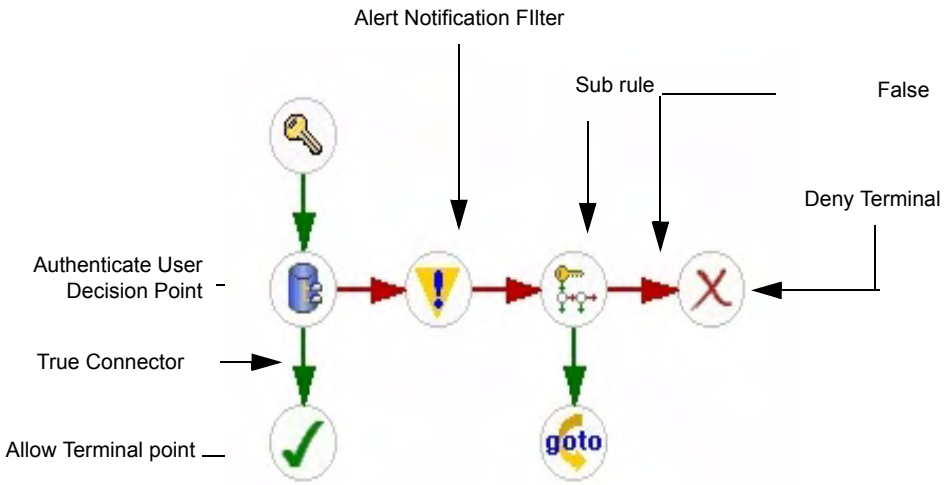

Figure 12 Example Rule

How the Validator API Works

During startup, the Policy Validator loads and initializes all Validator API plugins. The Policy Validator:

1 Searches for plugins: When the Policy Validator is started, it reads all the shared libraries in the directory `<SA_Install>/bin/plugins`.

2 Loads plugins: The Policy Validator calls the static `init()` method to obtain a table of plugins. Each row in the table contains a plugin entry containing the plugin name, its type, and a pointer to its `factory()` method. A `null` entry in the plugin table is used to identify the end of the table.

3 After the Policy Validator has started, it begins receiving requests from Enforcer plugins. The first time a Policy Validator plugin is used to evaluate a policy, the Policy Validator creates an instance of the plugin. The Policy Validator obtains the XML configuration from the Policy Store and creates an instance of the plugin by calling the plugin's `factory()` method with the XML configuration passed as an argument.

For more information on the `init()` method and the `factory()` method, see sections Creating Authentication Plugin Instances with the factory() Method on page 54 and Creating Decision Point Plugin Instances on page 69.

- If Select Auth is enabled for the resource in the Policy Matrix: the Policy Validator performs authentication based on the Select Auth configuration. The Policy Validator:
  - Determines which Authentication plugin to call. Select Auth may be configured to support one or more authentication methods. The Policy Validator executes the plugins in the order configured in Select Auth.
  - Calls the `authenticate()` method of the Authentication plugin. The `authenticate()` method attempts to validate the identity and returns a response code to the Policy Validator.
  - Determines what action to perform next based on the Authentication plugin result code. For example, if the identity is authenticated, the Policy Validator stops processing Authentication plugins and begins evaluating access control rules. If the identity could not be verified, the Policy Validator attempts to authenticate the identity by calling next the Authentication plugin configured in the Select Auth rule. For a complete list of result codes, see Authenticating Identities on page 61.

    

    If the Select Auth rule is configured to use an Authentication plugin that does not exist, the Policy Validator generates an error.

- If no identity authentication is required, or if an Authentication plugin successfully authenticates a identity: the Policy Validator begins processing the authorization policy for the resource. If the access control rule allows or denies access based only on the identity or group, the Policy Validator generates the appropriate XML response and returns it to the Enforcer plugin.
- If the resource is configured to use a conditional rule: the Policy Validator begins processing Decision Point plugins. For conditional rules, the Policy Validator:
  - Determines which Decision Point plugin to call. The Policy Validator begins by processing the first Decision Point plugin specified in the rule.
  - Calls the `decide()` method of the Decision Point plugin. The `decide()` method evaluates the request and returns `true` or `false`. The `decide()` method returns a result code to the Policy Validator. The result code may indicate that an error occurred or trigger the Policy Validator to take other actions, as described below.
  - The Policy Validator determines which decision point plugin to call next based on the `decide()` method result code. For example, if a decider plugin returns `true`, the next decision point plugin on the `true` branch is called. If access is denied, the next decision point plugin on the `false` branch is called. For a complete list of result codes, see section Evaluating a Policy Node on page 70.

— Continues processing Decision Point plugins until it reaches a Terminal point plugin in the decision tree. Common Terminal points include `allow`, `deny`, `redirect`, and `logoff`.

— When a Terminal point is reached, the Policy Validator stops processing and sends the appropriate XML response to the Enforcer plugin.

You should not place passwords or other sensitive information in the XML response. The information may be logged, causing accidental disclosure.

Authentication plugins and Decision Point plugins can communicate with each other by adding XML properties to the request. They can also add XML properties to the response to communicate with the Enforcer plugin. For example, if an Authentication plugin could not locate credentials in the request, it may add properties to the response to instruct the Enforcer plugin to prompt the identity for a name and password.

Validator API Classes and Utilities

To create Policy Validator plugins, you will need to be familiar with several Select Access classes and utilities.

The following two classes are parent classes for all Policy Validator plugins. Your Policy Validator plugin must inherit from one of these two classes.

- `AuthPlugin`: This class is the parent class for Authentication plugins. The `AuthPlugin` class defines the Authentication plugin result codes, a base constructor that your plugin should call, and the virtual methods `authenticate()` and `handleUserInfo()`. You must override the `authenticate()` method to perform identity authentication. The `handleUserInfo()` method automates processing of transient identities and populating personalization data. While it is possible to override the `handleUserInfo()` method, you should not need to do so. The `AuthPlugin` class is covered in detail in Creating an Authentication Plugin: File-based Authentication Example on page 52.
- `Decider`: This class is the parent class for all Decision Point plugins. The `Decider` class defines the Decision Point plugin result codes, the conditional rule branches, and the virtual `decide()` method. You must override the `decide()` method to determine what conditional rule branch to follow, `true` or `false`. This class is covered in detail in Creating a Decision Point Plugin: Directory Attributes Example on page 66.

The Validator API also includes several internal classes used to manage identity data. You will use these classes for creating transient identities, retrieving personalization information, accessing the cache, and obtaining an LDAP connection. The `AttributeLogic` plugin demonstrates how to use these classes to locate additional identity information. For a detailed example of how to use transient identities, the `UserSource`, and the `UserCache`, see LdapConnection, User, UserSource, and UserCache on page 72.

- `UserCache`: The Policy Validator caches identity data to increase performance. This enables the Policy Validator to access the local cache instead of performing a directory server search. Additionally, transient identities are not stored in LDAP, and are only located in the cache.
- `User`: The `User` class stores all pertinent identity data. This class contains information such as what dynamic groups and group memberships a identity possesses. You can use this class to determine if a identity is a transient identity. You can also use this class to obtain the `UserSource` where the identity is located.

- `UserSource`: The `UserSource` class represents a directory server and a location in the directory name space where identities are located. You can use this class to get a list of all the currently configured `UserSources`. If you know the DN for an identity, you can use this class to locate the specific `UserSource` that stores the identity data. Once you know which `UserSource` stores a particular identity, you can obtain an `LdapConnection` to the directory server that stores the identity data. This is useful when you want to query the directory server for additional identity data.

Select Access provides XML classes that you may use to process Policy Validator plugin configurations, Enforcer plugin requests, and responses. The Select Access XML classes are used by Authentication plugins, Decision Point plugins, and Enforcer plugins. For more information on manipulating XML in Select Access, see Understanding the Importance of XML on page 25.

- `XmlParser`: This class parses a block of memory or a file and creates an XML document tree. Typically you will not need to directly use this class, because Select Access creates XML documents for all configurations, requests, and responses. You can use this class if you need to merge an external XML document with a configuration, request or response. You may also use this class if you want to build a configuration, request, or response manually.
- `XmlTreeNode`: This class stores the XML nodes of a parsed XML document. This class provides low-level access to XML attributes. Typically you will use the `PropertyListElement` and `PropertyElement` classes to access XML data.
- `PropertyElement`: This class stores a name-value pair. A `PropertyElement` is an XML element defined by Select Access. The `PropertyElement` tag requires one an attribute called `name` that identifies the name of the name-value pair. The value is stored as the data between begin and end tags. An example `PropertyElement` that stores the name-value pair for the color red is: `<PROPERTY NAME="color">red</PROPERTY>`.
- `PropertyListElement`: This class stores an XML node that contains zero or more `PropertyElement` tags. Property lists may be nested; a `PropertyListElement` may contain property lists as well as property elements. An example of a `PropertyListElement` is: `<PROPERTYLIST NAME="colorList"> <PROPERTY NAME="color">red</PROPERTY></PROPERTYLIST>`.

Your Policy Validator plugin should support logging, handle exceptions, and leverage the Select Access string and memory utilities. Most plugins will need to interact with the following Select Access classes and utilities:

- `Logger`: The `Logger` class provides an interface to Select Audit. This class supports multiple levels of logging support: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `FATAL`.
- `EnforcerException`: This class is the parent class for the Decision Point plugin exceptions: `EvaluatorFatal`, `EvaluatorNonFatal`, and `EvaluatorInternal`. Decision Point plugins may throw any of the above exceptions, including the `EnforcerException`. Decision Point plugin exceptions are covered later in this chapter; see Evaluating a Policy Node on page 70.
- `AuthPluginException`: This class is defined in the `AuthPlugin.h` header, and should be used when an Authentication plugin exception occurs.
- `enforcer_sys.h`: This header files defines `SYS_STRDUP`, `STREQ`, and other platform independent string manipulation macros. Your plugin should use these macros when manipulating strings.

Creating a Policy Validator Plugin

When you create a Policy Validator plugin, you create a shared library. A shared library file has an extension of `.so` or `.sl` on UNIX platforms, and a `.dll` extension on Windows platforms. The shared library must contain one or more Policy Validator plugins.

To create your plugin

1. Create a C++ file that will contain one or more plugins.
2. Include the `AuthPlugin.h` header for Authentication plugins or the `Decider.h` header for Decision Point plugins. Include other headers as needed.
3. Create one or more plugin classes that inherit from the `AuthPlugin` or the `Decider` classes. Authentication plugins inherit from `AuthPlugin`, Decision Point plugins inherit from `Decider`.
4. Provide a constructor and a destructor for each plugin class.
5. Provide a static `factory()` method for each class. The `factory()` method is used by the Policy Validator to create instances of your plugin. It does not have to be named `factory()`; any legal C++ name is acceptable. The `factory()` method receives an `XmlTreeNode` as an arguments, and returns a pointer to a `Decider` or `AuthPlugin` object.
6. Override the `authenticate()` method or the `decide()` method, depending on the type of plugin.
7. Register your plugins by providing a single external `init()` method. This function provides a table of plugin entries to the Policy Validator. Each plugin entry contains the name of the plugin, the type of the plugin, and a pointer to a `factory()` method. The `init()` method should include an entry for each plugin class in the file.
8. Compile your plugin and link the resulting object file with the Validator API libraries. For more information on compiling plugins, see Building the AttributeLogic Example on page 79.
9. Install your plugin. See Installing Policy Validator Plugins on page 51.
10. Test your plugin.

Overriding the authenticate() Method

You can override the `authenticate()` method. To do this, your `authenticate()` method must:

1. Obtain the credentials from the Enforcer plugin request.
2. Validate the identity. This may consist of comparing a name and password with a database, verifying a digital signature, verifying a Kerberos ticket, forwarding the request to a RADIUS server, or some other custom method.
3. Optionally update the XML response. Authentication plugins can communicate with the Enforcer plugin by adding properties to the XML response. For instance, if credentials were not present in the request, the Authentication plugin can instruct the Enforcer plugin to prompt the identity for a name and password.

4 Optionally call the `handleUserInfo()` method. The `handleUserInfo()` method sets the identity's personalization information. For details on personalization, see Chapter 6, Using Personalization Attributes: the Personalization API. While not required, Authentication plugins should call the `handleUserInfo()` method, since it also caches the profile for future access. If the identity does not have a profile, this method will create a transient profile in the cache. Transient profiles are never written to the directory server. For more information on transient identities, see LdapConnection, User, UserSource, and UserCache on page 72.

5 Return a result code to the Policy Validator. The result code instructs the Policy Validator how to continue processing the request. For information on Authentication plugin result codes, see Authenticating Identities on page 61.

See Creating an Authentication Plugin: File-based Authentication Example on page 52 for a detailed example of how to build an Authentication plugin.

Overriding the decide() Method

You can override the `decide()` method. To do this, your `decide()` method must:

1 Validate the access policy. The Decision Point plugin may use any of the request data to make the decision, such as the identity, dynamic group memberships, timestamp, IP address, or even custom data embedded by the Enforcer plugin. The Decision Point plugin may also use external resources to gather additional information. For example, the plugin may query a database or a directory server. Decision Point plugins can modify the original request. This enables plugins to communicate with each other.

2 Optionally update the XML response. Decision Point plugins can communicate with the Enforcer plugin by adding properties to the XML response. For instance, a Decision Point plugin can inform the Enforcer plugin that an identity must authenticate before granting access.

3 Return a result code to the Policy Validator. The result code instructs the Policy Validator how to continue processing the request. For information on Decision Point plugin result codes, see Evaluating a Policy Node on page 70.

See Creating a Decision Point Plugin: Directory Attributes Example on page 66 for a detailed example of how build a Decision Point plugin.

Installing Policy Validator Plugins

To install a Policy Validator plugin, place the shared library containing the plugin in the `<SA_Install>/bin/plugins` directory. Install a copy of the plugin on each Policy Validator server. You only need to install the plugin on servers that are configured to use the plugin as part of the access rule. You must restart the Policy Validators for the changes to take effect.

Removing Policy Validator Plugins

To remove a plugin from the Policy Validator, simply delete the shared library containing your plugin from the `<SA_Install>/bin/plugins` directory, or move the file to another directory. You will need to restart the Policy Validators for the changes to take effect.

Make sure you remove all rules and authentication services that use the plugin before restarting the Policy Validator. Use the Policy Builder to remove the plugin references from Select Auth and conditional rules.

This procedure only removes the Policy Validator plugin. Policy Validator plugins have a corresponding Policy Builder plugin which must be removed separately. For details, see Removing Policy Builder Plugins on page 40.

Creating an Authentication Plugin: File-based Authentication Example

Select Access ships with a set of Authentication plugins. This section will review the source code for the File-based Authentication plugin. In this section, you will learn:

- How to include the Authentication plugin header files.
- How to register plugins using the `init()` method.
- How to create a `factory()` method that creates a plugin instance.
- How to override the `authenticate()` method to perform authentication.
- How to instruct Enforcer plugin to prompt for credentials.
- How to communicate personalization data to Enforcer plugin.
- How to change the behavior of the Policy Validator using result codes.

The `FileAuthenticator` class:

1. Sets the `UserSource`. A `UserSource` specifies a directory server branch where the identity profile is located.
2. Sets the login forms.
3. Loads the password file into the cache.
4. Extracts names and passwords from Enforcer plugin requests.
5. Triggers the Enforcer plugin to challenge the identity for a user name and password if these properties were not provided.
6. Authenticates the identity against the password cache.

The source code for the `FileAuthenticator` class can be located in *`<SDK_install_path>`*`/source/validator/plugins/FileAuthenticator.cpp`.

Including the Authentication Plugin Header Files

The `FileAuthenticator` Authentication plugin includes the header file `AuthPlugin.h`, which all Authentication plugins must include. The `validator_util.h` header includes `validator.h`:

```
#include "enforcer_pragmas.h"
#define ACE_BUILD_SVC_DLL
#include <ace/svc_export.h>
#include "auto_free.h"
#include <ctype.h>
```

```
#include <stdio.h>
#include <vector>
#include <string>
#include <map>

#include <new>
#include <string>
#include <iostream>
#include <stdio.h>

#include <PropertyElement.h>
#include <PropertyListElement.h>
#include "User.h"
#include "UserCache.h"
#include "UserSource.h"
#include "validator_util.h"
#include "AuthPlugin.h"
#include "EnforcerException.h"
#include "LdapAttributes.h"
```

FileAuthenticator Class Constants

The following constants are used in the remainder of the code and are provided here for reference.

Sample FileAuthenticator class constants declaration

The following code sample illustrates how class constants are declared.

```
#define TRANSIENT_LOCATION_TAG   "ArtificialSource"
#define LOGIN_FILENAME_TAG       "LoginForm"
#define PASSWORD_FILENAME_TAG    "PasswordFile"
#define USER_SOURCE_TAG          "UserSource"
#define ATTRIBUTE_PROP_LIST_TAG  "SelectedAttributes"
#define ATTRIBUTE_TAG            "Attribute"

#define MAXLINE_SIZE             1024
#define PASSWORD_ATTR            "userPassword"
#define UID_ATTR                 "uid"


typedef vector<string> strVector;
typedef map<string, strVector> attributeMap;
typedef map<string, attributeMap> userMap;
```

Registering an Authentication Plugin

Your plugin must register itself before the Policy Validator will be able to use your Authentication plugin. To do this you need to include an `init()` method in your plugin library. The Policy Validator calls the `init()` method for every shared library located in the Policy Validator directory. The `init()` method registers one or more plugins that are located in the shared library, by registering a table of plugin entries. Each table entry contains a triplet consisting of:

- The plugin name: The name used in the `component.xml` file to associate an Authentication plugin with a Select Auth authentication method.
- The plugin type: Authentication plugins are registered as type `AUTH`.
- A pointer to a `factory()` method: The `factory()` method provides the Policy Validator with a pointer to an Authentication plugin instance. The `factory()` method should return a pointer to a method, as identified by the type definition `voidp_func`.

The table of plugin entries is terminated with a `NULL` plugin entry. Registering the plugin on page 54 demonstrates how to create a plugin entry table and register your Authentication plugin. The code registers one plugin, the `FileAuthenticator` Authentication plugin.

Registering the plugin

The following code sample illustrates how the plugin is registered:

```
// table of plugins
static PluginEntry plugins[] = {
        { "FileAuthenticator", AUTH,
(voidp_func)&FileAuthenticator::factory
        },
        { NULL, AUTH, NULL }
};

PluginEntry * init()
{
    return plugins;
}
```

Creating Authentication Plugin Instances with the factory() Method

The purpose of the `factory()` method is to provide the Policy Validator with an instance of your Authentication plugin. The Authentication plugin instance represents an authentication service in Select Auth. The Policy Validator retrieves the Authentication plugin's configuration from the Policy Store and provides the XML to the `factory()` method. The `factory()` method creates an Authentication plugin instance using the XML to initialize the Authentication plugin's state. The `factory()` method is defined as:

```
static AuthPlugin * factory(const XmlTreeNode *props);
```

Creating a FileAuthenticator plugin instance on page 55 demonstrates how to build a `factory()` method. The example returns a `FileAuthenticator` Authentication plugin instance configured using the XML properties provided in the `props` variable. The `FileAuthenticator` class inherits from the `AuthPlugin` class, so the returned pointer is an Authentication plugin object.

The `FileAuthenticator::factory()` method parses the XML configuration and ensures that all the necessary data is present. New versions of the plugin allow administrators to specify a login form name to use in place of the default.

The `factory()` method calls the `FileAuthenticator` class constructor to create an Authentication plugin instance using the extracted configuration values. It then loads the password file and attribute list into the cache.

The `factory()` method can have any legal C++ name. Select Access uses the `init()` method to obtain a pointer to the `factory()` method.

Creating a FileAuthenticator plugin instance

The following code sample illlustrates how to create a `FileAuthenticator` plugin instance with the `factory ()` method:

```
AuthPlugin * FileAuthenticator::factory
(
      const string & name,
      const XmlTreeNode * props
)
{
      // Must have properties
      if (! props)
            {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
                  ">#<No File Authenticator properties specified");
               return NULL;
       }
       // Parameters
       const char * transientUserLocName  = props->getChild
       StringValue(TRANSIENT_LOCATION_TAG);
       const char * loginFormName         = props->getChild
       StringValue(LOGIN_FILENAME_TAG);
       const char * passwordFileLocName   = props->getChild
       StringValue(PASSWORD_FILENAME_TAG);
       const char * userSourceName        = props->getChild
       StringValue(USER_SOURCE_TAG);
      const UserSource * userSource = NULL;
      FILE * passwordFile = NULL;

      if (!sNameToUserSource(name.c_str(), userSourceName, userSource))
      {

        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
                  ">#<FileAuthenticator: Could not create a valid User
                      Source");
            return NULL;
      }
      if (! transientUserLocName ||
```

```
      ! sValidateSyntheticUserLocation(transientUserLocName,
        userSource))
{
      Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
            ">#<FileAuthenticator: Transient user location not
               located below the User Source");
      return NULL;
}
if (! loginFormName)
{
         Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
            ">#<FileAuthenticator: Could not locate the login form
               in the configuration");
      return NULL;
}
if (! passwordFileLocName)
{
         Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
            ">#<FileAuthenticator: Could not locate the password
                file in the configuration");
      return NULL;
}
else if (access(passwordFileLocName, 4) == -1)
{
      if (errno == ENOENT)
      {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
                  ">#<FileAuthenticator: The password file %s does
                     not exist.", passwordFileLocName);
      }
      else if (errno == EACCES)
      {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
                  ">#<FileAuthenticator: The password file %s is
                      not readable.", passwordFileLocName);
      }
      else
      {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
                  ">#<FileAuthenticator: An unknown error occurred
                  while trying to open the password file %s.",
                  passwordFileLocName);
      }
      return NULL;
}
```

```
        const XmlTreeNode *xmlAttributeList = props->
        getChild(ATTRIBUTE_PROP_LIST_TAG);

        try
        {
              passwordFile = fopen(passwordFileLocName, "r");
              FileAuthenticator * plugin = new FileAuthenticator
              ( name.c_str(), userSource, transientUserLocName,
                loginFormName);
              if(!plugin->getAttributesToCopy(xmlAttributeList))
                    return NULL;
              if(!plugin->getPasswords(passwordFile))
                    return NULL;
              return plugin;
        }
        catch (...)
        {
          Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
                    ">#<FileAuthenticator: Could not create a File
                        Authenticator plugin instance");
              return NULL;
        }
}
```

The `FileAuthenticator` constructor passes the plugin name and the `UserSource` to the parent constructor. All Authentication plugins must provide the parent class with the plugin name and the `UserSource`. If the `UserSource` is `NULL at runtime`, then the Policy Validator will try each available `UserSource` in the order configured. The constructor also sets fields to store the transient identity location and the login and challenge forms. For more information on transient identities, see LdapConnection, User, UserSource, and UserCache on page 72. If the log level of Select Audit is set to include DEBUG messages, the constructor also logs a message indicating that it was called.

The FileAuthenticator constructor

The following code sample is an example of a FileAuthenticator constructor:

```
FileAuthenticator::FileAuthenticator
(
        const char                    * name,
        const UserSource              * userSource,
         const char                    * transientUserLoc,
         const char                    * loginForm
)
:       AuthPlugin(name, userSource),
        transientUserLoc_(SYS_STRDUP(transientUserLoc)),
        loginForm_(SYS_STRDUP(loginForm)),
        cache_user_flag_(true)
{
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG, "FileAuthenticator:
```

```
      Constructing FileAuthenticator plugin");
      Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG, "FileAuthenticator:
      transientUserloc: %s", transientUserLoc);
      Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG, "FileAuthenticator:
      loginForm: %s", loginForm);
}
```

After creating an instance of the Authentication plugin, the `FileAuthenticator::factory()` method initializes the list of XML attributes. It calls to get these attributes with the `setToCopy()` method which loads them into the cache. Finally, it calls the `getPassword()` method to load the passwords into the cache.

When building Policy Validator plugins, it is important to properly handle error conditions. plugin errors often cause a valid request to be denied. They can be difficult to locate. You should log all errors so they may be tracked. Use the ERROR log level for these conditions.

Your `factory()` method should return NULL if it was unable to configure the plugin instance.

Loading the Password file

The following code sample shows how to load this file into the cache:

```
bool FileAuthenticator::getPasswords(FILE * passwordFile)
{
      if (passwordFile == NULL)
      {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
                  ">#<FileAuthenticator: An unknown error occurred while
                   trying to open the password file.");
            return false;
      }
        fetchPasswordData(passwordFile);
      return true;
}

void FileAuthenticator::processLine(char * line, userMap & users, string
& currentUserKey, string & currentAttributeKey, attributeMap & user-
Record )
{
      if (!users.empty() && currentUserKey != "")
      {
            userMap::iterator entry = users.find(currentUserKey);
            if (entry != users.end())
                  userRecord = entry->second;
      }
      strVector attributeValues;
      if (!userRecord.empty() && currentAttributeKey != "")
      {
            attributeMap::iterator attr = userRecord.find
            (currentAttributeKey);
```

```
        if (attr != userRecord.end())
                attributeValues = attr->second;
}
int i = 0;
bool allSpace = true;
while (i < MAXLINE_SIZE && line[i] != '\0' && allSpace)
{
        if (line[i++] != ' ')
                allSpace = false;
}
if (allSpace)
{
        if (userRecord.size() > 0)
        {
                Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
                "FileAuthenticator: Read user record for %s",
                currentUserKey.c_str());
                users[currentUserKey] = userRecord;
                currentUserKey = "";
                currentAttributeKey = "";
                userRecord.clear();
        }
        return;
}
i = 0;
if (line[0] == '\0' || line[0] == '#')
        return;
if (line[0] == ' '&& line[1] != ' ')
{
        //printf("multi-line attribute\n");
        if (attributeValues.size() > 0)
        {
                string value = attributeValues.back();
                attributeValues.pop_back();
                while (line[++i] != '\0' && i < MAXLINE_SIZE)
                {
                        value += line[i];
                }
                attributeValues.push_back(value);
                userRecord[currentAttributeKey] = attributeValues;
                if (currentUserKey != "")
                {
                        users[currentUserKey] = userRecord;
                }
                return;
        }
```

```
            return;
      }
      bool nameFinished = false;
      string name;
      string value;
      while (i < MAXLINE_SIZE && line[i] != '\0')
      {
            if (nameFinished)
            {
                  value += line [i];
            }
            else
            {
                  if (line[i] == ':')
                  {
                        nameFinished = true;
                        i++;
                  }
                  else
                  {
                        name += line [i];
                  }
            }
            i++;
      }
      if (currentAttributeKey == "")
            currentAttributeKey = name;
      if (name != currentAttributeKey)
      {
            currentAttributeKey = name;
            attributeValues.clear();
      }
      attributeValues.push_back(value);
      userRecord[currentAttributeKey] = attributeValues;
      if (name == "uid")
      {
            string newName = name + "=" + value;
            currentUserKey = newName;
      }
      if (currentUserKey != "")
            users[currentUserKey] = userRecord;
}

void FileAuthenticator::fetchPasswordData(FILE * fp)
{
        char line[MAXLINE_SIZE];
```

```
  char c;
  int i = 0;
bool processedLine = false;
  if (!fp)
          return;

  nullLine(line);
string currentUserKey = "";
string currentAttributeKey = "";
attributeMap tmpUserRecord;
  while ((c = fgetc(fp)) != EOF)
  {
          if (c != '\n')
          {
            processedLine = false;
                line[i] = c;
                i ++;
          }
          else
          {
            processedLine = true;
                line[i] = '\0';
            i = 0;
                processLine (line, users_, currentUserKey,
                currentAttributeKey, tmpUserRecord);
                nullLine(line);
          }
  }
if (!processedLine)
          processLine (line, users_, currentUserKey,
          currentAttributeKey, tmpUserRecord);
}
```

Destroying Authentication Plugin Instances

Your Authentication plugin will also have to provide a destructor. Make sure you free any external resources. The `FileAuthenticator` class does not manage any external resources, since C++ strings are automatically freed.

```
FileAuthenticator::~FileAuthenticator() {}
```

Authenticating Identities

To authenticate identities, the Policy Validator calls the `authenticate()` method of an Authentication plugin. Every Authentication plugin must override this method. The Policy Validator passes the `authenticate()` method four parameters:

- A pointer to the request data. This XML property list contains the resource access request sent by a Enforcer plugin, and may contain additional data embedded by other Authentication plugins. Select Access plugins can pass information to other plugins by adding data to the request.
- A pointer to the current XML response which contains any existing XML properties from Authentication plugins that have already been processed.
- A `PropertyListElement` which is used to hold personalization information. For more details on personalization, see Chapter 6, Using Personalization Attributes: the Personalization API.
- A trace flag that indicates whether the Decision Point plugin should include detailed information regarding policy decisions. in the response used to send back to the Enforcer plugins. It is usually used only for debugging purposes, and is usually set to `false`.

The `authenticate()` method returns a result code that instructs the Policy Validator how to continue processing. Authentication plugins are allowed to return the following values:

- `R_PERMIT`: Stop processing Authentication plugins; the identity has been authenticated.
- `R_DENY`: Stop processing Authentication plugins; the identity has been denied access. This is typically used when the profile has been deactivated.
- `R_RESTART`: Reauthenticate by processing the first Authentication plugin in the Select Auth. This usually occurs when new information concerning a identity has been discovered. This allows a plugin that could not originally authenticate the identity to retry using the new identity. The Policy Validator determines which Authentication plugins to use by evaluating the Policy Matrix and Select Auth.
- `R_ERROR`: This indicates that an error occurred while authenticating the identity. The Policy Validator will stop processing authentication and deny access. Your plugin should log an error level message explaining what error occurred.
- `R_CONTINUE`: The identity was not authenticated; attempt to authenticate the identity with the next Authentication plugin in the Select Auth.

To support password management, the following codes are used internally by Select Access. This enables Select Access to handle special cases, such as when a password has expired. Your plugin should not return these values.

- `R_DENY_FORM`
- `R_PERMIT_FORM`
- `R_DENY_AUTHENTICATED`
- `R_DENY_AUTHENTICATED_FORM`
- `R_DENY_USER_NOT_FOUND_FORM`

Authenticating requests

The following code sample illustrates how requests become authenticated:

```
AuthPlugin::Result
FileAuthenticator::authenticate(
      PropertyListElement *data,
      PropertyListElement *response,
      const PropertyListElement * const personalizer,
      const bool trace
)
```

```
{
      Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG, "FileAuthenticator:
      Invoking FileAuthenticator plugin");
      AuthPlugin::Result result = R_DENY;
      if (isRestart(data, name_.get()))
      {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
            "FileAuthenticator: Authentication restart");
            result = validateRestartData(data, response, personalizer);
      }
      else if (!fetchData(user_, ENFORCER_USER,
               ValidatorGetPostDataList(data)) ||
                   !fetchData(password_, ENFORCER_PASSWD,
                    ValidatorGetPostDataList(data)))
      {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
            "FileAuthenticator: No uid or password, sending login form");
            addPasswordHint2Response(response);
      }
      else if (validateUser(user_, password_))
      {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
            "FileAuthenticator: User authenticated");
            result = R_PERMIT;

            if (canCacheUser())
            {
                  Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
                  "FileAuthenticator: Caching user record");
                  result = handleUserInfo(data, response,
                  personalizer, user_.c_str(), transientUserLoc_.get());
            }
      }
      else
      {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
            "FileAuthenticator: User login failed");
            addPasswordHint2Response(response);
      }
      setResponseAction(result, response);
      return result;
}
```

Note how the Authentication plugin handles requests where the identity has not yet authenticated or provided credentials. The Authentication plugins returns the `R_DENY` code to inform the Enforcer plugin not to allow access to the requested resource.

The Authentication plugin also adds authentication hints to the XML response. Authentication hints inform the Enforcer plugin what types of authentication methods the Policy Validator allows for the requested resource. If the Enforcer plugin supports one of the listed methods, the authentication hint will trigger the Enforcer plugin to interact with the identity to obtain the identity's credentials. This is typically done with a login form that prompts the identity for a user name and a password, though any mechanism can be supported.

Validating the Password

The following code same shows how the validates a password:

```
bool FileAuthenticator::validateUser(const string & user, const string &
password)
{
        if (users_.empty())
        {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG, "No users
            present in the login map.");
            return false;
        }
        string userId = UID_ATTR;
        userId += "=";
        userId += user;
        userMap::iterator iter = users_.find(userId);
        if (iter == users_.end())
        {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG, "User %s not
            found in login map.", user.c_str());
            return false;
        }
        attributeMap userRecord = iter->second;
        attributeMap::iterator iter2 = userRecord.find(PASSWORD_ATTR);
        if (iter2 == userRecord.end())
        {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG, "No password
            found for user in login map.");
            return false;
        }
        strVector values = iter2->second;
        for (strVector::iterator iter3 = values.begin(); iter3 !=
             values.end(); iter3++)
        {
            if (password == (*iter3))
                return true;
        }
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG, "Password did not
                    match.");
return false;
}
```

The `authenticate()` method sets the XML response to reflect the result code from the `validateUser()` method. The preprocessor definitions for the XML string values used in Setting an XML response on page 65 are defined in `enforcer.h`. The Policy Validator sends the resulting XML response to the Enforcer plugin when processing is complete.

Setting an XML response

The following code sampe shows how to set an XML response:

```
void FileAuthenticator::setResponseAction
(
   AuthPlugin::Result result,
   PropertyListElement   * response
)
{
        switch (result)
   {
        case R_ERROR:
      {
                ValidatorSetAction(response, ENFORCER_ACTION_ERROR);
        }
      break;
      case R_DENY:
      {
                ValidatorSetAction(response, ENFORCER_ACTION_DENY);
        }
      break;
        case R_PERMIT: {
                ValidatorSetAction(response, ENFORCER_ACTION_ALLOW);
        }
      break;
        case R_RESTART: {
                // No ACTION to be inserted into response
        }
      break;
        default: {
                ValidatorSetAction(response, ENFORCER_ACTION_ERROR);
        }
      break;
        }
}
```

The `passwordHint2Response()` method adds XML properties to the response. These properties inform the Enforcer plugin to request identity authentication. The location of the login form is included in the response.

Creating a Decision Point Plugin: Directory Attributes Example

Select Access ships with a set of Decision Point plugins. One of the most flexible Decision Point plugin is the `attributelogic` plugin. This plugin compares identity attributes to a configured search filter. The search filter is configured by the `AttributeLogicPanel` Policy Builder plugin described in the section Creating a Policy Builder Plugin on page 34.

A search filter is composed of a series of Boolean expressions. Each expression specifies an attribute, a comparison operator, and a value. For instance, a search filter might specify that identities must have an attribute named “department” with a value equal to “sales”.

In this section, you will learn:

- How to include the Decision Point plugin header files.
- How to register plugins using the `init()` method.
- How to create a `factory()` method that creates a plugin instance.
- How to override the `decide()` method to perform authorization.
- How to use the `User`, `UserCache`, `UserSource`, and `LdapConnection` classes to obtain identity information.
- How to control which `DECIDER_BRANCH` the Policy Validator will follow. Decider plugins are configured with a `true` and a `false` branch. A Decision Point plugin is associated with each branch. The Policy Validator determines which Decision Point plugin to process next based on the `DECIDER_BRANCH` value.
- How to control the behavior of the Policy Validator using result codes.
- How to handle `EvaluatorExceptions`.

The `attributelogic` class is used as an example of an Authorization plugin. The `attributelogic` class:

1. Builds an LDAP compliant search filter from its XML configuration.
2. Extracts the identity located in the request XML.
3. Finds the identity in the cache.
4. Identifies the `UserSource` containing the identity. A `UserSource` specifies a directory server branch where the identity’s LDAP entry is located.
5. Obtains an `LdapConnection` from the `UserSource`. This enables the `attributelogic` plugin to search the directory server for additional identity information.
6. Compares the identity attributes and memberships to the configured search filter. If the search filter is configured to use LDAP, the plugin searches the directory server for the identity’s attributes. Otherwise, identity attributes are extracted from the XML request.
7. Instructs the Policy Validator to follow the `true` branch if the identity has the required attributes or the `false` branch if the identity does not.

The source code for the `attributelogic` class can be located in *`<SDK_install_path>`*`/source/validator/plugins/attributelogic.cpp`.

Including the Decision Point Plugin Header Files

The `attributelogic.cpp` file includes the `attributelogic.h` header. This file includes the `Decider.h` header necessary for all Decision Point plugins. The plugin also uses the XML classes `XmlTreeNode`, `PropertyElement`, and `PropertyListElement`. To obtain a directory server connection and dynamic groups, the plugin includes the `User`, `UserSource`, `UserCache`, and `LdapConnection` classes. The `evaluator.h` header defines the decision point plugin exceptions.

```
#include <algorithm>
#include "attributelogic.h"
#include "attributelogic_tags.h"
#include "UserSource.h"
#include "XmlTreeNode.h"
#include "PropertyElement.h"
#include "PropertyListElement.h"
#include "attribute_compare.h"
#include "ldap_util.h"
#include "LdapConnection.h"
#include "User.h"
#include "UserCache.h"
#include "validator_util.h"
#include "evaluator.h"
```

Attributelogic Class Constants

The constants defined in Attributelogic constants on page 67 define the types of Boolean operators and comparison operators used when building and evaluating the search filter.

Attributelogic constants

The following code example shows the constants for the attribute logic plugin:

```
const char *attributelogic::logicalOperatorNames[] = {"AND", "OR" };
const bool attributelogic::logicalOperatorSigns[] =  {true,  false};
const int attributelogic::logicalOperatorSize = 2;
const char *attributelogic::operatorNames[] =
{ "Equal",    "NotEqual",  "GreaterThan" "GreaterEqual",
  "LessThan", "LessEqual", "Set",        "NotSet" };
const char *attributelogic::operatorSigns[] =
{ "=",        "!=",        "!<=",         ">=",
  "!>=",      "<=",        "=*",          "!=*"    };
const bool attributelogic::operatorUnary[] =
{ false,      false,       false,          false,
  false,      false,       true,           true     };
const char *attributelogic::operatorLogic[] =
{ "|",        "|",         "|",            "|",
  "|",        "|",         "|",            "&"      };
const int attributelogic::operatorSize = 8;
const char *attributelogic::scopeNames[] =
{"UserData",      "GroupData",     "AnyData"};
const bool attributelogic::scopeUser[] =
```

```
{true,            false,           true     };
const bool attributelogic::scopeGroup[] =
{false,           true,            true     };
const int attributelogic::scopeSize = 3;
```

Registering the Decision Point Plugin

You must register your plugin before the Policy Validator will be able to use your Decision Point plugin. The Policy Validator calls the `init()` method for every shared library located in the Policy Validator directory. The `init()` method registers one or more plugins that are located in the shared library.

When you build a Decision Point plugin, you need to include an `init()` method in your plugin library. The `init()` method registers a table of plugin entries. Each table entry contains a triplet consisting of:

- The plugin name: The name used in the `component.xml` file to associate a Decision Point plugin with a Rule Builder plugin. The plugin name must match the `EVALUATOR` attribute value in the `COMPONENT` tag for the corresponding Policy Builder plugin. For details on the `component.xml` file, see Creating a component.xml File on page 37.
- The plugin type: Decision Point plugins are registered as type `DECIDER`.
- A pointer to a `factory()` method: The `factory()` method provides the Policy Validator with a pointer to a Decision Point plugin instance.

The table of plugin entries is terminated with a `NULL` plugin entry. The code registers one plugin, the attribute logic Decision Point plugin.

Creating and registering the plugin

The following code example demonstrates how to create a plugin entry table and register your Decision Point plugin:

```
extern "C" ACE_Svc_Export PluginEntry *init();
// A table of plugin factories located in this class
static PluginEntry plugins[] =
{
    // The Attribute Logic Validator plugin
    {  "attributelogic",  DECIDER,
(voidp_func)&attributelogic::factory },
    // Terminate the table with a "null" entry
    {   NULL,             DECIDER,       NULL }
};
PluginEntry * init()
{
    return plugins;
}
```

Creating Decision Point Plugin Instances

The `factory()` method provides the Policy Validator with an object that represents a decision point in a conditional rule. The Policy Validator retrieves the decision point configuration from the Policy Store and provides the XML to the `factory()` method. The `factory()` method creates a Decision Point plugin instance using the XML to initialize the Decision Point plugin state. The `factory()` method is defined by the template:

```
static Decider * factory(const XmlTreeNode *props);
```

The factory() method on page 69 demonstrates how to build a `factory()` method. The example returns an `attributelogic` instance configured using the XML properties provided in the `props` variable. The attributelogic class inherits from the `Decider` class, so the returned object pointer is a `Decider` object

The `factory()` method can have any legal C++ name. Select Access uses the `init()` method to obtain a pointer to the `factory()` method.

The factory() method

The following sample shows how to use the`factory()` method to build plugin instances:

```
Decider * attributelogic::factory(const XmlTreeNode *props) {
    if ( ! props ) {
        throw EnforcerException(EnforcerException::E_NULL_ARG,
            "attributelogic factory()", "Invalid Property List");
    }
    return new attributelogic(*props);
}

// constructor
attributelogic::attributelogic(const XmlTreeNode &propertyList, bool
useLdap)
    :   m_expression(propertyList),
        m_useLdap(useLdap)
{
    Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
        "Constructing attributelogic plugin");
    if ( ! m_expression.isPropertyList() ) {
        throw EnforcerException(EnforcerException::E_INVALID_ARG,
            "attributelogic::constructor()", "Invalid Property List");
    }
}

attributelogic::attributelogic(
    const XmlTreeNode &,  //propertyList
    bool = true           // useLdap
);
```

Note that the example `factory()` method calls a constructor that accepts only the `XmlTreeNode`. This constructor is defined in the header file `attributelogic.h`, and is included in the code example above. This single parameter `constructor` calls the constructor shown above, with the `useLdap` variable set to `true`.

Destroying Decision Point Plugin Instances

Your Decision Point plugin will also have to provide a destructor. Make sure you free any external resources. The attributelogic class does not manage any external resources.

```
attributelogic::~attributelogic(){}
```

Evaluating a Policy Node

As the Policy Validator processes a conditional rule, it evaluates one or more decision points. Each decision point represents a decision point plugin. When the Policy Validator needs to evaluate a decision point, it calls the `decide()` method of the corresponding Decision Point plugin. The `decide()` method determines whether the Policy Validator will continue processing using the `true` or `false` branch.

The `decide()` method is the core of the Decision Point plugin. Your plugin must overload this method; it is a virtual method defined in the parent `Decider` class.

The Policy Validator provides the `decide()` method with:

- A pointer to the request data. This XML node contains the resource access request sent by a Enforcer plugin, and may contain additional data embedded by other evalutators or Authentication plugins. Select Access plugins can communicate with other plugins by adding data to the request.
- A pointer to the current XML response which contains any existing XML responses from Decision Point plugins that have already been processed as part of the conditional rule.
- A trace flag that indicates whether debugging has been turned on at the command line.
- A `decider_branch_path` string. This string must be set to reflect which path the Policy Validator should follow, `true` or `false`. The two valid settings for the response string are the predefined strings `DECIDER_BRANCH_TRUE` and `DECIDER_BRANCH_FALSE`.

The `decide()` method returns a result code that instructs the Policy Validator whether to continue processing, abort due to an error, or take some other action. The codes are:

1. `CONTINUE`: Process the next the Decision Point plugin in the conditional rule, as determined by the response variable (i.e. `DECIDER_BRANCH_TRUE` or `DECIDER_BRANCH_FALSE`). Most Decision Point plugins return this value under normal conditions.
2. `DONE`: Stop processing Decision Point plugins and return the XML response to the Enforcer plugin. A Decision Point plugin typically returns this value only for special cases where access is to be allowed or denied regardless of the remaining conditional rule. For example, you may want to always deny access to an identity whose profile is inactive.
3. `ALLOW`: Stop processing and allow access. This is a reserved value for Select Access Terminal points. Your Decision Point plugin should not return this value.
4. `FAILED`: Stop processing due to an error.
5. `RESTART`: Re-evaluate the conditional rule from the first decision point. A Decision Point plugin usually returns this code after requesting that the identity authenticate.
6. `SEND_FORM`: This is a reserved code used by internally by Select Access. Your plugin should not return this value.

If your Decision Point plugin returns the value `DONE` or `ALLOW`, it must also set a value in the XML result to indicate whether access is allowed or denied. To do this, the Decision Point plugin must set the XML property `ENFORCER_ACTION` with a value of

ENFORCER_ACTION_ALLOW or ENFORCER_ACTION_DENY. The preprocessor definitions for these strings are defined enforcer.h, located in the *<SDK_install_path>*/source/enforcer directory.

The decide() method on page 71 illustrates the decide() method of the attributelogic class. This method sets the default decider_branch_path to DECIDER_BRANCH_FALSE. It then evaluates the request. If the identity has the appropriate attributes, the decide() method sets the decider_branch_path to DECIDER_BRANCH_TRUE. After evaluating the request, the method returns CONTINUE.

The decide() method

The following code sample show the decide() method in use.

```
Decider::Result attributelogic::decide
    (
        PropertyListElement *request, // the Enforcer XML request and
                                        data from other
                                     // evaluators.
        PropertyListElement *response, // the XML response to send back
        const bool trace,              // whether tracing is requested
       string& decider_branch_path    // tells Validator which branch to
                                        evaluate next
    ) {
    Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG, "Invoking
    attributelogic plugin");
    // default response will be to take the false branch
    decider_branch_path = DECIDER_BRANCH_FALSE;
    // evaluate can throw EvaluatorInternal if something is wrong
    if ( evaluate(*request) ) {
        decider_branch_path = DECIDER_BRANCH_TRUE;
    }
    return Decider::CONTINUE;       // keep processing
}
```

Note that attributelogic::evaluate() may throw an EvaluatorInternal exception. Decision Point plugin exceptions are defined in *<SDK_install_path>*/source/validator/evaluator.h. The defined exceptions are:

- EvaluatorFatal: Indicates that a serious error has occurred and the Policy Validator should restart.
- EvaluatorNonFatal: Indicates that a temporary error has occurred, but only this request is affected. The request is rejected.
- EvaluatorInternal: Indicates that an error occurred inside the evaluator module, but the Policy Validator can continue processing the remainder of the conditional rule. Processing will continue using the DECIDER_BRANCH_FALSE path. You should avoid using this method. You should use the EvaluatorFatal, EvaluatorNonFatal, and EnforcerException classes.

LdapConnection, User, UserSource, and UserCache

The `attributelogic` Decision Point plugin compares identity attributes against the configured security policy. If the identity attributes match the configured rule, the Decision Point plugin returns `true`. The `attributelogic` class may be configured to use the directory server to validate the identity attributes, or it can obtain them from the XML request. The `evaluate()` method in The evaluate() method on page 72 checks the `m_useLdap` flag to determine where to obtain identity attributes.

The evaluate() method

The following is an example of the `evaluate()` method:

```
bool attributelogic::evaluate(PropertyListElement &data) {
    return m_useLdap ? evaluateUsingLdap(data) :
        evaluateExpression(&m_expression, &data);
}
```

This section introduces two important classes, the `UserSource` and `UserCache`. Select Access stores and retrieves identity information from the directory server. An identity may be located in any of the configured identity locations. A `UserSource` is an object that stores the necessary information to locate identity profiles, such as a handle to the LDAP connection, the context base, the search base, etc.

For identities who are not located in a directory server, Select Access can be configured to use a pseudo-LDAP entry, called a transient identity. These profiles are not located in the directory server itself, but are represented in the cache like other profiles. When an Authentication plugin authenticates a identity who is not located in a directory server, it creates a transient identity in the Validator cache.

Obtaining references to the UserSource and UserCache on page 72 demonstrates how to obtain references to the `UserSource` and `UserCache`. The `evaluateUsingLdap()` method validates the information in the `UserSource` and `UserCache`. The `evaluateUsingLdap()` method:

1. Extracts the authenticated identity DN from the request data. The authenticated DN is only present in the request if the identity has already authenticated successfully.
2. Obtains the `UserSource` for a identity based on their authenticated DN.
3. Obtains the LDAP connection from the `UserSource`.
4. Obtains the user object from the `UserCache`. The `UserCache` uses the `UserSource` and the authenticated DN of the identity to locate the cached object.
5. Obtains the group and dynamic group memberships for a identity. These are located in the cached user object.
6. Calls `evaluateExpressionUsingLdap()` to validate that the identity attributes match the configure rule.

If any of the references in the `UserSource` or `UserCache` are invalid, an `EvaluatorInternal` exception is thrown and the Policy Validator follows the `DECIDER_BRANCH_FALSE` path.

Obtaining references to the UserSource and UserCache

The following code example shows how to obtain the appropriate references:

```
bool attributelogic::evaluateUsingLdap(const PropertyListElement &data)
{
```

```
    static const char LOCATION[] =
"attributelogic::evaluateUsingLdap()";
    const char *dn =
data.getChildStringValue(ENFORCER_AUTHENTICATED_DN);
    if ( ! dn ) {
        throw EvaluatorInternal(EnforcerException::E_INVALID_ARG,
        LOCATION,
            "Could not find Authenticated User dn in the query");
    }
    const UserSource *userSrc = UserSource::sGetByDN(dn);
    if ( ! userSrc ) {
        string message("Could not find User Source for User dn: ");
        message += dn;
        throw EvaluatorInternal(EnforcerException::E_INVALID_ARG,
        LOCATION, message);
    }
    LdapConnection *ldapConn = userSrc->getLdapConnection();
    if ( ! ldapConn ) {
        throw EvaluatorInternal(EnforcerException::E_INVALID_ARG,
        LOCATION,
            "User source has an invalid LdapConnection");
    }
    User *user = UserCache::sLocateByDN(userSrc, dn);
    if ( ! user ) {
        throw EvaluatorInternal(EnforcerException::E_INVALID_ARG,
        LOCATION,
            "User cannot be found in User Cache");
    }
    const UserMemberships **membership = user->getMemberships();
    if ( ! membership ) {
        throw EvaluatorInternal(EnforcerException::E_INVALID_ARG,
        LOCATION,
            "User Cache entry contains invalid membership info");
    }
    return (
        evaluateExpressionUsingLdap(m_expression,*ldapConn,*userSrc,dn,
        membership)
    );
}
```

The `evaluateExpressionUsingLdap()` method is called by `evaluateUsingLdap()`. The `evaluateExpressionUsingLdap()` method performs two functions. First, it builds an LDAP search filter based on the XML configuration. Then, it calls `evaluateUserSearchFilter()` or `evaluateGroupSearchFilter()`, depending on whether the filter is based on identity attributes or group memberships.

To build the search filter, `evaluateExpressionUsingLdap()` calls itself recursively to parse each filter row. The `attributeLogicPanel` configuration editor stores multiple filter rows in the same XML branch if the logical operator joining the rows is the same. Therefore, when the `evaluateExpressionUsingLdap()` method parses a logical operator, it recursively calls

itself to process each XML child node. Each child node represents a filter row. This method uses boolean short-circuit logic; it automatically stops processing when an `AND` expression returns `false` or when an `OR` expression returns `true`.

When the `evaluateUserSearchFilter()` method parses an expression, it extracts the comparison operator, the scope (i.e. identity, group, both), the attribute name, and the required attribute value. From these properties, the method builds an LDAP search filter. The `evaluateUserSearchFilter()` method calls `evaluateUserSearchFilter()` or `evaluateGroupSearchFilter()` to query the directory server using the filter. The result is passed back in the recursive call.

Search expressions

The following code sample shows how to use a search filter in your plugin:

```
bool attributelogic::evaluateExpressionUsingLdap(
    const XmlTreeNode &expression,
    LdapConnection &ldapConn,
    const UserSource &userSrc,
    const string &userDn,
    const UserMemberships ** membership
) {
    static const char LOCATION[] =
"attributelogic::evaluateExpressionUsingLdap()";
    const char *expressionName = expression.getName();
    if ( ! expressionName ) {
        throw EnforcerException(EnforcerException::E_INVALID_ARG,
        LOCATION,
            "Missing attribute name in expression");
    }
    if ( STREQ(expressionName, attributelogic_tags::LOGICAL_OPERATOR) )
{
        const char *operatorName =
            expression.getAttributeValue(attributelogic_tags::OPERATOR);
        if ( ! operatorName ) {
            throw EnforcerException(EnforcerException::E_INVALID_ARG,
            LOCATION,
                "Missing Logical Operator in expression");
        }
        const char **it = find_first_of(logicalOperatorNames,
        logicalOperatorNames
            + logicalOperatorSize, &operatorName, &operatorName + 1,
            streq);
        if ( it == logicalOperatorNames + logicalOperatorSize ) {
            throw EnforcerException(EnforcerException::E_INVALID_ARG,
            LOCATION,
                "Unknown Logical Operator in expression");
        }
        for (int i = 0; i < expression.getNumChildren(); ++i) {
            XmlTreeNode *child = expression.getChild(i);
            if ( ! child ) {
```

Parsing a logical operator joining expressions

Parse each expression in the XML node

```
                throw EnforcerException(EnforcerException::E_INVALID_ARG,
                LOCATION,
                     "Invalid XML");
            }
            bool result = evaluateExpressionUsingLdap(*child, ldapConn,
            userSrc,
                userDn, membership);
            if ( result != logicalOperatorSigns[it -
               logicalOperatorNames] ) {
                // stop if (... AND false) or (... OR true)
                return result;
            }
        }
        return true;
    }
    else if ( STREQ(expressionName,
              attributelogic_tags::COMPARE_OPERATOR) ) {
        // get Operator
        const char *operatorName =
            expression.getAttributeValue(attributelogic_tags::OPERATOR);
        if ( ! operatorName ) {
            throw EnforcerException(EnforcerException::E_INVALID_ARG,
            LOCATION,
                "Missing Compare Operator");
        }
        const char **opIt = find_first_of(operatorNames, operatorNames +
        operatorSize,
            &operatorName, &operatorName + 1, streq);
        if ( opIt == operatorNames + operatorSize ) {
            throw EnforcerException(EnforcerException::E_INVALID_ARG,
            LOCATION,
                "Unknown Compare Operator");
        }
        int operatorIndex = opIt - operatorNames;
        const char *operatorSign = operatorSigns[operatorIndex];
        bool operatorIsBinary = ! operatorUnary[operatorIndex];
        // get Scope
        const char *scopeName =
            expression.getAttributeValue(attributelogic_tags::SCOPE);
        if ( ! scopeName ) {
            throw EnforcerException(EnforcerException::E_INVALID_ARG,
            LOCATION,
                "Missing scope in expression");
        }
        const char **opSc = find_first_of(scopeNames, scopeNames +
        scopeSize,
            &scopeName, &scopeName + 1, streq);
        if ( opSc == scopeNames + scopeSize ) {
```

Evaluate expressions with recursive call

Parsing a filter expression

Extracting the scope

```
            throw EnforcerException(EnforcerException::E_INVALID_ARG,
            LOCATION,
                "Unknown Scope");
        }
        bool isUserScope = scopeUser[opSc - scopeNames];
        bool isGroupScope = scopeGroup[opSc - scopeNames];
        // get Attribute names (could be many)
        const PropertyElementVector attrNames =
        expression.getPropertyValues
        (attributelogic_tags::ATTRIBUTE_NAME);
        if ( attrNames.empty() ) {
            throw EnforcerException(EnforcerException::E_INVALID_ARG,
            LOCATION,
                "Missing attribute name");
        }
        // get Attribute value (only one)
        const PropertyElementVector attrValues =
expression.getPropertyValues(attributelogic_tags::ATTRIBUTE_VALUE);
        const char *attrValue = NULL;
        if ( operatorIsBinary &&
            ( attrValues.empty()
           || (attrValue = attrValues.front()->getStringValue()) == NULL
            || strlen(attrValue) < 1 ) ) {
            throw EnforcerException(EnforcerException::E_INVALID_ARG,
            LOCATION,
            "Missing AttributeValue property in expression");
        }
        // construct search filter
        // for each attribute in the CompareOperator list add the
           corresponding
        // condition to the search filter
        string searchFilter("(");
        searchFilter += operatorLogic[operatorIndex];
        for (PropertyElementVector::const_iterator
             nameIt(attrNames.begin());
            nameIt != attrNames.end(); nameIt++) {
            const char * attrName = (*nameIt)->getStringValue();
            if ( !attrName || *attrName == '\0' ) {
                continue;
            }
            string attrFilter;
            attrFilter += "(";
            attrFilter += attrName;
            // attr!* is represented as !(attr=*), attr!=value as
               !(attr=value)
            attrFilter += ( *operatorSign != '!' ) ? operatorSign :
            operatorSign + 1;
            if ( operatorIsBinary ) {
```

Extracting the attribute name

Extracting the list of attribute values

Building the search filter

```
                attrFilter += attrValue;
            }
            attrFilter += ")";
            if ( *operatorSign == '!' ) {
                attrFilter = "(!" + attrFilter + ")";
            }
            // Make sure that attribute exists before doing relational
               comparisons
           const char lastChar = operatorSign[strlen(operatorSign) - 1];
            if (lastChar != '*') {
                string str("(&(");
                str += attrName;
                str += "=*)";
                attrFilter = str + attrFilter + ")";
            }
            searchFilter += attrFilter;
        }
        searchFilter += ")";
        // done with the search filter, perform the search
        bool result = false;
        if ( isUserScope ) {
           result = evaluateUserSearchFilter(searchFilter, userDn,
           ldapConn);
        }
        if ( isGroupScope && !result) {
            return evaluateGroupSearchFilter(searchFilter, userDn,
            ldapConn,
                userSrc, membership);
        }
        return result;
    }
    else if ( STREQ(expressionName,
attributelogic_tags::ATTRIBUTE_LOGIC) ||
        STREQ(expressionName, attributelogic_tags::EXPRESSION_ROOT)) {
        if ( expression.getNumChildren() <= 0 ) {
            throw EnforcerException(EnforcerException::E_INVALID_ARG,
            LOCATION,
                "Missing Logical Attribute Property list");
        }
        XmlTreeNode *child = expression.getChild(0);
       if ( ! child ) {
           throw EnforcerException(EnforcerException::E_INVALID_ARG,
           LOCATION,
                "Invalid Logical Attribute Property list");
        }
        return evaluateExpressionUsingLdap(*child, ldapConn, userSrc,
            userDn, membership);
    }
```

The start of the plugin configuration

Recursive call to process remaining expression

```
    throw EnforcerException(EnforcerException::E_INVALID_ARG, LOCATION,
        "Unknown property in expression");
}
```

The `evaluateUserSearchFilter()` method in the Identity search filter searches only the profile in the directory server. Since the identity is already known, this reduces the load on the directory server by searching only the specified profile instead of searching the entire server. The `LdapConnection` obtained earlier is used to perform the search. The search will return `true` if the identity possesses the attributes required by the search filter. If the profile did not match the search filter, the method returns a value of `false`.

Identity search filter

The following code sample illustrates how to use the `evaluateUserSearchFilter()` method:

```
bool attributelogic::evaluateUserSearchFilter
(
    const string &searchFilter,
    const string &userDn,
    LdapConnection &ldapConn
) {
    auto_LDAPMessage searchResult;
    int rc = ldapConn.search(userDn.c_str(), LDAP_SCOPE_BASE,
    searchFilter.c_str()
        LdapAttributes::sNoAttributes, 0, NULL, NULL, LDAP_NO_LIMIT,
        LDAP_NO_LIMIT,
        searchResult);
    return ( rc == LDAP_SUCCESS &&
ldapConn.firstEntry(searchResult.get()) );
}
```

The `evaluateGroupSearchFilter()` method in Group search filter on page 78 searches every group and dynamic group membership the identity possesses. For each membership, this method searches each group entry in the directory server. Each group is compared against the group search filter. If any of the groups match the filter, the LDAP search will return exactly one entry, indicating success. When a match is found, this method return `true`. If none of the identity memberships match the group search filter, this method returns `false`.

Group search filter

The following code sample illustrates how to use the `evaluateGroupSearchFilter()` method:

```
bool attributelogic::evaluateGroupSearchFilter(
    const string &searchFilter,
    const string &userDn,
    LdapConnection &ldapConn,
    const UserSource &userSrc,
    const UserMemberships **membership
) {
    for (; *membership != NULL; ++membership) {
        for (UserMemberships::const_iterator it = (*membership)->begin()
            ; it != (*membership)->end(); ++it) {
```

```
            auto_LDAPMessage searchResult;
            int rc = ldapConn.search(*it, LDAP_SCOPE_BASE,
            searchFilter.c_str(),
                LdapAttributes::sNoAttributes,  0, NULL, NULL,
                LDAP_NO_LIMIT, LDAP_NO_LIMIT, searchResult);
            if ( rc == LDAP_SUCCESS &&
                 ldapConn.firstEntry(searchResult.get()) ) {
                return true;
            }
    }
}
        return false;
}
```

Building the AttributeLogic Example

To build the `attributelogic` example, install GNU make and execute the `gmake` command from the *<SDK_install_path>*`/source/validator/plugins` directory. . Refer to Building Examples in the SDK on page 23 for more information about the `gmake` command.

The `gmake` command should produce the output shown in Makefile output on page 79. If you are using a different compiler you will need to configure it to mimic the options specified below. When you modify the `Makefile`, make sure the include and library directory paths reflect the current operating system.

Makefile output

The following is sample output for the Decision Point plugin:

```
g++ -g -O2 -pipe  -DUSE_SSL   -D_PTHREADS -D_REENTRANT -DPOSIX_THREADS
    -DFD_SETSIZE=8192 -Wall -D_POSIX_THREAD_SAFE_FUNCTIONS -DACE_HAS_SSL
    -I../../solaris/include -I../../include -I.. -I../../enforcer -I../
      ../radius
    -DACE_HAS_EXCEPTIONS  attributelogic.cpp  -g -O2 -pipe  -L../../
     <platform?/lib
    -lenforcer -lopenssl -licuuc -licudata -lACE_SSL -lldap50 -laceclnt
    -lradlib
    -lcurl -lsocket -lnsl -ldl -lthread  -o attributelogic
```

5 Custom Security Services: the Enforcer API

The Enforcer API allows you to integrate Select Access security services with most products and custom components. This chapter describes how to use the Enforcer API to provide security services for Java, C/C++, and COM objects.

Chapter Overview

Topics in this chapter include subjects that describe the Enforcer plugin, its API, and the plugins you can build with this API to customize the Enforcer plugin's behaviour:

- Understanding the Enforcer plugin on page 81
- Understanding the Enforcer API on page 82
- Creating an Enforcer Plugin With the Java Enforcer API on page 87
- Creating an Enforcer Plugin With the C/C++ Enforcer API on page 104
- Creating an Enforcer Plugin With the COM Enforcer API on page 123
- Including Site-specific Data on page 142

Understanding the Enforcer plugin

Figure 13 illustrates how the Enforcer plugin works with the Policy Builder and Policy Validator to extend security services. Enforcer plugins control access to network resources. Enforcer plugins intercept resource requests and delegate authorization decisions to a Policy Validator. The Enforcer is responsible for enforcing the Policy Validator response. The Enforcer plugin and the Policy Validator use XML to communicate requests and enforcement instructions. This allows Enforcer plugins to send Policy Validators all necessary information for determining access control. It also enables the Policy Validator to cue the Enforcer plugin to request identity authentication, determine what type of authentication is allowed, locate login forms, or take any other action.

Enforcer plugins typically use a server API to intercept resource requests from a product. For example, the Sun/Netscape/iPlanet Enforcer plugin uses the NSAPI to intercept and authorize HTTP requests.

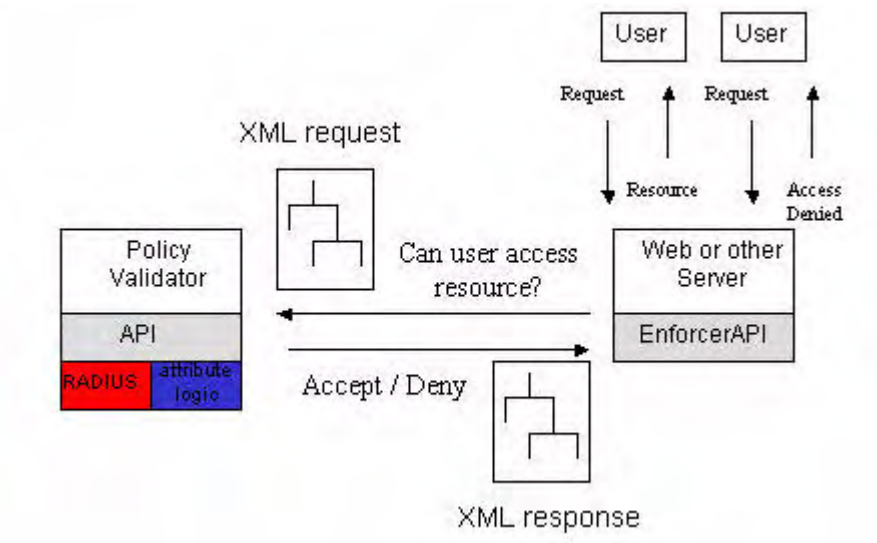

Figure 13 How Enforcer Plugins Work with Select Access Components

Understanding the Enforcer API

The primary function of a Enforcer plugin is to send authentication and authorization requests to the Policy Validator and enforce the response. The Enforcer API is a collection of interfaces, classes, and utilities that facilitate communication with Policy Validators. Select Access provides an Enforcer API interface for developing C/C++, Java, and COM components. Using the Enforcer API, you can quickly add security services:

- Identity authentication
- Single Sign-on
- Authorization
- Secure remote logging
- Personalization attributes

> For details on how to process personalization attributes, see Chapter 6, Using Personalization Attributes: the Personalization API.

To enable developers to integrate Select Access security services with the widest variety of products, the Enforcer API supports Java, C/C++, and COM interfaces. Three example Enforcer plugins are examined in this chapter, one for each Enforcer API interface:

- Servlet Enforcer plugin: This Enforcer plugin demonstrates the Java Enforcer API. This enforcer connects the servlet API to the Enforcer API. The servlet acts as a security filter that controls access to web resources using Select Access.
- Apache 2 Enforcer plugin: This Enforcer plugin demonstrates the C/C++ Enforcer API. It connects the Apache API to the Enforcer API. This plugin provides Select Access security services for the Apache server.
- WSE Enforcer plugin: This Enforcer plugin demonstrates the COM Enforcer API. It connects the .Net SOAP interface to the Enforcer API. This plugin provides Select Access security services to .Net resources.

How the Enforcer API Works

The Enforcer API encapsulates secure, reliable communication with Policy Validators and Select Audit. The API provides an interface to send requests and receive responses from Policy Validators. Behind the scenes, the Enforcer API manages the secure connections with Policy Validators. The API encapsulates the channel security, failover support, session time-out, and correlating responses with requests.

To create an Enforcer plugin

1. Create an Enforcer object. This step is usually performed only when the Enforcer plugin is initialized. Enforcer objects are used to communicate with the Policy Validator. When you create an Enforcer object, the Enforcer API locates the Enforcer plugin's configuration in the Policy Store. A local bootstrap configuration file, `enforcer.xml`, is used to locate the Enforcer plugin's configuration entry in the Policy Store.
2. Extract necessary data from the environment. This may include processing a form to obtain the ID and password, accessing server API variables to obtain the requested resource URL, reading environment variables to get SSL parameters, etc.
3. Build an XML request to send to a Policy Validator. The request must include the requested URL so the Policy Validator knows which authentication and authorization rules to evaluate. Typical XML requests include properties such as the ID, password, source IP, source hostname, SSL properties, single sign-on nonces, etc.
4. Use the Enforcer object to send the request to a Policy Validator.
5. Evaluate the response. The Enforcer plugin must take different actions depending on the Policy Validator response. Typical actions include prompting a identity for authentication credentials, providing access to the resource, displaying an access denied page, associating a nonce with a session (e.g. adding a nonce to a web browser cookie), providing personalization data to applications, etc.
6. Process the next request.

The Enforcer API uses XML to communicate with the Policy Validators. A sample XML request is shown in Example of a simplified Policy Validator request. The service and path properties are required properties. Any XML property may be added to a request.

Example of a simplified Policy Validator request

```
<PolicyValidatorQuery>
    <PROPERTY NAME="service">http://mycompany.com:8080</PROPERTY>
    <PROPERTY NAME="path">/secure/form</PROPERTY>
    <PROPERTY NAME="srcIP">10.10.10.6</PROPERTY>
    <PROPERTY NAME="srcPort">1107</PROPERTY>
    <PROPERTY NAME="dstIP">10.10.10.7</PROPERTY>
    <PROPERTY NAME="dstPort">8080</PROPERTY>
    <PROPERTY NAME="native_auth">native_password</PROPERTY>
    <PROPERTY NAME="native_auth">native_register</PROPERTY>
    <PROPERTY NAME="site_data">time: Tue Jul 10 13:14:40 2001</PROPERTY>
    <PROPERTY NAME="owner">gandalf</PROPERTY>
    <PROPERTY NAME="size">268</PROPERTY>
    <PROPERTY NAME="method">GET</PROPERTY>
    <PROPERTY NAME="queryID">1</PROPERTY>
 </PolicyValidatorQuery>
```

Example of a simplified Policy Validator reply shows a sample reply from a Policy Validator query. Policy Validators can return any type of XML data to Enforcer plugins. The sample reply allows access to the requested resource.

Example of a simplified Policy Validator reply

The following example is a simplified Policy Validator reply:

```
<PolicyValidatorReply>
   <PROPERTY NAME="queryID">1</PROPERTY>
   <PROPERTY NAME="authenticated_dn">uid=ssmith,ou=users,o=ca.hp.com
   </PROPERTY>
   <PROPERTY NAME="devo_groups">users</PROPERTY>
   <PROPERTY NAME="nonce">bWFub3dhci5jYS5iYWx0aW1vcmUuY29tOjk5ODh8dWlkPW
    dhbmRhbGYsb3U9dXNlcnMsb3U9c3RldmUsbz1jYS5iYWx0aW1vcmUuY29tfEx8
    cGFzc192YWxpZGF0b3J8O0zd4DuLSR2cw0K7Yp5pTb04TSWWxJJqqc7rUszF16
    atYOqTLnVqt+O4a6OFrWQegqu89L17Kwzv4n3XWIwFpsP+wJ8V1eEw4KG9c+REkJFs67
    bLKZuWZ6xNz2Xpgs7FLb5s2O3iwswvuDvcBNOZqF2pbOproktsiuaC36SqxmKLSzY/
    </PROPERTY>
   <PROPERTY NAME="action">ALLOW</PROPERTY>
</PolicyValidatorReply>
```

For a complete list of predefined XML property tags, see the *HP OpenView Select Access 6.2 Developer's Reference Guide*.

Enforcer API Classes and Utilities

Select Access ships with classes that store and process XML documents, `PropertyLists` and `Properties`. You must use the Select Access classes to build Policy Validator requests and process responses. Select Access also provides Enforcer plugin classes that encapsulate communication with the Policy Validator. Select Access provides an implementation for each object model: C/C++, Java, and COM.

The Enforcer API interface is dependent on the object model you are using. Consult the appropriate section for details on the Enforcer API classes and utilities:

- For Java classes and utilities, see Java Enforcer API Classes and Utilities on page 88.
- For C/C++ classes and utilities, see C/C++ Enforcer API Classes and Utilities on page 105.
- For COM classes and utilities, see The COM Enforcer API Classes and Utilities on page 126.

Configuring the Enforcer API

Every Enforcer plugin must be configured so that it knows what Policy Validators to communicate with, which files do not require authorization, how to process single sign-on and multiple domain single sign-on, etc. Using the Select Access Setup Tool you can configure these properties for a local Enforcer plugin. From the Setup Tool, click the **Configure** button for the appropriate type of Enforcer plugin you are configuring, e.g. Apache 2 Enforcer plugin. If you are configuring a custom Enforcer plugin, select the **Generic Enforcer** option.

When you create an Enforcer object, you specify an Enforcer plugin name. The Enforcer API locates the bootstrap file named *<SA_Install>*`/bin/enforcer_`*<enforcerName>*`.xml`. An example bootstrap XML file is shown in Code example<$paratext>. The bootstrap file tells the

Enforcer API which Policy Validator to contact to retrieve the Enforcer plugin configuration from the Policy Store. For more information on initializing the Enforcer API with bootstrap files, see the following sections:

- To create Java Enforcer objects, see Creating an Enforcer Object on page 90.
- To create C/C++ Enforcer objects, see Creating an Enforcer Object on page 107.
- To create COM Enforcer objects, see Creating an Enforcer Object on page 127.

An example enforcer_<enforcerName>.xml file

The following is an example of a bootsrap XML file for an Enforcer plugin:

Certificate data used for SSL connections

The enforcer ID, used to locate the Enforcer's configuration

The Validator host and port to use to obtain configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<enforcerBootConfig>
    <clientSSLCert>
        -----BEGIN CERTIFICATE-----
        ... Base64 encoded certificate ...
        -----END CERTIFICATE-----
    </clientSSLCert>
    <clientSSLKey>
        -----BEGIN RSA PRIVATE KEY-----
        ... Base64 encoded private key ...
        -----END RSA PRIVATE KEY-----
    </clientSSLKey>
    <enforcerID>chat.myHost.com:ChatEnforcer</enforcerID>
    <serverCertAllowSelfsigned>false</serverCertAllowSelfsigned>
    <serverCertAllowUnknownCA>false</serverCertAllowUnknownCA>
    <serverCertCA>
        -----BEGIN CERTIFICATE-----
        ... Base64 encoded certificate
        -----END CERTIFICATE-----
    </serverCertCA>
    <serverCertOcspUrl />
    <serverHost>validator.myHost.com:9988</serverHost>
    <serverPort>9988</serverPort>
    <serverUseSSL>true</serverUseSSL>
</enforcerBootConfig>
```

Once an Enforcer plugin has been configured, you may manage it remotely using the Policy Builder. To manage Enforcer plugins from the Policy Builder, select the **Tools** → **Component Configuration**. You can then select the ID of the Enforcer plugin you wish to configure.

Development Considerations

Every plugin is unique, depending on the server or component being secured. The examples used in this chapter focus on web-oriented Enforcer plugins. The Enforcer API may be used to secure different protocols and services. This section details some of the common tasks an Enforcer plugin must perform.

Determining if authorization is required

The Setup Tool and Policy Builder allow you to specify files and domains that should not be processed for authorization. A Enforcer plugin should use the demonstrated calls to the Enforcer API to ensure that a request needs to be authorized. These Enforcer API calls are demonstrated in the code examples provided.

Presenting login screens

Many times an identity will attempt to access a resource before they have authenticated. When this happens, the Policy Validator returns a `DENY` code to the Enforcer plugin. The Policy Validator response includes properties that identify whether authentication should be performed, what type of authentication is allowed, and where the login screens can be located.

It is the Enforcer plugin's responsibility to actually present the login screen to the identity. The examples in this chapter use the HTTP response output stream to display HTML login forms to identities. Your Enforcer plugin will have to provide a mechanism to prompt an identity for authentication credentials.

Maintaining session state

When you prompt an identity for authentication, it is important to save the session state before beginning the authentication process. This prevents the application from losing the identity's original request data while performing authentication. This is particular important for stateless protocols, such as HTTP.

The example Enforcer plugins save the original HTTP request data in a special property before redirecting the seb browser to a login page. The original request data is restored when the identity has completed the login process.

Handling single sign-on nonces

In order to facilitate single sign-on, there must be a mechanism for the client to securely send a nonce, or token, to the server. The nonce contains encrypted information that identifies the authenticated identity. It is important that the nonce be communicated securely. If the nonce is not properly secured, a hacker could intercept the nonce and assume the identity.

For the example Enforcer plugins in this chapter, web cookies are used to transmit nonces. After a identity authenticates, the Enforcer plugins place the nonce in a web cookie. The browser will then send the cookie to all servers in the same domain. This provides single sign-on within a domain for web transactions.

Performing multiple domain single sign-on

The cookie security model prohibits the browser from sending cookies to servers on a domain that is outside of the namespace where the cookie originated. This prevents single sign-on across different domains. To provide multiple domain single sign-on, web-based Enforcer plugins must take additional steps. The example Enforcer plugins provide multiple domain single sign-on by following these steps:

1. Determine if the Enforcer plugin is configured to allow the referring host to provide multiple domain single sign-on.
2. Redirect the browser to an authentication URL on the referring host. Save the session state and original URL in the request variables.

3 The referring server authenticates the identity. The referring server redirects the client's browser back to the original server. A nonce is placed inside of the redirect URL.

4 The original server extracts the nonce, queries the Policy Validator, generates cookies for the new domain, and redirects the identity to the original URL.

The Enforcer API provides methods to determine if multiple domain single sign-on is allowed for the referring host. The Enforcer plugin may be configured to allow multiple domain single sign-on using the Enforcer plugin configuration tool. Use the Setup Tool or the Policy Builder to configure an Enforcer plugin for multiple domain single sign-on. The Multiple Domain Single Sign-on on page 119 illustrates how to support additional domains for single sign-on. Configure the Enforcer plugin Multiple DNS domain SSO panel to reflect the login URLs of Enforcer plugins located on different domains.

Creating an Enforcer Plugin With the Java Enforcer API

This section demonstrates how to build a Java Enforcer plugin. In this section you will learn how to:

- Initialize an Enforcer object
- Extract authorization data
- Build a Policy Validator request
- Query a Policy Validator
- Enforce the Policy Validator decision
- Perform multiple domain single sign-on
- Deploy a sample application

The ServletFilter Example

The `ServletFilter` class provides an implementation of a servlet filter that enforces policy using Select Access. Servlet filters are a standard way to filter or transform servlet HTTP requests and responses. A servlet filter can be used to perform authorization before calling the requested servlet.

If you intend to build your own servlet Enforcer plugin, ensure you download the `jce1_2_2.jar` file from Sun's web site. Due to legal restrictions, Select Access can not distribute the file with its installer.

The `ServletFilter` connects the servlet filter interface to the Java Enforcer API. HTTP request variables are used to query a Policy Validator. Depending on the response, `ServletFilter` will either display an error page, a login page, an access denied page, or allow the requested resource to generate the output.

The `ServletFilter` class also demonstrates how to place the single sign-on nonce in a cookie for future access and process multiple domain single sign-on. To use the example, you should be familiar with Java, servlets, and HTTP.

Java Enforcer API Classes and Utilities

The Java Enforcer API classes are located in the archive *<SDK_install_path>*/shared/EnforcerAPI.jar. Enforcer plugins will also need to access the common XML and logging classes located in *<SDK_install_path>*/shared/shared.jar. Be sure to include these archives in your CLASSPATH.

The Java Enforcer API consists of the Enforcer class. The Enforcer class is the core of the Enforcer API. The Enforcer class represents a Java Enforcer object. It provides methods to create and send Policy Validator queries. This class also provides methods to access the Policy Validator response.

The shared.jar archive contains the common classes:

- Logger: The Logger provides an interface for recording messages to a number of destinations, including Select Audit.
- XmlElement: The XmlElement class stores generic XML nodes.
- Property: The Property interface is implemented by PropertyElement and PropertyListElement.
- PropertyElement: The PropertyElement class represents an XML element that stores a name-value pair. The value may be any data type.
- PropertyListElement: The PropertyListElement class represents a list of PropteryElements and nested PropertyListElements.

Importing the Enforcer Classes

The following example shows the Enforcer API classes you should import. These classes are located in the EnforcerAPI.jar and shared.jar archives.

```
import com.hp.selectaccess.enforcer.*;
import com.hp.selectaccess.util.Logger;
import com.hp.selectaccess.util.XmlElement;
import com.hp.selectaccess.util.PropertyElement;
import com.hp.selectaccess.util.PropertyListElement;
import com.hp.selectaccess.util.Property;
```

The ServletFilter and ServletTransaction Classes

As shown in the example below, the ServletFilter class has two important methods: init() and doFilter(). The servlet framework calls the init() method to initialize the filter when it is first used. This method creates a configured Enforcer plugin object and stores it in the m_enforcer field. After initialization, the doFilter() method is called to process each HTTP request. The skeleton for the ServletFilter class is shown below.

```
public class ServletFilter implements Filter {
    public static final String SERVLET_FILTER_NAME = "ServletFilter";
    public static final String ENFORCER_CONF_FILE = "enforcer_conf";
    public static final String ENFORCER_DEBUG_LEVEL = "debug_level";
    public static final String VIRTUAL_HOSTNAME = "virtual_hostname";
    public static final String ENFORCER_HANDLE = "enforcer_handle";
    private FilterConfig m_config = null;
```

```
    private Enforcer m_enforcer = null;
    private String m_virtual_hostname = null;
    private String m_server_software = null;

    public void init(FilterConfig config) throws ServletException { ... }
    public void doFilter
        (
            ServletRequest request,
            ServletResponse response,
            FilterChain chain
        ) throws IOException, ServletException { ... }
    public void destroy() {m_config = null; m_enforcer = null;}
}
```

Filtering servlet requests

The `doFilter()` method uses the `ServletTransaction` class to process the HTTP request. The following code sample illustrates how the `doFilter()` method passes the HTTP request and response to the `ServletTransaction` object and calls the `isAuthorized()` method to determine if access to the requested resource is allowed. If access is allowed, the remaining filters in the filter chain are executed. The HTTP response is generated by the requested servlet. If access is denied or authentication is required, the `isAuthorized()` method builds the appropriate HTTP response.

```
/**
 * Filter an HTTP request by creating a servlet transaction,
 * and then having it check with the validator to see whether
 * the operation is allowed.
 *
 *          request          The incoming HTTP request.
 *          response         The response being generated.
 *          chain            Downstream filters and servlets.
 */
public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain)
    throws IOException, ServletException
{
    HttpServletRequest httpReq = (HttpServletRequest) request;
    HttpServletResponse httpRes = (HttpServletResponse) response;
    try {
        ServletTransaction servletTransaction =
        new ServletTransaction(httpReq, httpRes, m_virtual_hostname,
         m_enforcer);
        servletTransaction.setServerSoftware(m_server_software);
        if (servletTransaction.isAuthorized()) {
            chain.doFilter(request, response);
        }
    }
    catch (Exception e) {
```

Check authorization

```
            m_enforcer.log(Logger.LOG_LEVEL_ERROR,
                "ServletFilter error in ServletTransaction: " + e);
        }
    }
```

The ServletTransaction class

The `ServletTransaction` class extends the `WebTransaction` class. The `WebTransaction` class has one important method, `isAuthorized()`. This method determines if authorization is required. If authorization is required, `isAuthorized()` builds an XML request from the HTTP request values, queries the Policy Validator, and enforces the response. Depending on the response from the Policy Validator, the `isAuthorized()` method calls the abstract methods `allow()`, `deny()`, `sendRedirect()`, `sendBasicAuthPage()`, etc. The `ServletTransaction` class overrides these methods to provide servlet specific interaction. These methods use the HTTP response output stream to display login forms and error pages. The skeleton for the `ServletTransaction` class, with inherited methods, is provided in the following example.

```
public class ServletTransaction extends WebTransaction {
    protected XmlElement              m_query;
    protected ValidatorResponse    m_val_response;

    ... Other field declarations ...

    public ServletTransaction
        (
            HttpServletRequest request,
            HttpServletResponse response,
            String serverName,
            Enforcer enforcer
        ) throws EnforcerException { ... }

    public boolean isAuthorized() { ... }

    protected boolean allow() { ... }
    protected boolean deny() { ... }
    protected boolean sendRedirect(String url) { ... }
    protected boolean sendBasicAuthPage(String realm) { ... }

    ... More internal methods ...

}
```

Creating an Enforcer Object

When `ServletFilter` is initialized, it creates an Enforcer object and stores it in the `m_enforcer` field. The Enforcer object is used to parse URLs for dangerous characters, determine if authorization is needed, query the Policy Validator, and obtain the response. The

Enforcer object can process multiple authorization requests and it is safe to use in multithreaded environments. The `ServletFilter` class uses a single Enforcer object to authorize each of the HTTP requests.

Instantiating new instances of an Enforcer object

When instantiating a new Enforcer object, you need to use the following attributes:

- `String saConfigFileName`: The name of the Select Access configuration file. If it's null, the constructor loads the default configuration file, `<SA_install_path>/selectaccess.conf`.
- `String enforcerConfigFileName`: The name of the Enforcer's configuration file. If it's null, the constructor loads the default Enforcer XML file from `<SA_install_path>/bin/enforcer.xml`.

  If you set the filename to null, administrators can run the Setup Tool to set the parameters for the default XML configuration file. For details, see the *HP OpenView Select Access 6.2 Installation Guide*.

- `String loggingName`: The customizable name of the Enforcer used in log outputs.
- `String enforcerTypeName`: The customizable string to identify the Enforcer type.
- `boolean initLogging`: A parameter that turns logging of the Enforcer on and off.
- `int debug_level`: A numeric value that sets the debug level.

For specific details on any of these parameters, see Chapter 10, Java Enforcer API in the *HP OpenView Select Access 6.2 Developer's Reference Guide*.

Initializing the servlet

To create an Enforcer object, you must provide the constructor with a bootstrap Enforcer plugin configuration file. This file locates the Enforcer object's configuration in the Policy Store. As shown in the following code example, the `init()` method for `ServletFilter` extracts the Enforcer plugin configuration file name from the servlet configuration. For details on the `enforcer.xml` file, see Configuring the Enforcer API on page 84.

```
public void init(FilterConfig config) throws ServletException {
    m_config = config;
    ServletContext context = m_config.getServletContext();
    m_server_software = context.getServerInfo();
    m_virtual_hostname = config.getInitParameter(VIRTUAL_HOSTNAME);
    // Try to configure an enforcer handle for this servlet filter.
    String enforcerConf = config.getInitParameter(ENFORCER_CONF_FILE);
    if (enforcerConf != null) {
        int dbg = 0;
        String debugLevel =
config.getInitParameter(ENFORCER_DEBUG_LEVEL);
        if (debugLevel != null) {
            dbg = Integer.parseInt(debugLevel);
        }
        m_enforcer = new Enforcer(null, enforcerConf,
        SERVLET_FILTER_NAME,
            "servlet", true, dbg);
```

Enforcer config file name

Creating a new Enforcer

```
        }
        // If configuration for this filter was not specified, try to inherit
        // a handle from the servlet container.
        else {
            m_enforcer = (Enforcer)context.getAttribute(ENFORCER_HANDLE);
            if (m_enforcer == null) {
                throw new ServletException("Unable to find/construct enforcer
                handle");
            }
        }
        // Clear the enforcer handle's user data character set, since the
        // servlet container will be handling character set conversion
        m_enforcer.clearUserDataCharacterSet();
    }
```

Extracting Authorization Data

Once an Enforcer object has been created, the `ServletFilter` is ready to process requests. The `doFilter()` method creates a `ServletTransaction` object to process each HTTP request. The `ServletTransaction` constructor uses the HTTP request to extract all the data that might be used during policy evaluation. The HTTP request contains the requested resource name, the HTTP method, SSL parameters, HTTP headers, and browser cookies. This information will be used to build a Policy Validator query.

```
public ServletTransaction(HttpServletRequest request,
HttpServletResponse response,
    String serverName, Enforcer enforcer) throws EnforcerException
{
    super(enforcer, request.getScheme(),
        serverName != null ? serverName : request.getServerName(),
        request.getServerPort(), UnescapeUrl(request.getRequestURI()),
        request.getQueryString(), request.getMethod(),
        request.getRemoteAddr());
    m_request = request;
    m_response = response;
    m_app_name = "ServletTransaction";
    m_referer = m_request.getHeader(REFERER_HEADER);
    m_host_header = m_request.getHeader(HOST_HEADER);
    if (m_host_header == null)
        m_host_header = m_server_name;
    Cookie[] cookies = m_request.getCookies();
    if (cookies != null) {
        for (int i=0;i<cookies.length;i++) {
            if
(cookies[i].getName().equals(DataStrings.AUTH_COOKIE_NAME))
                m_auth_cookie = cookies[i].getValue();
            if
(cookies[i].getName().equals(DataStrings.REGISTER_COOKIE_NAME))
                m_reg_cookie = cookies[i].getValue();
```

Extracting HTTP request data

Check for SSO cookie

```
        }
    }

    ... Initialize personalization data ...

}
```

Determining if Authorization is Needed

The `isAuthorized()` method of the `ServletTransaction` class is used to determine if access is allowed. Before querying the Policy Validator, the `isAuthorized()` method uses the `m_enforcer` object to determine if authorization is required. There are several reasons why authorization might not be necessary:

- The Enforcer plugin can not be configured. In this case, the Policy Validator can not be contacted, and access should be denied.
- The URL contains invalid characters. URLs that contain invalid characters are not be processed, and the request is rejected. An example of a dangerous character is the "." directory operator.
- The Enforcer plugin is configured to allow unrestricted access to the domain.
- The Enforcer plugin is configured to allow unrestricted access to the file.
- The request is part of a multidomain single sign-on redirect.

The following example illustrates how to check for each of these conditions:

```
public boolean isAuthorized() {
    if (!m_enforcer.isConfigured()) {
        m_enforcer.configure();
        // bail out early if we could not configure
        if (!m_enforcer.isConfigured())
            return deny();
    }
Suspicious URL
    if (m_enforcer.isEvilURL(m_path)) {
        log(Logger.LOG_LEVEL_WARNING, "rejecting suspicious url '" +
        m_path + "'");
        disableCaching();
        return deny();
    }
    String full_url = m_protocol + "://" + m_host_header + m_path;
    if (m_http_query != null && m_http_query.length() > 1)
        full_url = full_url + "?" + m_http_query;
    StringBuffer orig_url = new StringBuffer("");
Multidomain SSO
    if (isSSORedirectURL(m_http_query, orig_url)) {
        // MD-SSO redirect 2
        String r2 = constructSSOAuthURL(m_auth_cookie,
        orig_url.toString());
        return ssoRedirect(r2);
    }
    if (m_enforcer.isPassthroughDomain(m_server_name))
```

```
        return allow();
    if (m_enforcer.isIgnoredFile(m_path))
        return allow();

    ... Build the XML query ...

    ... Query the Validator ...

    ... Enforce the response ...

}
```

The multiple domain single sign-on redirect extracts the nonce for the current domain from an existing SSO cookie. The nonce from the existing cookie is added to the URL in the redirect. This effectively passes the nonce back to the original server using the URL to store the nonce. For more information on multiple domain single sign-on, see Performing multiple domain single sign-on on page 86.

Building the Policy Validator Request

After the `isAuthorized()` method determines that authorization is needed, it builds a Policy Validator request. The XML query is initialized using the `XmlQueryInit()` method. This creates the Policy Validator query element and adds properties for the service, path, and a query identity. The service and path properties are used by the Policy Validator to locate the security policy for the resource. The query identity is useful for correlating audit logs. Other properties are added using the `appendPropertyValue()` method as shown in following cod sample. You can add any property to the Policy Validator request. The Enforcer API provides constants for common properties.

```
public boolean isAuthorized() {

    ... Verify that authorization is required ...

    m_query = Enforcer.XmlQueryInit(m_service, m_path);
    if (m_enforcer.isEnableCookies()) {
        m_query.appendPropertyValue(DataStrings.ENFORCER_NATIVE_NONCE,
            DataStrings.ENFORCER_ENABLE);
    }
```

Adding a request property

```
    m_query.appendPropertyValue(DataStrings.ENFORCER_PROTOCOL,
    m_protocol);
    m_query.appendPropertyValue(DataStrings.ENFORCER_SRCIP,
    m_client_addr);
    if (m_auth_cookie != null && !m_auth_cookie.equals("logout"))
        m_query.appendPropertyValue(DataStrings.ENFORCER_NONCE,
        m_auth_cookie);
    if (m_reg_cookie != null)
        m_query.appendPropertyValue(DataStrings.ENFORCER_REGISTER_NONCE,
        m_reg_cookie);
    if (!m_enforcer.getQueryLevel().equals(DataStrings.QUERY_MINIMAL)) {
        if (m_http_query != null && m_http_query.length() > 1) {
            try {
```

```
                m_query.appendPropertyValue(DataStrings.ENFORCER_HTTP_QUERY,
                m_http_query);
                    addUrlEncoded(DataStrings.ENFORCER_HTTP_QUERY_LIST,
                    m_http_query);
                } catch (EnforcerException e) { }
            }
            m_query.appendPropertyValue(DataStrings.ENFORCER_METHOD,
            m_method);
        }
        if (m_enforcer.getQueryLevel().equals(DataStrings.QUERY_MAXIMAL)) {
            if (m_server_software != null && m_server_software.length() > 1)
{
                m_query.appendPropertyValue(DataStrings.ENFORCER_SERVER,
                    m_server_software);
            }
        }
```

Adding a SSO nonce

```
        StringBuffer sso_nonce = new StringBuffer("");
        if (isSSOAuthURL(full_url, sso_nonce)) {
            // set SSO nonce in query before sending query to validator
            m_query.setChildStringValue(DataStrings.ENFORCER_NONCE,
            sso_nonce.toString());
        }

        ... Add other properties to the request ...

        ... Query the Validator ...

        ... Enforce the response ...

}
```

If the identity has already authenticated, an authentication cookie will exist containing a single sign-on nonce. If a nonce is available, it should be included with the request. The Policy Validator uses the nonce to identify identities who have already authenticated. For more information on supporting single sign-on, see Handling single sign-on nonces on page 86.

Querying the Policy Validator

Once the Policy Validator query is built, use the `XmlQuerySend()` method to send the query to the Policy Validator. This method blocks communication until the Policy Validator response is received. The XML response and a return code are stored in the `ValidatorResponse`.

```
m_val_response = m_enforcer.XmlQuerySend(m_query);
```

Enforcing the Policy Validator Response

After querying the Policy Validator, the response must be processed to enforce the decision. The `getRetVal()` method may be used to determine if the request was allowed, denied, or encountered an error. The `getXmlElement()` method returns the XML element containing the Policy Validator reply. Processing a Policy Validator response on page 96 illustrates how to use both the `getRetVal()` and the `getXmlElement()` methods.

The `allowedByValidator()` method uses the `getRetVal()` method determine if the request was allowed or denied. If the request was allowed, the `allowedByValidator()` method looks for a single sign-on nonce and a registration nonce. These are obtained from the XML reply returned by the `getXmlElement()` method. If present, the nonces are added to HTTP cookies. The cookies distinguish the identity during future transactions. For more information on supporting single sign-on, see Handling single sign-on nonces on page 86.

If the request was denied, the `allowedByValidator()` method looks to see if the XML reply specified any authentication hints. Authentication hints instruct the Enforcer plugin how to authenticate identities. The Enforcer plugin uses this property to determine what type of login form to send to the identity.

Processing a Policy Validator response

The following example illustrates uses both the `getRetVal()` and the `getXmlElement()` methods to process Policy Validator responses.

```
public boolean allowedByValidator() {

    ... Query the validator ...

    if (m_val_response.getRetVal() == Enforcer.ENFORCER_ERROR_ACTION) {
        return false;
    }

    ... Set personalization ...

    m_reply = m_val_response.getXmlElement();
    String nonce = m_reply.getChildStringValue
          (DataStrings.ENFORCER_NONCE)
    if (nonce != null) {
        setCookie(DataStrings.AUTH_COOKIE_NAME, nonce, "/",
            m_enforcer.getCookieDomainName(), -1);
    }
    String registrationCookie =
        m_reply.getChildStringValue(DataStrings.ENFORCER_REGISTER_NONCE)
    if (registrationCookie != null) {
        String lifetime = getStringFromReply
             (DataStrings.ENFORCER_REGISTER_LIFE);
        int expiry = -1;    // by default, make it a session cookie
        if (lifetime != null && lifetime.length() > 0) {
            expiry = Integer.parseInt(lifetime);
        }
        setCookie(DataStrings.REGISTER_COOKIE_NAME, registrationCookie,
"/",
```

Checking Validator return code

Obtaining the full XML response

Checking for a nonce

```
                m_enforcer.getCookieDomainName(), expiry);
        }
        if (m_val_response.getRetVal() == Enforcer.ENFORCER_ALLOW_ACTION) {
            return true;
        }
        m_auth_hint = m_enforcer.getAuthHint(m_reply, m_pass_hint);
        if (m_auth_hint != null) {
            m_form = m_enforcer.getTransformedForm(m_auth_hint, null);
        }
        else {
            // zero auth_hints, so check for generic form data
            m_form_hint = m_reply.getChild(DataStrings.ENFORCER_FORM_DATA);
            if (m_form_hint != null) {
                m_form = m_enforcer.getTransformedForm(m_form_hint, null);
            }
        }
        return false;
    }
```

Checking for login request

Setting a web cookie in the HTTP response

If a single sign-on nonce is included in the Policy Validator reply, a web cookie is set so that future web transactions will include the nonce and the identity will not have to re-authenticate. The following example shows how to set a web cookie from a filter or servlet. The following code may be used to create any type of cookie:

```
/**
 * formats a cookie in accordance with rfc 2109, and sends a Set-cookie
 * header to the Web client
 *
 * param   name       the cookie name
 * param   value      the cookie value
 * param   path       the cookie path, typically "/"
 * param   domain     the cookie domain, null to disable
 * param   duration   the cookie lifetime in seconds, -1 for session
   cookies, 0 for delete now
 */
protected void setCookie(String name, String value, String path,
    String domain, int duration)
{
    Cookie cookie = new Cookie(name, value);
    if (path != null) {
        cookie.setPath(path);
    }
    if (domain != null && domain.length() > 0) {
        cookie.setDomain(domain);
    }
    cookie.setMaxAge(duration);
    m_response.addCookie(cookie);
}
```

Displaying Allowed, Denied, and Login messages

When the `allowedByValidator()` method returns true, the `WebTransaction` framework calls the abstract `allow()` method. If identity authentication is required, the `WebTransaction` extracts the login form name and calls the abstract `sendDynamicForm()` method to display the login page. If access is denied and there are no authentication hints, the `WebTransaction` framework calls the abstract `deny()` method. These are defined in `ServletTransaction` and is illustrated below.

Send a dynamic login form

```
/**
 * Send a dynamic form to the Web client
 *
 * param    url    an optional redirect or refresh URL to send in the form
 * return   false
 */
protected boolean sendDynamicForm(String url) {
    if (url != null) {
        String header = "1; URL=" + url;
        m_response.addHeader(REFRESH_HEADER, header);
    }
    try {
        java.io.PrintWriter out = m_response.getWriter();
        m_response.setContentType(HTML_CONTENT_TYPE);
        if (m_form == null) {
            m_form = FORM_ERROR;
        }
        out.println(m_form);
        out.close();
    }
    catch (java.io.IOException e) {
        logError("sendDynamicForm failed: " + m_form);
        sendError(HttpServletResponse.SC_FORBIDDEN);
    }
    return false;
}
```

Send login request using HTTP Basic authentication

```
/**
 * Send an http-basic-auth page to the Web client
 *
 * param     realm    the authentication realm
 * return    false
 */
protected boolean sendBasicAuthPage(String realm) {
    String header = "Basic realm=\"" + realm + "\"";
    m_response.addHeader(AUTHENTICATE_HEADER, header);
    sendError(HttpServletResponse.SC_UNAUTHORIZED);
    return false;
}
```

Send the requested resource. Add saved state information.

```
/**
 * tell the Web server framework we got "access allowed" for this request
 *
 * return    true
 */
protected boolean allow() {
    if (EXPORT_POST_DATA) {
        String data = getPostDataBuffer();
        if (data.length() > 0) {
            Map postDataMap = new HashMap();
            m_request.setAttribute(SA_POSTDATA, postDataMap);
            postDataMap.put("posted", getPostDataBuffer());
        }
    }
    return true;
}
```

Send access denied page

```
/**
 * Send the Web client an "access denied" reply.
 *
 * return false
 */
protected boolean deny() {
    if (setDenyViaForm()) {
        sendDynamicForm(null);
    }
    else {
        sendError(HttpServletResponse.SC_FORBIDDEN);
    }
    return false;
}
```

Multidomain Single Sign-on

After successful authentication, the nonce in the Policy Validator reply is used to maintain single sign-on. The nonce is an encrypted token. By placing the nonce in a web browser cookie, the browser will include the nonce in all future requests. Since web cookies are tied to a specific domain, this only provides single sign-on within a single domain.

To provide single sign-on across multiple domains, Enforcer plugins must:

1. Test to see if the referring site is part of the multiple domain single sign-on list.
2. Redirect the browser to the referring site and request a nonce.
3. The referring site redirects the browser back to the original site, with the nonce included in the URL.
4. Place the nonce in a cookie for this domain.
5. Redirect the identity to the original URL.

Performing Multiple Domain Single Sign-on on page 100 gives a sample that demonstrates how to perform multiple domain single sign-on. If a Policy Validator request is denied, the `isAuthorized()` method checks to see if multiple domain single sign-on could be used to identify the identity. If so, the method redirects the identity to the referring site and requests for a single sign-on nonce.

When a request is first received, the `isAuthorized()` method checks to see if the request is for a multiple domain single sign-on nonce. If so, it constructs the URL with the nonce, and redirects the client to the original site.

If the request is allowed by the Policy Validator, the `isAuthorized()` method checks to see if the requested URL contains a multiple domain single sign-on nonce. If so, the method creates a new cookie for the nonce, and provides access to the requested resource.

Performing Multiple Domain Single Sign-on

The following example shows how Enforcer plugins typically perform single sign-on across multiple domains:

```
public boolean isAuthorized() {

    ... Verify that authorization is required ...

    if (isSSORedirectURL(m_http_query, orig_url)) {
        String r2 = constructSSOAuthURL(m_auth_cookie,
                    orig_url.toString());
        return ssoRedirect(r2); // MD-SSO redirect 2
    }


    ... Build validator query ...

    ... Query the validator ...

    boolean isAllowed = allowedByValidator();
    if (isAllowed) {
        sso_nonce = new StringBuffer("");
        if (isSSOAuthURL(full_url, sso_nonce)) {
            String r3 = constructSSOOrigURL(m_path, m_http_query);
           setCookie(DataStrings.AUTH_COOKIE_NAME, sso_nonce.toString(),
                     "/",
            m_enforcer.getCookieDomainName(), -1);
            return sendAcceptedPage(r3);        // MD-SSO redirect 3
        }
        return allow();
    }
    if (couldSSOHelp(m_reply, m_referer, full_url)) {
        String r1 = constructSSORedirectURL(m_referer, full_url);
        return ssoRedirect(r1);    // MD-SSO redirect 1
    }

    ... Display login page ...
```

Send nonce from this domain

Redirect to original URL

See if referred can provide a nonce

```
    return deny();    // sends deny.html or 403 error
}
```

Multiple Domain Single Sign-on Methods

The following methods are helper methods for the multiple domain single sign-on feature. They are included here for reference.

```
/**
 * tests if the referring site is in our multi domain SSO protected sites
   list
 *
 * referer          the referring URL
 * return           true if the site is SSO-protected, false otherwise
 */
public boolean isDomainProtected(String referer) {
    boolean result = m_enforcer.isDomainProtected(referer);
    return result;
}

/**
 * create the SSO redirect 1 URL, which requests that the next server
 * sends back a nonce
 *
 * referer            the referring URL
 * currentURL         the current URL
 * return             the redirect URL
 */
public String constructSSORedirectURL(String referer, String currentURL)
{
    StringBuffer buf = new StringBuffer("");
    String url = currentURL.replaceFirst("\\?", "&");
    // new url has referer URL prepended and the current URL appended to
       the SSO tag
    buf.append(referer);
    buf.append("?");
    buf.append(SSO);
    buf.append("=");
    buf.append(url);
    return buf.toString();
}

/**
 * create the SSO redirect 2 URL, which contains nonce
 *
 * nonce              the current cookie
 * currentURL         the current URL
 * return             the redirect URL
 */
```

```
public String constructSSOAuthURL(String nonce, String currentURL) {
    StringBuffer buf = new StringBuffer("");
    String delimiter = "?";
    String url = currentURL.replaceFirst("\\&", "?");
    if (!url.equals(currentURL))
        delimiter = "&";
    String value = NO_NONCE;
    if (nonce != null && isDomainProtected(currentURL))
        value = nonce;
    // new url has referer URL prepended and the current URL appended to
the SSO tag
    buf.append(url);
    buf.append(delimiter);
    buf.append(SSO_AUTH_COOKIE_NAME);
    buf.append("=");
    buf.append(value);
    return buf.toString();
}

/**
 * create the SSO redirect 3 URL, which sends browser back to the
   original URL
 *
 * currentPath        the filename component of the current URL
 * http_query         the current http_query
 * return             the redirect URL
 */
public String constructSSOOrigURL(String currentPath, String http_query)
{
    StringBuffer buf = new StringBuffer("");
    buf.append(currentPath);
    int index = http_query.indexOf(SSO_AUTH_COOKIE_NAME);
    if (index > 0) {
        buf.append("?");
        buf.append(http_query.substring(0, index-1));
    }
    return buf.toString();
}

/**
 * tests if this request contains an SSO redirect desiring a nonce
 *
 * url               the current URL
 * orig_url          the original url
 * return            true if we are currently at a RedirectURL,
 *                   and need to do MD-SSO redirect 2
 */
public boolean isSSORedirectURL(String url, StringBuffer orig_url) {
```

```
    if (url == null)
        return false;
    int index = url.indexOf(SSO);
    if (index >= 0) {
        if (orig_url != null)
            orig_url.append(url.substring(index + SSO.length() + 1));
        return true;
    }
    return false;
}

/**
 * tests if this request contains an SSO nonce
 *
 * url               the current URL
 * nonce             the current cookie
 * return            true if we are currently at an AuthURL,
 *                   and need to do MD-SSO redirect 3
 */
public boolean isSSOAuthURL(String url, StringBuffer nonce) {
    int index = url.indexOf(SSO_AUTH_COOKIE_NAME);
    if (index > 0) {
        if (nonce != null)
            nonce.append(url.substring(index +
SSO_AUTH_COOKIE_NAME.length() + 1));
        return true;
    }
    return false;
}

/**
 * tests if multi-domain SSO should be attempted for this request
 *
 * reply             the validator's XML reply
 * referer           the referring URL
 * url               the current URL
 * return            true if validator denied because it needs auth and
                     MD-SSO
 *                   could help, so do MD-SSO redirect 1
 */
public boolean couldSSOHelp(XmlElement reply, String referer, String
url) {
    if (referer == null || ! isDomainProtected(referer))
        return false;      // referer is not a protected domain
    if (isSSOAuthURL(url, null))
        return false;      // prevent loops due to failed auth
    if (isSSORedirectURL(referer, null))
        return false;      // prevent double-redirect
```

```
    final String SKIP_STRING    = "https://";
    int slash = referer.indexOf('/', SKIP_STRING.length());
    String Website = referer.substring(0, slash + 1).toLowerCase();
    String url2 = url.toLowerCase();
    if (url2.startsWith(Website))
        return false;      // referral came from within same site
    if (m_enforcer.replyNeedsAuth(reply))
        return true;       // SSO could provide the auth we need
    return false;
}
```

Deploying the Sample Web Application with the Servlet Filter

To use the `ServletFilter`, you must deploy the filter on a web server or application server. For details on how you can achieve this, see Chapter 6, Servlet Engine Integrations, in the *HP OpenView Select Access 6.2 Network Integration Guide*.

Creating an Enforcer Plugin With the C/C++ Enforcer API

This section demonstrates how to build a C/C++ Enforcer plugin. In this section you will learn how to:

- Initialize an Enforcer object
- Extract authorization data
- Build a Policy Validator request
- Query a Policy Validator
- Enforce the Policy Validator decision
- Perform multiple domain single sign-on
- Install Apache plugins

The Apache Example

The Apache 2 Enforcer plugin is a plugin to the Apache web server. The Apache 2 Enforcer plugin uses the Apache API to control access to the web server. This Enforcer plugin connects the Apache API to the Enforcer API. The Apache 2 Enforcer plugin uses the Enforcer API to query Policy Validators and determine if a request is allowed. Depending on the response, the Apache 2 Enforcer plugin will either display an error page, a login page, an access denied page, or the requested URL. The Apache 2 Enforcer plugin supports single sign-on by placing a nonce inside a Web cookie. Multiple domain single sign-on is also supported.

To follow this example, you should be familiar with C, the Apache API, and HTTP requests.

The Apache API performs the following tasks:

1 Apache loads the modules specified in its `httpd.conf` configuration file when it starts up.

2. As each module is loaded, it passes registration information to the server. Some of this information:
   a. Specifies the module's initialization and cleanup handlers.
   b. Indicates which phases of each HTTP transaction and kinds of content it is interested in. In the case of the Apache 2 Enforcer plugin, this information specifies that processing is to be intercepted at the access control phase. In this phase, the plugin queries the Policy Validator to see whether the request is allowed.
3. When the Apache 2 Enforcer plugin is loaded, it passes four methods to the server:
   a. The access control handler: When it is called, the Apache web server passes it a structure that contains all of the information the server has about the request and its environment. The handler can both inspect and modify this structure in order to control the server's actions and further processing. This module builds Policy Validator requests, queries the Policy Validator, and processes the response to enforce the policy decision.
   b. A module initializer: This is called once to set up the module as a whole. The Apache 2 Enforcer plugin establishes a long-lived socket connection with the Policy Validator by creating a shared Enforcer object. This minimizes any delays that SelectAccess may introduce.
   c. A child processes initializer: This is called to initialize the module inside any child processes that the Apache web server spawns to handle requests.
   d. A clean up method: This is called to tidy up when those processes terminate.

The Apache server may be configured to use multiple security mechanisms, such as native HTTP authentication. These other mechanisms may prevent Select Access login and error pages from being displayed. To circumvent this, the Apache 2 Enforcer plugin uses dynamic content to deliver login forms, challenge forms, denial pages, etc.

C/C++ Enforcer API Classes and Utilities

The core of the C/C++ Enforcer API is the `EnforcerHandle` class. This class represents an Enforcer object that is used to communicate with the Policy Validator. An pointer to an `EnforcerHandle` is returned by the `EnforcerInit()` method calls that are demonstrated later in this section.

Select Access provides XML classes that you should use to build Policy Validator requests and process Policy Validator responses.

- `XmlParser`: This class parses a block of memory or a file and creates an XML document tree. Typically you will not need to directly use this class, because Select Access creates XML documents for all configurations, requests, and responses.
- `XmlTreeNode`: This class stores the XML nodes of a parsed XML document. This class provides low-level access to XML attributes. Typically you will use the `PropertyListElement` and `PropertyElement` classes to access XML data.
- `PropertyElement`: This class stores a name-value pair. A `PropertyElement` is an XML element defined by Select Access. The `PropertyElement` tag requires one an attribute called `name` that identifies the name of the name-value pair. The value is stored as the data between begin and end tags. An example `PropertyElement` that stores the name-value pair for the color red is: `<PROPERTY NAME="color">red</PROPERTY>`.

- `PropertyListElement`: This class stores an XML node that contains zero or more `PropertyElement` tags. Property lists may be nested; a `PropertyListElement` may contain zero or more property lists. An example of a `PropertyListElement` is: `<PROPERTYLIST name="colorSet"> <PROPERTY NAME="color">red</ PROPERTY></PROPERTYLIST>`.

Your Enforcer plugin should support logging, handle exceptions, and leverage the Select Access string and memory utilities. Most plugins will need to interact with the following Select Access classes and utilities:

- `Logger`: The `Logger` class provides an interface to Select Audit. This class supports multiple levels of logging support: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `FATAL`.
- `EnforcerException`: This is the base class for all Enforcer plugin exceptions.
- `enforcer_sys.h`: This header files defines `SYS_STRDUP`, `STREQ`, and other platform independent string manipulation macros. Your plugin should use these macros when manipulating strings.

Including the Enforcer Libraries

When developing an Enforcer plugin, the plugin will need to include the Enforcer API headers, as shown in the example that follows. The Apache 2 Enforcer plugin was designed to support multiple platforms and multiple versions of the Apache API (versions 1.3 and 2.0). To simplify the code, the code specific to previous versions of the Apache web server API have been omitted from these examples.

```
/*
 * mod_enforcer.c - An Apache security module using the hp Enforcer API
 *
 * In access control phase [#3], enforcer_check() queries the validator
 * using the Enforcer API to see if request is allowed.
 * If the request requires authentication a special
 * response is sent to the client, and we return DONE
 * to exit the apache request loop.
 */

#include <enforcer.h>
#include <enforcer_sys.h>
#include <XmlTreeNode.h>
#include "mod_enforcer.h"
#include <enforcer_Web.h>
#include "auto_free.h"

#if defined(SYS_WINDOWS)
# include "enforcer_load.h"
# ifdef _DEBUG
#  ifdef ORACLE_APACHE
#   define MODULE_NAME "oracle_apache_Web32d.dll"
#  else
#   define MODULE_NAME "apache_Web32d.dll"
#  endif
# else
```

```
#  ifdef ORACLE_APACHE
#   define MODULE_NAME "oracle_apache_Web32.dll"
#  else
#   define MODULE_NAME "apache_Web32.dll"
#  endif
# endif
#endif

EnforcerHandle *Enforcer_Handle = NULL;
static void enforcer_init(void);
```

Creating an Enforcer Object

An Enforcer object is used to communicate with the Policy Validator. To create an Enforcer object, the Enforcer API must be initialized. To initialize the Enforcer API, the Apache 2 Enforcer plugin calls the `EnforcerConfigInit()` method when the web server starts. The `EnforcerConfigInit()` method uses the `enforcer.xml` file to configure the Enforcer API. If a configuration file location is not provided the Enforcer API looks for a default `enforcer.xml` file.

By default, the Enforcer API uses the plugin name to identify which `enforcer.xml` file to use. For example, if the plugin name is apache, then the Enforcer API will look for the file `<SA_Install>/bin/enforcer_apache.xml`.

For Windows platforms, an extra call is needed to load the Enforcer plugin libraries and create registry keys.

Initialization

The following example demonstrates how to initialize the Enforcer API:

```
static int mod_enforcer_initialized = 0;
EnforcerHandle *Enforcer_Handle = NULL;

/*
 * initializes the Enforcer API
 */
static void enforcer_init(void){
    static const char * loggingName =
    ENFORCER_COMPANY_SHORT_NAME " " ENFORCER_PRODUCT_NAME " Apache "
    ENFORCER_SERVER_PLUGIN_NAME;
#if defined(SYS_WINDOWS)
    static const char * dllNames[] = { NULL };
    if (! EnforcerLoadLibraries(MODULE_NAME, dllNames, loggingName)) {
        fprintf(stderr, "Failed to load enforcer libraries\n");
        exit(1);
    }
    EnforcerCreateRegistryKeys(MODULE_NAME, loggingName);
#endif
    if (!EnforcerInit()) {
        EnforcerLog(ENFORCER_LOG_ERROR, ">#<EnforcerInit error");
```

```
        return;
    }
    Enforcer_Handle = EnforcerConfigInit(NULL, loggingName, "apache");
    if (Enforcer_Handle == NULL) {
        EnforcerLog(ENFORCER_LOG_ERROR, ">#<EnforcerConfigInit error");
        return;
    }
    mod_enforcer_initialized = 1;
}
```

Determining if Authorization is Required

After the Apache initialization modules terminate, the web server is ready to process HTTP requests. The `enforcer_check()` method is called by the Apache server framework. The `enforcer_check()` method first verifies that authorization is required for the requested resource. There are several reasons why authorization might not be necessary:

- The Enforcer API can not be configured. In this case, the Policy Validator can not be contacted, and access should be denied.
- The URL contains invalid characters. URLs that contain invalid characters are not be processed, and the request is rejected. An example of a invalid character is the “.” directory operator.
- The Enforcer plugin is configured to allow unrestricted access to the domain.
- The Enforcer plugin is configured to allow unrestricted access to the file.
- The request is part of a multidomain single sign-on redirect.

Determining the conditions for authorization

The following example illustrates how to check for each of the conditions.

```
/*
 * use Enforcer API to see if request should be allowed - access control
   phase [#3]
 */
extern "C" {
    static int enforcer_check(request_rec *r){

        ... Declarations ...

        if(Enforcer_Handle == NULL) {
            EnforcerLog(ENFORCER_LOG_ERROR, ">#<EnforcerAPI error: not
                        initialized");
            return HTTP_FORBIDDEN;
        }
        if(EnforcerIsEvilURL(r->uri, Enforcer_Handle)) {
            EnforcerLog(ENFORCER_LOG_WARNING, ">#<rejecting suspicious
                        url '%s'", r->uri);
            return HTTP_FORBIDDEN;
        }
```

Check for dangerous URLs

```
        if(!ap_is_initial_req(r)) {
            if (EnforcerGetDisableCaching(Enforcer_Handle)) {
                disable_caching(r, 0);
            }
            return DECLINED;
        }
        if(EnforcerGetP13Ncompat(Enforcer_Handle) == 0 &&
           suspicious_headers(r)) {
            return HTTP_FORBIDDEN;
        }
        virt_domain = (char*) ap_get_server_name(r);
        if (EnforcerIsPassthroughDomain(Enforcer_Handle, virt_domain)) {
            EnforcerLog(ENFORCER_LOG_DEBUG, "unsecured_domain: %s",
                        virt_domain);
            return DECLINED;
        }
        if (EnforcerIsIgnoredFile(Enforcer_Handle, r->uri)) {
            EnforcerLog(ENFORCER_LOG_DEBUG, "Ignore: %s", r->uri);
            return DECLINED;
        }

        ... Build the query ...

        ... Query the validator ...

        ... Process returnAction and XML reply ...

    }
}
```

Building the Policy Validator Request

Once the `enforcer_check()` method verifies that authorization is required, it builds a Policy Validator query. The `enforcer_check()` method uses the `initialize_query_object()` method to initialize and populate the query as shown in the example below.

```
/*
 * use Enforcer API to see if request should be allowed - access control
   phase [#3]
 */
extern "C" {
    static int enforcer_check(request_rec *r){

        ... Declarations ...

        ... Determine if authorization is required ...

        referer = (char *) ap_table_get(r->headers_in, "Referer");
```

Initialize a Validator query

```
        cookies = util_parse_cookie(r);
        if ((got = initialize_query_object(r, &query,
             &selectaccess_needs_refresh,
            http_query, referer, cookies)) != OK)
        {
            return got;
        }

        ... Query the validator ...

        ... Process returnAction and XML reply ...

    }
}
```

Initializing and populating the XML

The `initialize_query_object()` method initializes the query object using the `EnforcerAddrsQueryInit()` method. This initializes the query object with the appropriate `PolicyValidatorQuery` element and the required service and path properties. It also adds properties for the local and remote IP addresses, server name, and HTTP method.

After initializing the query object, the `initialize_query_object()` method adds XML properties for all the pertinent information available from the Apache API.

- `EnforcerAddEncoded()`: This method is used to add data that must be UTF-8 encoded.
- `EnforcerAddFormData()`: This method is used to add data from HTTP forms which is located in the input stream.
- `EnforcerAddQueryData()`: This method is used to add data from the HTTP query which is located in the URL.
- `appendChild()`: This method adds a new XML object to the query.
- `appendChildString()`: This method adds a new name-value pair to the query.

The example that follows demonstrates how to add properties and property lists to the query using these mechanisms:

```
/*
 * fill in XML query object with everything we can find out - helper for
   phase [#3]
 */
static int initialize_query_object
    (
        request_rec *r, XMLnode *query, int *needs_refresh,
        char *http_query, char *referer, table *cookies
    ){
    conn_rec *con = r->connection;
    char *password = NULL, *hvalue;
    char *post_data = NULL;
    int query_level = EnforcerGetQueryLevel(Enforcer_Handle);
    char *base_url;
    int url_changed;
```

```
    if (!EnforcerGetBasenameOfURL(r->uri, &base_url, &url_changed))
        return HTTP_FORBIDDEN;
    if (!EnforcerAddrsQueryInit(&con->remote_addr->sa.sin,
        &con->local_addr->sa.sin, ap_http_method(r),
        ap_get_server_name(r), base_url, query_level, query))
    {
        if (url_changed)
            delete base_url;
        return HTTP_FORBIDDEN;
    }
    if (url_changed)
        delete base_url;
    /* if they sent a basic-auth header, parse out the login and passwd
*/
    if ((hvalue = (char *) ap_table_get(r->headers_in, "Authorization"))
         != NULL) {
        // EnforcerLog(ENFORCER_LOG_DEBUG, "Authorization: %s", hvalue);
       if(enforcer_ap_get_basic_auth_pw(r, (const char **) &password) ==
          OK) {
            EnforcerAddEncoded(*query, ENFORCER_USER, r->user,
                               Enforcer_Handle);
            EnforcerAddEncoded(*query, ENFORCER_PASSWD, password,
                               Enforcer_Handle);
        }
    }
    /* if they sent a cookie header, parse out the cookies we recognize
*/
    if (cookies != NULL) {
        ap_table_do(add_auth_cookies, *query, cookies, NULL);
    }
    /* if this is a magic_query, read the POST data and add it to the XML
      query */
    if (magic_query(http_query) && (util_read(r, (const char**)
        &post_data) == OK)) {
        table *post_data_tab = NULL;
        *needs_refresh = EnforcerAddFormData(*query, post_data,
         Enforcer_Handle);
        /* also add post data to notes, so other handlers can access the
         data */
        if((tabify_post_data(r, &post_data_tab, post_data) == OK) &&
            post_data_tab && !ap_is_empty_table(post_data_tab))
        {
            ap_table_do(note_form_data, r, post_data_tab, NULL);
        }
    }
    if (EnforcerGetEnableCookies(Enforcer_Handle)) {
        /* enable Web session cookies */
        (*query)->appendChildString(ENFORCER_NATIVE_NONCE,
            SYS_STRDUP(ENFORCER_ENABLE));
```

Initialize the query

Add username & password

Add single sign-on nonce to request

Add SSO cookie data

Copy HTTP posted data

```
    }
    if ((hvalue = (char *) ap_table_get(r->notes, ENFORCER_SITE_DATA)) !=
       NULL)
        (*query)->appendChildString(ENFORCER_SITE_DATA,
          SYS_STRDUP(hvalue));
    const char *service =
(*query)->getChildStringValue(ENFORCER_SERVICE);
    /* if this is an SSL connection, add encryption info and certificate
       */
    if((service != NULL ) && STREQN(service, "https://", strlen("https:/
       /"))) {
       add_ssl(r, query, "SSL_CIPHER_USEKEYSIZE", ENFORCER_SSL_KEYSIZE);
        add_ssl(r, query, "SSL_CLIENT_CERT", ENFORCER_CERT);
        add_ssl(r, query, "SSL_CLIENT_S_DN", ENFORCER_SSL_CLIENT_DN);
        add_ssl(r, query, "SSL_PROTOCOL", ENFORCER_SSL_PROTOCOL);
        add_ssl(r, query, "SSL_CIPHER", ENFORCER_SSL_CIPHER);
    }
```

Adding HTTP query data

```
    if (query_level > QUERY_MINIMAL) {
        if (http_query != NULL) {
            EnforcerAddQueryData(*query, http_query, Enforcer_Handle);
        }
        /* if apache successfully stat'ed the file, add the owner and
         size */
        if (r->finfo.st_mode) {
            char tmp[BUFSIZ];
            struct passwd *pw;
            pw = getpwuid(r->finfo.st_uid);
            if (pw != NULL) {
                snprintf(tmp, BUFSIZ, "%s", pw->pw_name);
               (*query)->appendChildString(ENFORCER_OWNER,
                 SYS_STRDUP(tmp));
            }
            snprintf(tmp, BUFSIZ, "%ld", r->finfo.st_size);
           (*query)->appendChildString(ENFORCER_SIZE, SYS_STRDUP(tmp));
        }
        if (is_method_valid(r->method)) {
            (*query)->appendChildString(ENFORCER_METHOD,
SYS_STRDUP((char*)r->method));
        }
```

Adding a property list

```
        if (query_level == QUERY_MAXIMAL) {
            XMLnode header_list = XmlTreeNode::sNew
                                  (ENFORCER_TAG_PROPERTYLIST,
               ENFORCER_HTTP_HEADER_LIST);
            (*query)->appendChild(header_list);
            ap_table_do(add_table_to_xml, header_list, r->headers_in,
                        NULL);
            hvalue = (char *) ap_get_server_version();
            if (hvalue != NULL)
```

```
                (*query)->appendChildString(ENFORCER_SERVER,
                    SYS_STRDUP(hvalue));
        }
    }
    if (EnforcerGetDebugLevel(Enforcer_Handle) > 9)
        (*query)->appendChildString(ENFORCER_TRACE,
                 SYS_STRDUP(ENFORCER_ENABLE));
    return OK;
}
```

Adding authentication cookies to the XML query

The `initialize_query_object()` method uses the `ap_table_do()` method to extract cookies from the request header and add them to the Policy Validator request. The `ap_table_do()` method is a generic method that uses callback methods to perform specific actions. The following example illustrates the callback method that searches for single sign-on nonces in web cookies and adds them to the Policy Validator XML request.

```
/*
 * add authentication cookies to an XML query - helper for phase [#3]
 * callback for ap_table_do
 */
static int add_auth_cookies(void *data, const char *key, const char
                            *val){
    XmlTreeNode * object = (XmlTreeNode *)data;
    if(STREQ(key, AUTH_COOKIE_NAME)) {
        object->appendChildString(ENFORCER_NONCE, SYS_STRDUP(val));
    }
    if(STREQ(key, REGISTER_COOKIE_NAME)) {
        object->appendChildString(ENFORCER_REGISTER_NONCE,
                  SYS_STRDUP(val));
    }
    return TRUE;
}
```

Adding SSL information to the request

The following optional Apache methods are used to look up the SSL parameter values from the Apache API. The `add_ssl()` method is used to lookup SSL parameter values by name, and adds the property to the XML query.

```
/*
 * add SSL information to the XML query
 */
static void add_ssl
    (
        request_rec *r, XMLnode *query, char *ssl_name,
        const char *query_tag
    ){
    char *val = NULL;
    APR_DECLARE_OPTIONAL_FN(char *, ssl_var_lookup, (apr_pool_t *,
    server_rec *,
```

```
        conn_rec *, request_rec *, char *));
    APR_OPTIONAL_FN_TYPE(ssl_var_lookup) *var_lookup;
    var_lookup = APR_RETRIEVE_OPTIONAL_FN(ssl_var_lookup);
    if (var_lookup != NULL) {
       val = var_lookup(r->pool, r->server, r->connection, r, ssl_name);
    }
    if(val != NULL) {
        if (STREQ(query_tag, ENFORCER_SSL_KEYSIZE)) {
            char tmp[BUFSIZ];
            snprintf(tmp, BUFSIZ, "%s bit", val);
            (*query)->appendChildString(query_tag, SYS_STRDUP(tmp));
        }
        else
            (*query)->appendChildString(query_tag, SYS_STRDUP(val));
    }
}
```

Adding data to an XML object

The following example shows a generic method that adds name-value pairs to the Policy Validator XML request. This method is a callback method that is called by the `ap_table_do()` method. It demonstrates how to add generic properties to a Policy Validator request.

```
/*
 * add arbitrary table data to an XML object - helper for phase [#3]
 * callback for ap_table_do
 */
static int add_table_to_xml(void *data, const char *key, const char
                            *val){
    XmlTreeNode * d = (XmlTreeNode *)data;
    d->appendChildString(key, SYS_STRDUP(val));
    return TRUE;
}
```

Redirecting the web browser to a URL with a Magic query

If a identity requests a resource, but has not been authenticated, the Apache 2 Enforcer plugin presents the identity with a login form. The login form contains form contains the identity and credentials used for authentication. When a login form is presented to the identity, the Apache 2 Enforcer plugin uses a special query tag in the URL so that the enforcer knows to extract the identity, credentials, and other data from the form. This is referred to as a magic query. The following example shows how the Apache 2 Enforcer plugin redirects the browser to a URL containing a magic query.

```
/* redirect to a URL that apache will treat specially */
static int
send_magic_redirect(request_rec *r, char *orig_query)
{
    char url[1024];
    if (orig_query == NULL)
        snprintf(url, 1024, "%s?%s", r->uri, APACHE_MAGIC_QUERY);
```

```
    else
        snprintf(url, 1024, "%s?%s&%s", r->uri, orig_query,
                 APACHE_MAGIC_QUERY);
    return send_redirect(r, url);
}
```

Checking to see if the query contains original form data embedded

Once a identity submits a login form, the Enforcer plugin must determine if the URL contains a magic query. The following code examines the requested URL to determine if login data should be extracted from the login form. If the requested URL does not contain a magic query, the Apache 2 Enforcer plugin knows that the form data does not pertain to the login, and will not extract the posted form data.

```
/* returns 1 if a string contains our magic query string, 0 otherwise */
static int magic_query(char *q){
    if (q == NULL)
        return 0;
    if (strstr(q, APACHE_MAGIC_QUERY) == NULL)
        return 0;
    return 1;
}
```

Saving form data for other Apache plugins

If the requested URL contains a magic query, the Apache 2 Enforcer plugin extracts the credentials to add to the Policy Validator request. Due to a limitation in the Apache server, form data may only be read once by Apache 2 Enforcer plugins. To make the form data available to other Apache 2 Enforcer plugin, the form data is copied to a notes structure. This example shows how to copy the form data so that it is available to other plugins.

```
/*
 * add consumed POSTed form data to the notes area, so other handlers can
   access it
 */
static int note_form_data(void *data, const char *key, const char *val){
    #define FORM_NOTE_PREFIX         "SA_POSTED:"
    request_rec *r = (request_rec *)data;
    int len = strlen(key) + strlen(FORM_NOTE_PREFIX) +1;
    char *data_name = (char*) malloc(len);
    if (data_name == NULL)
    {
        EnforcerLog(ENFORCER_LOG_PERR, "note-form-data: failed to malloc
                    %d", len);
        return 0;
    }
    snprintf(data_name, len, "%s%s", FORM_NOTE_PREFIX, (char *) key);
    ap_table_add(r->notes, data_name, (char*) val);
    free(data_name);
    return TRUE;
}
```

Querying the Policy Validator

Once the Policy Validator query is complete, the `EnforcerQuerySend()` method is used to send the query to Policy Validators. The method returns a code indicating whether access should be allowed, denied, or if an error occurred. The XML reply from the Policy Validator is stored in the `reply` parameter. The following demonstrates how to use the `EnforcerQuerySend()` method:

```
/*
 * use Enforcer API to see if request should be allowed - access control
   phase [#3]
 */
extern "C" {
    static int enforcer_check(request_rec *r){

        ... Declarations ...

        ... Verify authorization is required ...

        ... Build a validator query ...

       returnAction = EnforcerQuerySend(Enforcer_Handle, query, &reply);

        ... Process returnAction and XML reply ...
    }
}
```

Enforcing the Response

After querying the Policy Validator, the response must be processed to enforce the decision. The `enforcer_check()` method uses the return value from the `EnforcerQuerySend()` method to determine if the Policy Validator encountered an error. If no error occurred, the `enforcer_check()` method searches the XML reply for a single sign-on nonce or a registration nonce. If present, the nonces are added to HTTP cookies. The cookies identify the identity during future transactions.

If the request was denied, the `enforcer_check()` method looks to see if the XML reply specified any authentication hints. Authentication hints instruct the Enforcer plugin how to authenticate identities. Based on these hints, the Apache 2 Enforcer plugin might prompt an identity for a login credentials.

Enforcing the Policy Validator decision

The following example shows how the Enforcer plugin is typically programmed to enforce Policy Validator decision:

```
/*
 * use Enforcer API to see if request should be allowed - access control
   phase [#3]
 */
extern "C" {
    static int enforcer_check(request_rec *r){
```

```
... Declarations ...
... Determine if authorization is required ...
... Build XML query ...
... Query Policy Validator ...

if(returnAction != ENFORCER_ERROR_ACTION) {
    /* you can't look inside the reply if it might be NULL (on
       error) */
    EnforcerSetenv(reply, r);
    /* if XML reply object had a nonce, add it as an
       AUTH_COOKIE_NAME cookie */
   if((s = reply->getChildStringValue(ENFORCER_NONCE)) != NULL)
{
        set_cookie(Enforcer_Handle, r, AUTH_COOKIE_NAME,
            (char*) s, COOKIE_LIFE, 0);
        // don't cache logout pages!
        if(STREQ(s, "logout")) {
            disable_caching(r, 0);
        }
    }
    /* if XML reply object had a registration cookie, add it */
    if((s = reply->getChildStringValue(ENFORCER_REGISTER_NONCE))
        != NULL) {
        long cookie_life = 0;
        if ((! reply->getChildLongValue(ENFORCER_REGISTER_LIFE,
             &cookie_life))
           || (cookie_life == 0))
        {
            cookie_life = COOKIE_LIFE;
        }
        set_cookie(Enforcer_Handle, r, REGISTER_COOKIE_NAME,
            (char*) s, cookie_life, 0);
    }
}
switch (returnAction) {
    case ENFORCER_ALLOW_ACTION:

        ... Handle multiple domain single sign-on ...

        response = DECLINED; /* apache-speak for 'let other
                    handlers decide */
        break;
    case ENFORCER_DENY_ACTION:

        ... Handle multiple domain single sign-on ...
```

Setting SSO cookies

Access allowed. Continue processing.

```
redirect_URL = reply->
          getChildStringValue(ENFORCER_REDIRECT);
if(redirect_URL != NULL
    && EnforcerAuthFailed(query, reply,
      selectaccess_needs_refresh))
{
    response = send_redirect(r, (char *)redirect_URL);
}
else if (EnforcerGetAuthHint(reply, &auth_plist,
         &hint_flags)) {
    response = DONE;
    if (hint_flags == HINT_PASSWORD
       && EnforcerGetLoginViaForm(Enforcer_Handle) == 0)
    {
        s = auth_plist->
           getChildStringValue(ENFORCER_FORM_REALM);
        send_basic_auth_page(r, (char *) s);
    } else {
        if (!magic_query(http_query)) {
          response = send_magic_redirect(r, http_query);
        } else {
            send_dynamic_form(r, NULL, auth_plist);
        }
    }
    if (hint_flags == HINT_FREE) {
        delete auth_plist;
    }
}
else if ((auth_plist = reply->
          getChild(ENFORCER_FORM_DATA)) &&
    auth_plist->isPropertyList())
{
    // they have a site-specific auth plugin or dynamic
       form
    response = DONE;
    if (!magic_query(http_query)) {
        response = send_magic_redirect(r, http_query);
    } else {
        send_dynamic_form(r, NULL, auth_plist);
    }
}
else if(redirect_URL != NULL) {
    response = send_redirect(r, (char *)redirect_URL);
} else {
    EnforcerLog(ENFORCER_LOG_DEBUG, "reply is flat out
    deny");
}
break;
```

Sending a login form

Access denied

```
                case ENFORCER_ERROR_ACTION:
                case ENFORCER_USER_DEFINED_ACTION:
                default:
                   ap_log_error(APLOG_MARK, APLOG_ERR, AP2_STATUS r->server,
                        "mod_enforcer ERROR %s %s", r->method, r->uri);
                    break;
            }
            delete query;
            delete reply;
            if (response == DECLINED) {
                if (selectaccess_needs_refresh) {
                    send_accepted_page(r);
                    response = DONE;
                }
                else if (EnforcerGetDisableCaching(Enforcer_Handle)) {
                    disable_caching(r, 0);
                }
            }
            return response;
            /* give other access control handlers a chance */
        }
    }
```

Multiple Domain Single Sign-on

After successful authentication, the nonce in the Policy Validator reply is used to maintain single sign-on. The nonce is an encrypted token. By placing the nonce in a web browser cookie, the browser will include the nonce in all future requests. Since web cookies are tied to a specific domain, this only provides single sign-on within a single domain.

To provide single sign-on across multiple domains, a Enforcer plugin must follow these additional steps:

1. Test to see if the referring site is part of the multiple domain single sign-on list.
2. Redirect the browser to the referring site and request a nonce.
3. The referring site redirects the browser back to the original site, with the nonce included in the URL.
4. Place the nonce in a cookie for this domain.
5. Redirect the identity to the original URL.

If a Policy Validator request is denied, the `enforcer_check()` method checks to see if multiple domain single sign-on could be used to identify the identity. If so, the method redirects the identity to the referring site and requests a single sign-on nonce.

When a request is first received, the `enforcer_check()` method checks to see if the request is for a multiple domain single sign-on nonce. If so, it constructs the URL with the nonce, and redirects the client to the original site.

If the request is allowed by the Policy Validator, the `enforcer_check()` method checks to see if the requested URL contains a multiple domain single sign-on nonce. If so, the method creates a new cookie for the nonce, and provides access to the requested resource.

Processing Multiple Domain Single Sign-on web redirection

The following sample demonstrates how to perform multiple domain single sign-on:

```
/*
 * use Enforcer API to see if request should be allowed - access control
   phase [#3]
 */
extern "C" {
    static int enforcer_check(request_rec *r){
        ... Declarations ...
        ... Determine if authorization is required ...

        /* we need http_query, referer and cookies for SSO support */
        if (r->args && strlen(r->args)) {
            http_query = r->args;
        }
        referer = (char *) ap_table_get(r->headers_in, "Referer");
        cookies = util_parse_cookie(r);
        if (EnforcerIsSSORedirectURL(http_query, &old_url)) {
            // do SSO redirect 2
            char *sso_authURL;
            if (cookies != NULL) {
               nonce = (char *) ap_table_get(cookies, AUTH_COOKIE_NAME);
            }
            EnforcerConstructSSOAuthURL(Enforcer_Handle, old_url, nonce,
                                        &sso_authURL);
            send_redirect(r, sso_authURL);
            delete[] sso_authURL;
            return HTTP_MOVED_TEMPORARILY;
        }

        ... Build the Policy Validator query ...
        ... Process nonce and create cookies ...

        switch (returnAction) {
            case ENFORCER_ALLOW_ACTION:
                response = DECLINED; /* apache-speak for 'let other
                           handlers decide */
                if (EnforcerIsSSOAuthURL(http_query, &nonce)) {
                    // SSO worked, send browser back to the original URL
                    // do SSO redirect 3
                    char *sso_origURL;
                    EnforcerConstructSSOOrigURL(r->uri, http_query,
                     &sso_origURL);

set_cookie(Enforcer_Handle,r,AUTH_COOKIE_NAME,nonce,COOKIE_LIFE,0);
                    send_refresh(r, 0, sso_origURL);
                    response = DONE;
```

The server we need a nonce from

Step 3: Add multiple domain nonce to URL

Step 4: Add nonce to local domain cookie

Step 5: Send the identity to the original resource

```
                    delete[] sso_origURL;
                }
                break;
            case ENFORCER_DENY_ACTION:
                // need to figure out the full_url to make SSO decision
                host_header = (char *) ap_table_get(r->headers_in,
                              "Host");
                if (host_header != NULL) {
                    snprintf(full_url, 1024, "%s://%s%s",
                        ap_http_method(r), host_header, r->uri);
                } else {
                    snprintf(full_url, 1024, "%s://%s%s",
                      ap_http_method(r), ap_get_server_name(r), r->uri);
                }
                if (http_query != NULL) {
                    strcat(full_url, "?");
                    strcat(full_url, http_query);
                }
                redirect_URL = reply->
                               getChildStringValue(ENFORCER_REDIRECT);
                if (EnforcerCouldSSOHelp(reply, referer, full_url,
                    Enforcer_Handle)) {
                    // SSO could result in an authentication without
                       prompting user
                    // do SSO redirect 1
                    char *sso_redirectURL;
                    EnforcerConstructSSORedirectURL(referer, full_url,
                        &sso_redirectURL);
                    response = send_redirect(r, sso_redirectURL);
                    delete[] sso_redirectURL;
                }
            }
        return response;
        }
}
```

Step 1: See if multiple domain SSO can help

Step 2: Redirect identity to referring server

Special UNIX Considerations

For UNIX platforms, the Apache server spawns child processes to handle web requests. When these processes terminate, the Enforcer plugin must free the Enforcer object created during the initialization process. The following sample demonstrates how to free the Enforcer object on UNIX platforms:

```
#ifndef APACHE2
/*
 * stuff to do before apache children exit, unix-only
 */
extern "C" {
    static void child_exit(server_rec *s, pool *p){
```

```
#ifdef __unix__
        if (mod_enforcer_initialized) {
            /* Enforcer API cleanup */
            EnforcerFree(Enforcer_Handle);
            mod_enforcer_initialized = 0;
        }
#endif
    }
}
#endif
```

Registering Apache Plugins

In order for the Apache 2 Enforcer plugin to method, the module must register itself with the Apache server. This is accomplished by creating registration modules as shown in following example. Only the Apache 2 registration is shown here.

```
static void enforcer_register_hooks(apr_pool_t *p){
    ap_hook_child_init(child_init, NULL, NULL, APR_HOOK_MIDDLE);
    ap_hook_access_checker(enforcer_check, NULL, NULL, APR_HOOK_MIDDLE);
}
/* Dispatch list for API hooks */
module AP_MODULE_DECLARE_DATA enforcer_module = {
    STANDARD20_MODULE_STUFF,
    NULL,                       /* create per-dir    config structures */
    NULL,                       /* merge  per-dir    config structures */
    NULL,                       /* create per-server config structures */
    NULL,                       /* merge  per-server config structures */
    NULL,                       /* table of config file commands       */
    enforcer_register_hooks     /* register hooks                      */
};
```

Compiling and Installing the Apache Enforcer Plugin

You can do the following with Apache 2 Enforcer plugin:

- Compile it into the server statically.
- Implement it as dynamically loaded libraries (DLLs) on Windows.
- Implement it as shared objects (`.so`) files on UNIX. The examples shown in this chapter use this technique, as it requires less effort to install dynamic plugins. You can configure the Apache 2 Enforcer plugin described below for specific sites simply by redefining some constants at the top of the `mod_enforcer.h` file and recompiling Apache 2 Enforcer plugin. To build the `mod_enforcer.h` file, change directories to *<SDK_install_path>*`/source/server_plugins/apache` and execute the `gmake` command. Refer to Building Examples in the SDK on page 23 for more information about the `gmake` command.

The output from the `gmake` command should look something like the example below. If you are building a custom Apache 2 Enforcer plugin, you will need to provide the options, header paths, and libraries to your compiler.

- To compile the Apache 2 Enforcer plugin using the provided `Makefile`, you must have compiled the Apache server with the `mod_so` module enabled. This is the default on Red Hat and many other operating systems.
- The `Makefile` for Apache included in the SDK specifies the installation location of the Apache 2 server. You can modify the file to install Apache in any folder you wish.

```
g++ -o ../../../solaris/bin/mod_enforcer.so -Wc,-Wall -Wc,-g
-Wl,-Bsymbolic
    -D_PTHREADS -D_REENTRANT -DPOSIX_THREADS -I../../../enforcer
    -L../../../solaris/lib
    -lenforcer -lcurl -lssl -lcrypto -licuuc -licudata -lcurl
    -lsocket -lnsl -ldl -lthread  -lstdc++
    ../mod_enforcer.cpp ../util.cpp
```

Creating an Enforcer Plugin With the COM Enforcer API

This section demonstrates how to utilize the COM Enforcer API. The Web Service Enhancement (WSE) Enforcer plugin is used as an example. In this section you will learn how to:

- Initialize an Enforcer object
- Extract authorization data from a SOAP message
- Build a Policy Validator request
- Query a Policy Validator
- Enforce the Policy Validator decision

The WSE Enforcer Plugin Example

The WSE Enforcer plugin is a COM component that secures access to .Net web services. The WSE Enforcer plugin connects the .Net API to the Enforcer API. SOAP message properties are used to build Policy Validator queries. The WSE Enforcer plugin will either allow the access to the web service, or deny access. If access is denied, the WSE Enforcer plugin will provide authentication hints to the web services client. The client should then present the appropriate credentials to the WSE Enforcer plugin.

The Helper class

The `Helper` class is used by both the `InputFilter` and the `OutputFilter` classes. The `Helper` class defines SOAP namespace constants, XML tag constants, and other constants used to process SOAP messages, Policy Validator queries and Policy Validator replies. The static `m_handle` field is used to store the Enforcer object used by the `InputFilter` and

`OutputFilter` to interact with the Policy Validator. The `Helper` class also defines methods to obtain and configure an Enforcer object used to communicate with the Policy Validator. The `Helper` class framework is shown in the following sample:

Shared Enforcer object for all filter instances

```
... using libraries ...
namespace com.hp.selectaccess.enforcer.wse
{
    public class Helper
    {
        protected static EnforcerCOMHandle m_handle = null;

        ... Define other constants ....

        public Helper(){}
        public static void InitEnforcer() { ... }
        public static EnforcerCOMHandle GetEnforcerCOMHandle() { ... }
        public static int GetRandomNumber() { ... }
        public static SoapException ConstructException
            (
                string faultCode,
                string faultCodeNS,
                string faultString
            ) { ... }
    }
}
```

The InputFilter class

The `InputFilter` class overrides the `ProcessMessage()` method. This enables the `InputFilter` class to extract SOAP message properties, build the Policy Validator query, and enforce the response. The `InputFilter` class framework is shown below:

```
... using libraries ...
namespace com.hp.selectaccess.enforcer.wse
{
    public class InputFilter:SoapInputFilter
    {
        public InputFilter(){}
        public override void ProcessMessage(SoapEnvelope envelope)
          { ... }
        public EnforcerXMLTree constructValidatorQuery
            (
                XmlElement body,
                SoapContext requestContext,
                EnforcerCOMHandle valHandle
            ) { ... }
        public void SendValidatorQuery
            (
                EnforcerXMLTree query,
                EnforcerCOMHandle valHandle
```

```
        ) { ... }
    public void GetValidatorCookieFromReply
        (
            EnforcerXMLTree reply,
            EnforcerCOMHandle valHandle
        ) { ... }
    public void AddPersonalizationData
        (
            EnforcerXMLTree reply,
            EnforcerCOMHandle valHandle
        ) { ... }

    ... Methods to extract encryption and signature attributes ...

} // class CustomInputFilter
} //CustomLibrary
```

The OutputFilter class

The `OutputFilter` class is called after the .Net web service has created a SOAP response. The `OutputFilter` class provides methods to insert any Select Access single sign-on nonces as cookies to the SOAP response. If authentication is required, the `OutputFilter` class inserts authentication hints as policy SOAP headers in the SOAP response. The `OutputFilter` class also provides methods to encrypt and sign the SOAP message. This is used to protect the single sign-on nonce from potential hackers. The `OutputFilter` class framework is shown in below:

```
... using libraries ...
namespace com.hp.selectaccess.enforcer.wse
{
    public class OutputFilter:SoapOutputFilter
    {
        public OutputFilter() { ... }
        public override void ProcessMessage(SoapEnvelope envelope)
          { ... }
        public void AddValidatorCookie
            (
                SoapEnvelope envelope,
                EnforcerCOMHandle valHandle
             ) { ... }
        public void AddValidatorCookieToSoap
            (
                SoapEnvelope envelope,
                string nonce,
                EnforcerCOMHandle valHandle
            ) { ... }
        public void AddSignedPolicyHeader
            (
                SoapEnvelope envelope,
```

```
                string[] authTypes,
                bool signPolicy,
                EnforcerCOMHandle valHandle
            ) { ... }
        public void EncryptResponseMessage
            (
                SoapContext context,
                EnforcerCOMHandle valHandle,
                bool mustEncrypt
            ) { ... }
        public void SignResponseMessage
            (
                SoapContext context,
                Reference reference,
                EnforcerCOMHandle valHandle
            ) { ... }

       ... More signature helper methods ...

    }
}
```

The COM Enforcer API Classes and Utilities

The `EnforcerCOMHandle` class located in the `ENFORCERLib` library is the principal Enforcer API class. This class represents an Enforcer object used to communicate with the Policy Validator.

To build the Policy Validator queries and responses, and to process the SOAP XML, the standard `System.xml` library classes may be used.

Importing the COM Libraries

To import the COM Enforcer API libraries, include the `ENFORCERLib` library as show in the following example. The code example includes the other libraries that the WSE Enforcer plugin requires.

```
using System;
using System.Collections;
using System.Diagnostics;
using System.IO;
using System.Web;
using System.Web.Services.Protocols;
using System.Xml;
using Microsoft.Web.Services;
using Microsoft.Web.Services.Security;
using ENFORCERLib;
```

Creating an Enforcer Object

The `Helper` class creates an Enforcer object using the `InitEnforcer()` method. This method initializes the Enforcer API with the `m_handle.Init()` call. The `InputFilter` and `OutputFilter` classes use the `GetEnforcerCOMHandle()` method to retrieve the configured Enforcer object. The following example demonstrates this.

```
public static void InitEnforcer()
  {
         /*
          *  LOG_TYPE = "wse"
          *  At the time of initialization, this WSE Enforcer will look
             for the file"
          *  "INSTALL_DIR/bin/enforcer_wse.xml"
          */
          if (m_handle == null)
          {
               m_handle = new EnforcerCOMHandle();
               string loggingName = m_handle.GetCompanyShortName() + " "
                    + m_handle.GetProductName() + " " + LOG_TYPE + " "
                    + m_handle.GetServerPluginName();
                m_handle.Init(null, loggingName, LOG_TYPE.ToLower());
               string msg = "WSE Enforcer Validator Handle Initialized";
                m_handle.EnforcerLogMsg(m_handle.GetInfoLogType(), msg);
          } //m_handle == null
    } //InitEnforcer

    public static EnforcerCOMHandle GetEnforcerCOMHandle()
    {
        if (m_handle == null)
        {
            InitEnforcer();
        }
        return m_handle;
    }
```

Create a new Enforcer object

Initialize the Enforcer object

Provide the Enforcer object to filters

Building an XML Request

The `InputFilter` class overrides the `ProcessMessage()` method to validate the request using the Policy Validator. The method calls the `constructValidatorQuery()` method to build an XML request. The properties for the Policy Validator query are extracted from the SOAP request.

The `constructValidatorQuery()` method extracts the required service and path properties from the `HttpRequest` in the SOAP request. The `constructValidatorQuery()` method then adds the ID and password and X.509 certificate authentication credentials to the XML query, if these properties are present in the request. If the SOAP header or HTTP header includes any single sign-on nonces in a cookie, `constructValidatorQuery()` adds the nonce to the query. If the Enforcer plugin is configured to enable cookies, the

NATIVE_NONCE property is added to the query to inform the Policy Validator to include a single sign-on nonce in the reply. The final query is returned to the ProcessMessage() method to send the query to the Policy Validator.

```
public EnforcerXMLTree constructValidatorQuery
    (
        XmlElement body, SoapContext requestContext, EnforcerCOMHandle
        valHandle
    )
{
    try
    {
        EnforcerXMLTree query = null;
        string httpClientCert = null;
        HttpRequest httpRequest = HttpContext.Current.Request;
        Uri uri = httpRequest.Url;
        string scheme = uri.Scheme;
        string host = uri.Host;
        int port = uri.Port;
        string path = uri.AbsolutePath;
        if (String.Compare(host, Helper.LOCALHOST, true) == 0
            || (String.Compare(host, "127.0.0.1", true)== 0))
        {
            string hostname = Dns.GetHostName();
            IPHostEntry hostEntry = Dns.GetHostByName(hostname);
            host = hostEntry.HostName;
        }
        string service = scheme + "://" + host + ":" + port;
        string resource = getResource(body);
        path = path + "/" + resource;
        string msg = "Input Filter:: Web Service URL being accessed:\n"
                + service + "\n" +path;
        valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
        // get client side certificate from the HTTP connection
       HttpClientCertificate clientCert = httpRequest.ClientCertificate;
        if (clientCert != null)
        {
            byte[] certBytes = clientCert.Certificate;
            if (certBytes != null && certBytes.Length > 0)
            {
                httpClientCert = Convert.ToBase64String(certBytes);
                msg = "Input Filter:: Client certificate received in the
                       HTTP"
                    + " connection:\n " + clientCert;
                valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(),
                msg);
            }
        }
        query = valHandle.newQuery(service, path);
```

Extract the HTTP request data for the XML query

Extract client certificate from HTTP connection

Create a new XML query with required properties

```
// Get embedded Security tokens
bool passAdded = false;
bool certAdded = false;
// log
int count = requestContext.Security.Tokens.Count;
msg = "Input Filter:: Total number of SecurityTokens received in
"
    + " the SOAP Message is "+count;
valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
foreach (SecurityToken tok in requestContext.Security.Tokens)
{
    if (tok is UsernameToken && !passAdded)
    {
        string username = ((UsernameToken)tok).Username;
        string password = ((UsernameToken)tok).Password;
        if (username != null && password != null)
        {
            string property = valHandle.GetTagPropertyList();
            EnforcerXMLTree propListElemNode =
                query.newNode(valHandle.GetTagPropertyList());

propListElemNode.setName(valHandle.GetTagPostDataList());
            EnforcerXMLTree userNode =
                query.newNode(valHandle.GetTagProperty());
            userNode.setName(valHandle.GetTagUser());
            userNode.setStringValue(username);
            EnforcerXMLTree passNode =
                query.newNode(valHandle.GetTagProperty());
            passNode.setName(valHandle.GetTagPassword());
            passNode.setStringValue(password);
            propListElemNode.appendChild(userNode);
            propListElemNode.appendChild(passNode);
            query.appendChild(propListElemNode);
            passAdded = true;
            msg = "Input Filter:: UsernameToken received in the
                  SOAP"
                + " message containing the "
                + "username '" + username
                + "has been added to the validator query.";
               valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(),
               msg);
        }
    }
    else if (tok is UsernameToken && passAdded)
    {
        string username = ((UsernameToken)tok).Username;
        string password = ((UsernameToken)tok).Password;
        msg = "Input Filter:: UsernameToken with username '"
```

Add a user name and password if present

Do not add more than one user name and password

```
                + username
               + "'received in the SOAP message will not be added to
                 the "
                + "validator query because the username and password
                   from "
                + "another UsernameToken have already been added.";
            valHandle.EnforcerLogMsg(valHandle.GetWarningLogType(),
            msg);
        }
```

Add certificate properties for login identity

```
        else if (tok is X509SecurityToken && !certAdded)
        {
            X509SecurityToken certToken = (X509SecurityToken)tok;
            X509Certificate cert = certToken.Certificate;
            string base64cert = cert.ToBase64String();
            EnforcerXMLTree certNode =
                query.newNode(valHandle.GetTagProperty());
            certNode.setName(valHandle.GetTagCert());
            certNode.setStringValue(base64cert);
            query.appendChild(certNode);
            certAdded = true;
           msg = "Input Filter:: Added to the validator query is the
                 "
                + " X509SecurityToken " + "containing the cert:\n"
                + base64cert;
            valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(),
            msg);
        }
```

Only add one certificate identity

```
        else if (tok is X509SecurityToken && certAdded)
        {
            X509SecurityToken certToken = (X509SecurityToken)tok;
            X509Certificate cert = certToken.Certificate;
            string base64cert = cert.ToBase64String();
            msg = "Input Filter:: Another X09SecurityToken received
                  in the "
              + "SOAP message has been added to the validator query.
                "
                + "The X509SecurityToken will not be added to the
                   validator"
                + "query. The certificate contained inside this "
                + "X509SecurityToken is:\n" + base64cert;
            valHandle.EnforcerLogMsg(valHandle.GetWarningLogType(),
            msg);
        }
```

Do not add other security tokens

```
        else
        {
           msg = "Input Filter:: A SecurityToken of unknown type has
                  been"
                + " received in the SOAP message.  "
              + "It will not be added to the validator query.  Only "
```

```
                + "UsernameTokens and X509SecurityTokens are sent to
                   the "
                + "validator to be authenticated.  "
                + "The type of the received token
                   is:\n"+tok.GetType();
            valHandle.EnforcerLogMsg(valHandle.GetWarningLogType(),
            msg);
        }
    } //for
```

If no SOAP certificate, add the client certificate from the HTTP connection

```
    if (!certAdded && httpClientCert != null)
    {
        msg = "Input Filter:: X509SecurityToken was not received in
              the "
            + "SOAP request, adding client certificate to the
              validator "
            + "XML query.";
        valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
        EnforcerXMLTree certNode =
            query.newNode(valHandle.GetTagProperty());
        certNode.setName(valHandle.GetTagCert());
        certNode.setStringValue(httpClientCert);
        query.appendChild(certNode);
    }
    // look for cookies in SOAP header first, then in HTTP cookie
       header
```

Look for SSO cookies in SOAP header

```
    SoapEnvelope envelope = requestContext.Envelope;
    XmlElement header = envelope.Header;
    XmlNodeList list =
       header.GetElementsByTagName(valHandle.GetDefaultCookieName(),
        Helper.BALT_NS);
    string nonce = null;
    if (list != null && list.Count > 0)
    {
        XmlNode baltNonce = list[0];
        if (baltNonce != null)
        {
            nonce = baltNonce.InnerText;
           msg = "Input Filter:: Validator nonce received in the SOAP
             "
                + "message will be added to the validator query.  "
                + "It is:\n"+nonce;
            valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(),
            msg);
        }
    }
```

If no SOAP cookies, look for SSO cookies in the HTTP header

```
    if (nonce == null)
    {    // now look in HTTP cookie header
        msg = "Input Filter:: Validator nonce was not received in the
       SOAP"
```

```
            + " message, now checking for cookies in HTTP header.";
        valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
        string name = valHandle.GetDefaultCookieName();
        HttpCookieCollection collection = httpRequest.Cookies;
        for(int iter = 0; iter < collection.Count; iter++)
        {
            string key = collection.GetKey(iter);
            if (String.Compare(key, name, true) == 0)
            {
                nonce = collection[iter].Value;
                msg = "Input Filter:: Validator nonce received as an
                HTTP"
                    + " cookie.  It is:\n"+nonce;
               valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(),
               msg);
                break;
            }
        }
    }
```

Add the SSO nonce to the XML request

```
    if (nonce != null)
    {
        // add the cookie extracted from SOAP or from HTTP header
        // to the validator request
        EnforcerXMLTree nonceNode =
            query.newNode(valHandle.GetTagProperty());
        nonceNode.setName(valHandle.GetTagNonce());
        nonceNode.setStringValue(nonce);
        query.appendChild(nonceNode);
    }
```

There was no SSO cookie provided

```
    else
    {    // log msg
        msg = "Input Filter:: Validator nonce not found in the
        request.";
        valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
    }
    // add 'native_nonce' property
```

If cookies are supported, signal SSO nonce support

```
    int enableCookie = valHandle.GetEnableCookies();
    if (enableCookie > 0)
    {
        EnforcerXMLTree nativeNonceNode =
            query.newNode(valHandle.GetTagProperty());
        nativeNonceNode.setName(valHandle.GetTagNativeNonce());
        nativeNonceNode.setStringValue(valHandle.GetValueEnable());
        query.appendChild(nativeNonceNode);
    }
```

SSO nonces are not supported for this client

```
    else
    {
```

```
                msg = "Input Filter:: The Enforcer Web cookies support not
                enabled";
                valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
            }
            // log the validator query
            string queryStr = query.toString(1);
            msg = "Input Filter:: Validator query constructed.\n"+queryStr;
            valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
            return query;
        }
        catch (Exception e)
        {
            string msg =
                "Input Filter:: Exception when constructing validator
                 query:\n "
                + e.Message;
            SoapException se =

Helper.ConstructException(Helper.FC_SERVER_UNABLETOAUTHENTICATE,
                Helper.BALT_NS, Helper.SERVERFAULTSTRING);
            throw se;
        }
    } //constructValidatorQuery
```

Return the XML query

Querying the Policy Validator

The `ProcessMessage()` method in `InputFilter` uses the `SendValidator()` method to query the Policy Validator and process the reply. The query is sent to the Policy Validator using the code shown in the following example:

```
public void SendValidatorQuery(EnforcerXMLTree query, EnforcerCOMHandle
valHandle)
{
    try
    {
        EnforcerXMLTree reply = valHandle.SendQuery(query);

        .... Process the reply ...

    }
    catch (Exception e)
    {
        ... Process reply exceptions ...
    }
} //SendValidatorQuery
```

Enforcing the Policy Validator Response

The `SendValidatorQuery()` method is used to query the Policy Validator and process the reply. The following example demonstrates the `SendValidatorQuery()` method. If the Policy Validator allows the requested action, the `SendValidatorQuery()` method adds a SSO nonce to the current `HttpContext` so future request will not require additional user authentication. The cookie is added to the `HttpContext` using the `GetValidatorCookieFromReply()` method shown in the following example:

```
public void SendValidatorQuery(EnforcerXMLTree query, EnforcerCOMHandle
valHandle)
{
    try
    {

        ... Query validator ...

        if (reply != null)
        {
            // log reply
            string replyStr = reply.toString(1);
            String msg = "Input Filter:: Validator reply
                          received:\n"+replyStr;
            valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
            // read validator 'action' property
            string action = reply.getChildStringValue
                            (valHandle.GetTagAction());
            if (String.Compare(action, valHandle.GetAllowAction(), true)
                == 0)
            {
                // access allowed, get validator nonce and add to
                   HttpContext Items
                // so that it can be returned to the client by the output
                   filter.
                msg = "Input Filter:: Validator action is ALLOW, locating
                      validator "
                    + "nonce to return to client.";
                valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(),
msg);

                GetValidatorCookieFromReply(reply, valHandle);
                AddPersonalizationData(reply, valHandle);
            }
            else if (String.Compare(action, valHandle.GetDenyAction(),
                     true) == 0)
            {
                //deny access, throw exception and return <Policy>
                msg = "Input Filter:: Validator action is DENY. The
                      Enforcer will "
                    + "return WS-Policy in the SOAP Response Message.";
```

Access allowed.

Get the SSO nonce returned in the reply

Access was denied, throw a SOAP exception indicating login failed

```
                    valHandle.EnforcerLogMsg(valHandle.GetWarningLogType(),
                    msg);
                    SetWSPolicy(reply, valHandle);
                    SoapException se =

        Helper.ConstructException(Helper.FC_CLIENT_FAILEDAUTHENTICATION,
                        Helper.BALT_NS, Helper.CLIENTFAULTSTRING);
                    throw se;
                }
                else
                {
                    // error or unknown action
                    msg = "Input Filter:: Validator action is '" + action +
                          "', will "
                        + "return 'UnableToAuthenticate' faultcode.";
                    valHandle.EnforcerLogMsg(valHandle.GetWarningLogType(),
                    msg);
                    SoapException se =

        Helper.ConstructException(Helper.FC_SERVER_UNABLETOAUTHENTICATE,
                        Helper.BALT_NS, Helper.SERVERFAULTSTRING);
                    throw se;
                }
            }
            else
            {
                String msg = "Input Filter:: Validator reply could not be
                             read. The client "
                    + "cannot be authenticated.";
                valHandle.EnforcerLogMsg(valHandle.GetErrorLogType(), msg);
                SoapException se =

        Helper.ConstructException(Helper.FC_SERVER_UNABLETOAUTHENTICATE,
                        Helper.BALT_NS, Helper.SERVERFAULTSTRING);
                throw se;
            }
        }
        catch (SoapException se)
        {
            // this will probably be the SOAP exception constructed above,
               throw it again
            throw se;
        }
        catch (Exception e)
        {
            valHandle.EnforcerLogMsg(valHandle.GetErrorLogType(), "Input
            Filter::\n"
                + e.Message);
            SoapException se =
```

An error occurred during validation

Could not get the reply from the Validator

```
Helper.ConstructException(Helper.FC_SERVER_UNABLETOAUTHENTICATE,
            Helper.BALT_NS, Helper.SERVERFAULTSTRING);
        throw se;
    }
} //SendValidatorQuery
```

Adding SSO nonces to the HttpContext

Following this activity, the `GetValidatorCookieFromReply()` method extracts an SSO nonce from the Policy Validator reply, if a nonce was present. The following example demonstrates this:

```
public void GetValidatorCookieFromReply
    (
        EnforcerXMLTree reply,
        EnforcerCOMHandle valHandle
    )
{
    string nonce = null;
    nonce = reply.getChildStringValue(valHandle.GetTagNonce());
    if (nonce != null)
    {
        String msg = "Input Filter:: The nonce contained in the validator
                     reply is:\n"
            + nonce;
        valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
        HttpContext.Current.Items.Add(Helper.SACLIENTCOOKIE, nonce);
    }
    else
    {
        String msg = "Input Filter:: The validator reply does not contain
                     a nonce.";
        valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
    }
} //GetValidatorCookieFromReply
```

Setting the WSE Enforcer plugin policy elements

If access was denied, the WSE Enforcer plugin uses the `SetWSPolicy()` method to include XML properties that indicate whether authentication could have been used to access the resource, and if so, which authentication methods are available. An exception is thrown to prevent the resource from being accessed. The exception is serialized as a SOAP fault. The authentication hints are added to the SOAP message using the `OutputFilter`, discussed next.

```
public void SetWSPolicy(EnforcerXMLTree reply, EnforcerCOMHandle
valHandle)
{
    int children = reply.getNumChildren();
    EnforcerXMLTree authServersElem = null;
    for (int i=0; i< children; i++)
```

```
{
    EnforcerXMLTree child = reply.getChild(i);
    string childName = child.getName();
    if (childName != null &&
        String.Compare(childName,
        valHandle.GetTagAuthenticationServerTypes(), true) == 0)
    {
        authServersElem = child;
        break;
    }
}
if (authServersElem != null)
{
    int count = authServersElem.getNumChildren();
    string[] authTypes = new string[count];
    bool authAdded = false;
    string msg = "Input Filter:: The SelectAccess authentication
                 method with "
        + "which the client is required to authenticate with:";
    for (int j=0; j<count; j++)
    {
        EnforcerXMLTree child = authServersElem.getChild(j);
        string childName = child.getName();
        if (childName != null &&
            String.Compare(childName,
                valHandle.GetTagAuthenticationMethod(), true) == 0)
        {
            authTypes[j] = (string)child.getStringValue();
            msg = msg + "\n" + authTypes[j];
            authAdded = true;
        }
    }//for
    if (authAdded)
    {
        // log
        valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
        HttpContext.Current.Items.Add(Helper.AUTHENTICATIONTYPES,
        authTypes);
    }
    else
    {
        string mesg = "Input Filter::  The validator authentication
                      methods will "
            + "not be added to the WS-Policy in the SOAP response.";
        valHandle.EnforcerLogMsg(valHandle.GetWarningLogType(),
        mesg);
    }
}
```

Check to see if any login hints were included in reply

Get the authentication hint

Add authentication methods to the HTTP context

```
    else
    {
        // log
        string msg = "Input Filter:: <"
+valHandle.GetTagAuthenticationServerTypes()
            + "> was not found in the validator reply.  WS-Policy in the
              SOAP "
            + "response will not be set.";
        valHandle.EnforcerLogMsg(valHandle.GetWarningLogType(), msg);
    }
} //SetWSPolicy
```

Filtering SOAP output

Once the `InputFilter` has processed the request and allowed access to the .Net web service, the `OutputFilter` is used to add cookies to the HTTP response and encrypt and sign the SOAP message. The `OutputFilter` class overrides the `ProcessMessage()` method to filter SOAP responses. The code for the `ProcessMessage()` method is shown in Filtering SOAP output on page 138.

Under normal circumstances, the `ProcessMessage()` method will add the SSO nonce to a SOAP header and to the HTTP header. The `AddValidatorCookie()` method, shown in Adding nonces to the SOAP header on page 139, extracts the `HttpContext` nonces and places them in the SOAP header. The SOAP response message is then encrypted and signed.

If the `ProcessMessage()` method determines that a SOAP fault occurred, the method calls the `HandleFault()` method, shown in Adding authentication hints to a SOAP response on page 140, to copy the authentication hints added to the `HttpContext` during the `InputFilter` processing.

```
public override void ProcessMessage(SoapEnvelope envelope)
{
    EnforcerCOMHandle valHandle = Helper.GetEnforcerCOMHandle();
    if (valHandle != null)
    {
        XmlElement bodyElem = envelope.Body;
        XmlDocument doc = bodyElem.OwnerDocument;
        // check if the SOAP Response contains a SOAP Fault
        XmlNodeList nodelist =
envelope.GetElementsByTagName(Helper.SOAP_FAULT,
            Helper.SOAP_NS);
        nodelist = doc.GetElementsByTagName(Helper.SOAP_FAULT,
                   Helper.SOAP_NS);
        if (nodelist != null && nodelist.Count>0)
        {
            HandleFault(envelope, nodelist, valHandle);
            // HandleFault calls SignResponseMessage
            EncryptResponseMessage(envelope.Context, valHandle, false);
        }
        else
        {
            AddValidatorCookie(envelope, valHandle);
```

Look for any SOAP exceptions

If no exceptions, add nonce and encrypt / sign message

```
                SignResponseMessage(envelope.Context, null, valHandle);
                EncryptResponseMessage(envelope.Context, valHandle, true);
            }
```

Could not get a valid handle to the Validator

```
        }
        else
        {
            string mesg = "Output Filter:: Validator Handle is Null, sending
                          faultcode: "
                +Helper.FC_SERVER_UNABLETOAUTHENTICATE;
            valHandle.EnforcerLogMsg(valHandle.GetErrorLogType(), mesg);
            SoapException se =

Helper.ConstructException(Helper.FC_SERVER_UNABLETOAUTHENTICATE,
                Helper.BALT_NS, Helper.SERVERFAULTSTRING);
            throw se;
        }
    } //processMessage
```

Adding nonces to the SOAP header

The `InputFilter` may have added SSO nonces to the `HttpContext`. The `OutputFilter` uses the `AddValidatorCookie()` method to extract the nonces from the `HttpContext` and adds them as headers in the SOAP response. The `AddValidatorCookie()` and the helper method `AddValidatorCookieToSoap()` methods are illustrated in below.

```
public void AddValidatorCookie(SoapEnvelope envelope, EnforcerCOMHandle
valHandle)
{
    // check if we need to send a validator nonce to the client
    string nonce = (string)
                    HttpContext.Current.Items[Helper.SACLIENTCOOKIE];
    if (nonce != null)
    {
```

Add the Validator nonce to the HTTP header

```
        HttpCookie cookie = new HttpCookie
                            (valHandle.GetDefaultCookieName(),nonce);
        HttpContext.Current.Response.Cookies.Add(cookie);
        AddValidatorCookieToSoap(envelope, nonce, valHandle);
        string msg = "Output Filter:: Sending validator nonce to the Web
                     service "
            + " client:\n" + nonce;
        valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
    }
    else
    {
        string msg = "Output Filter:: The validator nonce was not
                     found.";
        valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
    }
} //SendValidatorCookie
```

```
public void AddValidatorCookieToSoap
    (
        SoapEnvelope envelope,
        string nonce,
        EnforcerCOMHandle valHandle
    )
{
    XmlElement header = envelope.CreateHeader();
    XmlDocument document = header.OwnerDocument;
    XmlNode baltNonce = document.CreateNode(XmlNodeType.Element,
        Helper.BALT_NS_PREFIX,
        valHandle.GetDefaultCookieName(),
        Helper.BALT_NS);
    baltNonce.InnerText = nonce;
    header.AppendChild(baltNonce);
}//AddValidatorCookieToSoap
```

Add the nonce to a new SOAP header

Adding authentication hints to a SOAP response

If a SOAP fault occurs, there are several possible reasons. One potential reason is that the `InputFilter` could not authenticate the identity. If this is the case, the `HandleFault()` method extracts all the `HttpContext` authentication hints added by the `InputFilter`, and adds the authentication hints to the SOAP envelope.

```
public void HandleFault
    (
        SoapEnvelope envelope,
        XmlNodeList nodelist,
        EnforcerCOMHandle valHandle
    )
{
    string mesg = "Output Filter:: The Web Service or a Web Service
                   filter is "
        + "sending a fault.";
    valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), mesg);
    // will contain only one fault element
    XmlNode faultNode = nodelist[0];
    XmlNodeList faultList = faultNode.ChildNodes;
    if (faultList != null)
    {
        string faultCode = null;
        string faultString = null;
        string faultDetail = null;
        string msg = "Output Filter::";
        for (int i=0; i<faultList.Count; i++)
        {
            XmlNode child = faultList[i];
            if (String.Compare(child.LocalName, Helper.SOAP_FAULTCODE,
                true) == 0)
            {
```

See if a SOAP fault occurred

Get the fault details

```
                faultCode = child.InnerText;
                msg = msg + "\n" +"The FaultCode is: "+faultCode;
            }
            else if
(String.Compare(child.LocalName,Helper.SOAP_FAULTSTRING,true) == 0)
            {
                faultString = child.InnerText;
                msg = msg + "\n" +"The FaultString is: "+faultString;
            }
            else if
(String.Compare(child.LocalName,Helper.SOAP_FAULTDETAIL,true) == 0)
            {
                faultDetail = child.InnerText;
                msg = msg + "\n" +"The FaultDetail is: "+faultDetail;
            }
        }
        valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
        if (faultCode != null)
        {
            int index = faultCode.IndexOf(":");
            string code = faultCode.Substring(index+1,
                          faultCode.Length-(index+1));
```

See if the fault code was related to a failed login

```
            if (String.Compare(code,
                Helper.FC_CLIENT_FAILEDAUTHENTICATION, true) == 0)
            {
               msg = "Output Filter::The Web Service client could not be
                     "
                   + "authenticated, WS-Policy will be added to the SOAP
                     response.";
                valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(),
                msg);
                string[] authTypes =
```

See if there were any login hints provided

```
(string[])HttpContext.Current.Items[Helper.AUTHENTICATIONTYPES];
                if (authTypes != null)
                {
                    int total = authTypes.GetUpperBound(0);
                    if (total >= 0)
                    {
                        bool signPolicy = false;
                        int sign = valHandle.GetEnableSOAPSigning();
                        if (sign > 0)
                        {
                            signPolicy = false;
                        }
```

Add the list of login types to the SOAP header

```
AddSignedPolicyHeader(envelope,authTypes,signPolicy,valHandle);
                    msg = "Output Filter:: WS-Policy added to the SOAP
                    response.";
```

```
valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(), msg);
                    }
                }
No login hints      else
available           {
                        SignResponseMessage(envelope.Context, null,
                                            valHandle);
                       msg = "Output Filter:: Failed to add WS-Policy to the
                              SOAP "
                           + "response. The authentication method required
                             for client "
                           + "authentication is not known.";

valHandle.EnforcerLogMsg(valHandle.GetWarningLogType(), msg);
                    }
A non-login fault  }
was found          else
                   {
                       SignResponseMessage(envelope.Context, null, valHandle);
                       msg = "Output Filter:: The faultcode is not recognized.
                             WS-Policy "
                           + "will not be added to the SOAP response.";
                       valHandle.EnforcerLogMsg(valHandle.GetDebugLogType(),
                       msg);
                   }
               }
           } //faultList != null
       } //HandleFault
```

Building the WSE Enforcer Plugin

To build the WSE Enforcer plugin, you must first make sure that the Enforcer COM DLL is installed on your system and that your project contains a reference to it. The web service filter uses the Enforcer COM API to contact the validator.

If Select Access is installed on the machine, the Enforcer COM DLL is already installed. If not, you will need to register `enforcer32d.dll` with the operating system. To do this, execute the following command:

```
C:\>regsvr32 /s "<SA_Install_Directory>\bin\enforcer32.dll"
```

Including Site-specific Data

You can create your own custom plugins to add site-specific data to an XML query. Locate example `site_data` plugins for all three web servers in *<SDK_install_path>*`\source\server_plugins`. The examples show how to use a custom plugin to extract time data.

How Site Data is Injected into the Policy Validator Query

When the server loads the sample plugin, each request the Enforcer plugin sends to the Policy Validator has an XML query element called `site_data`. How this element appears varies for each of the following Enforcer plugins:

- Apache 2 Enforcer plugin: `apache time is <time>`
- IIS Enforcer plugin: `IIS_Sample Plugin, Adding Header : site_data`
- Sun/Netscape/iPlanet Enforcer plugin: `iplanet time is <time>`

The `site_data` plugins work is by injecting a header before the Enforcer plugin processes the request. The web server the sends time data to the client.

To load the site_data plugin

1. Include the following calls:
   - For Apache web servers:

     ```
     ap_table_add(r->notes, "site_data", "value")
     ```

   - For IIS web servers:

     ```
     pHeaders->AddHeader(pFC, "site_data", "value")
     ```

     where `pFC` is a pointer to `HTTP_FILTER_CONTEXT`

   - For Sun ONE (iPlanet) web servers:

     ```
     pblock_nvinsert("site_data", "value", rq->vars)
     ```

2. Configure your server to load and execute the `site_data` plugin as required for each specific web server:
   - For Apache web servers: Immediately after the `enforcer_module line`, add the following statement:

     ```
     LoadModule site_data_module "<path_to_site_data_plugin>"
     ```

   - For IIS web servers: Use the console to add the `site_data` plugin in the call chain before the IIS Enforcer plugin.
   - For Sun ONE (iPlanet) web servers: Add the following lines to the respective configuration files as needed.

     `Obj.conf`: Before the `enforcer_check` line add:

     ```
     PathCheck fn="add_site_data"
     ```

     `Magnus.conf`: Append the following two lines:

     ```
     Init fn="load-modules" shlib="path_to_plugin"
     funcs="add_site_data"
     ```

   This ensures that details not normally supported by the Enforcer plugin are extracted and forwarded to the Policy Validator for further processing.

6 Using Personalization Attributes: the Personalization API

Select Access enables Enforcer plugins to provide identity personalization data. Personalization data are identity attributes that are not necessarily used for access control. This chapter describes how to configure Select Access to support personalization, how Policy Validators provide personalization attributes to Enforcer plugins, and how Enforcer plugins can make this information available to applications.

In order for personalization to work, the Policy Builder must be configured to enable it. For details, see the *HP OpenView Select Access 6.2 Policy Builder Guide*.

Chapter Overview

Topics in this chapter include subjects that describe the function of personalization and its API, as well as how you can manipulate data with this feature to suit your business needs:

- Understanding the Personalization API on page 145
- Supporting Personalization in Policy Validators on page 146
- Supporting Personalization in Enforcer Plugins on page 146
- Accessing Personalization Data from Resources on page 150

Understanding the Personalization API

The Personalization API is not a library of methods per se. It is a feature that is provided by cooperation between the Policy Builder, the Validator API and the Enforcer API. Since Enforcer plugins and Policy Validators use XML to communicate, Policy Validators can pass virtually any information to Enforcer plugins, including additional identity data.

This personalization data is transmitted in a special XML property list and is included in the Policy Validator XML reply. The property list is identified by the constant, `ENFORCER_P13NINFO`, which is defined as the string `"personalization"`. Attributes are URL encoded as properties within the personalization property list. The name of the attribute property is determined by the Policy Builder configuration, as discussed below. The following sample demonstrates the personalization property list and personalization properties.

Identity attributes

```
<PolicyValidatorReply>
    <PROPERTY NAME="queryID">2</PROPERTY>
    <PROPERTY NAME="authenticated_dn">cn=john doe,ou=users,dc=hp,dc=com
    </PROPERTY>
    <PROPERTYLIST NAME="personalization">
        <PROPERTY NAME="email">steve$40hp%2Ecom</PROPERTY>
```

```
        <PROPERTY NAME="cname">Steve%20Kotsopoulos</PROPERTY>
        <PROPERTY NAME="user">
            cn%3Djohn%20doe%2Cou%3Dusers%2C%2Cdc%3Dhp%2Cdc%3Dcom
        </PROPERTY>
        <PROPERTY NAME="groups">users,grinders</PROPERTY>
    </PROPERTYLIST>
    ... federation server properties ...
    <PROPERTY NAME="action">ALLOW</PROPERTY>
</PolicyValidatorReply>
```

Supporting Personalization in Policy Validators

In order for Enforcer plugins to receive personalization attributes, authentication plugins in the Policy Validator must include the personalization property list in the XML response.

After an authentication plugin has authenticated an identity, the authentication plugin calls the Validator API method, `handleUserInfo()`, to add the identity to the `UserCache`. This method also extracts the personalization attributes for the current Select Auth node and embeds them in the personalization property list. Authentication plugins that call this method do not need to perform any additional steps to support personalization. For more details on the `handleUserInfo()` method, see Authenticating Identities on page 61.

Supporting Personalization in Enforcer Plugins

When an Enforcer plugin requests a resource access authorization, the Policy Validator will provide personalization data upon successful identity authentication. The personalization attributes are available in the Policy Validator plugin response in a special property list.

The Enforcer plugin is responsible for extracting personalization attributes from the Policy Validator reply and placing these attributes in the local environment. This allows servlets, CGI programs, or other applications to make use of the identity's attributes to customize the display.

Depending on the type of server being secured, the personalization data may be populated in different locations. For web servers, the HTTP request headers are set, which has the side-effect of also populating environment variables. For servlets, a `HashMap` is added to the request object. Other applications may need to publish the attributes to other environments.

If you are supporting personalization using HTTP headers, such as with the Apache 2 Enforcer plugin, the Enforcer plugin must check for spoofed personalization headers. If personalization headers exist in the HTTP request, the request should be rejected. This prevents identitys from embedding false personalization attributes in an HTTP request.

For example, if the Enforcer plugin set environment variables using the Select Access personalization prefix (i.e. `SA`), it should scan the request for all variables beginning with `SA` before processing the request. For servlets, the personalization data must be removed from the HTTP request headers. Initializing the personalization variables prevents unauthorized insertion of personalization attributes.

This section uses the example Enforcer plugins from Chapter 4, Custom Authentication Methods and Rules: the Validator API to demonstrate how to provide personalization support in Enforcer plugins.

The Servlet Enforcer Plugin Example

Before building a Policy Validator request, the servlet Enforcer plugin initializes the personalization `HashMap`. The following code example demonstrates how to do this.

The personalization property list is identified by the constant `ENFORCER_P13NINFO`. Accessing personalization data is no different than accessing any other property list in the Policy Validator reply.

```
public ServletTransaction
    (
        HttpServletRequest request,
        HttpServletResponse response,
        String serverName,
        Enforcer enforcer
    ) throws EnforcerException {
    ... Call parent constructor ...

    ... Build Policy Validator request ...

    // initialize personalization map
    m_request.removeAttribute(SA_PERSONALIZATION);
    m_p13nMap = new HashMap();
    m_request.setAttribute(SA_PERSONALIZATION, m_p13nMap);
}
```

Adding personalization data to the servlet HTTP headers

After the personalization map has been initialized, the servlet Enforcer plugin queries the Policy Validator. If access is allowed, the personalization attributes are extracted and populated in the `HashMap`.

Extracting identity data property list from the reply

```
/**
 * sets personalization data, by walking through the ENFORCER_P13NINFO
   nested
 * propertylist in the XML reply, and calling application-specific code
   to
 * actually set each personalization name/value pair
 *
 */
public void setPersonalization() throws EnforcerException {
    XmlElement p13nXml = m_reply.getChild(
                            DataStrings.ENFORCER_P13NINFO);
    if (p13nXml == null) {
        return;
    }
```

```
    if (! (p13nXml instanceof PropertyListElement)) {
        logError("personalization data exists, but is not a property
                list");
        return;
    }
    PropertyListElement p13nList = (PropertyListElement)p13nXml;
    try {
        Enumeration e = p13nList.getPropertyValues();
        while (e.hasMoreElements()) {
            XmlElement child = (XmlElement)e.nextElement();
            String name  = child.getName();
            String value = child.getText();
            value = URLDecoder.decode(value, "UTF8");
            setPersonalization(name, value);
        }
    }
    catch (UnsupportedEncodingException e) {
        throw new EnforcerException("failed to translate personalization
                                    data", e);
    }
}

/**
 * Set personalization data by walking through the ENFORCER_P13NINFO
   nested
 * propertylist in the XML reply, and calling application-specific code
   to
 * actually set each personalization name/value pair.
 */
protected void setPersonalization(String name, String value) {
    m_p13nMap.put(name, value);
}
```

Extracting identity attributes from the property list

The Apache Enforcer Plugin Example

Before building a Policy Validator request, the Apache 2 Enforcer plugin looks for HTTP headers that contain the Select Access personalization prefix. If a request contains any personalization headers, the request is rejected. The following example demonstrates how to search for these HTTP headers:

```
/*
 * make sure client isn't trying to inject their own personalization
   headers
 */
static int suspicious_headers(request_rec *r){
    char *p;
    ap_table_do(sanitize_request_headers, r, r->headers_in, NULL);
    if ((p = (char *) ap_table_get(r->notes, ENFORCER_P13NPREFIX)) !=
         NULL) {
```

```
            return 1;
        }
        return 0;
    }

    /*
     * check for bad http request headers
     * callback for ap_table_do
     */
    static int sanitize_request_headers(void *data, const char *key, const
    char *val){
        request_rec *r = (request_rec*) data;
        if (strncasecmp(key, ENFORCER_P13NPREFIX,
            strlen(ENFORCER_P13NPREFIX)) == 0) {
            EnforcerLog(ENFORCER_LOG_WARNING,
               ">#<rejecting suspicious header '%s=%s' for url %s", key, val,
                r->uri);
            ap_table_add(r->notes, ENFORCER_P13NPREFIX, key);
            return 0;
        }
        return TRUE;
    }
```

Looking for identity data in HTTP headers

Exporting to the environment

If no personalization headers were found in the identity's request, the Apache 2 Enforcer plugin queries the Policy Validator. If access is allowed, the personalization attributes are extracted and populated in the corresponding HTTP header and environment variables.

Note that the personalization property list is identified by the constant `ENFORCER_P13NINFO`. Accessing personalization data is no different than accessing any other property list in the Policy Validator reply.

```
    /*
     * personalization support: export the user info as environment variables
     */
    void EnforcerSetenv(XMLnode reply, request_rec *r){
        XMLnode tempNode = NULL;
        char empty = '\0';
        int nVarCount, i;
        XmlTreeNode * info = reply->getChild(ENFORCER_P13NINFO);
        if (! info) {
            return;
        }
        nVarCount = info->getNumChildren();
        for (i=0; i<nVarCount;i++) {
            tempNode = info->getChild(i);
            auto_free<char> name(SYS_STRDUP(tempNode->getName()));
            const char * value = tempNode->getStringValue();
            if (name == NULL) {
```

Extracting identity data property list from the reply

Extracting each attribute from the property list

```
                continue;
            } else {
                // convert variable name to upper case
                unsigned int j;
                for (j=0; j < strlen(name.get()); j++) {
                    if (isalpha((int) name[j])) {
                        name[j] = (char) toupper(name[j]);
                    }
                }
            }
            if (value == NULL) {
                value = &empty;
            }
            ApacheSetenv(name.get(), value, r);
        }
    }
    /*
     * Apache-specific way of exporting environment variables
     */
    void ApacheSetenv(char *name, const char *val, request_rec *r){
        // add the variables to the request header, which also sets
           environment variable
        if(EnforcerGetP13Ncompat(Enforcer_Handle)) {
            ap_table_add(r->headers_in, name, (char*) val);
        } else {
            int str_len = strlen(name) + strlen(ENFORCER_P13NPREFIX) +1;
            auto_free<char> secure_name((char*) malloc(str_len));
           snprintf(secure_name.get(), str_len, "%s%s", ENFORCER_P13NPREFIX,
                    name);
            ap_table_add(r->headers_in, secure_name.get(), (char*) val);
        }
        // EnforcerLog(ENFORCER_LOG_DEBUG, "Setenv %s=%s", name, val);
        // ap_table_add(r->subprocess_env, name, (char*) val);
    }
```

Adding the identity data to the HTTP header and environment

Accessing Personalization Data from Resources

This section provides some examples of code that extract environment variables and display the settings. The code examples demonstrate several common server environments, including CGI, servlet, and COM. The following code examples merely display the current environment. Your application will likely use the extracted data to create customized output based on the personalization data extracted.

Displaying UNIX Environment Variables

This example uses a server-side include to execute a common gateway interface (CGI) program that will display all the available environment variable names and their values. Place the following Apache 2 Enforcer plugin configured. Use the following command to call the CGI perl script shown below.

```
<!--#exec cgi="/sktest/allow/UNIX-printenv.cgi"-->
```

The following script displays environment variables in Perl:

```
#!/usr/bin/perl
# Adjust the above line to find perl on your system

#  test-env.cgi Copyright 1995 by David Efflandt efflandt@xnet.com
#     May be freely distributed and modified.
#     Author is not responsible for its use.
#  Note: Variable list will differ if run locally from UNIX

# Print content-type for HTTP/1.0 compatibility
print "Content-type: text/html\n\n";

# Print title and initial heading
print "<Head><Title>Environment</Title></Head>\n";
print "<Body><H1>List of all Environmental Variables</H1><HR>\n";

# Sort and print all environmental variables
foreach $key (sort keys(%ENV)) {print "$key = $ENV{$key}<p>\n";}

# Resolve REMOTE_HOST into name
$ip_address = $ENV{'REMOTE_ADDR'};
@numbers = split(/\./, $ip_address);
$ip_number = pack("C4", @numbers);
($name) = (gethostbyaddr($ip_number, 2))[0];
if (defined $name) {print "<HR>REMOTE_HOST resolves to: $name";}

# Print closing to point to your home page
print "<HR><P>\n";
print 'Return to my <A HREF=".">home page</A>.<P></BODY>';
exit;
```

Displaying HTTP Headers from a Servlet

The `ServletEnvPrint` class, demonstrated in the code sample below, extracts not only the Select Access personalization attributes, but also other servlet parameters. `ServletEnvPrint` extracts all request attributes from the `HttpServletRequest` class and displays the name and value of the attribute. If the attribute is not a string, the servlet displays the Java class name of the attribute. For post data and personalization attributes inserted by Policy Validators, the example illustrates how to iterate the maps of name/value pairs stored in the `HttpServletRequest`.

```
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ServletEnvPrint extends HttpServlet
{
    public void init(ServletConfig config)
        throws ServletException
    {
        m_config = config;
    }
    public void doPost(HttpServletRequest req, HttpServletResponse
                       res)
        throws ServletException, IOException
    {
        doGet(req, res);
    }
    public void doGet(HttpServletRequest req, HttpServletResponse
                      res)
        throws ServletException, IOException
    {
        m_count += 1;
        res.setContentType("text/html; charset=UTF-8");
        PrintWriter out = res.getWriter();
        out.println("<html>");
        out.println("<head><title>Servlet Environment</title>
                  </head>");
        out.println("<body><h1>Servlet Environment</h1></body>");
        out.println("<h2>Configuration Parameters</h2>");
        out.println("<table>");
        Enumeration enum = m_config.getInitParameterNames();
        while (enum.hasMoreElements()) {
            String name = (String)enum.nextElement();
            String value = m_config.getInitParameter(name);
            out.println("<tr><td>" + name + "</td><td>=</td><td>"
                + value + "</td></tr>");
        }
        out.println("</table>");
        out.println("<h2>Request Attributes</h2>");
        out.println("<table>");
        enum = req.getAttributeNames();
        while (enum.hasMoreElements()) {
            String name = (String)enum.nextElement();
            Object value = req.getAttribute(name);
            if (value == null) {
                out.println("<tr><td>" + name
                    + "</td> <td>=</td><td>-NULL-</td></tr>");
            }
```

Display configuration properties

Display all attribute name and value

```
            else if (value instanceof String) {
               out.println("<tr><td>" + name + "</td> <td>=</td><td>"
                     + value + "</td></tr>");
            }
            else {
                String className = value.getClass().getName();
                out.println("<tr><td>" + name + "</td> <td>=
                            </td><td><b><em>"
                    + className + "</em></b></td></tr>");
            }
        }
        out.println("</table>");
        out.println("<h2>Post Data Attributes</h2>");
        Map mm = (Map)req.getAttribute("SA_PostData");
        if (mm == null) {
            out.println("<tr><td>no data</td></tr>");
        }
        else {
           for (Iterator it=mm.keySet().iterator(); it.hasNext(); ) {
                String name = (String)it.next();
                String value = (String)mm.get(name);
                out.println("<tr><td>" + name + "</td> <td>" + value
                     + "</td></tr>");
            }
        }
        out.println("</table>");
        out.println("<h2>Personalization</h2>");
        out.println("<table>");
        Map m = (Map)req.getAttribute("SA_Personalization");
        if (m == null) {
            out.println("<tr><td>no data</td></tr>");
        }
        else {
            for (Iterator it=m.keySet().iterator(); it.hasNext(); ) {
               String name = (String)it.next();
               String value = (String)m.get(name);
               out.println("<tr><td>" + name + "</td><td>" + value +
                           "</td></tr>");
            }
        }
        out.println("</table>");
        out.println("</html>");
    }

    protected ServletConfig m_config;
    protected int m_count = 0;
}
```

For non-String objects, display the class name

Retrieve and display the Select Access post data

Retrieve and display the identity attributes

Displaying Server Variables With Visual BASIC

The following example uses Visual BASIC to display the personalization attributes from an Active Server Page (ASP):

```
<%@ Language=VBScript %>
<html>
<body>
<table>
<%
For Each elem In Request.ServerVariables
    Response.Write "<tr><td>"
    Response.Write "<b>" & elem & "=</b>" & _
        Request.ServerVariables(elem)
    Response.Write "</td></tr>"
Next
%></table>
</body>
</html>
```

7 Querying for Multiple Resources: the Policy API

Select Access enables Enforcer plugins to provide to query the Policy Validator for access to multiple resources. The Policy API is an extension to the Enforcer API. The Policy API contains methods that simplify building and processing queries that contain requests for multiple resources. This chapter describes how to use multiple resource requests.

Chapter Overview

Topics in this chapter include subjects that multiple resource queries, the Policy API, how to program components to support this functionality:

- Understanding Access Control on page 155
- Understanding XML and the Policy API on page 155
- Using the Java Policy API on page 157
- Using the C/C++ Policy API on page 158
- Using the COM Policy API on page 159

Understanding Access Control

Part of access control is preventing unauthorized access to resource. Another aspect of access control that is often overlooked is preventing an identity from even knowing that a resource exists. For instance, it is not good practice to present the identity with a list of menu options if the identity is only authorized to access a few of the menu options.

To build a usable interface, it is often necessary to know whether an identity could access a resource before ever building the menu option that allows the identity to select the resource. Since menus often contain many items, it is necessary to query the Policy Validator with a list of resources.

Understanding XML and the Policy API

Select Access provides a method of asking the Policy Validator to authorize a set of resources. To do this, a special property list is added to the XML request that is sent to the Policy Validator. The property list is identified by the constant `MULTIPLERESOURCELIST`, which is the defined as the string `"multipleResourceList"`. Within a multiple resource property list are one or more resources, each identified by a resource property list. A resource property list is identified by the constant `ENFORCER_RESOURCE`, which is defined as the string

"resource". A resource property list must contain a service tag to identify the resource. Service tags are added using the standard ENFORCER_SERVICE property, which is defined as the string "service".

Multiple resource request

The following sample shows the XML for a multiple resource request.

```
<PolicyValidatorQuery>
    <PROPERTY NAME="queryID">67</PROPERTY>
    <PROPERTY NAME="service">http://dev01.ca.hp.com:8050</PROPERTY>
    <PROPERTY NAME="path">/foo</PROPERTY>
    <PROPERTY NAME="user">sk</PROPERTY>
    <PROPERTY NAME="password">XXXXXX</PROPERTY>
    <PROPERTYLIST NAME="multiResourceList">
        <PROPERTYLIST NAME="resource">
            <PROPERTY NAME="path">/sktest/allow</PROPERTY>
        </PROPERTYLIST>
        <PROPERTYLIST NAME="resource">
            <PROPERTY NAME="path">/sktest/deny/deny.html</PROPERTY>
        </PROPERTYLIST>
        <PROPERTYLIST NAME="resource">
            <PROPERTY NAME="path">/sktest/auth</PROPERTY>
        </PROPERTYLIST>
        <PROPERTYLIST NAME="resource">
            <PROPERTY NAME="path">/four.html</PROPERTY>
        </PROPERTYLIST>
        <PROPERTYLIST NAME="resource">
            <PROPERTY NAME="path">/five.html</PROPERTY>
        </PROPERTYLIST>
        <PROPERTYLIST NAME="resource">
            <PROPERTY NAME="path">/six.html</PROPERTY>
        </PROPERTYLIST>
    </PROPERTYLIST>
</PolicyValidatorQuery>
```

Multiple resource reply

When evaluating multiple resource, the Policy Validator returns a property list for the multiple resources. Each resource is embedded in its own property list, containing the original service property and the allow and deny message for the resource. The following code sample demonstrates the Policy Validator reply for a multiple resource request. Note that each resource has an allow or deny value in the action property.

```
<PolicyValidatorReply>
    <PROPERTY NAME="queryID">67</PROPERTY>
        <PROPERTYLIST NAME="multiResourceList">
          <PROPERTYLIST NAME="resource">
            <PROPERTY NAME="path">/sktest/allow</PROPERTY>
            <PROPERTY NAME="action">ALLOW</PROPERTY>
          </PROPERTYLIST>
```

```
            <PROPERTYLIST NAME="resource">
              <PROPERTY NAME="path">/sktest/deny/deny.html</PROPERTY>
              <PROPERTY NAME="action">DENY</PROPERTY>
            </PROPERTYLIST>
            <PROPERTYLIST NAME="resource">
                <PROPERTY NAME="path">/sktest/auth</PROPERTY>
                <PROPERTY NAME="action">ALLOW</PROPERTY>
            </PROPERTYLIST>
            <PROPERTYLIST NAME="resource">
                <PROPERTY NAME="path">/four.html</PROPERTY>
                <PROPERTY NAME="action">ALLOW</PROPERTY>
            </PROPERTYLIST>
            <PROPERTYLIST NAME="resource">
                <PROPERTY NAME="path">/five.html</PROPERTY>
                <PROPERTY NAME="action">ALLOW</PROPERTY>
            </PROPERTYLIST>
            <PROPERTYLIST NAME="resource">
                <PROPERTY NAME="path">/six.html</PROPERTY>
                <PROPERTY NAME="action">ALLOW</PROPERTY>
            </PROPERTYLIST>
        </PROPERTYLIST>

        ... other properties ...
        ... personalization attributes ...

        <PROPERTY NAME="action">DENY</PROPERTY>
</PolicyValidatorReply>
```

While you can use the standard XML classes provided with Select Access to build multiple resource requests and process multiple resource responses, the code examples in this section demonstrate how to use the multiple resource methods in the Java and COM Enforcer APIs. C/C++ developers must use the standard XML classes, as no helper methods exist for multiple resources in the C/C+ Enforcer API.

Using the Java Policy API

The following example uses Java Server Pages (JSP) to create a Policy Validator request with multiple resources. The JSP displays the configured policy for each of the resources requested.

```
<%@page contentType="text/html"%>
<%@page import="com.hp.selectaccess.util.XmlElement" %>
<%@page import="com.hp.selectaccess.enforcer.*" %>
<%@page import="java.util.*" %>
<%@page import="javax.servlet.*, javax.servlet.http.*,
javax.servlet.http.HttpUtils" %>
<%! static private Enforcer enforcer = null;
    public void jspInit() {
        if (enforcer == null) {
```

```
            enforcer = new Enforcer(null, null, "policy_api", "jsp",
true, 1);
        }
    }
  %>

<%
        XmlElement q = enforcer.XmlQueryInit
                       ("http://dev01.ca.hp.com:8050",
            "/foo");
        q.appendPropertyValue(DataStrings.ENFORCER_USER, "sk");
        q.appendPropertyValue(DataStrings.ENFORCER_PASSWD, "sksksk");
        enforcer.XmlAppendMultiResource(q,"/sktest/allow");
        enforcer.XmlAppendMultiResource(q,"/sktest/deny/deny.html");
        enforcer.XmlAppendMultiResource(q,"/sktest/auth");
        String[] resArray = {"/four.html","/five.html","/six.html"};
        enforcer.XmlAppendMultiResources(q,resArray);
        ValidatorResponse vr = enforcer.XmlQuerySend(q);
        XmlElement xr = vr.getXmlElement();
        Hashtable ht = enforcer.getMultiResourceResult(xr);
        Enumeration keys = ht.keys();
        String currKey = null;
        %>
<html>
<head><title>JSP Page</title></head>
<body>
<h1>Basic Java PolicyAPI JSP Program for SelectAccess 6.0</h1>

<table>
<tr><th><i>Resource</i></th><th><I>Action</I></th></tr>
  <%      while (keys.hasMoreElements()) {
            currKey = (String)keys.nextElement(); %>
<tr><td><%= currKey %></td><td><%= ht.get(currKey) %></td></tr>
  <%      } %>

</table>
</body>
</html>
```

Create a query

Add multiple resources to the request

Get the multi-resource responses

Using the C/C++ Policy API

There are no helper methods in the C/C++ Policy API for multiple resource requests. You must use the standard XML manipulation methods to build and process the requests. Use Multiple resource request and Multiple resource reply on page 156 as a guide for building requests and processing responses.

Using the COM Policy API

The following example illustrates how to use Visual BASIC to create a Policy Validator query with multiple resources in the request. This Active Server Page (ASP) will display each of the resources with the configured policy for that resource.

```
<%@ Language=VBScript %>
<% option explicit %>
<% response.Expires=-1 %>
<%
Sub Test_Multi_Resource_Query
    Dim xmlTree, qresponse, response_dict

    Rem  The following line initializes a new Query
    Set xmlTree =
        Application("EnforcerHandle").newASPQuery(
        "testdriver://host.testbed.private:1025",
        "/enf/web/papi/enf_web_papi_0011.asp")

    Rem add some resources to the multi-resource query
    xmlTree.addMultiResource("/enf/web/papi/multi-resource images/
                              cheese.jpg")
    xmlTree.addMultiResource("/enf/web/papi/multi-resource images/
                              flamingo.jpg")
    xmlTree.addMultiResource("/enf/web/papi/multi-resource images/
                              hippo.jpg")
    xmlTree.addMultiResource("/enf/web/papi/multi-resource images/
                              rhino.jpg")

    set xmlTree = addUserPassword(xmlTree)

    Rem send the query to the validator and get back the result
    Set qresponse = Application("EnforcerHandle").SendQuery(xmlTree)
    Response.write "The response is <br><pre>" &
        EscapeXML(qresponse.toString(1)) & "</pre>"
End Sub

Private Function addUserPassword(xmlTree)
    Dim user_node, password_node
    set user_node = xmlTree.newNode("Property")
    user_node.setName("user")
    user_node.setStringValue("papi_three")
    xmlTree.appendChild(user_node)
    set password_node = xmlTree.newNode("Property")
    password_node.setName("password")
    password_node.setStringValue("password")
    xmlTree.appendChild(password_node)
```

Create a Validator query

Add multiple resources to the request

```
    Rem Response.write "<pre>" & EscapeXML(xmlTree.toString(1)) &
            "</pre>"
    Set addUserPassword = xmlTree
End Function

Private Function EscapeXML(strXMLElement)
    strXMLElement = Replace(strXMLElement, "&", "&amp;")
    strXMLElement = Replace(strXMLElement, "<", "&LT")
    strXMLElement = Replace(strXMLElement, ">", "&GT")
    rem strXMLElement = Replace(strXMLElement, """", "&quote;")
    rem strXMLElement = Replace(strXMLElement, """", "&quote;")
    strXMLElement = Replace(strXMLElement, "'", "&apos;")
    EscapeXML = strXMLElement
End Function
%>

<HTML>
<HEAD>
<title>multi-resource query</title>
</HEAD>
<BODY>
    Functions of the policy api tested are:<br>
    <ul>
        <li>IEnforcerCOMHandle.newASPQuery</li>
        <li>IEnforcerCOMHandle.SendQuery</li>
        <li>IEnforcerXMLTree.addMultiResource</li>
        <li>IEnforcerXMLTree.toString</li>
        <li>IEnforcerXMLTree.newNode</li>
        <li>IEnforcerXMLTree.setName</li>
        <li>IEnforcerXMLTree.setStringValue</li>
        <li>IEnforcerXMLTree.appendChild</li>
    </ul>
    <br>
    Tests that password managment requests are ignored for resources in
    the multi-resource query The response should have a propertylist
    called <i>multiResourceList</i> with four propertys called
    <i>resource</i>.
    The <i>path</i> of each should be to cheese, flamingo, rhino and
    hippo and the
    <i>action</i> should be for your policy on each.<br><br>

    <% Call Test_Multi_Resource_Query %>
</BODY>
</HTML>
```

8 Transient Directory Profiles: the User API

Select Access provides an API to store persistent identity attributes for identitys that do not have an profile on the directory server. In this case, the Policy Validator can not provide personalization attributes for them, unless the authentication plugin creates something known as a transient identity profile. This chapter describes how those profiles are created.

For more explicit discussion of transient identity profiles, refer to the *HP OpenView Select Access 6.2 Policy Builder Guide*.

Chapter Overview

Topics in this chapter include subjects that describe transient identity profiles, the User API, and how to create these types of profiles once the Authentication plugin authenticates an identity:

- Understanding the User API on page 161
- The File Authentication Plugin Example on page 162

Understanding the User API

The User API is provided by the `UserCache` class whose methods allow Authentication plugins to create profiles in the directory server for transient identies. The LdapConnection, User, UserSource, and UserCache on page 72 section earlier in this guide also discussed this class. The `UsesrCache` methods create a permanent profiles for the identity. Identity attributes are stored in this entry, and may be accessed via the methods discussed in Chapter 6, Using Personalization Attributes: the Personalization API.

The User API provides write access to the directory server and location represented by a `UserSource`. To create a identity profile in the directory, the Authentication plugin calls the method outlined in the example below.

```
User::UserRefPtr UserCache::sCreateTemporary
(
    const UserSource * userSource,
    const char * subjectName,
    const char * baseDn,
    UserCacheAttributes & attributes
);
```

In this example, the profile is created via a user entry, which is added to the folder identified by `baseDn`. The relative distinguished name is created by appending the `subjectName` to "`CN=`". For example, if you call `sCreateTemporary()` with a `subjectName` equal to "`John`

Doe" and a baseDN set to "dc=acme, dc=com", the LDAP entry would have the DN "CN=John Doe, dc=acme, dc=com". Note that the baseDn must be located in the name space defined by the userSource.

The sCreateTemporary() method adds the necessary directory attributes, such as the inetOrgPerson objectclass, to identify the directory object as an identity. Any attributes provided by the caller are also added.

If sCreateTemporary() is unable to complete the operation, the method returns NULL. LDAP operations might fail for a number of reasons, such as insufficient privileges, schema violations, server unavailable, and so on.

To learn how to manipulate attributes, examine the UserCacheAttributes class in the the UserCache.h header. The UserCacheAttributes class is a collection of key-value maps. The most important method in the class is:

```
void UserCacheAttributes::add(const char * key, const char * value);
```

This method adds an attribute key and an attribute value. The UserCacheAttributes stores a vector of values for each key, so repeated calls to the add() method using the same key parameter will result in additional values being associated with the key. This is to support multi-valued directory attributes.

Note that the UserCacheAttributes keys are case sensitive. This means that "CN" and "cn" are considered to be two different attributes in the UserCacheAttributes.

Attribute keys and values must conform to attribute definitions in the LDAP schema. Invalid attributes will cause sCreateTemporary() method to fail and return NULL.

Do not use attributes with language specifiers. Language-specific attributes are not used by the Personalization API. Language-specific attributes in the directory server may also cause problems when using the Policy Builder to manage directory objects.

The File Authentication Plugin Example

The file authenticator is a simple Authentication plugin that demonstrates the User API. The authenticator uses a password file to authenticate identities. This file must be installed on each Policy Validator.

Once an identity has authenticated, the plugin checks to see if the identity has a directory server entry. If an entry exists, the plugin calls the handleUserInfo() method to obtain personalization data and update the response XML. If the identity does not have a directory server entry, the file authenticator extracts the identity attributes from the password file and creates a directory server entry for the identity.

Do not use the File Authenticator plugin in a production environment. The password file is not encrypted, and identity passwords are not protected. This plugin is provided for demonstration purposes only.

Installing the File Authenticator

The following instructions assume that you already have a compiled version of the plugin. If you do not have a compiled plugin, you will need to first build the plugin.

To build the plugin

1. Go to the `<install directory>/source/validator/plugins` directory.
2. Execute the `make` command. Refer to Building Examples in the SDK on page 23 for more information about the `make` command.
3. Verify the `FileAuthenticator.so` library was created.

To install the File Authenticator, you will need to install the Policy Builder plugin, the Policy Validator plugin, and optionally HelpSet documentation. You will also need a Enforcer plugin to test the plugin.

To install the Policy Builder plugin

1. Select **Tools** → **Configure Policy Plugins** from the main menu bar.
2. Click the **Browse** button next to the **Authentication Service Plugins** field.
3. Select the `<install_directory>/source/Java/jar/FileAuthenticator` directory.
4. Click **Upload**.

To install the Policy Validator plugin

1. Copy the `<install_directory>/source/validator/plugins/FileAuthenticator.so` plugin to `<install_directory>/bin/plugins` directory.
2. Make sure the Policy Validator can access a copy of the `password.ldif` file.
3. Restart the Policy Validator.
4. Repeat for each Policy Validator.

To install the online help

1. Go to the product installation directory.
2. Unpack the help files on the administration server using the command: `jar xf source/java/FileAuthenticator/HelpSet.jar`.
3. Repeat this for each Administration server. The `HelpSet` is not loaded into the directory, and must be installed on each Administration server.

Configuring the File Authenticator

After installing the File Authenticator, you must create an authentication service instance, configure and add it to Select Auth.

To create an authentication service

1. From the Policy Builder, click **Tools** → **Authentication Services**. The **Authentication Services** dialog box appears.
2. Click **Add** to display the **Authentication Method** dialog box. This dialog displays a radio button for each Authentication plugin already uploaded. If the Policy Builder component has been properly installed, you will see an option for **File Authenticator**.

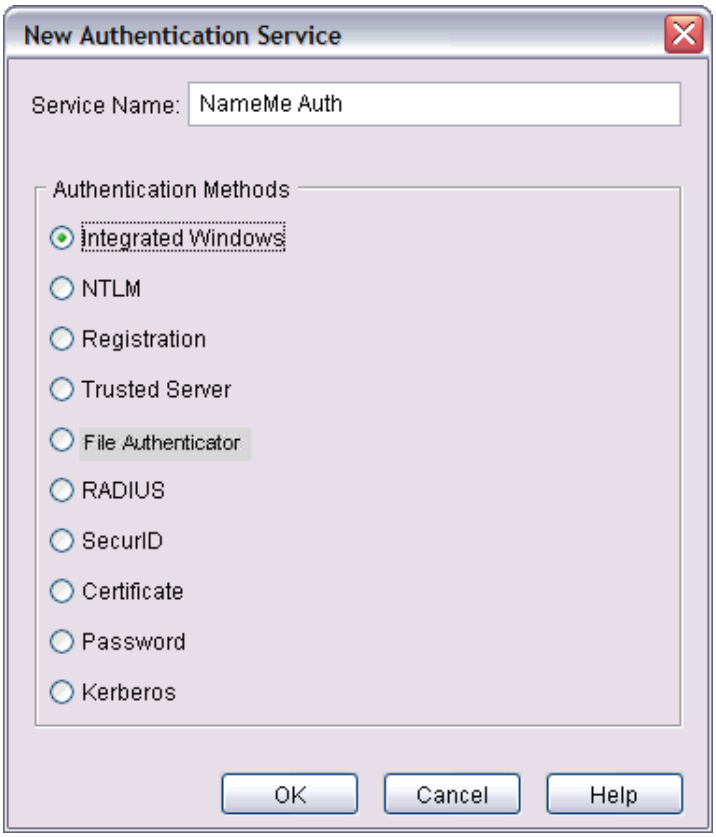

Figure 14 Authentication Method Dialog Box

3 Click the **File Authenticator** method.

4 Type a **Server name**.

5 Click **OK** to create a new file authenticator service, and display its configuration panel. Configure the authentication properties. You must select:

- Identity data location
- Transient identity location
- Password file name

  The password file must be located on the Policy Validator machine. The password file used by the file authenticator must conform to the Lightweight Directory Interface Format (LDIF). For an example, see Sample password.ldif entry on page 165.

- Login form name

  The login form must be located on the Enforcer plugin machine.

6 Select which attributes in the password file need to be copied to LDAP with the User API, by clicking on the **Configure Attributes** tab in the file authenticator Service Properties dialog. The **File Authenticator Properties** dialog appears as shown in Figure 15.

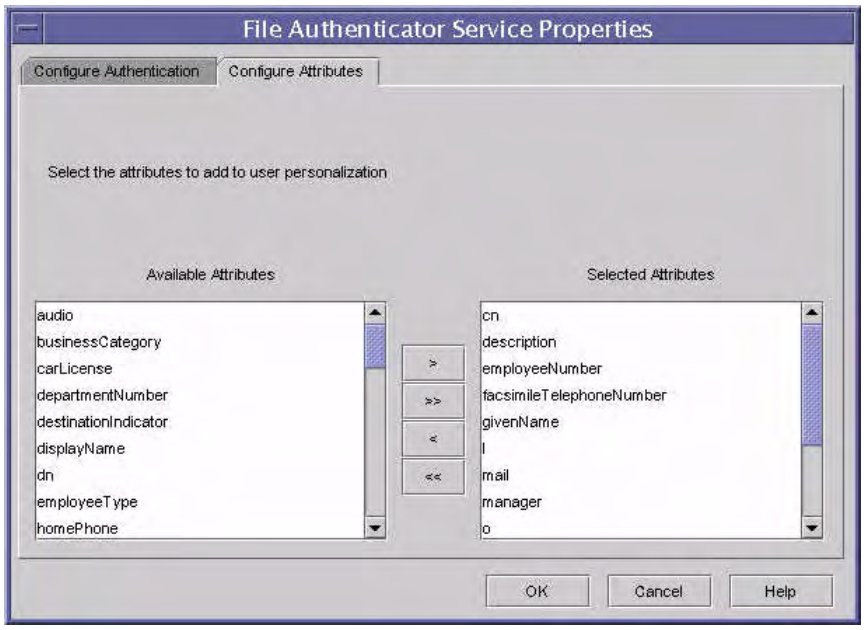

Figure 15 Configuring Attributes

7 Select the attributes and click **OK**.

Sample password.ldif entry

The following is an LDIF-compliant password entry:

```
dn: uid=duser,ou=People, dc=cryptoknights,dc=com
givenName: Demo
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetorgperson
sn: User
cn: Demo User
uid: duser
userPassword: passw0rd
```

How the File Authenticator Works

Like all Policy Validator plugins, the file authenticator uses the static method `init()` to register itself with the Policy Validator. The `init()` method registers the `factory()` method shown below. This method is responsible for validating the XML properties configured by the Policy Builder and creating a `FileAuthenticator` instance. The code shown in the example below validates the XML properties, ensures the password file is readable, creates a plugin instance, configures the allowable attributes, and stores the password file entries as a map of identity profiles.

```
// Creates an instance of the FileAuthenticator
AuthPlugin * FileAuthenticator::factory
(
    const string & name,
```

```
    const XmlTreeNode * props
)
{
    // Must have XML properties
    if (! props)
    {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
            ">#<No File Authenticator properties specified");
        return NULL;
    }
    // Extract the XML properties
    const char *transientUserLocName =
        props->getChildStringValue(TRANSIENT_LOCATION_TAG);
    const char * loginFormName =
        props->getChildStringValue(LOGIN_FILENAME_TAG);
    const char * passwordFileLocName =
        props->getChildStringValue(PASSWORD_FILENAME_TAG);
    const char * userSourceName = props->
                                  getChildStringValue(USER_SOURCE_TAG);
    const UserSource * userSource = NULL;
    // Make sure a user source was provided
    if (!sNameToUserSource(name.c_str(), userSourceName, userSource))
    {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
            ">#<FileAuthenticator: Could not create a valid User
               Source");
        return NULL;
    }
    // Make sure a transient user location was provided and that it is
       valid
    if (! transientUserLocName ||
        ! sValidateSyntheticUserLocation(transientUserLocName,
                                         userSource))
    {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
            ">#<FileAuthenticator: Transient location not located within
               User Source");
        return NULL;
    }
    // Make sure a login form was provided. There is no way to validate
       the form exists
    if (! loginFormName)
    {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
           ">#<FileAuthenticator: Could not locate the login form in the
               configuration");
        return NULL;
    }
```

Extract the XML properties

Validate the UserSource and transient user location

Validate the login form name is not null

Validate the password field is not null

```
// Make sure a password file name was provided
if (! passwordFileLocName)
{
    Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
        ">#<FileAuthenticator: Could not locate password file in
           configuration");
    return NULL;
}
```

Validate the permissions on the password file

```
// Make sure we have read privileges to the file
else if (access(passwordFileLocName, 4) == -1)
{
    // If the file does not exist...
    if (errno == ENOENT)
    {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
            ">#<FileAuthenticator: The password file %s does not
               exist.",
            passwordFileLocName);
    }
    // If we don't have read privileges...
    else if (errno == EACCES)
    {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
            ">#<FileAuthenticator: The password file %s is not
               readable.",
            passwordFileLocName);
    }
    // If we encountered some other error
    else
    {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
            ">#<FileAuthenticator: An unknown error occurred while
               opening file %s.",
            passwordFileLocName);
    }
    // Could not read password file, so can not try to create plugin
    return NULL;
}
// Try to construct a plugin instance
FileAuthenticator * plugin = NULL;
try
{
    // Make sure we can open the password file
```

Try to open the password file

```
    FILE * passwordFile = NULL;
    passwordFile = fopen(passwordFileLocName, "r");
    if (passwordFile == NULL)
    {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
```

```
                    ">#<FileAuthenticator: An unknown error occurred while
                       opening file %s.",
                    passwordFileLocName);
                return NULL;
            }
            // Create a plugin instance
            plugin = new FileAuthenticator( name.c_str(), userSource,
                                        transientUserLocName, loginFormName);
            // Configure plugin instance with usermap and attribute vector
            if (plugin != NULL)
            {
                // Extract the XML property list containing allowable
                   attributes
                // Attributes not on this list will be ignored
                const XmlTreeNode *xmlAttributeList =
                    props->getChild(ATTRIBUTE_PROP_LIST_TAG);
                XmlTreeNodeVector xmlAttributes =
                    xmlAttributeList->getChildren(ATTRIBUTE_TAG);
                // Init the vector of attributes this plugin will copy to the
                  User API
                plugin->getAttributesToCopy(xmlAttributes);
                // Init the map of user ID and attributes
                plugin->fetchPasswordData(passwordFile);
            }
        }
        catch (...)
        {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
                ">#<FileAuthenticator: Could not create File Authenticator
                   plugin instance");
            return NULL;
        }
        return plugin;
    }
```

Extract list of attributes to copy to the User API

Read the user entries from the password file

Return file authenticator plugin instance

Authenticating identities

The `FileAuthenticator::factory()` method creates a `FileAuthenticator` instance using the XML properties configured by the Policy Builder. The `UserSource`, synthetic (also known as transient) identity location, and login form are passed to the `FileAuthenticator` constructor.

Once an instance is created, `FileAuthenticator::factory()` calls `getAttributesToCopy()` to create a vector of attribute names that the plugin should copy to the User API. It then calls `fetchPasswordData()` to create a map of identity profiles. The identity name is used for the map key. An identity entry is a map of attributes where the attribute name is the map key. An attribute is a vector of values.

Once a file authenticator instance is created, it is ready to authenticate identities. The `FileAuthenticator::authenticate()` method is not very different from most other authenticators. First, it checks to see if a restart code was issued by the Policy Validator. Next

it determines whether the an identity name and password have been provided. If not, it provides a hint to the Enforcer plugin to prompt for credentials. Lastly, if a identity name and password are provided, it authenticates the identity by matching the provided password with the one stored in the identity map. If it cannot, it adds an authentication hint to the response to trigger a login form.

The following example demonstrates these:

```
AuthPlugin::Result FileAuthenticator::authenticate
(
    PropertyListElement *data,
    PropertyListElement *response,
    const PropertyListElement *personalizer,
    const bool trace
)
{
    Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
        "FileAuthenticator: Invoking FileAuthenticator plugin");
    AuthPlugin::Result result = R_DENY;
    // Determine if we already authenticated the user and Validator sent
       an R_RESTART
    if (isRestart(data, name_.get()))
    {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
            "FileAuthenticator: Authentication restart");
        result =  validateRestartData(data, response, personalizer);
    }
    // Could not find credentials, send hint to Enforcer to prompt for
       uid & password
    else if (!fetchData(user_, ENFORCER_USER,
                        ValidatorGetPostDataList(data)) ||
        !fetchData(password_, ENFORCER_PASSWD,
                   ValidatorGetPostDataList(data)))
    {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
           "FileAuthenticator: No uid or password, sending login form");
        addPasswordHint2Response(response);
    }
    // See if the user name and password match a record in the user map
    else if (validateUser(user_, password_))
    {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
                    "FileAuthenticator: User authenticated");
        result = R_PERMIT;
        if (canCacheUser())
        {
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
                "FileAuthenticator: Caching user record");
            bool success = true;
           // User was authenticated, now see if they have an LDAP entry
```

Check whether a R_RESTART code isissued

No user name or password provided, send a login

Validate the credentials

If no LDAP entry is found, create one

```
            if (!isUserInLdap(user_.c_str()))
            {
                // No LDAP entry, so try to create one
                if (!addNewUserRecord(user_.c_str()))
                    success = false;
            }
            // If entry was created in LDAP, make sure the operation was
               successful
            if (success)
                result = handleUserInfo(data, response, personalizer,
                         user_.c_str(),
                    transientUserLoc_.get());
        }// end if can cache user
    } //  end if user authenticated
    // User name not found or bad password
    else
    {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG,
            "FileAuthenticator: User login failed");
        addInitialForm(response);
    }
    setResponseAction(result, response);
    return result;
}
```

The `authenticate()` method calls `isUserInLdap()` to determine if the identity already has a profile. This `UserCache::sLocateByUID()` method is used to locate the identity in LDAP. It searches each `UserSource` for the identity name.

Searching for existing profiles

Once the identity authenticates, the `authenticate()` method checks to see if the identity already has a profile in the directory. If not, the `authenticate()` method calls the `addNewUserRecord()` method to create a permanent profile for the identity. The `addNewUserRecord()` method uses the User API to add the identity to the directory server.

If the identity already had a directory server profile, or if the `addNewUserRecord()` successfully adds a directory server profile for the identity, the `handleUserInfo()` method is called.

The following code sample demonstrates these actions:

```
bool FileAuthenticator::isUserInLdap(char * user)
{
    // Search for the userId in the LDAP user locations
    UserSourceVector& sources = all_user_sources();
    UserRefPtr userp;
    bool foundUserId = false;
    // Search all the configured user sources for the user LDAP entry
    for ( UserSourceVector::iterator iter = sources.begin(); iter !=
          sources.end();
        iter++ )
```

Check all user sources to see if the user has a directory server entry

```
    {
        try
        {
            // Try to locate the user in the current user source
            userp = UserCache::sLocateByUID(*iter, user);
            if ((userp != NULL) && userp->exists() )
            {
                // User was found, validate that either the user record
                   is not
                // synthetic, or that this plugin created the synthetic
                   entry
                if
((!userp->isSynthetic())||(userp->getUserSource()==syntheticSource))
                {
                    return true;
                }
            }
        }
        catch (EnforcerException &e)
        {
            if ( e.getErrorNum() != EnforcerException::E_LDAP )
            {
                throw e;
            }
        }
    }
    // If we get this far, the user has not been found
    return false;
}
```

Found the user in LDAP

Did not find the user

Creating a new identity profile

As demonstrated by the code sample below, if `authenticate()` determines that an identity does not have a directory server profile, it calls `addNewUserRecord()`, which then calls `UserCache::sCreateTemporary()` to create the directory entry fro the new identity profile. Note that the `CN` attribute is used to identify the entry in the directory. The code below uses the provided identity name as the value for the `CN` attribute. The `CN` attribute may have multiple values, but the value used to identify the directory location is the second parameter in the `sCreateTemporary()` method.

```
bool FileAuthenticator::addNewUserRecord(const char * user)
{
    // Get the user source where the entry will be added
    const UserSource *syntheticSource = UserSource::sGetByDN
                                        (synthetic_loc);
    if ( syntheticSource == NULL )
    {
        Logger::log(VAL_CHAN_AUTH, ENFORCER_LOG_ERROR,
            ">#<Unable to find synthetic location");
        return false;
```

Get the Synthetic user source

```
    }
    Logger::log(VAL_CHAN_AUTH, ENFORCER_LOG_DEBUG, "Found synthetic
                location");
    // Get the user attributes from the attribute map
    UserCacheAttributes attributes;
    getAttributes(user, attributes);
    // Create the user entry
    userp = UserCache::sCreateTemporary(syntheticSource, user,
                                        synthetic_loc, attributes);
    // See if the entry was created
    if ( userp == NULL )
    {
        Logger::log(VAL_CHAN_AUTH, ENFORCER_LOG_ERROR,
            ">#<Failed to create temporary user");
        return false;
    }
    Logger::log(VAL_CHAN_AUTH, ENFORCER_LOG_DEBUG, "Created temporary
                user %s",
        userp->getDN());
    return true;
}
```

Add a new entry on the directory server

Creating attributes

When the file authenticator needs to create an entry in LDAP, it calls `getAttributes()` to obtain the `UserCacheAttributes` to copy to LDAP. The `getAttributes()` method, demonstrated in the code sample below, looks up the identity in the identity map created in the `factory()` method. This method uses the vector of allowable attributes to determine which identity attributes should be added to the `userCacheAttributes`. Only the configured attributes will be copied; other identity attributes are ignored.

The `FileAuthenticator::getAttributes()` method is currently case-sensitive for attribute names. If an attribute in the password file is specified using a different case then what is configured in the Policy Builder, the file authenticator will ignore the attribute.

```
void FileAuthenticator::getAttributes(const char * user,
UserCacheAttributes & attr)
{
    // Create the key to lookup the user record
    string theUser = "uid=" + string(user);
    attributeMap userRecord;
    userMap::iterator aUser = users_.find(theUser);
    // See if the user has a record, if not, user name is unknown.
    if (aUser == users_.end())
    {
        Logger::log(VAL_CHAN_AUTH, ENFORCER_LOG_DEBUG,
            "Can not locate user record in login map.");
        return;
    }
```

Lookup identity data read from password file

```
        // If found, get the user attributes
        userRecord = aUser->second;
        strVector attributeValues;
        // Only look for attributes that are configured to copy
        for (strVector::iterator iter = attributes_.begin(); iter !=
             attributes_.end();
            iter ++)
        {
            string key = (*iter);
            // See if the user has the attribute
            attributeMap::iterator attribute = userRecord.find(key);
            if (attribute != userRecord.end())
            {
                // Get the attribute values
                attributeValues = attribute->second;
                // Add each attribute value
                for (strVector::iterator values = attributeValues.begin();
                    values != attributeValues.end(); values ++)
                {
                    string value = (*values);
                    Logger::log(VAL_CHAN_AUTH, ENFORCER_LOG_DEBUG,
                        "FileAuthenticator::getAttributes() Copying %s=%s",
                        key.c_str(), value.c_str());
                    attr.add(key.c_str(), value.c_str());
                }
            }
        }
    }
```

Only add attributes that are configured

See if the identity has the attribute

If multi-valued attribute, add each value

9 Administration API

The Administration API provides a programming interface for the Select Access Administration server that can be used to define resources, set policies, and modify user passwords. This chapter describes the functions of the Administration API.

Chapter Overview

Topics in this chapter describe the Administration server, the API, and how to use the API:

- Understanding the Administration Server on page 175
- Understanding the Administration API on page 176
- Getting Started on page 177
- Data Type Overview on page 177
- Common Data Types on page 178
- Resource-specific APIs on page 184
- Policy-specific APIs on page 191
- Helper APIs on page 207
- Modifying Passwords on page 207

Understanding the Administration Server

The Administration server handles SSL details and configuration information. As the configuration engine for Select Access components, the Administration server coordinates all setup details by:

- Collecting common parameters
- Handling requests sent by the Setup Tool to read and write component configuration information to and from the Policy Store
- Defining the Select Access common parameters that all components inherit
- Managing setup parameters among different Select Access components

The setup of the Administration server writes most parameters to a local XML file. These parameters are bootstrap parameters. The Administration server requires these parameters at startup, which is why it writes them to this local file.

Understanding the Administration API

The Administration API is an environment based on a published WSDL for developers working in languages with web service support for externally defining resources and setting policies.

The Administration API is mainly tested in the Axis framework.

The API can be used to do the following tasks:

- Add, modify or delete service and resource entries
- Add, modify or delete policy settings for users and resources using Allow, Deny or existing rules
- Allow users to modify their passwords

The Administration API is implemented via web services and is protected by the Axis Enforcer. The Setup tool automatically configures the Administration API environment and the Axis Enforcer.

Calls to the web service operations are intercepted by the Axis Enforcer and forwarded to the web service if an Allow is returned from the Validator. Any calls via the web service to perform an operation are examined by the Administration server and allowed only if the appropriate permissions are in place.

Exception Handling

There are three kinds of exceptions produced by the Administration API:

- **System errors**, e.g. network error, invalid SOAP messages, etc. These are exceptions that are not caused by the application. Usually, the web service framework will return an error or exception, for example, Axis returns a `soap:fault` to the client program.
- **Unknown application exceptions**, e.g. NullPointerExceptions. When such an exception occurs, a message is logged to the audit system (web service channel, `ERROR` level), and a `java.io.RemoteException` is thrown. Axis returns a `soap:fault` to the client.
- **Known exceptions**, including client input validation errors, and Administration server returned errors (i.e. entry already exists, policy signature error, etc.). These exceptions are returned as `wsdl:fault` to the client. Each `wsdl:fault` will have an error code and a brief error message indicating the cause of the error.

Logging

The web service layer does not log any user activities. The Administration server logs these with the login user DN. You can track all the web service access by setting up a specific user who has permission to access the service. Only unknown application errors are logged.

If you are using Java to write client code, you can import the SA certificate into the JRE keystore. The SA certificate can be obtained at `<SA install folder>\Select Access\shared\jetty\etc\certs\mcacert.cer`.

Getting Started

The Administration API WSDL is available at following URL:

`https://host:port/axis/services/wsadmin?wsdl`

where *<host>* represents the Administration server name, and *<port>* is the Administration API port. The default port is 9993.

If the input parameters to ANY Administration API call are invalid, an exception is thrown with the message, "`Invalid argument`".

Data Type Overview

The following table acts as a summary of the data types used in the Administration API.

Table 8 Administration API Data Types

Data Type	Description	Related Data Types
`resourcePath`	Specifies the path for the resource in the Policy Builder	`Network Resources Path` `Administrative Resources Path`
`Network Resources Path`	Corresponds to the Resource Access tree in the Policy Builder	`resourcePath` `Administrative Resources Path`
`Administrative Resources Path`	Corresponds to the Administrative Access tree in the Policy Builder	`resourcePath` `Network Resources Path`
`LdapSearchCriteria`	Used to search for a certain range of resources and identities	`LdapSearchCriteria Filter` `resourcePath`
`LdapSearchCriteria Filter`	Used for fine-grain searches	`LdapSearchCriteria`
`AdminFault`	Contains the error message returned from the Administration API call	`setUserPassword`

`getResource`	Retrieves the detail of a resource	`resourcePath`
		`ResourceServer`
		`setResource`
		`searchResources`

`ResourceServer`	Corresponds to a server in Resource server in the Policy Builder	`resourcePath`
		`getResource`
		`setResource`
		`searchResources`

Table 8 Administration API Data Types (cont'd)

Data Type	Description	Related Data Types

`setResource`	Modifies a resource	`resourcePath`
		`getResource`
		`ResourceServer`
		`searchResources`

`searchResources`	Searches for a list of resources	`LdapSearchCriteria`
`getPolicies`	Returns a portion of the Policy Grid	`setPolicy` `LdapSearchCriteria`
`setPolicy`	Sets the policy to a specific resource	`getPolicy`
`refreshValidators`	Clears the Validator cache.	
`getAuthServiceNames`	Returns the names of all authentication services defined through the Policy Builder	
`getRuleNames`	Returns a list of rule names	
`setUserPassword`	Changes user password	`AdminFault`

Common Data Types

resourcePath

A resource path is a string and referenced by a slash-separated ("/") path, for example:

`/network/http/<localhost>/index.html`

Valid resource paths begin with `/network`. If you set a resource path with a forward slash at the end, for example, `/network/http/<localhost>/`, the resource path is treated as though there is no forward slash at the end: `/network/http/<localhost>`.

Resource paths are case-insensitive. Leading and trailing whitespace in a path string will be ignored. The following examples demonstrate how whitespace is treated. In the examples, "_" represents whitespace.

Leading or trailing whitespace. Paths with leading or trailing whitespace are considered as equal and are valid. For example:

`_/network/http/<localhost>/index.html`

`/network/http/<localhost>/index.html`

Whitespace defined through the Policy Builder. If you define a resource with whitespace in the Policy Builder, for example:

`/network/h_ttp/Apa_che/index_._html`

both of the following will be valid when using it in Administration API:

```
/network/h_ttp/Apa_che/index_._html
```

```
_/network/h_ttp/Apa_che/index_._html
```

using whitespace within a component will return an error.

```
/network_/h_ttp/Apa_che/index_._html
```

Network Resources Paths

A Network Resources path corresponds to the **Resource Access** tree in the Policy Builder.

The Resource tree contains **Resource Access** and **Administrative Access** branches. Both are localizable.

Network Resources paths start with "`/network`" as the root component. Resource names are separated by "/". Below is a resource in Policy Builder and its path:

```
/network/http/<localhost>/index.html
```

A Network Access Resource path is shown below:

```
"/network/<folder-name>/<resource-server-name>/<resource-name>/
<sub-resource-name>"
```

Administrative Resources Paths

Administrative Resources paths correspond to the **Administrative Access** tree in the Policy Builder. Administrative Resource paths start with "`/admin`" as the root component.

The Resource tree contains **Resource Access** and **Administrative Access** branches. Both are localizable.

An Administrative Resources path has following subordinates:

- Network Management Resource paths
- Schema Management paths
- Function Management paths
- Identity Management paths

Network Management Resource paths

A Network Management Resource path with "`/admin/network`" and will look like the following:

```
"/admin/network/<folder-name>/<resource-server-name>/<resource-name>/
<sub-resource-name>"
```

The character "/" cannot be in the resource name. "/" is always treated as a resource path symbol.

Schema Management paths

Attribute schema entries are grouped by object classes in the Policy Builder but the Schema Management path is not the same as it appears in the Policy Builder. Schema Management paths start with "`/admin/schema`".

To query an attribute schema entry, the path should start with "`/admin/schema/attribute`":

"`/admin/schema/attribute/<attribute-name>`"

To query an object class schema entry, the path should start with "`/admin/schema/object_class`". The following is a valid object class management path:

"`/admin/schema/object_class/<object-class-name>`"

Function Management paths

A Function Management path starts with "`/admin/functions`". A valid function management path is shown below:

"`/admin/functions/<function-name>`"

The *<function-name>* is not the localized name that appears in the Policy Builder. Each function has a predefined name. A list of all the currently defined function names is given below:

`authentication_service_configuration`

`component_configuration`

`identity_editor_plugin_configuration`

`identity_location_configuration`

`network_discovery`

`password_policy_configuration`

`password_reset_configuration`

`policy_data_signing`

`report_viewer`

`rule_builder`

`sub-delegation_ability`

`workflow_alert_template_configuration`

`workflow_configuration`

Identity Management paths

Identity Management paths all start with "`/admin/identity/`", followed by the real identity DN, for example:

"`/admin/identity/select_auth`"

"`/admin/identity/unknown_identities`"

"`/admin/identity/known_identities`"

"`/admin/identity/known_identities/cn=foobar,ou=HR,dc=hp,dc=com`"

identityDN

An `identityDN` is a string and referenced by the original directory entry Distinguished Name, for example:

```
cn=foobar,ou=HR,dc=hp,dc=com
```

Special identity entries in the **Identities Tree** such as SelectAuth, UnknownIdentities and KnownIdentities are referenced as below:

`select_auth`

`unknown_identities`

`known_identities`

LdapSearchCriteria

Use `LdapSearchCriteria` to search for a range of resources and identities. `LdapSearchCriteria` is defined as:

```
<complexType name="LdapSearchCriteria">
     <sequence>
       <element name="threshold" type="int" minOccurs="0"/>
       <element name="pageSize" type="int"/>
       <element name="continuedID" minOccurs="0" type="string"/>
       <element name="baseID" type="string"/>
       <element name="scope" minOccurs="0">
         <simpleType>
             <annotation>
               <documentation>This data type is to set the scope of
               LDAP search. BASE is to search the entry only, ONE is
               to search one level down, SUB is to search all
               subordinates.</documentation>
             </annotation>
             <restriction base="string">
                 <enumeration value="BASE"/>
                 <enumeration value="ONE"/>
                 <enumeration value="SUB"/>
             </restriction>
         </simpleType>
       </element>
       <element name="filter" nullable="true" minOccurs="0"
       type="impl:LdapSearchCriteriaFilter" />
     </sequence>
</complexType>
```

LdapSearchCriteria parameters

The following table lists the `LdapSearchCriteria` parmeters:

Table 9 LdapSearchCriteria Parameters

Parameter	Description	Required	Value
`threshold`	`Threshold` controls the maximum number of entries that the LDAP search may return. If it's `0` (zero), the maximum number of entries is dependent with the size limit of LDAP server.	No	Default value is `0`. Note: Setting the threshold with too small a value may result SIZE_LIMIT_EXCEEDED exception.
`pageSize`	Use `pageSize` to define the maximum number of records the Administration API operation may return. There's no limit if `pageSize` is `0` (zero).	No	Default value is `0`. This means to return as many as the server can support.
`continuedID`	Use `continuedID` to retrieve part of the returned records. Setting `continuedID` to null means this is the first page.	No	The value of `continuedID` must be `resourcePath` if it's used in the Resources Tree, and must be `identityDN` if it's used in the Identities Tree.
`baseID`	The root from where the search starts.	Yes	The value of `baseID` must be `resourcePath` if it's used in the Resources Tree, and must be `identityDN` if it's used in the Identities tree.
`scope`	Specifies that the scope of a search.	No	The valid values are `BASE`, `ONE` and `SUB`. The default value is `SUB`. • `BASE` means the search includes only the `baseID`. • `ONE` means the search includes only the entries one level below the `baseID`. • `SUB` means the search includes the `baseID` and all entries at all levels beneath `baseID`.
`filter`	Search filter specifying search criteria.	No	Default value is the default value of `LdapSearchCriteriaFilter`. Refer to LdapSearchCriteriaFilter on page 183 for more information.

To support a paging navigation, the client can specify the `continuedID` and `pageSize`. The flag `isContinued` in the last returned entry indicates if there are more entries to return. The client uses this value in the next search to fetch the next page. The first record on the next page is the one next to `continuedID`, or an empty page if `continuedID` is the last record. The order is dependent upon Select Access's internal sorting algorithm.

Setting `continuedID` to null means this is the first page. Setting `pageSize` to `0` means to return as many as the server can support.

LdapSearchCriteriaFilter

`LdapSearchCriteriaFilter` is used for fine-grain searches. `LdapSearchCriteriaFilter` is defined as:

```
<complexType name="LdapSearchCriteriaFilter">
     <simpleContent>
       <extension base="string">
         <attribute name="type" use="optional">
           <simpleType>
                <restriction base="string" >
                     <enumeration value="LDAP" />
                     <enumeration value="NAME" />
                </restriction>
           </simpleType>
         </attribute>
       </extension>
     </simpleContent>
</complexType>
```

LdapSearchCriteriaFilter parameters

The following table lists the `LdapSearchCriteriaFilter` parameters:

Table 10 LdapSearchCriteriaFilter Parameters

Parameter	Description	Required	Value
`type`	This attribute defines the search is either an LDAP-based search or a NAME-based search. If it's an LDAP type search, the filter value is compliant with RFC2254 (*The String Representation of LDAP Search Filters*). If it is a NAME type, the filter value is a string, and a wildcard (*) can be used.	No	Default type is `LDAP`.

AdminFault

`AdminFault` contains the error message returned from the Administration API call. Its definition is given below:

```
<complexType name="AdminFault">
 <sequence>
   <element name="errorCode" type="int"/>
   <element name="errorMessage" nullable="true" type="string"/>
   <element name="errorDetailCode" type="int"/>
   <element name="errorDetailMessage" nullable="true" type="string"/>
   <element name="errorObjects" nullable="true" type="string"
   minOccurs="0" maxOccurs="unbounded"/>
 </sequence>
</complexType>
```

Resource-specific APIs

Resource-specific API operations apply to network access resources only.

getResource

Use `getResource` to retrieve the detail of a resource.

Input

`getResource` takes `resourcePath` as input. Only Network Resources paths and Network Management Resource paths can be used as a `resourcePath`. (Refer to Network Resources Paths on page 179 and Network Management Resource paths on page 179 for more information.)

Output

`getResource` returns an instance of Resource, if there is no error. A Resource data type is defined as:

```
<complexType name="Resource">
<sequence>
     <element name="resourcePath" type="string"/>
     <element name="type">
       <simpleType>
        <annotation>
           <documentation>Resource type can be one of the Folder,
           Service, and Resource types.</documentation>
        </annotation>
        <restriction base="string">
               <enumeration value="Folder" />
               <enumeration value="Service" />
```

```
                <enumeration value="Resource" />
            </restriction>
          </simpleType>
        </element>
        <element name="charSet" nullable="true" type="string"/>
        <element name="server" nullable="true" type="impl:ResourceServer"
        minOccurs="0" maxOccurs="unbounded"/>
        <element name="continued" type="boolean"/>
</sequence>
</complexType>
```

getResource parameters

The following table lists the `getResource` parmeters:

Table 11 getResource Parameters

Parameter	Description	Required	Value
`resourcePath`	The `resourcePath` of the returned resource.	Yes	A string with the format of `resourcePath` described above
`type`	The resource type.	Yes	Can be `Folder`, `Service` or `Resource`
`charSet`	Only valid when the resource type is `Service`. It specifies the character set used by this service.	No	
`server`	Only valid when the resource type is `Service`. It's an instance of the `ResourceServer` data type.	Yes if the resource type is `Service`.	See ResourceServer on page 185
`continued`	Only useful when searching resources with `pageSize` defined in `LdapSearchCriteria`. It indicates whether there are more entries to return. If it is true, the `resourcePath` of this resource can be used as the `continuedID` in `LdapSearchCriteria` to read the next page.	No	Default value is `false`

ResourceServer

`ResourceServer` corresponds to a server in **Resource Server** in the Policy Builder. Its definition is:

```
<complexType name="ResourceServer">
 <sequence>
```

```
    <element name="url" nullable="true" type="string"/>
    <element name="representative" type="boolean"/>
   </sequence>
  </complexType>
```

ResourceServer parameters

The following table lists the `ResourceServer` parmeters:

Table 12 ResourceServer Parameters

Parameter	Description	Required	Value
`url`	The URL of this resource server.	Yes	Can be null
`representative`	True if this server is the representative server. When you run resource discovery on this resource server, only the representative server is scanned. Refer to the *HP OpenView Select Access 6.2 Policy Builder Guide* for more information.	Yes	`True` or `false`

Sample program

All sample programs are Java programs based on the Axis framework.

```
    public void getResourceSample() throws Exception {
        // WsadminSoapBindingStub is generated from the WSDL by Axis's
        wsdl2java tool

com.hp.ov.selectaccess.adminserver.wsadmin.client.WsadminSoapBindingStub
adminApi;
        // WsadminServiceLocater is generated from the WSDL by Axis's
        wsdl2java tool
        try {
            adminApi = (com.hp.ov.selectaccess.adminserver.wsadmin.
            client.WsadminSoapBindingStub)
                            new
com.hp.ov.selectaccess.adminserver.wsadmin.client.WsadminServiceLocator
().getwsadmin();
        }
        catch (javax.xml.rpc.ServiceException jre) {
            if(jre.getLinkedCause()!=null)
                jre.getLinkedCause().printStackTrace();
        }
        // Time out after a minute
        adminApi.setTimeout(60000);
```

```
            // set the credentials of Admin API call
            // The user id used in the call must be delegated through Policy
            Builder and has necessary permissions
            adminApi.setUsername("adminApi_user_id");
            adminApi.setPassword("adminApi_user_password");
            // Resource is generated from the WSDL by wsdl2java
            // See above for the definition of Resource
            com.hp.ov.selectaccess.adminserver.wsadmin.client.Resource
            value = null;
            try {
                value = adminApi.getResource("/network/http");
                String charset = value.getCharSet();
                String path = value.getResourcePath();
                // ResourceType is created from WSDL by wsdl2java
                ResourceType type = value.getType();
                if (type.equals(ResourceType.Service)) {
                ResourceServer[] servers = value.getServer();
                for (int i = 0; i < servers.length; i++) {
                  ResourceServer server = servers[i];
                  String url = server.getUrl();
                  boolean isRepresentative = server.isRepresentative();
                }
                }
            }  catch
    (com.hp.ov.selectaccess.adminserver.wsadmin.client.AdminFault e1) {
               System.err.println("Error code: " + e1.getErrorCode());
               System.err.println("Error message: " + e1.getErrorMessage());
            }
        }
```

setResource

Use `setResource` to modify a resource.

Input

`setResource` takes two input fields. It has different effects, based on the values of the input. The input definition is:

```
<complexType>
 <sequence>
  <element name="oldResource" nullable="true" type="impl:Resource"/>
  <element name="newResource" nullable="true" type="impl:Resource"/>
 </sequence>
</complexType>
```

setResource input parameters

The following table lists the `setResource` input parmeters:

Table 13 setResource Input Parameters

Parameter	Description	Required	Value
`oldResource`	A resource instance represents a resource to be modified.	Yes	Can be null
`newResource`	A resource instance represents the modified resource.	Yes	Can be null

setResource behavior

The following table describes the `setResource` behavior:

Table 14 setResource Behavior

oldResource value	newResource value	Description
Null	Null	Throw exception with invalid argument message
Null	Not null and valid	Create a new resource
Not null and exists	Null	Delete `oldResource`
Not null and exists	Not null and valid	Modify `oldResource` to `newResource`

Only network resources can be passed to input. `newResource` must have valid ancestor. For example, to define `/network/node/http` as a `newResource`, `/network/node` must already exist.

Output

`setResource` returns a string. The returned string is defined as:

```
<simpleType name="ReturnString">
  <restriction base="string">
     <enumeration value="SUCCESS" />
     <enumeration value="FAILURE" />
     <enumeration value="PENDING_WORKFLOW" />
  </restriction>
</simpleType>
```

setResource output parameters

The following table lists the **setResource** output parmeters:

Table 15 setResource Output Parameters

Parameter	Description	Required	Value
SUCCESS	setResource executed successfully.	n/a	n/a
FAILURE	Operation failed.	n/a	n/a
PENDING_WORKFLOW	Request has been submitted and is waiting for approval.	n/a	n/a

Sample program

Below is a sample program for creating a new resource.

```
protected void setResourceSample() {
    // WsadminSoapBindingStub is created by wsdl2java
    com.hp.ov.selectaccess.adminserver.wsadmin.client.WsadminSoap
    BindingStub adminApi;
   // WsadminServiceLocator is created by wsdl2java
   try {
        adminApi = (com.hp.ov.selectaccess.adminserver.wsadmin.
        client.WsadminSoapBindingStub)
                new com.hp.ov.selectaccess.adminserver.wsadmin.
                client.WsadminServiceLocator().getwsadmin();
    }   catch (javax.xml.rpc.ServiceException jre) {
        if(jre.getLinkedCause()!=null)
            jre.getLinkedCause().printStackTrace();
    }
    // Time out after a minute
    adminApi.setTimeout(60000);

    adminApi.setUsername(username);
    adminApi.setPassword(password);
    try {
        // ReturnString is generated by wsdl2java
        ReturnString value = null;
        Resource resource = new Resource();
        resource.setCharSet("UTF-8");
        resource.setResourcePath("/network/http");
        resource.setType(ResourceType.Service);
        ResourceServer resourceServer = new ResourceServer();
        resourceServer.setRepresentative(true);
        resourceServer.setUrl("http://localhost:80");
        resource.setServer(new ResourceServer[] { resourceServer });
        value = adminApi.setResource(null, resource);
    }  catch (com.hp.ov.selectaccess.adminserver.wsadmin.
       client.AdminFault e1) {
        System.err.println("Error: " + e1.getErrorMessage());
```

```
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
```

searchResources

Use `searchResources` to search for a list of resources.

Input

`searchResources` takes `LdapSearchCriteria` as its input. Refer to LdapSearchCriteria on page 181 for more information.

Output

`searchResources` returns a list of Resources. Refer to Resource-specific APIs on page 184 for more information about Resource APIs.

Sample program

```
    public void searchResourcesSample() throws Exception {
        // WsadminSoapBindingStub is generated by wsdl2java

com.hp.ov.selectaccess.adminserver.wsadmin.client.WsadminSoapBindingStub
adminApi;
        try {
            adminApi = (com.hp.ov.selectaccess.adminserver.wsadmin.
            client.WsadminSoapBindingStub)
                  new com.hp.ov.selectaccess.adminserver.wsadmin.client.
                  WsadminServiceLocator().getwsadmin();
        }   catch (javax.xml.rpc.ServiceException jre) {
            if(jre.getLinkedCause()!=null)
                jre.getLinkedCause().printStackTrace();
        }
        // Time out after a minute
        adminApi.setTimeout(60000);
        adminApi.setUsername(username);
        adminApi.setPassword(password);
        com.hp.ov.selectaccess.adminserver.wsadmin.client.Resource[]
        value = null;
        try {
            // LdapSearchCriteria is generated by wsdl2java
            LdapSearchCriteria filter = new LdapSearchCriteria();
            filter.setThreshold(new Integer(3));
            filter.setBaseID("/network");
            // LdapSearchCriteriaScope is generated by wsdl2java
            filter.setScope(LdapSearchCriteriaScope.SUB);
            value = adminApi.searchResources(filter);
        } catch (Exception e1) {
            e1.printStackTrace();
        }
```

```
        try {
            LdapSearchCriteria filter = new LdapSearchCriteria();
            filter.setBaseID(“/network”);
            filter.setScope(LdapSearchCriteriaScope.BASE);
            value = adminApi.searchResources(filter);
        }  catch (Exception e1) {
             e1.printStackTrace();
        }
        try {
            // example of name search
           LdapSearchCriteria searchCriteria = new LdapSearchCriteria();
            searchCriteria.setBaseID("/network");
         searchCriteria.setScope(LdapSearchCriteriaScope.SUB);
            // LdapSearchCriteriaFilter is generated by wsdl2java
            LdapSearchCriteriaFilter filter = new
            LdapSearchCriteriaFilter();
            // LdapSearchCriteriaFilterType is generated by wsdl2java
            filter.setType(LdapSearchCriteriaFilterType.NAME);
            // use wildcard
            filter.set_value("*");
            searchCriteria.setFilter(filter);
            value = adminApi.searchResources(searchCriteria);
        }  catch (Exception e1) {
            e1.printStackTrace();
        }
    }
```

Policy-specific APIs

getPolicies

`getPolicies` returns a portion of the Policy Grid.

Input

`getPolicies` takes two input fields: the first defines the scope of resources, and the second defines the scope of identities. The definition of the input is as below:

```
<complexType>
 <sequence>
  <element name="resourceCriteria" type="impl:LdapSearchCriteria"/>
  <element name="identityCriteria" type="impl:LdapSearchCriteria"/>
 </sequence>
</complexType>
```

Both `resourceCriteria` and `identityCriteria` are instances of `LdapSearchCriteria`. Paging can be used.

Output

`getPolicies` returns a list of `ResourceRow`. If no resource or identity is found in the search, `AdminFault` returns an error code and error message.

The definition of `ResourceRow` is:

```
<complexType name="ResourceRow">
 <sequence>
  <element name="identityColumn" type="impl:IdentityColumn"
  minOccurs="0" maxOccurs="unbounded"/>
  <element name="continued" type="boolean"/>
 </sequence>
 <attribute name="resourcePath" type="string" />
</complexType>
```

getPolicies parameters

The following table lists the `getPolicies` parmeters:

Table 16 getPolicies Parameters

Parameter	Description	Required	Value
`identityColumn`	A list of `IdentityColumns`. Refer to IdentityColumn on page 192 for the description of `IdentityColumn`. Each `IdentityColumn` is a Policy Grid.	No	List of `IdentityColumn`. Refer to IdentityColumn on page 192 for more information.
`continued`	Indicates if there's more entry to return. Only useful when paging was set in the `resourceCriteria`.	Yes	Default is `false`.
`resourcePath`	Refer to resourcePath on page 178 for more information.	Yes	The `resourcePath` can be used in paging navigation.

IdentityColumn

Each of the returned `ResourceRows` contain a list of `IdentityColumn`. Each `IdentityColumn` contains the `identityDN` and the policy of this identity.

`IdentityColumn` is define as:

```
<complexType name="IdentityColumn">
 <sequence>
  <element name="policy" type="impl:Policy"/>
  <element name="continued" type="boolean"/>
 </sequence>
 <attribute name="identityDN" type="string" />
</complexType>
```

IdentityColumn parameters

The following table lists the `IdentityColumn` parmeters:

Table 17 IdentityColumn Parameters

Parameter	Description	Required	Value
`policy`	The policy of this grid.	Yes	Refer to IdentityColumn on page 192 for more information
`continued`	Indicates if there are more entries to return. It's only useful when paging was set in the `resourceCriteria`.	Yes	Default is `false`
`identityDN`	Refer to identityDN on page 181 for more information.	Yes	Refer to Policy on page 193 for more information

Policy

The base definition of `policy` is:

```
<complexType name="Policy" abstract="true">
  <sequence>
    <element name="state" type="string"/>
    <element name="inherited" type="boolean"/>
  </sequence>
</complexType>
```

Policy parameters

The following table lists the `policy` parmeters:

Table 18 Policy Parameters

Parameter	Description	Required	Value
`state`	The policy state.	Yes	Depends on the subtypes
`inherited`	Indicates whether the policy is inherited.	Yes	`True` means the policy is inherited (not explicitly defined)

`Policy` is an abstract type, thus cannot be passed to input directly. There are four subtypes of policy.

- `AccessPolicy`
- `WorkflowPolicy`
- `AdminPolicy`
- `SelectAuthPolicy`

AccessPolicy

Network resources have an access policy. The `AccessPolicy` is defined as:

```
<complexType name="AccessPolicy">
  <annotation>
    <documentation>When setting AccessPolicy state, AccessPolicyState
    should be used.</documentation>
  </annotation>
  <complexContent>
   <extension base="impl:Policy">
     <sequence>
       <element name="rule" nullable="true" type="impl:Rule"
       minOccurs="0" maxOccurs="unbounded"/>
     </sequence>
   </extension>
 </complexContent>
</complexType>
```

AccessPolicy parameters

The following table lists the `AccessPolicy` parmeters:

Table 19 AccessPolicy Parameters

Parameter	Description	Required	Value
`state`	The policy state. It's inherited from `policy`.	Yes	The state is one of `ALLOW`, `DENY` and `RULE`. `RULE` means the resource is enforced by conditional rules
`inherited`	Indicates whether the policy is inherited.	Yes	`True` means the policy not explicitly defined
`rule`	A list of rules.	No	If the state is `RULE`, it has the information of the conditional rules

Rule

A `rule` is defined as:

```
<complexType name="Rule">
 <sequence>
  <element name="state">
     <simpleType>
         <annotation>
            <documentation>A rule state can be one of VALID, INVALID,
            and MISSING</documentation>
         </annotation>
         <restriction base="string">
             <enumeration value="VALID" />
             <enumeration value="INVALID" />
```

```
            <enumeration value="MISSING"/>
          </restriction>
        </simpleType>
      </element>
      <element name="name" type="string"/>
    </sequence>
  </complexType>
```

Rule parameters

The following table lists the `rule` parmeters:

Table 20 Rule Parameters

Parameter	Description	Required	Value
`state`	The state of the rule.	Yes	One of `VALID`, `INVALID` or `MISSING`. • `VALID` means the rule is taking effect • `INVALID` means the rule is malformed • `MISSING` means there isn't a rule with the specified name
`name`	Rule name.	Yes	Rule name

WorkflowPolicy

`WorkflowPolicy` determines how a workflow applies to administrative resources. It is defined as:

```
<complexType name="WorkflowPolicy">
  <annotation>
     <documentation>When setting WorkflowPolicy state,
     WorkflowPolicyState should be used.</documentation>
  </annotation>
  <complexContent>
   <extension base="impl:Policy">
     <sequence>
       <element name="rule" nullable="true" type="impl:Rule"
       minOccurs="0" maxOccurs="unbounded"/>
     </sequence>
   </extension>
 </complexContent>
</complexType>
```

WorkflowPolicy parameters

The following table lists the `WorkflowPolicy` parmeters:

Table 21 WorkflowPolicy Parameters

Parameter	Description	Required	Value
`state`	The state of the workflow policy.	Yes	Either `ENABLED` or `DISABLED`
`inherited`	Indicates whether the policy is inherited.	Yes	`True` if the workflow policy was not explicitly defined
`rule`	A list of workflow rules that are taking effect. Refer to Rule on page 194 for more information.	Yes	Can be null

AdminPolicy

`AdminPolicy` is the policy used in administrative resources. Its definition is:

```
<complexType name="AdminPolicy">
  <annotation>
     <documentation>When setting AdminPolicy state, AdminPolicyState
     should be used.</documentation>
  </annotation>
  <complexContent>
   <extension base="impl:Policy">
     <sequence>
       <element name="workflowPolicy" nullable="true"
       type="impl:WorkflowPolicy"/>
     </sequence>
   </extension>
 </complexContent>
</complexType>
```

AdminPolicy parameters

The following table lists the `AdminPolicy` parmeters:

Table 22 AdminPolicy Parameters

Parameter	Description	Required	Value
`state`	The state of the Admin policy.	Yes	The possible values are `FULL`, `READ_ONLY`, `POLICY_ONLY`, `ADMIN_ONLY` and `NONE`. The valid value also depends on to what resource the policy is applying: • For identity administrative resources, only `FULL`, `POLICY_ONLY`, `ADMIN_ONLY` and `NONE` are valid. • For schema resources, only `FULL` and `READ_ONLY` are valid. • For function and network administrative resources, only `FULL` and `NONE` can be used.
`inherited`	Indicates whether the policy is inherited.	Yes	`True` if the Admin policy was not explicitly defined.
`workflowPolicy`	Define the workflow policy on the grid.	Yes	May be null. Refer to WorkflowPolicy on page 195 for more information.

SelectAuthPolicy

`SelectAuthPolicy` is used to define the authentication service on a specific resource. `SelectAuthPolicy` is defined as:

```
<complexType name="SelectAuthPolicy">
  <annotation>
     <documentation>When setting SelectAuthPolicy state,
     SelectAuthPolicyState should be used.</documentation>
  </annotation>
  <complexContent>
   <extension base="impl:Policy">
     <sequence>
       <element name="property" type="impl:SelectAuthPropertyType"
       minOccurs="0" maxOccurs="1" nullable="true"/>
     </sequence>
   </extension>
 </complexContent>
</complexType>
```

SelectAuthPolicy parameters

The following table lists the `SelectAuthPolicy` parmeters:

Table 23 SelectAuthPolicy Parameters

Parameter	Description	Required	Value
`state`	The state of the SeletAuth policy.	Yes	Either `ENABLED` or `DISABLED`
`inherited`	Indicates whether the policy is inherited.	Yes	`True` if the SelectAuth policy was not explicitly defined
`property`	The authentication service of the resource.	No	Refer to SelectAuthPropertyType on page 198 for more information

SelectAuthPropertyType

`SelectAuthPropertyType` is an XML schema definition of Select Access rule and authentication components. The document element is `COMPONENT` which is defined as:

```
<complexType name="ComponentType">
   <annotation>
     <documentation>The XML schema of Select Access rule and
     authentication component</documentation>
   </annotation>
   <sequence>
       <element ref="impl:PROPERTYLIST" minOccurs="1" maxOccurs="1"/>
   </sequence>
   <attribute name="TYPE" type="string" use="required" />
   <attribute name="CONDITION" type="string" use="required"/>
   <attribute name="NAME" type="string" use="required"/>
   <attribute name="DESCRIPTION" type="string" use="required"/>
   <attribute name="EVALUATOR" type="string" use="required"/>
   <attribute name="CONFIGURATOR" type="string" use="required"/>
</complexType>
```

SelectAuthPropertyType parameters

The following table lists the `SelectAuthPropertyType` parmeters:

Table 24 SelectAuthPropertyType Parameters

Parameter	Description	Required	Value
`PROPERTYLIST`	List of property.	Yes	Refer to PROPERTYLIST on page 200 for more information
`TYPE`	The type of component.	Yes	Must be `decision` if it's used in SelectAuthPolicy
`CONDITION`	Condition this component works in.	Yes	Must be `any` if it's used in SelectAuthPolicy

Table 24 SelectAuthPropertyType Parameters (cont'd)

Parameter	Description	Required	Value
NAME	The name of the component.	Yes	Must be authentication if it's used in SelectAuthPolicy
DESCRIPTION	Description of the component.	Yes	Must be Authenticate Users if used in SelectAuthPolicy
EVALUATOR	The evaluator of the component.	Yes	Must be authentication if it's used in SelectAuthPolicy
CONFIGURATOR	The Java class (a GUI screen) used to configure the component.	Yes	Must be com.hp.ov.selectaccess.rulebuilder.screens.AuthPropertiesDlg if it's used in SelectAuthPolicy

COMPONENT

Each COMPONENT has one, and only one, PROPERTYLIST. The name of this PROPERTYLIST must be authentication, The PROPERTY inside this PROPERTYLIST must be valid authentication service. PROPERTYLIST is defined as:

```
<complexType name="PropertyListType" mixed="false">
   <annotation>
     <documentation>PropertyList may have Property and PropertyList
     children</documentation>
   </annotation>
   <sequence>
       <element ref="impl:PROPERTYLIST" minOccurs="0"
       maxOccurs="unbounded"/>
       <element ref="impl:PROPERTY" minOccurs="0" maxOccurs="unbounded"
       />
   </sequence>
   <attribute use="required" name="NAME" type="string" />
</complexType>
```

COMPONENT parameters

The following table lists the COMPONENT parmeters:

Table 25 COMPONENT Parameters

Parameter	Description	Required	Value
PROPERTYLIST	Each PROPERTYLIST can contain one or more PROPERTYLIST.	No	n/a
PROPERTY	Refer to PROPERTY on page 200 for more information.	No	n/a
name	The name of this PROPERTYLIST.	Yes	na/

PROPERTYLIST

Each PROPERTYLIST must have a name attribute, and may or may not have one or more PROPERTYLIST and/or PROPERTY. A PROPERTY is basically a name and value pair in XML format. Its definition is:

```
<complexType name="PropertyType">
   <annotation>
      <documentation>
        This is a generic property element.
      </documentation>
   </annotation>
   <simpleContent>
      <extension base="string">
        <attribute use="required" name="NAME" type="string" />
      </extension>
   </simpleContent>
</complexType>
```

PROPERTY

If the PROPERTY refers to authetication service, the value of NAME attribute must be a valid authentication method, and the text of PROPERTY must be a valid authentication service name.

Sample SelectAuthProperty XML

A typical SelectAuthProperty XML is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!—DON’T change anything in the root document->
<COMPONENT TYPE="decision" CONDITION="any" NAME="authentication"
  DESCRIPTION="Authenticate Users" EVALUATOR="authentication"
  CONFIGURATOR="com.hp.ov.selectaccess.rulebuilder.screens.
  AuthPropertiesDlg">
  <!—NAME attribute must be “authentication”->
```

```
<PROPERTYLIST NAME="authentication">
     <!—for P13n propertylist, the NAME must be “Personalization”->
     <PROPERTYLIST NAME="Personalization">
          <!—NAME must be “UserAttributes” to define identity
          attribute->
          <PROPERTYLIST NAME="UserAttributes">
               <!—In identity attribute, the NAME must be valid LDAP
               attribute, the text value is the environment variable
               name->
               <PROPERTY NAME="member">id13n2</PROPERTY>
          </PROPERTYLIST>
          <!—to define group attribute, the NAME must be
          “GroupAttributes”->
          <PROPERTYLIST NAME="GroupAttributes">
              <!—the NAME must be valid LDAP attribute, and the text
              value is the environment variable->
              <PROPERTY NAME="aliasedObjectName">group13n2</PROPERTY>
          </PROPERTYLIST>
          <!—to define dynamic group attribute, the NAME must be
          “RoleAttributes”->
          <PROPERTYLIST NAME="RoleAttributes">
              <!—the NAME must be valid LDAP attribute, and the text
              value is environment variable name ‡
              <PROPERTY NAME="associatedDomain">role13n2</PROPERTY>
          </PROPERTYLIST>
          <!—to define identity DN, the NAME must be “UserDN”, and the
          text value is the environment variable used to store identity
          DN->
          <PROPERTY NAME="UserDN">id13n</PROPERTY>
          <!—to define group name, the NAME must be “GroupNames”, and
          the text value is the environment variable to store group
          names->
          <PROPERTY NAME="GroupNames">groupp13n</PROPERTY>
          <!—to define dynamic group names, the NAME must be
          “RoleNames”, and the text value is the environment variable
          to store dynamic group names->
          <PROPERTY NAME="RoleNames">rolep13n</PROPERTY>
     </PROPERTYLIST>
     <!—PROPERTY in this level must refer to authentication
     service, the value of NAME attribute must be valid
     authentication method, and the text value is the
     authentication service name.->
     <PROPERTY NAME="password">password</PROPERTY>
     <PROPERTY NAME="registration">Reg</PROPERTY>
</PROPERTYLIST>
</COMPONENT>
```

Sample program

```
public void getPoliciesSample() throws Exception {

com.hp.ov.selectaccess.adminserver.wsadmin.client.WsadminSoapBindingStub
adminApi;
```

```
        try {
            adminApi = (com.hp.ov.selectaccess.adminserver.wsadmin.
            client.WsadminSoapBinding Stub)
                new com.hp.ov.selectaccess.adminserver.wsadmin.client.
                WsadminServiceLocator ().getwsadmin();
        }
        catch (javax.xml.rpc.ServiceException jre) {
            if(jre.getLinkedCause()!=null)
                jre.getLinkedCause().printStackTrace();
        }
        // Time out after a minute
        adminApi.setTimeout(60000);
        adminApi.setUsername(username);
        adminApi.setPassword(password);
        com.hp.ov.selectaccess.adminserver.wsadmin.client.ResourceRow[]
value = null;

    // Get a portion of grids
        try {
           LdapSearchCriteria resourceFilter = new LdapSearchCriteria();
            resourceFilter.setBaseID("/network");
            resourceFilter.setScope(LdapSearchCriteriaScope.SUB);

            LdapSearchCriteria subjectFilter = new LdapSearchCriteria();
            subjectFilter.setBaseID(auth_dn);
            subjectFilter.setScope(LdapSearchCriteriaScope.SUB);
            value = adminApi.getPolicies(resourceFilter, subjectFilter);
            IdentityColumn idCol = value[0].getIdentityColumn(0);
            Policy policy = idCol.getPolicy();
        }
        catch (Exception e1) {
    System.err.println("Error: " + e1.getErrorMessage());
        }

        // get only one grid
        try {
           LdapSearchCriteria resourceFilter = new LdapSearchCriteria();
            resourceFilter.setBaseID(resource_path);
            resourceFilter.setScope(LdapSearchCriteriaScope.BASE);

            LdapSearchCriteria subjectFilter = new LdapSearchCriteria();
            subjectFilter.setBaseID(auth_dn);
            subjectFilter.setScope(LdapSearchCriteriaScope.BASE);
            value = adminApi.getPolicies(resourceFilter, subjectFilter);
            assertNotNull("getPolicies returns null", value);
            IdentityColumn idCol = value[0].getIdentityColumn(0);
            Policy policy = idCol.getPolicy();
        }
```

```
        catch (Exception e1) {
    System.err.println("Error: " + e1.getErrorMessage());
        }
    }
```

setPolicy

`setPolicy` sets policy to a specific resource.

Input

`setPolicy` has four input fields. The definition of the `setPolicy` input fields is:

```
<complexType>
 <sequence>
  <element name="resourcePath" type="string"/>
  <element name="identityDN" type="string"/>
  <element name="oldPolicy" type="impl:Policy"/>
  <element name="newPolicy" type="impl:Policy"/>
 </sequence>
</complexType>
```

setPolicy parameters

The following table lists the `setPolicy` parmeters:

Table 26 setPolicy Parameters

Parameter	Description	Required	Value
`resourcePath`	The resource to where the policy will be applied.	Yes	Refer to resourcePath on page 178 for more information
`identityDN`	The `idenityDN` to whom the policy will be applied.	Yes	Refer to identityDN on page 181 for more information
`oldPolicy`	An existed policy.	Yes	Depends on the resource, `oldPolicy` may be instance of `AccessPolicy`, `AdminPolicy`, `WorkflowPolicy` and `SelectAuthPolicy`. It must be valid.
`newPolicy`	The new policy will be applied.	Yes	Depends on the resource, `newPolicy` may be instance of `AccessPolicy`, `AdminPolicy`, `WorkflowPolicy` and `SelectAuthPolicy`. It must be valid.

Select Access supports multiple administrative sessions. In order to make sure the policies are consistent among all sessions to avoid security breach, Select Access needs the `oldPolicy` to validate the policy states and setting. The `newPolicy` must copy (clone) everything from `oldPolicy`, and make changes based on that. Unlike `setResource`, `setPolicy` can only change policy, it's not intented to clean or create policy.

The administration policy and workflow policy can be set in one function call but the implementation uses two steps for the operation. In Step 1, it sets the administration policy and in Step 2, it sets workflow policy. Because these operations are not atomic, there are seven possible results:

Step 1 status = failure, Step 2 will not proceed, return Exception with error message

Step 1 status = sccess, Step 2 = sccess, return sccess

Step 1 status = sccess, Step 2 = failure, return Exception with error message

Step 1 status = sccess, Step 2 = workflow_pending, return workflow_pending

Step 1 status = workflow_pending, Step 2 = sccess, return workflow_pending

Step 1 status = workflow_pending, Step 2 = workflow_pending, return workflow_pending

Step 1 status = workflow_pending, Step 2 = failure, return Exception with error message

An exception is thrown if there is a failure result, but the change in the first step is not rolled back because the rollback may bring more failures. You must determine how best to handle this situation.

Output

`setPolicy` returns a string of `SUCCESS`, `FAILURE` or `PENDING_WORKFLOW`. Refer to the Output section of setResource on page 187 for the definition.

Sample program

The sample program demonstrates how to set `SelectAuthPolicy` through the Administration API Java classes that are generated by `wsdl2java`.

```
public void setPolicySample() throws Exception {
    com.hp.ov.selectaccess.adminserver.wsadmin.client.Wsadmin
    SoapBindingStub adminApi;
    try {
        adminApi = (com.hp.ov.selectaccess.adminserver.wsadmin.
        client.WsadminSoapBindingStub)
            new com.hp.ov.selectaccess.adminserver.wsadmin.
            client.WsadminServiceLocator().getwsadmin();
    }
    catch (javax.xml.rpc.ServiceException jre) {
        if(jre.getLinkedCause()!=null)
            jre.getLinkedCause().printStackTrace();
    }

    // Time out after a minute
    adminApi.setTimeout(60000);
    adminApi.setUsername(username);
    adminApi.setPassword(password);
```

```
        ReturnString value = null;

// get old SelectAuthPolicy
        // define resource scope
        LdapSearchCriteria resourceFilter = new LdapSearchCriteria();
        resourceFilter.setBaseID(resource_path);
        resourceFilter.setScope(LdapSearchCriteriaScope.BASE);
        // define identity scope
        LdapSearchCriteria subjectFilter = new LdapSearchCriteria();
        subjectFilter.setBaseID(auth_dn);
        subjectFilter.setScope(LdapSearchCriteriaScope.BASE);

        ResourceRow[] oldPolicies = null;
        Policy oldPolicy = null;

        try {
            subjectFilter.setBaseID(SpecialIdentity.select_auth
            .getValue());
            // get grid policy
            oldPolicies = adminApi.getPolicies(resourceFilter,
            subjectFilter);
            SelectAuthPolicy oldSelectAuthPolicy = (SelectAuthPolicy)
            oldPolicies[0].getIdentityColumn(0).getPolicy();
            SelectAuthPolicy newPolicy = new SelectAuthPolicy();
           newPolicy.setState(SelectAuthPolicyState.ENABLED.getValue());
            // copy old policy properties
            newPolicy.setInherited(oldSelectAuthPolicy.isInherited());
            // build SelectAuthProperty
            SelectAuthPropertyType selectAuthProperty = new
            SelectAuthPropertyType();

            ComponentType component = new ComponentType();
            selectAuthProperty.setCOMPONENT(component);
            component.setCONDITION("any");
            component.setCONFIGURATOR("com.hp.ov.selectaccess.
            rulebuilder.screens.AuthPropertiesDlg");
            component.setDESCRIPTION("Authenticate Users");
            component.setEVALUATOR("authentication");
            component.setNAME("Authentication");
            component.setTYPE("decision");

            PropertyListType propertyList = new PropertyListType();
            component.setPROPERTYLIST(propertyList);
            propertyList.setNAME("authentication");

            PropertyType subProperty1 = new PropertyType();
            subProperty1.setNAME("password");
            subProperty1.set_value("password");
```

```
            PropertyType subProperty2 = new PropertyType();
            subProperty2.setNAME("certificate");
            subProperty2.set_value("cert");

            PropertyType[] subProperties = new PropertyType[]
            {subProperty1, subProperty2};
            propertyList.setPROPERTY(subProperties);

            PropertyListType p13n = new PropertyListType();
            p13n.setNAME("Personalization");
            propertyList.setPROPERTYLIST(new PropertyListType[]{p13n});

            PropertyType p13nProperty1 = new PropertyType();
            p13nProperty1.setNAME("UserDN");
            p13nProperty1.set_value("id13n");
            p13n.setPROPERTY(new PropertyType[] {p13nProperty1});

            PropertyListType p13nPropertyList1 = new PropertyListType();
            p13nPropertyList1.setNAME("UserAttributes");
            p13n.setPROPERTYLIST(new
            PropertyListType[]{p13nPropertyList1});

            PropertyType attributeProperty1 = new PropertyType();
            attributeProperty1.setNAME("dn");
            attributeProperty1.set_value("foo");
            p13nPropertyList1.setPROPERTY(new
            PropertyType[]{attributeProperty1});
            // SelectAuthProperty is built, set to newPolicy
            newPolicy.setProperty(selectAuthProperty);
            // call admin api to set the policy
            value = adminApi.setPolicy(resource_path,
            SpecialIdentity.select_auth.getValue(), oldSelectAuthPolicy,
            newPolicy);
        }
        catch (com.hp.ov.selectaccess.adminserver.wsadmin.client.
        AdminFault e1) {
    System.err.println("Error: " + e1.getErrorMessage());
        }
    }
```

Helper APIs

refreshValidators

`refreshValidators` clears the Validator cache. It takes no input.

getAuthServiceNames

`getAuthServiceNames` returns the names of all authentication services defined through the Policy Builder. It takes no input.

getRuleNames

`getRuleNames` returns a list of rule names.

Input

`getRuleNames` takes the `RuleType`, which must be `POLICY_RULE` or `WORKFLOW_RULE`.

Output

If the input `RuleType` is `POLICY_RULE`, it returns the names of all policy rules. If the input `RuleType` is `WORKFLOW_RULE`, it returns the names of all workflow rules.

Modifying Passwords

The Administration API supports changes to user passwords made via user self-management, password reset, or via the Administration API.

Changing User Passwords

To change a password, a user must login to a client application. The user must provide their LDAP DN, correct old password, and a new password.

In most cases, the user will already be logged into Select Access via an Enforcer plugin that makes the `authenticated_dn` readily available (`servletfilter`, `axisfilter`).

The application calls the Administration API to change the password for the user. The client application is responsible for sending the user's credential to the web service server. The web service server uses the authenticated DN to make RMI calls into the Administration server.

setUserPassword

`setUserPassword` changes user password.

Input

The input fields are:

```
<complexType>
 <sequence>
  <element name="identityName" type="string"/>
  <element name="oldPassword" type="string"/>
  <element name="newPassword" type="string"/>
 </sequence>
</complexType>
```

setUserPassword parameters

The following table lists the `setUserPassword` parmeters:

Table 27 setUserPassword Parameters

Parameter	Description	Required	Value
`identityName`	The identity whose password needs to be changed. Currently only `identityDN` is supported.	Yes	`identiyDN`. Refer to identityDN on page 181 for more information
`oldPassword`	The old password.	Yes	Old password
`newPassword`	The new password. If the password policy is set, the new password will be validated against the policy with certain limitation. Refer to the *HP OpenView Select Access 6.2 Policy Builder Guide* for information about the password policy.	Yes	New password

Output

If password was changed successfully, it returns nothing, otherwise, an `AdminFault` with error message is thrown.

Sample program

```
public void setPasswordSample() throws Exception {
    com.hp.ov.selectaccess.adminserver.wsadmin.client.Wsadmin
    SoapBindingStub adminApi;
    try {
        adminApi = (com.hp.ov.selectaccess.adminserver.wsadmin.
        client.WsadminSoapBindingStub)
            new com.hp.ov.selectaccess.adminserver.wsadmin.
            client.WsadminServiceLocator().getwsadmin();
```

```
    }
    catch (javax.xml.rpc.ServiceException jre) {
        if(jre.getLinkedCause()!=null)
            jre.getLinkedCause().printStackTrace();
    }

    // Time out after a minute
    adminApi.setTimeout(60000);
    adminApi.setUsername(username);
    adminApi.setPassword(password);

    String id_DN = "cn=foo, ou=bar, O=foobar";
    try {
        adminApi.setUserPassword(id_DN, "oldPassword",
        "newPassword");
    }
    catch (com.hp.ov.selectaccess.adminserver.wsadmin.client.
    AdminFault e1) {
System.err.println("Error: " + e1.getErrorMessage());
    }
```

Error Handling

If the provided DN and old password do not authenticate to LDAP, the returned `AdminFault` will not indicate whether the DN was valid or not. The API returns the error "`could not authenticate user`". This is to prevent this API from being used as a username/password guessing interface. If the new password is rejected, the API returns the message "`failed to change password`".

10 Administration Servlets: the Web Administration Interface

In addition to the Policy Builder applet, Select Access provides a Web Administration interface that allows administrators to perform Delegated Administration and Workflow tasks using a web browser. This chapter describes how you can customize the interface using servlets and the Web Administration interface.

Chapter Overview

Topics in this chapter include subjects that describe the Web Administration interface, its interface, and how to customize the interface for delegated administrators:

- Understanding the Web Administration Interface on page 211
- Customizing Web Administration on page 213
- About JSP Containers and JAR files on page 227

Understanding the Web Administration Interface

The Web Administration interface is a collection of classes that provide a subset of administrative tasks. Currently, the interface provides classes and methods to create, delete, and manage identity attributes, group memberships, and folders. The Web Administration interface can also submit or approve Workflow requests.

The Web Administration interface consists of the following classes:

- `Admin`: This class provides methods to search, add, modify, rename, and delete directory entries. The methods require `HttpServletRequest` and `HttpServletResponse` parameters to provide administrator credentials to an internal Enforcer plugin.
- `Attribute`: This class represents an LDAP entry's attribute. It provides methods to determine attribute type and access attribute values.
- `Entry`: This class represents an LDAP object. An `Entry` contains a collection of attributes. This class provides methods to access the entry attributes and process group memberships.
- `ErrorException`: Indicates that an error has occurred.
- `Node`: A `Node` represents a graphical tree used to display and navigate LDAP objects. A `Node` determines whether a branch in the tree is expanded or collapsed for viewing. The `Node` type is used to display different icons for identities, groups, folders, and dynamic groups.
- `SearchResults`: This class represents the set of directory entries returned from a directory server query.

`SetCharacterEncodingFilter`: This class is a servlet filter that sets the character encoding used to parse incoming requests.

Some of the Admin class methods

The `Admin` class is the core of the Web Administration interface. The more commonly used methods are shown in the code sample below. For more information on the `Admin` and other classes see the *HP OpenView Select Access 6.2 Developer's Reference Guide*.

```
Admin getWebAdmin()

void logout(HttpServletRequest request, HttpServletResponse response)

Entry getEntry
(
    HttpServletRequest request,
    HttpServletResponse response,
    String entryDN
)

Node getNode
(
    HttpServletRequest request,
    HttpServletResponse response,
    String entryDN
)

boolean setEntry
(
    HttpServletRequest request,
    HttpServletResponse response,
    Hashtable attrList
    Hashtable pwdResetList
)

boolean deleteEntry
(
    HttpServletRequest request,
    HttpServletResponse response,
    Entry entry
)

SearchResults searchTree
(
    HttpServletRequest request
    HttpServletResponse response,
    String startFolder,
    String searchFilter,
    boolean includeSubFolders,
```

```
    int sizeLimit,
    boolean withFolders,
    boolean withUserser,
    boolean withGroups,
    boolean withRoles,
    int pageSize,
    int pageIndex
)
```

`Admin.getWebAdmin()` is static. It is used to obtain an instance of the `Admin` class. `Admin.logout()` terminates an administrator's session. Most of the remaining methods in the `Admin` class provide access to identity, group, folder, and dynamic group data. Each of these methods requires an `HttpServletRequest` which is used to validate the administrator's session. The `HttpServletResponse` is required so the Web Administration Enforcer plugin can redirect administrators to a login page when necessary. Note that `getNode()` and `getEntry()` both take a string parameter containing the DN of the object, while the other methods require an `Entry` object.

Customizing Web Administration

The Web Administration server runs on the Jetty web server and servlet container. To customize Web Administration, you create and deploy servlets on the Web Administration server. You can not access the Web Administration interface from a different server. The server's document root is *<SA_install_path>*`/shared/jetty/java/webadmin`. Servlets are located in Java archive files in the `WEB-INF/lib` directory. Servlet deployment is configured by the `WEB-INF/web.xml` file.

For more information on the Jetty server, see `http://jetty.mortbay.org`.

Web Administration is based on an open source fame work known as struts. For details, see `http://struts.apache.org`.

The servlets create HTML pages and process administration requests. The HTML pages are dynamically generated with identity, group, folder, and workflow data obtained from the Web Administration interface. When an administrator performs an action, such as modifying an identity profile, the HTML form data is sent to a servlet that uses the Web Administration interface to perform the action, such as updating a profile in the Policy Store.

The form-based Delegated Administration action classes are an examples that show how to customize Web Administration by using the struts framework. Since the JSP source files no longer process business logic, the action logic is implemented via either action classes or helper classes. You need to modify these Java files to customize Web Administration.

The Build Process and Angle Brackets (>< or >#<)

When reviewing and/modifying the Java source code included in this SDK, notice that some strings are prepended with angle brackets. There are two kinds: >< and >#<. Known internally to HP development teams as "butterflies". Butterflies are used in the following places:

- >< butterflies appear in the Web administration source code.

- >#< butterflies appear in log messages.

Warning! If you modify the Java source code, it is extremely important that you do *not* do the following:

- Change strings not prepended by butterflies. These are constants required by the web application and should not be modified in any way.

Remove any butterflies; the build process automatically removes butterflies prepended to required strings.

JSP Files Used

The following JSP files are used by the Delegated Administration interface:

- `add.jsp`: Adds an identity, group, or folder.
- `browse.jsp`: Displays directory nodes.
- `common.jsp`: Contains CSS stylesheet links.
- `delete.jsp`: Deletes an identity, group, or folder.
- `errorpage.jsp`: Generates error messages.
- `logout.jsp`: Removes the session cookie and terminates the administrators session.
- `main.jsp`: Displays the main menu page with administrative tasks.
- `membership.jsp`: Manages group memberships.
- `modify.jsp`: Modifies an identity, group, or folder attributes.
- `navigate.jsp`: Displays the navigation tree for directory nodes.
- `pagetitle.jsp`: Displays the page title.
- `register.jsp`: Enrolls identities in self-registration.
- `rename.jsp`: Renames a directory server object.
- `search.jsp`: Searches directory for a person, group, or folder.
- `view.jsp`: Views directory nodes and entries.

When an administrator connects to the web interface, the web server delivers the static HTML file, `index.html`. This provides two options: Form-based Delegated Administration or Workflow Administration. The Delegated Administration link calls `controller.do`. Its URL is mapped to the `controllAction.java` which is configured in the `struts-config.xml` file.

All subsequent requests are routed to the `controller.do` action class which then delegates tasks to the other servlets. The `controller_jsp` servlet uses several parameters to determine which servlet should handle an action. The parameters are:

- `type`: Indicates whether the action is being performed on a `user`, `group`, `folder`, or `all_subjects`.
- `action`: Indicates whether to `ADD`, `DELETE`, `LOGOUT`, `MODIFY`, `RENAME`, or `VIEW` the specified object.

`sub_action`: Indicates additional subordinate actions. For example, if the primary action is `VIEW`, subordinate actions may be `BROWSE`, `SEARCH`, or `SHOW_ALL`.

Figure 16 shows the flow diagram for Form-based Delegated Administration. The rest of this section uses the `main.jsp`, `search.jsp`, and `modify.jsp` to demonstrate how to use the Web Administration interface to search for directory entries, display search results, select an identity, display current attributes, and modify attributes. For additional details on JSP pages and JAR files, see About JSP Containers and JAR files on page 227.

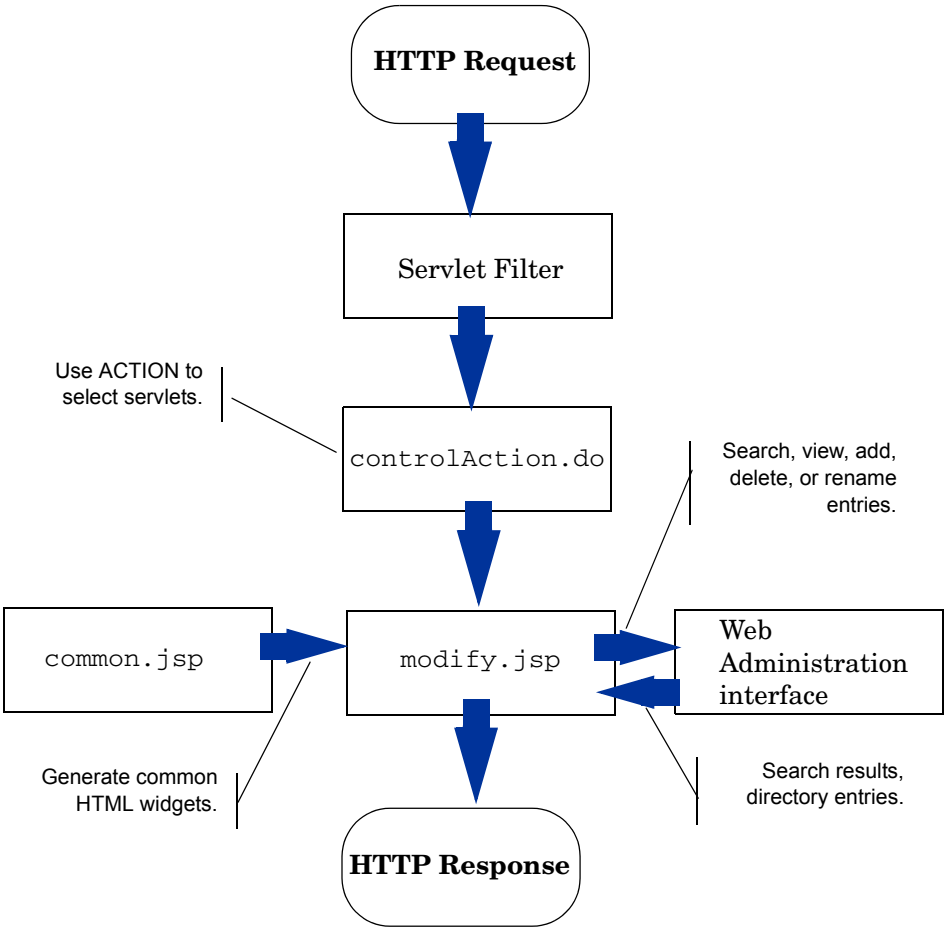

Figure 16 Flow Diagram of the Form-based Delegated Administration Tool

Providing a Menu

The first page displayed by the Form-based Delegated Administration servlet is the main menu page. The only dynamic portion is the status display, which displays the result of any previous actions. The menu options themselves do not change. Figure 17 shows the menu displayed in a web browser.

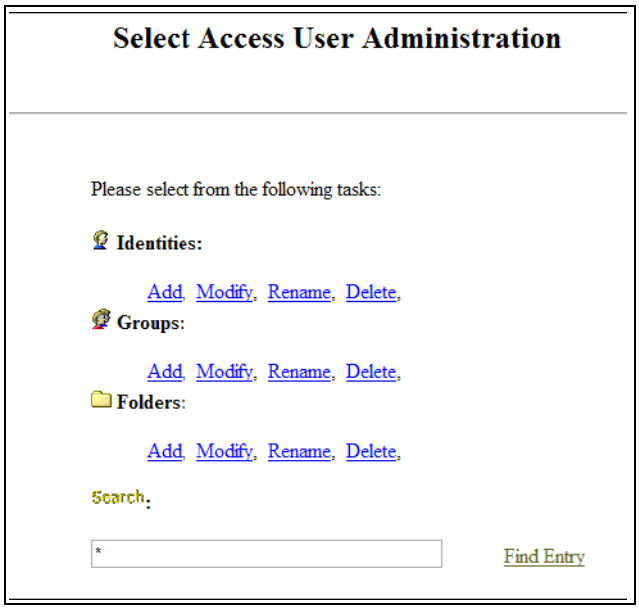

Figure 17 main.jsp

HTML main menu

The following HTML example is generated by `control.do`. Note how the URLs include the parameters that `control.do` uses to delegate tasks. For example, if an administrator clicks the link to add an identity, the `AddAction.java` is called to generate the form to add a new identity profile.

```
<html>
<body>
 ... Select Access User Administration header ...
<!-- Display a status string if one is found -->

<!-- Display all permitted operations -->
<div align="left">
Please select from the following tasks:
<h4><img src="images/user.gif">Users:<br></h4>
    <a href="controller.jsp?type=user&action=ADD">Add</a>
    <a href="controller.jsp?type=user&action=MODIFY">Modify</a>
    <a href="controller.jsp?type=user&action=RENAME">Rename</a>
    <a href="controller.jsp?type=user&action=DELETE">Delete</a>

... Group tasks ...
... Folder tasks ...

<h4><img src="images/search.gif">Search:<br></h4>
    <form NAME="SEARCH_FORM" METHOD="POST">
        <input type="text" name="filter" value="*" size="40">
        <input type="hidden" name="type" value="all_subjects">
        <input type="hidden" name="action" value="VIEW">
        <input type="hidden" name="sub_action" value="SEARCH">
```

```
            <a href="#" onClick="document.SEARCH_FORM.submit();"> Find
             Entry</a>
            <br><br>
        ... browse option ...
        ... view all option ...
        </form>
... logout option ...
... Static footer ...
</body>
</html>
```

The `MainAction.java` only generates the menu, and does not process any requests. It does not use the Web Administration interface and therefore will not be reviewed in this document.

Displaying Data: Directory Server Search Results

There are two types of data that Web Administration servlets must display, a `Node` or an `Entry`. Directory nodes represent the location of a directory server entry, and are typically used to navigate or select an entry. For instance, when adding, modifying, renaming, or deleting a directory server profile, the servlet must first locate it.

The **Search** option in the built-in Web Administration interface demonstrates how to search the directory and display the returned nodes. Figure 18 shows the results from a directory server search.

Figure 18 Directory Server Search Results

Searching a directory server

The search is performed in the `showSubjects()` method in the `HTMLSearchFormatter.java` file. A directory server search filter is provided. Based on the action being performed, the `showSubjects()` method determines whether the search should

include identities, groups, dynamic groups, or folders. As shown in the following example, an `Admin` object is created, and the `searchTree()` method is used to perform the LDAP search. The `includeSubFolders` parameter in `searchTree()` is used to control the scope of the directory search. The `startFolderId` parameter specifies where the search should begin in the directory hierarchy.

```
public static void showSubjects(HttpSession session,
   HttpServletRequest request, HttpServletResponse response,
   String searchFilter, String action, String entryType,
   String subAction, String membershipSubAction, String currentDN,
   String searchDN, StringBuffer content) throws Exception {

  String startFolderId = null;
  boolean includeSubFolders = true;
  int pageSize = Common.getPageSize();

  boolean withFolders = false;
  boolean withUsers = false;
  boolean withGroups = false;
  boolean withRoles = false;
  if (action.equals(Common.ADD_ACTION)) {
   if (membershipSubAction == null) {
    withFolders = true;
   } else {
    if (entryType.equals(Common.SUBJECT_USER)) {
     withGroups = true;
    } else if (entryType.equals(Common.SUBJECT_GROUP)) {
     withGroups = true;
     withUsers = true;
    }
   }
  } else if (entryType.equals(Common.SUBJECT_USER)) {
   if (membershipSubAction != null) {
    withGroups = true;
   } else {
    withUsers = true;
   }
  } else if (entryType.equals(Common.SUBJECT_GROUP)) {
   if (membershipSubAction != null) {
    withGroups = true;
    withUsers = true;
   } else {
    withGroups = true;
   }
  } else if (entryType.equals(Common.SUBJECT_FOLDER)) {
   withFolders = true;
  } else if (entryType.equals(Common.SUBJECT_ALL)) {
   withUsers = true;
   withGroups = true;
```

```
 withFolders = true;
 withRoles = true;
}

Admin webAdmin = Admin.getWebAdmin();
```

Performing a directory server search

```
SearchResults result = null;
try {
 result = webAdmin.searchTree(request, response, startFolderId,
   searchFilter, includeSubFolders, Common.getSearchLimit(),
   withFolders, withUsers, withGroups, withRoles, pageSize,
   searchDN);
} catch (Error e) {
 result = null;
 log.error(e);
 return;
}
String operation;
Hashtable pairs = new Hashtable();
pairs.put(Common.ACTION, action);
pairs.put(Common.TYPE, entryType);
if (membershipSubAction != null && membershipSubAction.length() > 0) {
 pairs.put(Common.SUBJECT_DN, request.getParameter
          (Common.SUBJECT_DN));
 pairs.put(Common.MEMBERSHIP_SUB_ACTION, membershipSubAction);

 if (membershipSubAction.equals(Common.ADD_MEMBER_ACTION)) {
  Entry currentEntry = webAdmin.getOriginalEntry(request,response);
  Node currentNode = currentEntry.getNode();

  // We don't allow to select already existing members.
  for (Enumeration en = result.getPageData().elements();
  en.hasMoreElements();) {
   Node node = (Node) en.nextElement();

   boolean isEnabled = false;
   if (currentNode.getIsGroup()) {
    isEnabled = !currentEntry.isHasGroupMember(node);
   } else if (currentNode.getIsUser()) {
    isEnabled = !currentEntry.isMemberOfGroup(node);
   }
   node.setIsEnabled(isEnabled);
  }
 }
 operation = Common.MEMBER_DN;
} else {
 for (Enumeration en = result.getPageData().elements();
 en.hasMoreElements();) {
```

Obtaining the nodes from the directory search

```
  Node node = (Node) en.nextElement();

  boolean isEnabled = node.getIsGroup() || node.getIsUser() ||
  node.getIsFolder();
  node.setIsEnabled(isEnabled);
 }
 operation = Common.SUBJECT_DN;
}

formatSimpleList(null, result.getPageData(), operation, pairs,
                 content);

if (searchFilter != null) {
 pairs.put(Common.SEARCH_FILTER, searchFilter);
}

HtmlPageNavigationFormatter.appendPrevNextPagingControls(
        request,
        currentDN,
        result.getFirstDN(),
  result.getContinuedDN(),
  pairs,
  content);
}
```

Displaying directory node information

As shown in the example below, once the search results are obtained, `HTMLSearchFormatter()` is called to display the results. This method demonstrates how to access the node DN and path.

```
public static void formatSimpleList(HttpServletRequest request,
   Vector data, String operation, Hashtable pairs, StringBuffer result)
   throws Exception {

  StringBuffer cmd = new StringBuffer();
  Common.constructBasicControllerAction(request, Common.CONTROL_PAGE,
                                        pairs, operation, cmd);

  HtmlFormatter.writeDivBegin(result, "center");
  HtmlFormatter.writeTableBegin(result);

  for (Enumeration et = data.elements(); et.hasMoreElements();) {
   Node childData = (Node) et.nextElement();
   String nodeDn = childData.getDN();
   String nodeName = childData.getNodePath();

   HtmlFormatter.writeRowBegin(result);
```

Obtaining the nodes from the directory search

Accessing a node's location

```
        HtmlFormatter.writeImage(childData, result);

Determine if
node should be
accessible
        if (childData.getIsEnabled()) {
         HtmlFormatter.writeHyperLink(result, cmd, nodeDn, nodeName);
        } else if(!childData.getIsRole()){
         HtmlFormatter.writeHyperLink(result, null, null, nodeName);
        }else if(childData.getIsRole()){
            HtmlFormatter.writeHyperLink(result, null, null, nodeName,true);
        }
        result.append("<br/>");

        HtmlFormatter.writeRowEnd(result);
       }////end-if

       if (result.length() > 0)
        result.append("\n");

       HtmlFormatter.writeTableEnd(result);
       HtmlFormatter.writeDivEnd(result);
      }
```

The DN and the path are included in the hypertext links so that if an administrator selects the link, the node location can be passed to a servlet that will then display the `Entry` attributes for the node.

Displaying Data: Directory Entries as Identity Profiles

Once a directory server node is selected, the entry located at the node may be displayed. An entry represents the a directory server entry and it's attributes. The `ModifyAction.java` file displays a the entry as a identity profile, with its current attributes before allowing an administrator to modify the entry. Figure 19 shows a directory server entry for a specific identity profile.

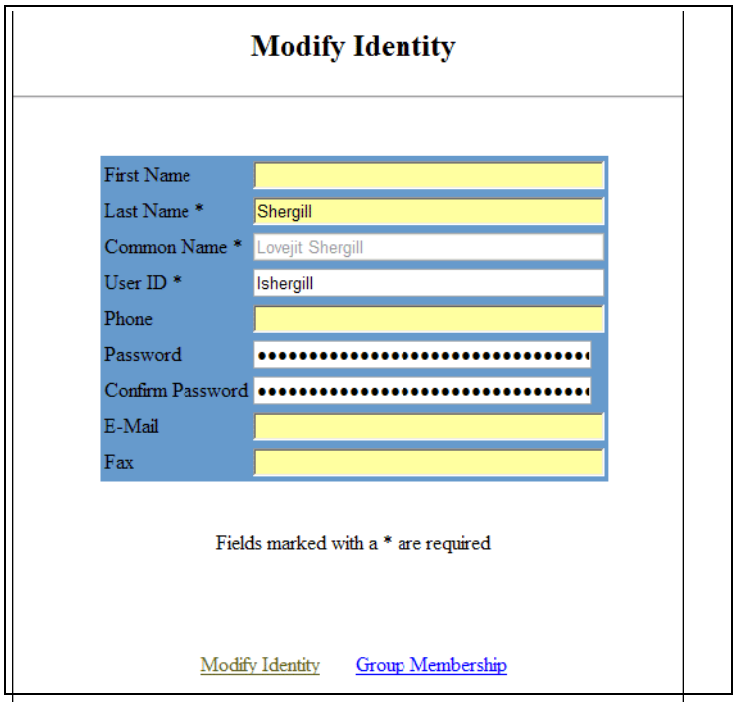

Figure 19 Modifying a Profile

Locating a directory server entry

To display an `Entry's` attributes, the servlet must first locate the `Entry`. This is accomplished by using the `Entry` DN obtained from the `Node` object in the previous section. The following code sample shows how a servlet can use the `Admin` object to extract an `Entry` using the `Entry's` DN:

Create an Admin object

Create an entry from a DN

```
StringBuffer pageContents = new StringBuffer();
StringBuffer statusMessage = new StringBuffer();
String method = (String)request.getMethod();
String type = request.getParameter(Common.TYPE);
String POST_PREFIX = "modify_";
int ID_LENGTH = 3;
Admin webAdmin = Admin.getWebAdmin();
String subjectDN = (String)request.getParameter(Common.SUBJECT_DN);
String action = request.getParameter(Common.ACTION);
HtmlModifyFormatter.SimpleTableFormatStatus formatStatus = null;
String formName = "MODIFY_FORM";

// User has already modified the entry, so now attempt to write to
   LDAP.
if ( method.equalsIgnoreCase(Common.POST_METHOD) )
{
   ... Update the object (discussed in next section).....

}
else
```

```
        {
         // Build content to represent the entry to be modified.

         // This status message is used in two places below, so create it
            just
         once here.

         if ( subjectDN != null )
         {
          // Display the properties of object specified by subjectDN
          // Note 1:: Using this version of getEntry(..) automatically
             stores the
          // entry represented by subjectDN in a session variable for later
             use.
          Entry entry = webAdmin.getEntryCached(request, response,
                                                subjectDN);
```

Display the entry's attributes

```
          if ( entry != null )
           formatStatus =
HtmlModifyFormatter.formatSimpleTableForModify(entry,
          POST_PREFIX, ID_LENGTH, true, true, pageContents);

              request.setAttribute(Common.RESULT,pageContents.toString());
              request.setAttribute(Common.FORMAT_STATUS,formatStatus);
         }

          ... Display whether the operation succeeded
```

Displaying a directory server entry as a profile

As demonstrated by the sample below, once the entry object is instantiated, the servlet can access the entry's attributes. The `formatSimpleTableForModify()` method shows how to do this.

```
public static SimpleTableFormatStatus formatSimpleTableForModify
 (
    Entry entry,
    String prefix,
    int idLength,
    boolean skipHiddenOrPwdAttribs,
    boolean useDescriptiveName,
    StringBuffer result
 ) throws Exception
 {
     SimpleTableFormatStatus status = new SimpleTableFormatStatus();

  result.append("\n");
  result.append("<table>");
```

Get the profile's attributes

Extracting an attribute

Determining if an attribute is hidden

Extracting the descriptive name of an attribute

Determining if an attribute is required

Disabling RDN and read-only attributes

Extracting the attribute value

```
boolean attrFound = false;
Vector attributes= entry.getAttributes(prefix, idLength);
for ( Enumeration ea = attributes.elements(); ea.hasMoreElements(); )
{
 Attribute attr = (Attribute)ea.nextElement();
 if ( skipHiddenOrPwdAttribs )
 {
             // do not display operational, 'objectClass' and password
              attributes
              if ( attr.isHidden() || attr.isPassword()) {
   continue;
              }
 }

 String val = attr.getStringValue();
 if ( val == null )
  val = "";

 result.append("\n");
 result.append("<tr>");
 result.append("<td>");

 if ( useDescriptiveName )
  result.append(attr.getAttributeDescriptiveName());
 else
  result.append(attr.getName());

 if ( attr.isRequired() || attr.isUid() || attr.isUserPrincipalName())
  result.append(" * ");

 result.append("</td>");
 result.append("<td>");

 result.append("<input");
 //QXCR1000226667 - Web interface Adminserver SelfManagement , can't
   change
 Common name
 if (attr.isReadOnly()||attr.isRdnAttribute())
  result.append(" disabled");
 result.append(" type=\"");
 if (attr.isPassword()) {
  result.append("password");
 } else {
  result.append("text");
 }

 String fieldName = attr.getFormFieldName();
 if (attr.isPassword())
```

```
    status.passwordFieldName = fieldName;
   if ( attr.isRequired() || attr.isUid() || attr.isUserPrincipalName())
{
    RequiredFieldStatus fieldStatus = new RequiredFieldStatus();
    fieldStatus.fieldName = fieldName;
    fieldStatus.attrName = attr.getAttributeDescriptiveName();
    status.requiredFieldNames.addElement(fieldStatus);
   }

   result.append("\" name=\"");
   result.append(fieldName);
   result.append("\" value=\"");
   result.append(val);
   result.append("\" size=\"40\" />");

   result.append("</td>");
   result.append("</tr>");

   attrFound = true;
  }

  result.append("\n");
  result.append("</table>");
  result.append("\n");
  result.append("<br><br>");

  if (attrFound)
      result.append("><Fields marked with a * are required");
  else
     result.append("><You do not have the required permissions to modify
      any
      attributes in your identity profile. Contact your administrator to
      make
     any changes you require.");

  result.append("<br><br>");
  result.append("\n");

  return status;
 }
```

Attributes that are not editable are disabled. Attributes that are hidden are not included in the HTML output. Hidden attributes should not be displayed. Alternatively, your code may include hidden attribute in hidden form elements or store them in HTTP session variables.

Modifying a Profile

The `Admin` class can be used to add, delete, modify, or rename an entry. The `ModifyAction.java` reviewed in the previous section calls the `modifyEntry()` method to update an entry. The `modifyEntry()` method uses the form data created by the `formatSimpleTableForModify()`, including modified values. The code shown below is part of the same method shown in Locating a directory server entry on page 222.

Modifying the profile's directory server entry

The following code sample shows the methods used to modify entries on the directory server:

```
StringBuffer pageContents = new StringBuffer();
StringBuffer statusMessage = new StringBuffer();
String method = (String)request.getMethod();
String type = request.getParameter(Common.TYPE);
String POST_PREFIX = "modify_";
int ID_LENGTH = 3;
Admin webAdmin = Admin.getWebAdmin();
String subjectDN = (String)request.getParameter(Common.SUBJECT_DN);
String action = request.getParameter(Common.ACTION);
HtmlModifyFormatter.SimpleTableFormatStatus formatStatus = null;
String formName = "MODIFY_FORM";

// User has already modified the entry, so now attempt to write to
   LDAP.
if ( method.equalsIgnoreCase(Common.POST_METHOD) )
{
       //Admin.TRACE("Modify.jsp::Got a post:: attempt to update
        "+type);

 Hashtable postDataList = webAdmin.getPostData(request, response,
 POST_PREFIX, ID_LENGTH);
 boolean bRetVal = webAdmin.setEntry(request, response,
                   postDataList,
 null);

    if ( bRetVal ) //success

statusMessage.append(HtmlModifyFormatter.getStatusMessage(request));
      else // failure
 {
  statusMessage.append(DataStrings.getI18nString("><Failed to
     modify an entry at DN = \"{0}\".", subjectDN));
  statusMessage.append("\n<br>\n");
  statusMessage.append("><Please try again or contact your
  administrator.");
 }
}
else
```

Create an Admin object

Modifying a directory server entry using HTML form data

```
        {
         ... Display the Entry attributes (shown in previous section)...
    }
```

The full code listing for `ModifyAction.java` uses JavaScript to ensure that the HTML form data contains the all the required attributes.

About JSP Containers and JAR files

Because Select Access' Jetty JSP container loads all JAR files located in the `<SA_install_path>\shared\jetty\policy_builder\webadmin` folder, you must ensure the servlets are correctly packaged. Therefore HP recommends that you check for the following:

- Ensure your modified and recompiled servlet files were moved to the `webadmin` folder of the `deploy` staging directory (described in To build and install the Web Administration interface on Windows or UNIX on page 24). If you do not find the files there, ensure you relocate them.
- If you have added or deleted any default files shipped with the Select Access SDK, manually copy `web.xml` to any of the following folders (depending on your customizations):

  ```
  <SA_install_path>\shared\jetty\policy_builder\webadmin\
  delegated_admin\WEB-INF\lib
  ```

  OR

  ```
  <SA_install_path>\shared\jetty\policy_builder\webadmin\
  self_management\WEB-INF\lib
  ```

  OR

  ```
  <SA_install_path>\shared\jetty\policy_builder\webadmin\
  self_registration\WEB-INF\lib
  ```

  HP recommends that you simply modify the contents of the action class file but not rename it. Otherwise you will need to also modify the `struts-config.xml` file accordingly.

11 Customizing Logging: the Logger API

Select Access provides a Logger API that allows any Java or C++ application to report events in a secure, structured way. The Logger API utilizes the Select Access logging classes which provide named log channels, log severity levels, and log destinations. This chapter shows you how to integrate logging into your application.

Chapter Overview

Topics in this chapter include subjects that describe Select Audit, its Logger API, and how to integrate third-party logging to applications that are not default Select Access components:

- Understanding the Logger API on page 229
- Using the Logger API on page 230
- The Logger API for C++ on page 231
- The Logger API for Java on page 234

Understanding the Logger API

There are two types of methods in the Logger API: those that create log filters and those that record log events. A log filter contains:

- Destination: the location where the event message is to be written. Currently, log entries may be written to the standard output stream, the system log, a file, or Select Audit.
- Channel(s): an arbitrary string that identifies the type of audit event, such as `Logger.ADMIN_CHANNEL`, or `Logger.CONFIG_CHANNEL`. The "`*`" may be used to signify that all log channels are accepted by this filter. Each channel must specify a severity level.
- Severity level(s): the minimum level that the channel accept. Messages of a lower severity are ignored. Currently, log messages may be one of the following levels: `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `FATAL`.

The Logger API allows you to configure multiple log filters. For example, one log filter may indicate that `FATAL` messages should be written to the system log for all log channels, while another log filter might specify that `DEBUG` messages for the "test" channel should be written to a local file.

When an application records an event, it specifies the event level, a log channel, and a message. The `Logger` class matches the channel and level to the configure log filters and determines which destinations will receive the message.

If `Logger` finds a log filter with a suitable log channel and severity level, it will write the message to the destination specified by the filter. If multiple filters accept the message, `Logger` writes the message to all of the specified destinations. If no filters accept the message, the message is ignored.

The recorded log entry includes the date, time, channel name, and message text. Sample log entries show how log entries from a web server might look. Note that Select Audit formats messages as XML. This is a feature of Select Audit, not the Logger API.

Sample log entries

The following example illustrates how sample events are logged with the Logger API:

```
2004/01/16 11:45:23 myApplicationChannel System startup, ready to accept
requests
2004/01/16 11:45:24 myApplicationChannel admin /index.html 200
2004/01/16 11:45:25 myApplicationChannel admin /images/icon.gif 200
2004/01/16 11:45:26 myApplicationChannel admin /banner.html 200
2004/01/16 11:45:45 myApplicationChannel jdoe /Secure/paylist.html 403
```

The Policy Validator and Enforcer plugin are examples of applications that manage audit events with the Logger API. They use an XML configuration stored by the Policy Builder to initialize log filters. Once the filters are initialized, they record component and plugins messages. For details on these plugins, see Chapter 4, Custom Authentication Methods and Rules: the Validator API and Chapter 5, Custom Security Services: the Enforcer API.

Using the Logger API

An application must configure a set of log filters before it records events. There are two ways to configure log filters: manually or with XML. This chapter will demonstrate how to configure log filters manually with the C++ Logger API, and how to use XML configurations with the Java Logger API

The examples shown in this chapter do not sign the audit messages. If you require signed audit messages, ontact professional services.

To configure C++ log filters manually

1. Initialize the Logger API using the `Logger::init()` method.
2. Create one or more log filters by instantiating the `LogFilter` class.
3. Set the destination for the filter using the `setFilter()` method. You will need to provide `a` destination object to `setFilter()`. The Logger API for C++ on page 231, shows how to create log destinations using static factory methods, such as `LogSystem::FactoryFunc()`, `LogStderr::FactoryFunc()`, and `LogFile::CreateFileLogger()`.
4. Add one or more channels to the filter. A channel consists of a channel name and a severity level. The channel name "`*`" indicates all channels. Severity level may be `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `FATAL`.
5. Add the filter to the local logging configuration.

To configure Java log filters using XML

1. Create an XML configuration. If the XML is stored in a string or file, create an XML document with a parser.
2. Use `SyslogConfig.configure()` to create log filters from the XML configuration.

The Logger API for C++

The C++ Logger API is defined in the `Logger.h` header file. You will need to include the `LogFilter.h` header file if you are performing manual filter configuration. Depending on the type of destination you are using, you may also need to include `LogStderr.h`, `LogSystem.h`, and `LogFile.h`. Note that many of the `Logger` methods are static.

Logging Messages With the Default Destination

`Logger::log()` takes three parameters: the channel name to use, the severity level of the message, and the log message itself. If used before filters have been initialized, the default system log is used to record messages with a priority of `WARNING` or higher. The following code sample shows how to log messages using the default filter and destination. Note the method `EnforcerSocketInit()` is used to initialize `winsock` on Windows platforms. On UNIX platforms, it does nothing.

```
#include <iostream>
#include "Logger.h"
#include "LogStderr.h"
#include "LogSystem.h"
#include "LogFilter.h"
#include "LogFile.h"
#ifdef SYS_UNIX
#define LOGFILE_NAME "/tmp/logtest"
#endif
#ifdef SYS_WINDOWS
#define LOGFILE_NAME "C:\\temp\\logtest"
#endif

int main(int argc, char *argv[])
{
    // Initialize winsock if needed
    EnforcerSocketInit();

    // First, try logging to the default destination
    Logger::log("LogTestChannel", ENFORCER_LOG_WARNING,
        ">#<Test 1::default output channel");
    Logger::log(LogTestChannel, ENFORCER_LOG_INFO, ">#<Test 2::Should
                not be logged");

    ... Manual configuration ...
```

Initialize Windows sockets

Log to the default destinations

```
    ... Test manually configured channels ...

}
```

Manually Configuring Log Filters

After testing the default filter, `LogTest.java` initializes the Logger API using the `Logger::init()` method. This static method clears any currently configured filters and initializes the API using the default logging destination. `LogTest.java` then creates a new filter and manually adds destinations and channels.

The `LogFilter::LogFilter()` constructor creates an empty filter. The sample code uses its `setDestination()` method to add a destination reference. Its parameter specifies a destination, which is typically generated by a destination factory method for the type of destination being used. For example, the `LogStderr::FactoryFunc()` method creates an object that logs to standard error, `LogSystem::FactoryFunc()` logs to the system log, and `LogFile::CreateFileLogger()` logs to a file. Note the different factory methods take different parameters. A log file requires a file name and parameter that specifies the maximum file size, while an audit factor requires the host and port of Select Audit.

For each filter, you must add one or more channels. Channels are added using the `LogFilter::addChannel()` method. This method's two parameters are a channel name and a log level. If `NULL` is used as the channel name, all channels are accepted. Only messages with a severity equal to or greater than the configured level will be logged. All other messages will be ignored. After all the channels are added to the filter, `LogTest.java` adds the filter to the `Logger` class using the `Logger::addFilter()` method.

Sample custom log filters configuration

The following code sample shows how to configure a system log, standard error and log file filters:

```
... Headers ...
int main(int argc, char *argv[])
{

    ... Test default channels...

    // Now, try building an interesting log channel
    Logger::init("LogTest");

    // Build a filter for stderr
    LogFilter *filt = new LogFilter();

    filt->setDestination(LogStderr::FactoryFunc("LogTest", NULL));
    filt->addChannel("LogTestChannelError", ENFORCER_LOG_ERROR);
    filt->addChannel("LogTestChannelDebug", ENFORCER_LOG_DEBUG);
    filt->addChannel(NULL, ENFORCER_LOG_FATAL);

    // Give the filter to the logger; the logger will delete it later
    Logger::addFilter(filt);
```

Initialize the Logger

Creating a filter

Setting a destination for StdErr

Adding a filter to the Logger

```
    // Build another filter for syslog
    filt = new LogFilter();
    filt->setDestination(LogSystem::FactoryFunc("LogTest", NULL));
    filt->addChannel(NULL, ENFORCER_LOG_WARNING);
    Logger::addFilter(filt);

    // And another filter for an output file
    filt = new LogFilter();
    filt->setDestination(LogFile::CreateFileLogger("LogTest",
                                                   LOFGFILE_NAME, 100));
    filt->addChannel("LogTestChannelDebug", ENFORCER_LOG_DEBUG);
    Logger::addFilter(filt);

    ... Send log entries to the custom channels...
    ... Clean up ...
}
```

Setting a destination for syslog

Setting a destination for a file

Logging Messages

As shown in the subsequent code sample, after log filters have been configured, `LogTest.java` records audit events with `Logger::log()`, which writes the message to various destinations. The method's three parameters are the channel name, the severity level, and the message. It uses the channel name and severity level determine the message destinations. Note that `Logger::log()` works like `printf`. Strings containing "*%*" variables accept additional parameters that are formatted and inserted into the message.

```
... Header ...

int main(int argc, char *argv[])
{

    ... Test default channels...
    ... Configure custom channels ...

    // Now, some interesting messages
    Logger::log("LogTestChannelError", ENFORCER_LOG_ERROR, ">#<test 3 ::
                 %s", "logged");
    Logger::log("LogTestChannelError", ENFORCER_LOG_WARNING, ">#<test 4
                :: %s", "only to system log");
    Logger::log("LogTestChannelDebug", ENFORCER_LOG_DEBUG, "test 5 ::
                %s", "logged");
    Logger::log("UndefinedChannel", ENFORCER_LOG_INFO, ">#<test 6 ::
                %s", "not logged");
    // You should have seen test messages 3 and 5
    // Messages 1, 3, and 4 should be in the system log;
    // Messages 3 and 4 should be in /tmp/logtest.1

    // Now, try to force /tmp/testlog to roll
```

Logging a message on a channel

Logging a message on a different channel

Testing log file roll over. Creates new logs when full.

```
    for (int i = 0 ; i < 100 ; i++)
    {
        Logger::log("LogTestChannelDebug", ENFORCER_LOG_DEBUG, "Roll test
                    %d", i);
    }
}
```

The Logger API for Java

The `LogTest.java` program demonstrates the Logger API for Java. To run this application, the Enforcer JAR files and dependent archives must be available in the classpath.

The Java Logger API is defined by the `Logger` class. You will also need to import XML classes. Note that many of the `Logger` methods are static.

Logging Messages With the Default Destination

If a `logClient` element does not contain any destination elements, the default destination is configured. The default destination for messages with a level of `WARNING` or higher, the message is written to the default system log. The following XML tags configure the Logger to use the default destination:

```
<logClient></logClient>
```

Using XML to Configure Log Filters

The Java Logger API does not expose methods to manually build filters. You must use an XML document to configure the Java `Logger` class. A sample XML configuration is show below. It demonstrates how to configure system log, standard error, log file, and Select Audit destinations. If you provide this XML document to `SyslogConfig.configure()`, it will configure four filters. Note that the standard error filter demonstrates how to configure multiple log channels for a destination.

```
<logClientConfig>
    <destination>
        <systemLog />
        <channel level="DEBUG" name="*" />
    </destination>
    <destination>
        <file unixName="/tmp/sa.log" windowsName="c:\temp\sa.log\"
                        maxSize="5000"/>
        <channel name = "*" level="DEBUG" />
    </destination>
    <destination>
        <stderr/>
        <channel name="applicationLog" level="ERROR"/>
        <channel name="debugLog" level="DEBUG"/>
    </destination>
    <destination>
```

```
        <logServer host="127.0.0.1" port="9989"/>
        <channel name="*" level="WARNING"/>
    </destination>
</logClientConfig>
```

The XML configuration

The following code sample shows the class imports that are required to access the Java Logger API. In addition to the Logger API packages, you will need to use the XML packages to initialize the Logger.

```
package examples.logtest;
// Import the Logging API classes
import com.hp.ov.auditserver.*;
import com.hp.ov.auditserver.config.*;
import com.hp.ov.selectaccess.util.Logger;
// Import other helper classes
import java.io.StringReader;
import com.sun.xml.tree.XmlDocument;
import com.hp.ov.selectaccess.util.XmlDocumentFactory;
import com.hp.ov.selectaccess.util.XmlElement;

public class LogTest
{

    // Test configuration (as XML)
    public static final String SampleConfig;
    public static final String ClientName = "LogTest";

    ... Method declarations to configure logging...
    ... Method declarations to log messages ...

}
```

The `SampleConfig` string stores the XML configuration, such as the one shown in in the previous example. `LogTest` creates a `StringReader` around the XML string, uses it to obtain a `LogClientConfiguration`, and validates the configuration using `LogClientConfiguration.validate()`. If valid, this is used to obtain an array of log filters. The `SyslogConfig.configure()` method is used to configure the `Logger` class using the provided log filter array, default system log, and name. The following code sample demonstrates how this method is typically used:

```
public LogTest
{
    ... Define constants ...

    // Create and run a logging test.
    public static void main(String[] args)
    {
        try
        {
```

```
                LogTest test = new LogTest(SampleConfig);
                // Run tests.
                test.runTests();
            } catch (Exception e) {
            }
        }
        // Create a LogTest instance, and configure logging.
        public LogTest(String configStr) throws Exception
        {
            StringReader reader = new StringReader(configStr);

            Destination[] destinations = null;

            LogClientConfig logClientConfig =
LogClientConfig.unmarshal(reader);
            logClientConfig.validate();

            if (logClientConfig != null)
                destinations = logClientConfig.getDestination();
            SyslogConfig.configure
            (
               ClientName,       // Name of program doing the logging
               false,            // always false, internal only
               destinations,     // an array of log filters
               Logger.getInstance().getSyslogLogger(), // the default system
                                                       logger
               false,             // useXML, should always be false
               null,              // always false, used for signing
               0                  // always 0, used for signing
            );
        }

        ... Define methods to log sample data ...
    }
```

Create a StringReader out of XML

Build a log filter config out of XML

Configure Logger log filters using SyslogConfig

Logging Messages

After log filters have been configures, `LogTest.java` records audit events with `Logger.log()`, which writes the message to various destinations. The method's five parameters are the Logger client name, the channel name, the human readable message, the extended messages, and the level. It uses the channel name and severity level determine the message destinations. The extended message can be `null`. Applications should always provide a human readable message.

Logging to custom channels

The following example demonstrates how to log messages using the static `Logger.log()` method. The method `runTests()` generates audit events at each severity level for each configured channel. Applications can create custom channels; however, they should only use the log levels defined in `Logger`.

```
public LogTest
{
    ... Define constants ...
    ... Define methods that configure Logger...

    // Run tests.
    public void runTests()
    {
        String channels[] = {
            Logger.ADMIN_CHANNEL,
            Logger.POLICY_CHANNEL,
            Logger.OPERATION_CHANNEL,
            Logger.CONFIG_CHANNEL,
            Logger.ENFORCER_PLUGIN_CHANNEL,
            Logger.WORKFLOW_CHANNEL
        };
        int levels[] = {
            Logger.LOG_LEVEL_DEBUG,
            Logger.LOG_LEVEL_INFO,
            Logger.LOG_LEVEL_WARNING,
            Logger.LOG_LEVEL_ERROR,
            Logger.LOG_LEVEL_FATAL
        };
        String levelNames[] = {
            "debug",
            "info",
            "warning",
            "error",
            "fatal"
        };
        for (int ic=0; ic<channels.length; ++ic)
        {
            String channel = channels[ic];
            for (int il=0; il<levels.length; ++il)
            {
                int level = levels[il];
                String levelName = levelNames[il];
               String shortMsg = "short(" + channel + ", " + levelName +
")";
                String longMsg  = "long(" + channel + ", " + levelName +
")";
              Logger.log(ClientName, channel, shortMsg, longMsg, level);
```

List of pre-defined logging channels

List of pre-defined log levels

Readable names for log levels

Test all the channels

Logging a message on the test channel

```
                }
            }
        }
    }
```

12 Custom Property Editors: the Subject Editor API

The Subject Editor API allows custom configuration screens that edit user, group, dynamic group, and folder properties. The default property panels were developed using this API. This chapter describes how to use the Subject Editor API to create custom property editor panels in the Policy Builder.

Chapter Overview

Topics in this chapter include subjects that describe how to create custom Subject Editors for identity profiles with the Subject Editor API:

- Understanding the Subject Editor API on page 239
- Creating Subject Editor Plugins on page 240
- Installing and Removing Subject Editor Plugins on page 247

Understanding the Subject Editor API

The Subject Editor API retrieves data from the Policy Store, which is displayed and edited in a `SubjectEditorPluginScreen`. This class is an extended `JPanel` that is displayed in a `JTabbedPane`. The Subject Editor API displays Subject Editor plugins for directory profiles based on the entry's `object class` attributes and an XML configuration file. A sample XML configuration is shown below.

```
<?xml version='1.0' encoding="UTF-8"?>
<SubjectEditorPluginConfig pluginDescription="Employee properties
plugin" >
    <SubjectEditorPluginMapping
        objectClass="inetOrgPerson"
        screenClass="EmployeeSubjectEditorPanel"
    />
    <SubjectEditorPluginMapping
        objectClass="organizationalPerson"
        screenClass="EmployeeSubjectEditorPanel"
    />
</SubjectEditorPluginConfig>
```

The Policy Builder searches the configuration and creates a set of Subject Editor plugins based on the object class of the directory entry. The example above will add the `EmployeeSubjectEditorPanel` when editing LDAP entries for `organizationalPerson`

and `inetOrgPerson` objects. The screen class should contain the fully specified class name, including packages. The plugin description is displayed in the **Configure Subject Editor plugins** dialog.

Do not use Subject Editor plugins to configure identity data for profiles containing attributes that have language specifiers, e.g. `cn;lang-fr`.

Creating Subject Editor Plugins

Subject Editor plugins are Java classes that extend the `SubjectEditorPluginScreen`, which is an abstract class that extends JPanel. Your plugin must override the abstract methods:

- `getScreenName()`: Returns the name to display as the tab title.
- `createScreen()`: Called when the screen is first created in the Policy Builder. Only called once per session.
- `openScreen()`: Called when the screen is first opened to edit an entry.
- `loadScreenData()`: Called every time a tab selection occurs.
- `validateScreenData()`: Called before a `saveScreenData()`.
- `saveScreenData()`: Called when the **OK** button is pressed.
- `closeScreenData()`: Called when the **OK** or **Cancel** buttons are pressed.
- `showHelp()`: Called when the **Help** button is pressed and the Subject Editor plugin's `JPanel` has focus.

These methods are called by the Policy Builder when a directory server entry is being modified.

To create a new Subject Editor plugin

1. Create a `SubjectEditorPluginScreen` to configure directory entry attributes.
2. Override the abstract methods listed above.
3. Compile your plugin and create a Java archive (JAR) file.
4. Create an XML configuration file for your plugin.
5. Load your plugin.
6. Test your plugin. Verify that the graphical components operate properly and that the identity, group, dynamic group, or folder attributes are properly updated.

   You will have to terminate any current Policy Builder sessions before testing your plugin, since the Subject Editor plugins are only initialized once per applet session.

Subject Editor plugin pverview

The following sample shows the structure of a typical plugin. The `EmployeeSubjectEditorPanel` is throughout this chapter to demonstrate the Subject Editor API.

```
import com.hp.ov.selectaccess.adminserver.interfaces.*;
import com.hp.ov.selectaccess.policybuilder.interfaces.*;
import com.hp.ov.selectaccess.policybuilder.common.*;
public class EmployeeSubjectEditorPanel extends
SubjectEditorPluginScreen
    implements ActionListener
{
    public EmployeeSubjectEditorPanel() {...}
    public void createScreen(SubjectEditorPluginMainFrame mainFrame)
        throws Exception {...}
    public void closeScreen() {...}
    public String getScreenName() throws Exception {...}
    public void loadScreenData() throws Exception {...}
    public void openScreen
    (
        SubjectNodeData sourceData,
        SubjectEntry originalEntryData,
        SubjectEntry entryData
    ) throws Exception {...}
    public void saveScreenData() throws Exception {...}
    public void showHelp() throws Exception {...}
    public boolean validateScreenData(javax.swing.JTabbedPane
tabbedPane) throws {...}
}
```

Building the Screen Interface

The first step to building a Subject Editor plugin is design the graphical interface that you will use to edit LDAP properties. If you are using a development environment, you can design a `JPanel` component and modify the class to extend the `SubjectEditorPluginScreen`. Figure 20 shows the graphical interface for the `EmployeeSubjectEditorPanel` class. It dynamically allocates one or more `JTextFields` to display and capture changes to attribute values. Multiple fields are displayed for multi-valued attributes. The **Add Value** and **Delete Value** buttons are used to add and remove additional values in a multi-valued attribute, such as `employeeType`.

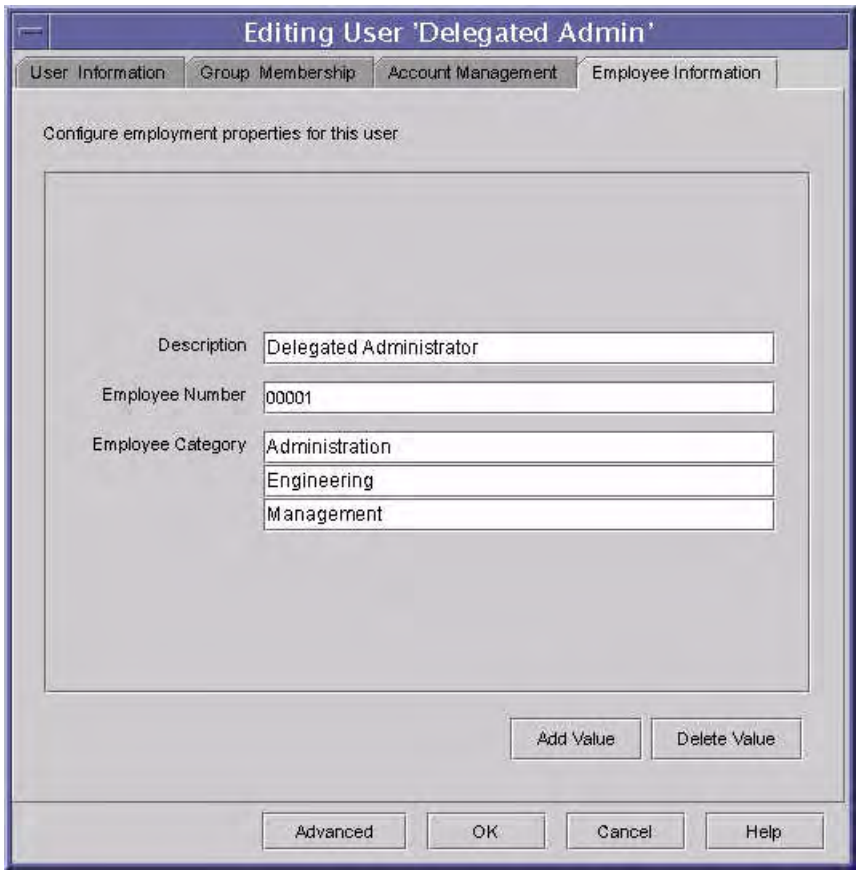

Figure 20 Creating a Customized Subject Editor Plugin

Overriding the getScreenName() method

The `getScreenName()` method does not require any parameters. This method is used to obtain the tab text displayed in the tab panel. The code below displays the tab text "Employee Information".

```
public String getScreenName() throws Exception
{
    return inl8n.getString(SCREEN_NAME);
}
```

Overriding the createScreen() method

As shown below, the Policy Builder calls `createScreen()` the first time the Subject Editor plugin is used. This method has one parameter, `mainFrame`, which should be saved for future access. It may be used to access helper methods such as `displayMessageBox()`.

```
public void createScreen(SubjectEditorPluginMainFrame mainFrame) throws
Exception
{
    this.mainFrame = mainFrame;
    // Remove any attribute value fields
    clearData();
}
```

Overriding the openScreen() method

The Policy Builder calls `openScreen()` each time a subject entry is edited. Its three parameters provide the directory entry data necessary to display current attributes, validate attributes, and update attributes. These values should be saved for use by other methods. The code example below demonstrates this.

```
public void openScreen(SubjectNodeData sourceData, SubjectEntry
originalEntryData, SubjectEntry entryData) throws Exception
{
    this.sourceData = sourceData;
    this.originalEntryData = originalEntryData;
    this.entry = entryData;
}
```

Overriding the loadScreenData() method

Once a screen has been created and opened, the Policy Builder calls `loadScreenData()` to set the graphical interface state. As deomonstrated in the sample that follows, this method is called the first time a screen is opened, and every time an administrator switches between tabs. Since the `EmployeeSubjectEditorPanel` dynamically creates `JTextFields` for each attribute value, the method shown below uses a flag so the GUI is only generated once. Without the `isConfigured` flag, new `JTextFields` would be added to an attribute every time an administrator switches tabs.

```
public void loadScreenData() throws Exception
{
    if (isConfigured)
        return;
    // Remove any attribute value fields
    clearData();
    SubjectEntryData entryData = (SubjectEntryData)
entryData.getEntryData();
    AttributeData attr = entryData.getAttribute(DESCRIPTION_ATTR);
    attr
    Vector values;
    values = attr.getAttributeStringValues(DESCRIPTION_ATTR);
    setFieldValues(descriptionPanel, values,
                   attr.isSingleValuedAttribute());
    values = entryData.getAttributeValues(EMPLOYEE_NUMBER_ATTR);
    setFieldValues(employeeNumberPanel, values,
                   attr.isSingleValuedAttribute());
    values = entryData.getAttributeStringValues(EMPLOYEE_TYPE_ATTR);
    setFieldValues(employeeTypePanel, values,
                   attr.isSingleValuedAttribute());
    isConfigured = true;
}

private void setFieldValues(javax.swing.JPanel panel, Vector values,
boolean isMulti)
{
    // No value, so provide an empty attribute value text field
```

```
    if ((values == null) || (values.size() == 0))
    {
        addAttributeTextField(panel);
        return;
    }
    Iterator iter = values.iterator();
    if (isMulti)
    {
        // Add a text field for each attribute value
        while (iter.hasNext())
        {
            addAttributeTextField(panel, (String) iter.next());
        }
    }
    else
        // Attribute is single valued, so Vector should have only one
           value
        addAttributeTextField(panel, (String) iter.next());
}
```

`EmployeeSubjectEditorPanel.loadScreenData()` uses the `SubjectEntryData` obtained from `openScreen()`. It uses the entry data to obtain the `AttributeData`, which provides access to the attribute values and indicates whether the attribute is multi-valued. The `setFieldValues()` method creates a `JTextField` for each attribute value. If an attribute has no values, or if it is single values, `setFieldValues()` creates a single text field.

Overriding the validateScreenData() method

The Policy Builder calls `validateScreenData()` when the **OK** button is pressed. If all the Subject Editor panels successfully validate their screen data, the `saveScreenData()` method is called. It should ensure that all the attribute values contain properly formatted values. Required attributes must have at least one proper value.

```
public boolean validateScreenData(javax.swing.JTabbedPane tabbedPane)
throws Exception
{
    // error handling
    SubjectEntryData entryData = (SubjectEntryData)
                                  entry.getEntryData();
    boolean success = true;
    for (int i = 0; ((i < panels.length) && i < (attributeNames.length));
                     i++)
    {
        AttributeData attr = entryData.getAttribute(attributeNames[i]);
        Vector values = getProperties(panels[i]);
        if ((attr.isRequiredAttribute()) && (values.size() < 1))
        {
            mainFrame.showMessageBox(inl8n.getString(MUST_HAVE_VALUE);
            success = false;
        }
```

```
        else if ((attr.isSingleValuedAttribute()) && (values.size() > 1))
        {
            mainFrame.showMessageBox(inl8n.getString(TOO_MANY_VALUES);
            success = false;
        }
    }
    return success;
}
```

Overriding the saveScreenData() method

Once all the Subject Editor tabs have validated their data, the Policy Builder calls `saveScreenData()` to update the attribute values. An important consideration when adding attribute data is the empty or `null` value. When an attribute has no value, the attribute should be deleted from the directory entry. Attributes that are not preferred attributes and read only attributes can not be updated.

```
public void saveScreenData() throws Exception
{
    SubjectEntryData entryData = (SubjectEntryData)
                                    entry.getEntryData();
    for (int i = 0; ((i < panels.length) && i < (attributeNames.length));
         i++)
    {
        AttributeData attr = entryData.getAttribute(attributeNames[i]);
        if ((!attr.isReadOnly) && (attr.isPreferredAttribute))
        {
            Vector values = getProperties(panels[i]);
            if (values.size() < 1)
                entryData.removeAttribute(attributeNames[i])
            else
                entryData.setAttributeValues(attributeNames[i], values);
        }
    }
}
```

Overriding the closeScreen() method

The Policy Builder calls `closeScreen()` when the screen is no longer being used. This may occurs when the administrator clicks either **OK** or **Cancel**. The `EmployeeSubjectEditorPanel` plugin removes all dynamically created `JTextFields`, and resets the `isConfigured` flag.

```
public void closeScreen()
{
    isConfigured = false;
    clearData();
}
```

Overriding the showHelp() method

The Policy Builder calls the `showHelp()` method when an administrator clicks the **Help** button and tabbed panel has focus. The `showHelp()` method shown below uses the Java `HelpSet` class to create a help window. The `HelpSet` documentation must be installed on the Policy Builder in the `<SA_install_path>/shared/jetty/policy_builder/protected` directory or sub-directory. It is a good idea to place your help files in a directory named `help/plugins` to keep them separate from the provided `HelpSet` documents.

The Java `HelpSet` classes are used by the Policy Builder, and thus the classes are already in the plugin classpath. It is worth noting that the Policy Builder uses a specialized class loader, `ClassLoaderFromJarBytes`. This class loader is used to create instances of plugin. It can locate and instantiate classes, but it does not override the `getResource()` and `getResourceAsStream()` methods. This means that you can not package your `HelpSet` documentation with your class JAR file. You must install the documentation on the Administration Server and build a URL to it.

Displaying Help

The following code sample shows how to use the `JApplet` to obtain a base URL to the Administration server. It adds the relative path of the documentation to the base URL and instantiates a `HelpSet`. If the documentation is not installed on the Administration server, the method locates the `HelpSet` documentation provided with Select Access and displays it.

```
public void showHelp() throws Exception
{
    if (helpSet == null)
    {
        // Locate and initialize the HelpSet for this panel
        createHelpSet();
        if (helpSet == null)
        {
            // Display error message, could not create HelpSet or default
               HelpSet

mainFrame.showMessageBox(inl8n.getString(ERROR_DEFAULT_HELP));
            return;
        }
        // Create a HelpBroker to display the HelpSet
        helpBroker = helpSet.createHelpBroker();
        // Set the help modal based on current JDialog
        WindowPresentation win =
            (DefaultHelpBroker) helpBroker).getWindowPresentation();
        win.setActivationWindow(this.getParentDialog());
    }
    // Display the HelpSet
    helpBroker.setViewDisplayed(true);
    helpBroker.setDisplayed(true);
}

private void createHelpSet()
{
```

```
        ClassLoader loader = getClass().getClassLoader();
        URL url = null;
        URL baseUrl = null;
        try
        {
            // Get the URL to the Policy Builder
            baseUrl = mainFrame.getMainApplet().getCodeBase();
            // Add the relative path of the documentation
            url = new URL (baseUrl, HELP_SET_URL);
            if (url != null)
                helpSet = new HelpSet(loader, url);
            else
            {
               mainFrame.showMessageBox(inl8n.getString(ERROR_HELP_LOCATE));
            }
        }
        catch (Exception e)
        {
            mainFrame.showMessageBox(inl8n.getString(ERROR_HELP_CREATE));
            getDefaultPolicyBuilderHelpSet(loader, baseUrl);
        }
        catch (ExceptionInInitializerError ex)
        {
            mainFrame.showMessageBox(inl8n.getString(ERROR_HELP_INIT));
            getDefaultPolicyBuilderHelpSet(loader, baseUrl);
        }
    }
```

Installing and Removing Subject Editor Plugins

You can install and remove any Subject Editor plugins you create. Removing your Subject Editor plugins reverts the interface back to Select Access defaults.

To install your Subject Editor plugin

1. Compile your changes.
2. Add the necessary classes to a Java archive.
3. Create an XML configuration.
4. Upload the Subject Editor plugin by clicking **Tools → Configure Subject Editor Plugins** from the Policy Builder. For more details, see the *HP OpenView Select Access 6.2 Policy Builder Guide*.
5. Restart the Policy Builder.

The Policy Builder initializes Subject Editor plugins only once per session. Future sessions will display your plugin screen in the tabbed panel for editing directory profiles.

Select Access uses a “best fit” algorithm to determine the order plugins are displayed in the tabbed pane. You cannot explicitly specify the order.

To delete your Subject Editor plugin

1. In the Policy Builder, click **Tools** → **Configure Subject Editor Plugins**.
2. In the **Configure Subject Editor Plugins** screen, select the plugin, and click **Delete**.

A Queries: Understanding and Evaluating Access

Policy Validator queries are the backbones of the Select Access access control system. By using XML to encode and transmit data, there are no restrictions on the form or the types of data that can be contained in the query. This appendix helps you to better understand the format and implementation of these queries.

Appendix Overview

Table 28 outlines major topics covered by this appendix.

Table 28 Query-related Topics

Topic Description	Details
An introduction to the process by which an Enforcer plugin sends an XML query to the Policy Validator.	What is a Query? on page 249
The structure of XML queries and responses.	What is the Structure of an XML Query on page 250
A list of query property tags that do not have any special-purpose plugins.	What Properties a Query Contains on page 252
An overview of the Query utility and how you can use it to evaluate the performance of Select Access components within your environment.	The Query Utilities on page 253

What is a Query?

A query is an XML-encoded request the Enforcer plugin makes on behalf of an identity for a particular resource on the Enforcer-protected web server. The Policy Validator then evaluates the identity's access policy in the Policy Store to determine if the identity is allowed or denied access to the network resource. The Policy Validator then returns the access decision to the Enforcer plugin.

How is a Query Used?

The process by which the Policy Validator and the Enforcer plugin evaluate an identity access request is summarized below:

1 The Enforcer plugin gathers information about an identity's incoming access request and packages data it discovers as an XML query to the Policy Validator.

2 When a query is received, the Policy Validator deciphers the query and determines where additional data lookups need to be performed for the identity and resource pair in question.

3 The Policy Validator accesses the required policy information for the identity and resource pair from the Policy Store and interprets the policy logic and conditions surrounding the request. This information becomes cached if it is required in the future.

4 If it discovers a conditional policy, the Policy Validator checks and evaluates the rule criteria defined within one or more conditional rules. A decision is reached.

5 The Policy Validator communicates the result of the policy logic to the Enforcer plugin making the request, which then enforces the policy decision.

What is the Structure of an XML Query

The Policy Validator receives queries from various Enforcer plugins as XML documents. The Enforcer plugin also encodes queries to the Policy Validator using XML. XML allows for complete extensibility with respect to modifying information in the query structure: you can arbitrarily add new attributes with values to a query.

Select Access ignores any attributes for which it has no use.

Thus, as with the authorization rules, any type of data can be passed to the Policy Validator. This can include:

- Binary objects like X.509 digital certificates
- Complex objects like complete emails
- Simple objects like standard text

For example, an XML query from the Enforcer plugin on behalf of Lance Mountain is shown in sample below. This query shows the that the Enforcer plugin's `Query Details` parameter has been set to maximal. Additionally, it shows Lance's personalization details forwarded by a federated partner, as well as a list of information that describes Steve Smith in the directory server. Notice how data is delimited by the following tag pairs:

- `<PolicyValidatorQuery>` and `</PolicyValidatorQuery>` delimit the entire query and all data contained in it.
- `<PROPERTY NAME="`*name*`"></PROPERTY>` delimit the value for the property with "*name*".
- `<PROPERTYLIST NAME="`*name*`">` delimit the value for the property list (in this case a nested property list) with "*name*".

```
<PolicyValidatorQuery>
<PROPERTY NAME="service">http://mymenu.foodcompany.com:8070</PROPERTY>
  <PROPERTY NAME="path">/test/auth/submit.cgi</PROPERTY>
  <PROPERTY NAME="dstIP">10.10.10.30</PROPERTY>
  <PROPERTY NAME="srcIP">10.10.10.110</PROPERTY>
  <PROPERTY NAME="dstPort">8070</PROPERTY>
  <PROPERTY NAME="srcPort">4001</PROPERTY>
  <PROPERTY NAME="dstHost">mymenu.foodcompany.com</PROPERTY>
  <PROPERTY NAME="protocol">http</PROPERTY>
  <PROPERTY NAME="srcHost">user_isp.com</PROPERTY>
```

```
  <PROPERTY NAME="nonce">AgAAAAAPl5SEQAAAAA+XlRqc29sMDAxLmNhLmJhb
       HRpbW9yZS5jb206OTk5OABwYXNzAGNuPXN0ZXZlIGtvdHNvcG91bG9zLG91
       PXN0ZXZlLGRjPWNhLGRjPWJhbHRpbW9yZSxkYz1jb20AAGp790LYZin
       TvdZ8UhpWv4CfkXtmu8885S0AeHA3vX7qrF353ASKBJ5RS2bJz1qCJXGfzK4v
       QqfVTT0mcE8yIgJQefoFX+GnVV092WaFYtiOe7QjfpSqXtaQmShZC1QlRnAYJV
       wo5Gx5s4B/THU5BgPKZyvUEJnK/pcsb2hOqCOd</PROPERTY>
    <PROPERTY NAME="fullname">Lance Mountain</PROPERTY>
    <PROPERTY NAME="arrived">10am</PROPERTY>
  </PROPERTYLIST>
  <PROPERTY NAME="native_nonce">enable</PROPERTY>
  <PROPERTY NAME="http_query">hungry=yes&amp;lunch=pizza</PROPERTY>
  <PROPERTYLIST NAME="http_query_list">
    <PROPERTY NAME="hungry">yes</PROPERTY>
    <PROPERTY NAME="lunch">pizza</PROPERTY>
  </PROPERTYLIST>
  <PROPERTY NAME="method">POST</PROPERTY>
  <PROPERTYLIST NAME="http_header_list">
    <PROPERTY NAME="Host">dev01:8070</PROPERTY>
    <PROPERTY NAME="User-Agent">Mozilla/5.0 (Windows; U; en-US;
                 rv:1.0.1) Gecko/20020823 Netscape/7.0</PROPERTY>
    <PROPERTY NAME="Accept">text/xml,application/xml,application/
                xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/
                x-mng,image/png,image/jpeg,image/gif;q=0.2,text/css,*/
                *;q=0.1</PROPERTY>
    <PROPERTY NAME="Accept-Language">en-us, en;q=0.50</PROPERTY>
    <PROPERTY NAME="Accept-Encoding">gzip, deflate, compress;q=0.9
    </PROPERTY>
    <PROPERTY NAME="Accept-Charset">ISO-8859-1, utf-8;q=0.66, *;q=0.66
    </PROPERTY>
    <PROPERTY NAME="Keep-Alive">300</PROPERTY>
    <PROPERTY NAME="Connection">keep-alive</PROPERTY>
    <PROPERTY NAME="Referer">http://dev01:8070/test/allow/postfix.html
    </PROPERTY>
    <PROPERTY NAME="Cookie">PolicyUser=AgAAAAAPl5SEQAAAAA+XlRqc29sMDA
         xLmNhLmJhbHRpbW9yZS5jb206OTk5OABwYXNzAGNuPXN0ZXZlIGtvdHNvcG91b
         G9zLG91PXN0ZXZlLGRjPWNhLGRjPWJhbHRpbW9yZSxkYz1jb20AAGp790LYZin
         TvdZ8UhpWv4CfkXtmu8885S0AeHA3vX7qrF353ASKBJ5RS2bJz1qCJXGfzK4v
         QqfVTT0mcE8yIgJQefoFX+GnVV092WaFYtiOe7QjfpSqXtaQmShZC1QlRnAYJV
         wo5Gx5s4B/THU5BgPKZyvUEJnK/pcsb2hOqCOd</PROPERTY>
    <PROPERTY NAME="Content-Type">application/x-www-form-urlencoded
    </PROPERTY>
    <PROPERTY NAME="Content-Length">54</PROPERTY>
  </PROPERTYLIST>
  <PROPERTY NAME="server">Apache/2.0.40 (Unix) DAV/2</PROPERTY>
  <PROPERTY NAME="queryID">2</PROPERTY>
</PolicyValidatorQuery>
```

What Properties a Query Contains

The table below outlines those query property tags that can be evaluated using string operations. If you frequently use one of these query properties and would like a custom decision point plugin for it, Select Access' extensible architecture allows you to build your own plugin. For details, see Chapter 4, Custom Authentication Methods and Rules: the Validator API.

Table 29 Query Properties Syntax

Property	Description	Value
`cert`	A property that sends information required by the certificate decision point.	A PEM-encoded X.509 digital certificate.
`client`	A property that sends information from the client software. Note: This data is not used by any of the standard server decision points.	The name and version number of the client software that initiated the request.
`dstHost` `srcHost`	A property that makes the host name explicit, when a single physical machine supports multiple virtual host names. Note: This data is not used by any of the standard server decision points.	The name of the host to which the request was sent.
`dstIP` `srcIP`	A property that retrieves destination IP information.	The IP address to which the request was sent.
`dstPort` `srcPort`	A property that retrieves destination port information. This data is required by the ports decision point.	The port to which the request was sent.
`protocol`	A property that describes which format is used for transmitting data between the browser and server.	Any accepted protocol. For example, `http`, `https`.
`method`	A property that describes what HTTP header command was used to encapsulate the data.	Any valid HTTP header command. For example, `GET`, `POST`, `HEAD`.
`server`	A property that describes what kind of web server is used.	Any supported web/application server. For example, iPlanet web server.

How the Policy Validator Replies to the XML Query

Replies are also formatted as XML. They can contain three things:

- An action telling the Enforcer plugin what to do next. Typical actions are:
  - Allow or Deny
  - Get more information by displaying a form
- Data that must be communicated to an application on the web server. For example, a Java application might require information to personalize the experience for an identity.
- Session information, namely a cookie or nonce that identifies the identity on subsequent visits to one or more configured cookie domains. For details on cookies and nonces see the *HP OpenView Select Access 6.2 Network Integration Guide*.

For example, an XML response from the Policy Validator is shown in the sample that follows. Notice how data is delimited by the following tag pairs:

- `<PolicyValidatorReply>` and `</PolicyValidatorReply>` delimit the entire response and all data it contains.
- `<PROPERTY NAME="name"></PROPERTY>` delimit the value for the property with "*name*".

```
<PolicyValidatorReply>
<PROPERTY NAME="queryID">2</PROPERTY>
  <PROPERTY NAME="authenticated_dn">cn=Lance Mountain,ou=customer,
      dc=com,dc=user_isp</PROPERTY>
  <PROPERTYLIST NAME="personalization">
    <PROPERTY NAME="User">lmountain</PROPERTY>
    <PROPERTY NAME="Phone">504%2D2325</PROPERTY>
    <PROPERTY NAME="Fax">504%2D2399</PROPERTY>
    <PROPERTY NAME="DName">cn%3Dlance%20mountain%2Cou%3Dlance%2Cdc
       %3Dcom%2Cdc%3Duser_isp%2Cdc</PROPERTY>
    <PROPERTY NAME="Groups">customer%20people</PROPERTY>
  </PROPERTYLIST>
  <PROPERTY NAME="login_time">Thu Feb 27 12:59:45 2003
</PROPERTY>
  <PROPERTYLIST NAME="authentication_server_types">
    <PROPERTY NAME="authentication_method">password</PROPERTY>
  </PROPERTYLIST>
  <PROPERTY NAME="action">ALLOW</PROPERTY>
</PolicyValidatorReply>
```

The Query Utilities

The Query utility is a command line application that sends queries to a Policy Validator. There are two implementations:

- About the C++ Implementation on page 254
- About the Java Implementation on page 257

About the C++ Implementation

You can use the Query utility to do large test-runs against a Policy Validator. With the C++ verion of the utility (`query.cpp`), you can:

- Check a single query's outcome.
- Evaluate performance within your environment.
- Review simple and advanced authentication.

  You can find the query program in the *<SDK_install_path>/query* directory.

Command line syntax

On Windows, start the Query utility by entering one of the following:

- `query [options] url`
- `query [options] file1.xml file2.xml ...`

On UNIX, start the Query utility by entering one of the following:

- `./query [options] url`
- `./query [options] file.xml file2.xml ...`

where:

- *url* is the fully qualified network location of the resource that is parsed and used to create the XML query. Entering only part of a URL results in the query failing, as all parts are needed by the Policy Validator. For example:

  `http://www.perftest.com:80/index.html`

  The parts of this URL are as follows:

  — `http` is the protocol.

  — `www.perftest.com` is the host.

  — `80` is the port.

  — `index.html` is the resource.
- *file.xml* is the file that contains one or more queries in XML format.
- `[options]` can be almost any combination of the items listed in Table 30.

  Use `-f`, `-t`, and `-l` options for the Policy Validator stress testing.

  You cannot combine some parameters with others. For example, you cannot use the `-A` parameter with the `-u` parameter, because it does not make sense to do so.

.

Table 30 Query Program Options for C++

Options	Usage
`-A user password`	Adds the username and password information into each query. Use this parameter to check the authentication process of the Policy Validator.
`-a <user><password>`	Adds the username and password information as separate arguments into each query. Use this parameter to check the authentication process of the Policy Validator for usernames and/or passwords that contain a colon.
`-C filename`	Defines the certificate used. `filename` is the name and location of the PEM format certificate.
`-c filename`	Specifies an Enforcer plugin's configuration file. `filename` is the name and location of the configuration file. If no `filename` is provided, the default is used.
`-d`	Enables internal debugging. You can increment the debug level by one for each parameter you use. For example, `-dd` increments the level of debugging to level 2 (which enables tracing).
`-e number`	Defines the kind of result you expect to receive from the Policy Validator. Options supported are: • `0`—ALLOW • `1`—DENY • `2`—ERROR • `3`—USER_DEFINED
`-f number`	For UNIX only. Forks the query into the corresponding *`number`* of parallel processes.
`-h`	Shows parameter descriptions and help.
`-l number`	Specifies the number of times the queries are sent.
`-M`	Specifies that the query being sent is a management action (for example, that the query contains information to allow the Policy Validator to re-configure itself).
`-m`	Enables nonce merging. If enabled, a nonce sent in a reply from the Policy Validator is merged into subsequent queries. This option is used to test the authentication process. This option can only be used with the `-n` option.
`-n`	Enables nonce support. To make reauthentication unnecessary, Select Access creates nonces to keep track of identities who have already been authenticated. These nonces are generated by the Policy Validator's authentication plugin.

Table 30 Query Program Options for C++ (cont'd)

Options	Usage
`-r filename`	Specifies a resource file with one URL per line. The Query program sends one query for each listed resource in the file.
`-q`	Prints the number of queries executed per second at the end of the run. Use this option with `-r` or `-u` when you are sending many queries with one command.
`-s sourceHost sourceIp`	Specifies the XML fields that contain the sender's hostname and IP address so you can simulate messages as if they were generated by different IP addresses or hosts. This option can be used, for example, to test authorization rules that administrators may have set for different identities, to ensure the Policy Validator is evaluating these rules correctly.
`-t number`	Specifies the number of threads inside each process. Each thread sends the request separately.
`-u filename`	A file that adds a series of usernames and passwords into each query. `filename` is the name and location of the user file. Use this parameter to check the authentication process of the Policy Validator.
`-V`	Returns the version number of the Query utility and exits.
-v	Always prints the result of each query.
-x	Keeps the Query program running even if one of the queries generates an error.

Usage scenarios

Depending on your business environment and your goal for using this utility, the command line syntax varies. However, we have included some common scenarios below.

Testing a single query's outcome

To test a single query, run the Query utility at a debug level of two, and include the anticipated outcome of that query. For example, on Windows run the following command:

```
query -dd -e 0 http://www.mycompany.com:80/demo
```

This example creates a single query with one thread that is sent to the Policy Validator once.

Simple performance and stress testing

To stress test and measure Policy Validator performance, run the Query utility with forking (available on UNIX only) and looping options for maximum results. For example, on Solaris run the following command:

```
./query -f 16 -l 2000 -e 0 http://www.mycompany.com:80/demo
```

This example creates 16 parallel processes with one thread each that are sent to loop the Policy Validator 2000 times. When the Query utility finishes running, it provides a summary as it exits. For example, `66206 queries per minute: 32000 queries in 29 seconds`.

Simple authentication tests

To query a resource that has SelectAuth enabled, run the Query utility with the authentication option enabled. For example, on Windows run the following command:

```
query -dd -a jdoe:secretword http://www.mycompany.com:80/demo
```

This example creates a single query with one thread that gets sent to the Policy Validator for a single instance.

Advanced authentication tests with nonce support

To query a resource that has SelectAuth enabled and confirm that nonce support is functioning correctly, run the Query utility with the authentication and nonce support options enabled. For example, run the following command:

```
query -dd -n -m -a jdoe:secretword http://www.mycompany.com:80/demo
```

This example creates a single query with one thread that gets sent to the Policy Validator. The results that are traced show that nonce support is enabled in the first query. The first Policy Validator reply then contained a new nonce. The Query utility subsequently adds that nonce into the second query. The second reply does not have a nonce because the Policy Validator knows the nonce is fresh.

About the Java Implementation

This SDK also includes a Java implementation of the Query utility. You can find this file, `QueryTest.java`, in the following location:

<SDK_install_path>`/source/Java/com/hp/ov/selectaccess/enforcer/`

This implementation is not as feature rich as the C++ implementation; HP provides this Java implementation as an example only. However, if you intend to run this example, copy the following JAR files from the Select Access core product:

- `msgsresources.jar`
- `jdom.jar`
- `jakarta-oro-2_0.jar`
- `bcprov-jdk14.jar`

Program Options

Like the C++ implementation, you can run the program with different options according to your need. Currently you can run the program with any combination of the parameters documented in Table 31.

Table 31 Query Program Options for Java

Options	Usage
`-a user:password`	Adds the username and password information into each query. Use this parameter to check the authentication process of the Policy Validator.
`-c filename`	Specifies an Enforcer plugin's configuration file. `filename` is the name and location of the configuration file. If no `filename` is provided, the default is used.
`-d number`	Enables internal debugging. You can increment the debug level by increasing the number used with this option (where 1 is the lowest level of debugging).
`-e`	Displays the current configuraiton of the Policy Validator.
`-f`	Displays any form data that were forwarded with the query.
`-h`	Shows parameter descriptions and help.
`-l number`	Specifies the number of times the queries are sent.
`-n`	Enables nonce support. To make reauthentication unnecessary, Select Access creates nonces to keep track of identities who have already been authenticated. These nonces are generated by the Policy Validator's authentication plugin.
`-p number`	Enables query pausing. You can increment the pausing duration by increasing the number used.
`-s filepath`	Sets the location to the `selectacces.conf` file.
`-V`	Returns the version number of the Query utility and exits.

Index

Numerics

32-bit operating system *See* Operating systems

Symbols

.Net API, Enforcer API, connecting with, 123

.Net interfaces, 82

.Net web service, creating, 125

A

Access control
- handlers, 105
- plugins for *See* Authorization plugin
- rules for *See* Rules
- users, validating *See* Validator plugins

AccessPolicy
- described, 194
- parameters, 194

ACE, 22

Adding
- add() method, 162
- add_ssl() method, 113
- addAuthHint2Response() method, 65
- addChannel() method, 232
- addFilter() method, 232
- addNewUserRecord() method, 170, 171
- AddValidatorCookie() method, 138, 139
- AddValidatorCookieToSoap() method, 139

Address, IP, 51, 110

AdminFault, 184

Administration
- Admin class, 211
- delegated *See* Delegated Adminisrtation
- form-based *See* Form-based administration
- requests for, 213
- server, for Select Access, 163, 246
- server described, 175
- workflow *See* Workflow

Administration API
- AccessPolicy, 194
- AdminFault, 184
- Administrative Resource paths, 179
- AdminPolicy, 196
- common data types, 178
- COMPONENT, 199
- data type overview, 177
- described, 176
- exception handling, 176
- Function Management path, 180
- getAuthServiceNames, 207
- getPolicies, 191
- getResource, 184
- getRuleNames, 207
- Helper APIs, 207
- identityColumn, 192
- identityDN, 181
- Identity Management paths, 180
- LdapSearchCriteria, 181
- LdapSearchCriteriaFilter, 183
- logging, 176
- modifying passwords, 207
- Network Management resource paths, 179
- Network Resources paths, 179
- password error handling, 209
- policy, 193
- policy-specific APIs, 191
- PROPERTYLISTy, 200
- refreshValidators, 207
- ResourceServer, 185
- resource-specific APIs, 184
- Rule, 194
- Schema Management paths, 180
- searching resource lists, 190
- searchResources, 190
- SelectAuthPolicy, 197
- SelectAuthPropertyType, 198
- setPolicy, 203
- setResourc, 187
- setUserPassword, 207
- starting, 177
- WorkflowPolicy, 195

AdminPolicy
- described, 196
- parameters, 197

Alerting, 32
allowedByValidator() method, 96
AND expressions, 73
ap_table_do() method, 113
Apache API
  connecting to
  including libraries for, 106
  supporting multiple versions, 106
  using with Enforcer API, 104
Apache Enforcer plugins
  .so files, 122
  access control handler, 105
  Apache API *See* Apache API
  authorization, checking if required, 108
  callback methods, 113
  child processes, 105
  compiling code, 122
  configuration data, 107
  cookies, extracting, 113
  credentials, extracting, 114
  decisions, enforcing, 116
  environment variables, displaying to CGI, 151
  exception handling, 106
  forms, login, 114
  forms, saving data from, 115
  gmake command, 122
  httpd.conf, loading, 104
  HTTP headers, sanitizing, 148
  HTTP transactions, 105
  installing, 122
  instances of, 105, 107
  invalid characters, 108
  libraries, including, 106
  logging, 106
  MD SSO, 119
  module initialization, 105
  native HTTP authentication, 105
  nonces, 113, 116
  personalization with, 148
  queries, 109, 116
  registering, 122
  registration data, passing, 105
  resources, freeing, 105
  SSL parameter lookup, 113
  SSO, 113
  string manipulation, 106
  UNIX considerations, 121
  URL, redirecting, 114
APIs
  Administration, described, 176
  Apache API *See* Apache API
  customizing Select Access with, 19
  Logging *See* Logger API
  NSAPI, 81
  Personalization *See* Personalization API
  Policy *See* Policy API
  servlet API *See* Servlet API
  Subject Editor *See* Subject Editor API
  User API *See* User API
Appending
  appendChild() method, 110
  appendChildString() method, 110
  appendPropertyValue() method, 94
Applications
  customizing, 146
  servers of, protecting, 20
Architecture
  plugin, 25
  Select Access, 17
  XML-based, 25
Archiving
  classpaths for, 234
  components, 39
  JAR files *See* JAR files
Arguments
  passing, 47
  XMLTreeNode, 50
Arrays, for log filters, 235
ASP
  multiple resource queries with, 159
  using, 154
AttributeLogic, 22

Attributes, 25
Attribute class, 211
attributelogic.h header, 67
attributelogic class, 66, 72
AUTHENTICATOR attribute, 38
component.xml overview, 37
CONDITION attribute, 38
CONFIGURATOR attribute, 38
Decision Point plugin for *See* Decision Point plugins
decoding values, 41
DESCRIPTION attribute, 37
evaluating, 32
EVALUATOR attribute, 38
extracting, 162
for personalization *See* Personalization
for search expressions, 66
hidden, 225
in component.xml, 37
language specifiers of, 162
managing remotely *See* Delegated Administration
multiple values for, 162
name, 49
NAME attribute, 37
nxPolicyComponent, 41
nxXml, 41
transient users *See* Transient users
TYPE attribute, 37
updating, 245
validating, 72
values, multiple, 241
vector of, 172

Audit logs, 94

Authentication, using nonces with, 255, 258

Authentication plugins
authenticate() method, 48, 50, 61, 62, 65, 168
authenticating users with, 61
authplugin.h file, 50
AuthPlugin class, 48
building, 79
constructor for, 48
creating, 29, 30
creating, steps, 50
destructor for, 50
errors, handling, 58
file authentication example *See* File Authentication plugin
FileAuthenticator, example, 52
group membership, obtaining, 72
instances, creating, 54
instances, destroying, 61
logging to Select Audit, 57
login, triggering, 65
passwords, managing, 62
personalization, setting, 51
personalization in, 62
registering, 54
result codes, 51, 62
role membership, obtaining, 72
trace flags, 62
transient users, creating *See* Transient users
users, authenticating, 50
validating users, 50
XML, classes for, 49
XML, parsing, 55
XML, responses, 65
XML, updating, 50

AUTHENTICATOR attribute, 38

Authorization plugins
AuthPlugin class, 48, 50
AuthPluginException class, 49
creating, 31
creating policy with *See* Policy Builder, 18
custom, 29
customizing rules with *See* Rule Builder
types of, 30

B

Basic, Visual
Active Server Page *See* ASP
multiple resource queries, 159

Boolean *See* Search

Bootstrap configuration, 84

Browsers. *See* Web browsers

Buttons
Browse, 163
Cancel, 33, 35, 36, 240
Configure, 84
creating, 35
Delete Value, 241
Help, 33, 35, 36, 240
OK, 33, 35, 165, 240
processing, 35
radio, 31, 37, 163
Upload, 163

C

C/C++
classes and utilities, 84, 105
COM interfaces for, 82
Enforcer API overview, 81
Enforcer API *See* Enforcer API
libraries, 20
Policy API *See* Policy API
Validator API overview, 44
Caches
Policy Validator, 41
user, 48, 51, 66, 72, 146, 161, 162
users, transient, 34
cancleClicked() method, 36
Certificates, query property for, 252
CGI
environment variables, displaying, 151
programs, customizing, 146
Characters, invalid, 93, 108
Classes
Admin, 211
Attribute, 211
AttributeLogic, 66, 72
AuthPlugin, 48, 50
AuthPluginException, 49
calling external for help window, 36
CDN, 34
Decider, 48, 50, 69
Enforcer, 88
EnforcerCOMHandle, 126
EnforcerException, 49, 71, 106
EnforcerHandle, 105
Entry, 211, 221
ErrorException, 211
EvaluatorFatal, 49, 71
EvaluatorInternal, 49
EvaluatorNonFatal, 49, 71
FileAuthenticator, 52, 53
HttpServletRequest, 151
InputFilter, 127, 133
LdapConnection, 49, 66, 67
Log, 229
LogFilter, 230
Logger, 49, 88, 106
Node, 211, 221
OutputFilter, 138
Property, 88
PropertyElement, 49, 67, 88, 105
PropertyListElement, 49, 67, 88, 105
RadiusPanel, 38
RuleComponenetPanel, 33
SearchResults, 211
ServletEnvPrint, 151
ServletFilter, 87, 88
ServletTransaction, 89
SetCharacterEncodingFilter, 212
SubjectEditorPluginScreen, 239, 240
TableSorter, 34
TableUtil, 34
User, 48, 67
UserCache, 48, 67, 72, 161
UserCacheAttributes, 162
UserSource, 49, 66, 67, 72
WebTransaction, 90
XML, 49
XmlElement, 88
XmlParser, 49, 105
XMLTreeNode, 67
XmlTreeNode, 49, 105

Classpath
dependent archives for logging, 234
for Configuration Editors, 35
for Java Enforcer plugins, 88
for Subject Editor plugins, 246
JAR files, adding to, 35

Cleanup handlers, 105

closeScreen() method, 245

Codes
restart, 168
result, 48, 51, 62
return, 70, 95

Code source, attributelogic example, 66

Colors, supported, 37

COM
classes, 84
Enforcer API overview, 81
for Enforcer plugins, 82, 126
for Enforcer plugins, building with *See* WSE Enforcer plugin
libraries of, 20
Policy API, 159
WSE Enforcers, 82

Commands
gmake, 79, 122
registering, 142

Common gateway interface *See* CGI

Comparison operators, 66, 67

Compiling, 79

COMPONENT
described, 199
parameters, 200

Components
component.jar, 35, 39
component.xml, 37, 38

Conditional
access *See* Rules
CONDITION attribute, 38
rules, 18
rules, evaluating, 70

config.xml, 41

Configuration Editors
Classpath for, 35
component.xml file for, 37
creating, 35
creating, steps, 35
criteria of, 35
displaying, 33
labeling, 37
Subject editors *See* Subject Editor plugins
using *See* Policy Builder, plugins

Configuration files
config.xml, 41
enforcer.xml, 83, 91, 107
Enforcer plugins, 83
httpd.conf, 104
Subject Editor plugin, 240
Subject editor plugin, 239

Configuring
CONFIGURATOR attribute, 38
configure() method, 231, 234
Select Access, 20

Constants
attributelogic example, 67
ENFORCER_P13NINFO, 145, 147, 149
ENFORCER_RESOURCE, 155
Enforcer API, 94
MULTIPLERESOURCELIST, 155
namespace, 123

Constructors
Authentication plugin example, 48
constructValidatorQuery() method, 127
FileAuthenticator example, 55, 57
for Decision Point plugins, 69
for Enforcer plugins, 91
for log filters, 232
for Servlet Enforcer plugins, 92
for Subject Editor plugins, 168
for Validator plugins, 50
UserSource parent, 57

Control handlers, 105

Cookies, 86
building, 92
creating, 83
extracting, 113, 134
for MD SSO, 99
for SOAP, 138
generating, 96
inserting for SOAP responses, 125
nonces for, 116

CreateFileLogger() method, 230, 232

createScreen() method, 242

Credentials
administrator, 211
user, 50, 83

cURL, 22

Customizing Select Access
configuration editors, 29
methods of, 19
option overview, 21

D

Data
authorization, extracting, 92
configuration, 107
environment, extracting, 83
navigating, 217
passing to the Policy Validator, 250
transferring via HTTP methods, 25
types, administration, 217
user, 49
validating configuration, 35

data types
common, 178
overview, 177

Decider
class, 48, 50, 69
class result codes, 48
Decide() method, 47, 48, 50, 51, 70
Decider.h header, 67
decider_branch_path string, 70

Deciders *See* Decision Point plugins

Decision Point plugins, 32
Attributelogic example, 66
attributelogic source, 66
constructor for, 69
creating, steps, 50
decide() method, using, 51
decider.h file, 50
Decider class, 48
destructor for, 70
directory server connections, 67
evaluating, 70
exception classes for, 49
exceptions, 67, 71
initializing, 69
instances, creating, 69
processing, 47
registering, 68
result codes, 51
return codes, 70
trace flags, 70
types of, 46
validating policy with, 51
XML, updating, 51
XML classes for, 49

Decoding attributes, 41, 148

Delegated Administration
API for *See* Web Administration interface
attributes, managing, 211
credentials, 211
folders, managing, 211
form-based *See* Form-based Administration
groups, managing, 211
membership, managing, 211

Descriptions
DESCRIPTION attribute, 37
for Subject Editors, 239

Destination, log, 236

Destructors
Authentication plugin example, 50
Decision Point plugin instances, 70
for Authentication plugins, 61
for Decision Point plugins, 70
for Validator plugins, 50

Digital certificate *See* Certificates

Directories
for custom components, 39
for preprocessor definitions, 71
for shared libraries, 46, 51

Directory attributes *See* Attributes

Directory servers
attributes, validating, 72
configuration entries in, 83
configuration properties, 84
connecting to, 72
connecting with, 67
creating user entries in, 161
DN, 34
example plugin entry, 41
folders, managing remotely, 211
form-based management, 217
namespace of, 49
new entry, creating, 171
object properties, 41
plugins, registering, 68
registering plugins, 54
removing plugins from, 40
search filters for, 73
storing XML, 25
user sources, listing, 49
displayMessageBox() methods, 242
Distinguished Name *See* DN
DLLs
for Enforcer plugins, 106
implementing Apache Enforcer plugin as, 122
implementing Policy Validator plugins as, 50
implementing WSE Enforcer plugin as, 142
required for Select Access, 22
DN
entry in password file, 165
extracting, 72
hypertext links for, 221
object string parameter, 213
obtaining user location for, 72
of user source, 49
user entry of, 222
Document trees, creating, 49
Document type definition *See* DTD
doFilter() method, 88
Domains, SSO across multiple *See* MD SSO
Dynamically loaded libraries *See* DLLs

E

Editors
configuration *See* Configuration Editors
subject *See* Subject Editor plugins
Encoding
character, setting, 212
personalization properties, 145, 147, 149
UTF-8, 85, 110, 239
Enforcer API
.Net API, connecting with, 123
adding services with, 82
C/C++ classes, 105
C/C++ utilities, 105
classes, 82, 84
classes, importing, 88
COM, importing libraries for, 126
COM classes, 126
COM *See* WSE Enforcer plugin
configuring, 84
constants for, 94
extending, 155
features of, 21
including libraries, 106
interfaces of, 82
jar file for, 88
Java classes, 88
overview, 81
servlet API, connecting to, 82
using, 82
using with Apache API, 104
utilities, 82, 84
EnforcerException class, 106

Enforcer plugins
Apache example *See* Apache Enforcer plugins
Apache *See* Apache Enforcer plugins
API libraries for, 20
authorization data, extracting, 92
building custom, 22
certificates, 85
classes, importing, 88
classpath for, 88
COM *See* WSE Enforcer plugins
configuration file for, 83
constructor for, 91
cookies, 83
creating, overview of, 83
creating, steps for C/C++, 104
creating, steps for Java, 87
customization overview, 20
development considerations, 85
encoding XML in, 25
enforcer32d.dll file, 142
enforcer.h header file, 65, 71
ENFORCER_ACTION_ALLOW property, 71
ENFORCER_ACTION_DENY property, 71
ENFORCER_ACTION property, 70
enforcer_check() method, 108, 109, 116
ENFORCER_P13NINFO constant, 145, 147, 149
ENFORCER_RESOURCE constant, 155
ENFORCER_SERVICE property, 156
enforcer_sys.h, 49
enforcer_sys.h header file, 106
EnforcerAddEncoded() method, 110
EnforcerAddFormData() method, 110
EnforcerAddQueryData() methods, 110
EnforcerAddrsQueryInit() methods, 110
Enforcer class, 88
EnforcerCOMHandle class, 126
EnforcerException class, 49, 71
EnforcerHandle class, 105
EnforcerInit() method, 105
EnforcerQuerySend() method, 116
EnforcerSocketInit() method, 231
environment data, extracting, 83
environment variables, reading, 83
failover support in, 83
forms, processing, 83
initializing instances, 83
logging, 82
logging in users, 86
login forms, locating, 81
MD SSO, 84, 86
nonces, 83, 86
overview, 18, 81
personalization, 82
personalization attributes, placing in local environment, 146
Policy Validator, delegating to, 81
Policy Validators, managing, 83
queries, 25, 249
remote management of, 85
requests, building, 83
resource requests, intercepting, 81
responses, evaluating, 83
responsibilities of, 20
security services, adding, 82
servlet *See* Servlet Enforcer plugin
session state, maintaining, 86
Setup Tool, configuring with, 84
site_data *See* site_data plugins
SSL, 85
SSL parameters, 83
SSO, 82, 83, 84
template for, 20
timeouts in, 83
URLs, obtaining, 83
users, authenticating, 82
users, authorizing, 82
WSE *See* WSE Enforcer plugins
XML classes for, 49
XML properties, adding, 83
Enterprise Java Beans *See* EJB, 22
Entry class, 211, 221
Envelopes, SOAP, 140
Environment
data, extracting, 83
personalization attributes, placing in, 146
servlets, 151
variables, extracting, 150
variables, for personalization, 146, 149
variables, reading, 83
ErrorException class, 211
Evaluation
evaluate() method, 72
evaluateExpressionUsingLdap() method, 72
evaluateUserSearchFilter() method, 73
evaluateUsingLdap() method, 72, 73
evaluator.h header file, 67
EvaluatorFatal class, 49, 71
EvaluatorInternal class, 49
EvaluatorNonFatal class, 49, 71
Evaluators *See* Decision Point plugins
EVALUATOR attribute, 38
Exceptions
handling, 49
throwing in Policy Validator, 72
Expat, 22
Expiry, passwords, 62
Expressions, search, 73

F

Factory
- factory() method, 47, 50, 54, 68, 165
- FactoryFunc() method, 230, 232

Failovers, 83

fetchPasswordData(), 168

fetchPasswordData() method, 168

File Authentication plugin
- attributes, getting, 172
- configuring, 163
- installing, 162
- LDIF standards, 164
- map, of user entries, 168
- map of entries, 165
- new entry, creating, 171
- password matching, 168
- permanent user entry, creating, 170
- processes of, 165
- restart code, 168
- User API, using, 162
- vector, of attributes, 172
- vector of attributes, 168

FileAuthenticator
- class, 52
- class constants, 53
- example Authentication plugin, 52

FileAuthenticator class, 52

Filters
- in rules, 32
- plugins, 46

Flags
- m_useLdap, 72
- trace, 62, 70

Folders
- data for, 213
- managing remotely *See* Delegated Administration

Formatting
- formatSimpleTableForModify(), 226
- formatSimpleTableForModify() method, 223
- simple lists, 220

Form-based administration
- attributes, hidden, 225
- data in, 217, 221, 226
- directory searches, 217
- DN links in, 221
- dynamic HTML generation, 216
- HTML, generation of, 213
- JSP pages, 213, 214
- menus, 215, 216
- servlet deployment, 213
- servlets for, 216
- URLs of, 216

Forms
- challenge, 52, 57, 105
- denial pages, 105
- error pages, 90
- list of JSP pages, 214
- login, 52, 57, 81, 86, 90, 105, 110, 114
- login, displaying, 98
- processing, 83
- saving data in, 115
- triggering login, 65
- using as input stream, 110

Functions *See* Methods

G

getAuthServiceNames, 207

GETing data, 25

getPolicies
- described, 191
- parameters, 192

getResource
- described, 184
- parameters, 185

getRuleNames, 207

Getting
- getAttributes() method, 172
- getAttributesToCopy() method, 168
- GetEnforcerCOMHandle() method, 127
- getEntry() method, 213
- getLightAdmin() method, 213
- getNode() method, 213
- getResource() method, 246
- getResourceAsStream() method, 246
- getRetVal() method, 96
- getScreenName() method, 242
- GetValidatorCookieFromReply() method, 134
- getXmlElement() method, 96

GIFs
directories for, 39
filenames, 39
requirements of, 37
rule pane icons, 37
toolbar icons, 37
gmake command, 79
GNU, using, 79
Groups
data for, 213
managing remotely *See* Delegated Administration
membership, obtaining, 72
GUI *See* User Interface

H

handleComponentShown() method, 36
Handlers
access control, 105
cleanup, 105
handleWindowOpened() method, 36
Handling
HandleFault() method, 138, 140
handleUserInfo() method, 48, 51, 146, 162, 170
Header files
attributelogic.h, 67
authplugin.h, 49, 50
Decider.h, 67
decider.h, 50
enforcer.h, 65, 71
enforcer_sys.h, 49, 106
evaluator.h, 67
LogFile.h, 231
Logger.h, 231
LogStderr.h, 231
LogSystem.h, 231
mod_enforcer.h, 122
validator_util.h, 52
Help
displaying, 246
helpClicked() method, 36
installing, 163
packaging, 246
Helper APIs, 207
Hidden attributes, 225
HTML
dynamic generation of, 213
static file, 214
HTTP
cookies, 96
denial pages, 105
forms as input stream, 110
headers, 127, 138
headers, building, 92
headers, displaying, 151
headers, personalizing with, 146
headers, sanitizing for personalization, 148
HTTPContext, 134
HttpContext, 138
httpd.conf configuration file, 104
HttpRequest property, 127
HttpServletRequest class, 151
HttpServletRequest parameter, 211
HttpServletResponse parameter, 211
method, 110
methods, 25
native authentication, 105
phases of transaction, 105
query data, 110
request, processing, 88, 92
requests, transforming, 87
resource requests, 81
response, building, 89
session variables, 225

I

Icons
for Rule Builder plugins, 37
Policy Matrix overview, 211
removing, 42
Rule Builder overview, 32
IdentityColumn
described, 192
parameters, 193
identityDN, 181
identity filter, 22
Identity filters, 22
includeSubFolders parameter, 218
inetOrgPerson objectclass, 162
INF, Web Administration, 213
init() method, 50, 88, 232
initialize() method, 35
Initializing
init() method, 47, 54, 68, 91, 165, 230
initialize_query_object() method, 109, 110, 113
InputFilter class, 127, 133

Installing
plugins, overview, 39
Select Access, 21
source code and libraries, 22
Validator plugins, 51
Interface *See* User Interface
Internationalization classes, 36
Invalid characters, 108
IP addresses, 51, 110
Is
isAuthorized() method, 89, 93, 100
isUserInLdap() method, 170
Iterating maps, 151

J

JAR files, 21
adding to CLASSPATH, 35
component.jar, 35, 39
EnforcerAPI.jar, 88
HelpSet.jar, 163
PolicyBuilder.jar, 35
shared.jar, 35, 88
Java
classes, 84
custom Enforcer plugins, 22
Enforcer API overview, 81
Enforcer classes, 88
Enforcer components, 82
libraries of, 20
log test program, 234
Policy API, 159
Server Pages *See* JSP
using archives for, 39
JDialog, 33
displaying configuration in, 35
displaying help, 246
JPanel, building, 33
JSP
creating, 157
creating Policy Validator request with, 157
for form-based administration, 213
list of files, 214

L

Languages
attributes for, 162
index of, 36
VB *See* Visual Basic
LdapConnection class, 49, 66, 67
LdapSearchCriteria
described, 181
parameters, 182
LdapSearchCriteriaFilter
described, 183
parameters, 183
LDAP *See* Directory servers, 25
LDIF
passwords, conforming to standards, 164
sample password entry for, 164
Libraries
migrating from previous releases, 22
shared, 68
types of *See* APIs, 20
Light Administration *See* Form-based Administration
Lightweight Directory Interface Format *See* LDIF
Linux requirements on Red Hat, 123
Loading plugins *See* Plugins
loadScreenData() methods, 243
log() method, 231, 233
LogFile.h header, 231
LogFilter() method, 232
Log filter, initializing, 232
LogFilter class, 230
Logger.h header file, 231
Logger class, 49, 88, 106, 229
Logging
filters, 46
integrating into applications *See* Logging API
levels of, 49
secure remote, 82
Logging API
classpaths, 234
configuration, validating, 235
filter, contents of, 229
filter, creating, 229
filter, initializing, 232
filters, configuring, 230
messages, writing, 236
overview, 229
testing, 234
variables, 233
XML, 230
XML configuration, 231
Logging in forms, 114
Logging out, 32
Log in, Enforcer plugin protocol, 86

Login forms, displaying, 98
logout() method, 213
Logs
  arrays for filters, 235
  audit, 94
  channels for, 229
  configuration, validating, 235
  destinations for, 236
  file, logging to, 232
  filter, creating, 229
  filters for, 230
  maximum size of, 232
  output destination, 229
  severity levels, 229
  syslog, 233
  variables for, 233
  XML configuration, 231
LogStderr.h header file, 231
LogSystem.h header, 231

M

m_enforcer, 93
m_handle, 123
m_handle.Init(), 127
m_properties, 33
Macros, string manipulation, 49, 106
mainFrame parameter, 242
Makefile, modifying, 79
Management, password, 62
Maps
  iterating, 151
  of user entries, 165
  URLs, 214
  user entries, 168
MD SSO, 84, 86, 93
  Apache Enforcer plugin, 119
  helper methods for, 101
  nonce, extracting, 94
  performing, 99
Membership
  administration, 211
  administrators, 211
  comparing, 66
  group, managing remotely *See* Delegated Administration
  group information, 48, 73, 78
  page for managing, 214
  role information, 48, 51, 78
Memory utilities, 49
Menus, displaying, 215, 216
Messages
  log, levels, 229
  logs, writing, 236

Methods
add, 113
add(), 162
addAuthHint2Response(), 65
addChannel(), 232
addFilter(), 232
addNewUserRecord(), 170, 171
AddValidatorCookie, 139
AddValidatorCookie(), 138
AddValidatorCookieToSoap(), 139
allowedByValidator(), 96
ap_table_do(), 113
appendChild(), 110
appendChildString(), 110
appendPropertyValue(), 94
authenticate(), 48, 50, 61, 62, 65, 168
closeScreen(), 245
configure(), 231, 234
constructValidatorQuery(), 127
CreateFileLogger(), 230, 232
createScreen(), 242
customization, 19
decide, 51
Decide(), 70
decide(), 47, 50
displayMessageBox(), 242
doFilter(), 88
enforcer_check(), 108, 109, 116
EnforcerAddEncoded(), 110
EnforcerAddFormData(), 110
EnforcerAddQueryData(), 110
EnforcerAddrsQueryInit(), 110
EnforcerInit, 105
EnforcerQuerySend(), 116
EnforcerSocketInit(), 231
evaluate(), 72
evaluateExpressionUsingLdap(), 72
evaluateUserSearchFilter(), 73
evaluateUsingLdap(), 72, 73
factory, 54
factory(), 47, 50, 54, 68, 165
FactoryFunc(), 230, 232
formatSimpleTableForModify(), 223, 226
getAttributes(), 172
getAttributesToCopy(), 168
GetEnforcerCOMHandle(), 127
getEntry(), 213
getLightAdmin(), 213
getNode(), 213
getResource(), 246
getResourceAsStream(), 246
getRetVal(), 96
getScreenName(), 242
GetValidatorCookieFromReply(), 134
getXmlElement(), 96
HandleFault(), 138, 140
handleUserInfo(), 48, 51, 146, 162, 170
helper, MD SSO, 101
init(), 47, 50, 54, 68, 88, 91, 165, 230, 232
initialize_query_object, 113
initialize_query_object(), 109, 110
isAuthorized, 100
isAuthorized(), 89, 93
isUserInLdap(), 170
loadScreenData(), 243
log(), 231, 233
LogFilter(), 232
logout(), 213
modifyEntry(), 226
openScreen(), 243
ProcessMessage(), 124, 127, 128, 133, 138
processRequest(), 65
runTests(), 237
saveScreenData(), 244, 245
sCreateTemporary(), 161, 171
searchTree(), 218
sendBasicAuthPage(), 90
sendDynamicForm(), 98
sendRedirect(), 90
SendValidator(), 133
SendValidatorQuery(), 134
setDestination(), 232
setFieldValues(), 244
setFilter(), 230
SetWSPolicy(), 136
showHelp(), 246
showSubjects(), 217
sLocateByUID(), 170
validate(), 235
validateScreenData(), 244
XmlQueryInit, 94
XmlQuerySend(), 95
Migrating libraries, 22
mod_enforcer.h header file, 122
modifyEntry() method, 226
Multiple domain single sign-on *See* MD SSO
MULTIPLERESOURCELIST constant, 155

N

NAME attribute, 37
Namespace
constants, defining, 123
directory servers, 49
single sign-on, 86
Node class, 211, 221

Nonces
extracting, 86, 94, 139
for MD SSO, 99
for registration, 116
for SSO, 116
for WSE Enforcer plugins, 125
including, 97
NATIVE_NONCE property, 127
passing, 94
Policy Validator, sending to, 134
registration, 96
single sign-on, 83, 86
SOAP, adding to, 139
SSO, 96
NSAPI, 81
NTEventLogAppender.dll, 22
Null entry, 47
nxPolicyComponent attribute, 41
nxXml attribute, 41

O

Object class
attributes, 239
for entry, 239
inetOrgPerson, 162
okClicked() method, 35
Online help, 246
building into plugins, 163, 246
installing, 163
JAR file for, 163
openScreen() method, 243
OpenSSL, 22
Operating systems
dll for 32-bit, 142
Linux, 123
registering plugins, 142
UNIX, environment variables for, 151
Windows, 231
Operators, comparison, 66, 67
OR expressions, 73
OutputFilter class, 138

P

Packaging classes, 35
Parameters
action for servlet, 214
constructor, 69
for Authentication plugins, 61
for Subject Editor plugin, 243
HttpServletRequest, 211
HttpServletResponse, 211
includeSubFolders, 218
key, 162
logging, 231
mainFrame, 242
reply, 116
servlet, 151
SSL, 83, 92, 113
startFolderId, 218
string, 213
Parsing
expressions, 74
logical operators, 73
requests, 212
URLs, 90
XML, 55, 105
XML documents, 49
Passwords
changing user, 207
error handling, 209
extracting, 162
LDIF, conforming to standards, 164
managing, 62
matching to user, 168
modifying, 207
processing, 83
PEM encoding, 252
Personalization
APIs used in, 20
attributes, placing in environment, 146
attributes for, 82
configuring Select Access to support, 21
data, accessing from resource, 150
data, obtaining, 162
data, providing, 83
enabling in Policy Builder, 145
environment variables, extracting, 150
in Authentication plugins, 62
maps, iterating, 151
Personalization API *See* Personalization API, 145
prefix, using, 146
property list for, 145
setting in Authentication plugin, 51
sharing attributes, 145
supporting in Policy Validators, 146
Visual Basic, using, 154

Personalization API
ENFORCER_P13NINFO constant, using, 145, 147, 149
features of, 21
language attributes, 162
libraries of, 145
overview, 145
Platforms
file extensions for, 50
specific logging methods for, 231
supporting multiple, 106
UNIX, 121
Windows, extra requirement for, 107
Plugins
authentication *See* Authentication plugins
authorization *See* Authorization plugins
creating icons for, 37
customizing Select Access with, 19
extracting XML configuration for, 33
FileAuthenticator source, 52
filter, 46
installing, 39
loading, 39
logging, integrating into, 229
Rule Builder types, 37
Subject Editor *See* Subject Editor plugins
subrule, 46
summary of, 17
terminal points, 46
Policy
described, 193
evaluation, 47
parameters, 193
Policy API
overview, 155
XML overview, 155
Policy Builder
API *See* Policy Builder API, 21
Enforcer plugins, remote management of, 85
installing plugins, 39
loading plugins, 39
personalization enabling, 145
plugins, 31
plugins, overview, 18
plugins, removing, 40
plugin types, 30
responsibilities of, 19
sessions, terminating, 240
Subject editors *See* Subject Editor plugins
Policy Builder API
cancleClicked() method, 36
handleComponentShown() method, 36
handleWindowOpened() method, 36
helpClicked() method, 36
initialize() method, 35
JDialog, 33
JPanels, 33
libraries of, 19
okClicked() method, 35
overview of, 30
RuleComponentPanel class, 33
using, 33
utilities, 34
Policy Builder plugins
creating, overview, 34
creating, procedure, 34
Policy Enforcer *See* Enforcer plugins
Policy plugins *See* Policy Builder, plugins for
policy-specific APIs, 191
Policy Store *See* Directory servers
Policy Validator, 18, 49
API *See* Validator API
communicating with other components, 43
configuring, 43
Decision Point plugins, calling, 47
decisions, enforcing, 116
decisions, sending, 96
EvaluatorInternal exception, 72
interfacing with Select Audit, 49
JSP-based queries, 157
logging, 49
managing, 83
multiple resources, querying, 155
overview, 43
personalization, supporting, 146
plugins, loading, 38
plugin types *See* Authentication plugins and Authorization plugins
queries, 84, 133
queries adding data to, 94
query replies, 253
refreshing cache for, 41
result codes, 51
rule evaluation with, 43
SOAP requests, building, 127
SOAP responses, 126
string manipulation macros, 49
testing performance with the Query Utility, 253
utilities, using, 49
validation process, 46, 47
XML communication with, 44

Policy Validator plugins
API libraries for *See* Validator API
creating overview, 50
creating process, 50
overview, 18
removing, 51
responsibilities of, 19
using XML, 25
POSTing data, 25
Prefix, personalization, 146
Preprocessor definitions, 71
printf, 233
Processes, termination of, 121
ProcessMessage() method, 124, 127, 128, 133, 138
processRequest() method, 65
Prompts, user, 50
Properties
elements of, 34
ENFORCER_ACTION, 70
ENFORCER_ACTION_ALLOW, 71
ENFORCER_ACTION_DENY, 71
ENFORCER_SERVICE, 156
HttpRequest, 127
in component.xml file, 37
lists of, 34
lists of elements, 34
NATIVE_NONCE, 127
objects in directory, 41
personalization, special list for, 145
Policy API, using, 155
validating data, 35
XML, including, 136
Property class, 88
PropertyElement class, 49, 67, 88, 105
PROPERTYLIST, described, 200
PropertyListElement, 62
PropertyListElement class, 49, 67, 88, 105
Protocols
filters, 22
HTTP *See* HTTP

Q

Queries
authorization, 90, 105
command line options, 255, 258
definition, 249
description of, 249
elements of, 94
generating, 25, 88
processing, 155
purpose, 249
replies from Policy Validator, 253
simplifying, 155
SOAP properties, 123
SSO, 87
structure of, 250
understanding, 249
usage scenarios, 256
using Visual Basic with, 159
utility, 253

R

RadiusPanel class, 38
RADIUS server example xml file, 38
Recording, messages *See* Logging API
Records *See* Users, entries
Redirecting, 32
Refreshing caches, 41
refreshValidators, 207
Registration
Apache Enforcer plugins, 105
nonce for, 96
nonces for, 116
Removing plugins, 40
Resource paths
Administrative, 179
Function Management, 180
Identity Management, 180
Network, 179
Network Management, 179
Schema Management, 180
Resources
.NET. *See* WSE Enforcer plugins
freeing, 105
freeing with destructors, 70
multiple, querying for, 156
multiple, with Visual Basic, 159
personalization data, accessing, 150
protecting with plugins, 29
querying for multiple, 155
requests for, interception, 81
URLs obtaining, 83

ResourceServer
described, 185
parameters, 186
Resource-specific APIs, 184
Restart code, 168
Result codes, 48, 51, 62
Return codes, 70, 95
Roles
data, 213
icons for, 211
information, 48, 67
membership, obtaining, 72
searching, 218
user, 67
Rule Builder
decision points in *See* Decision Point plugins
decisions tree parts, 46
evaluating rules, 47
icons, overview of, 32
plugins, 31
RuleComponentPanel, 33
extending, 35
overriding, 35
Rules
adding decision points in, 32
adding subrules in, 32
building custom, 31
conditional, 18
creating filters for, 32
described, 194
evaluating, 70
overview of, 32
parameters, 195
types of, 32, 46
using XML, 25
runTests() method, 237

S

Saving
saveScreenData() method, 245
saveScreenData() methods, 244
sCreateTemporary() method, 161, 171
Search expressions, 73
Search filters
evaluating, 67
expression for, 66
forms, 217
LDAP, 73
SearchResults class, 211
searchTree() method, 218
Security
customizing, 29
policies. *See* Policy Builder, 18
services, adding, 82
Security policies, evaluating, 47
Select Access
APIs of *See* APIs
backend application logic of, 20
component overview, 17
configuring, 20
custom configuration editors for, 29
customizing, 19
JAR files, 21
libraries shipped with, 22
logging, integrating, 229
overview, 17, 250
passwords in, 62
personalization prefix for, 146
plugin overview, 30
security services, integrating *See* Enforcer plugins
source code, 22
use of XML in, 26
XML overview, 25
Select Audit
communicating, 83
interfacing with, 49
logging to, 82
log level, 57
logs, correlating among components, 94
SelectAuthPolicy
described, 197
parameters, 198
SelectAuthPropertyType
described, 198
parameters, 198
XML, 200
SelectID
authentication methods *See* Authentication plugins
custom editors for, 30
enabling, 47
overview, 17
personalization attributes, extracting from, 146
servers, configuring, 45
sendBasicAuthPage() method, 90
sendDynamicForm() method, 98
sendRedirect() method, 90
SendValidator() method, 133
SendValidatorQuery() methods, 134

Servlet Enforcer plugins, 82
authorization, determining if required, 93
authorization data, extracting, 92
constructor for, 92
cookies, 94
cookies, creating, 92
decisions, enforcing, 96
forms, displaying, 98
HTTP response, building, 89
instance, creating, 90
MD SSO, 93, 99
nonce, creating, 96
nonce, extracting, 94
overview, 87
personalization with, 147
Policy Validator response, building, 94
queries, building, 95
return codes, 95
servlet AP, connecting to, 82
SSL parameters, 92
URLs, parsing, 90
ServletEnvPrint class, 151
ServletFilter class, 87, 88
Servlets
container for form-based administration, 213
customizing, 146
HTTP headers, displaying, 151
ServletTransaction class, 89
Session timeouts, 83
SetCharacterEncodingFilter class, 212
setDestination() method, 232
setFieldValues() methods, 244
setFilter() method, 230
setPolicy
described, 203
parameters, 203
setResource
behavior, 188
described, 187
input parameters, 188
output parameters, 189
setResources, 190
Setup Tool
configuring Enforcer plugins with, 20
Enforcer plugins, configuring, 21, 84
MD SSO, 84
MD SSO, configuring, 87
Policy Builder, configuring, 86
SSO, 84
setUserPassword
described, 207
parameters, 208
SetWSPolicy() method, 136
shared.jar, 88
Shared object files, implementing Apache Enforcer plugin as, 122
showHelp() method, 246
showSubjects() method, 217
Single sign-on *See* SSO
Single Sockets Layer *See* SSL
site_data plugin
configuration files, web servers, 143
injecting header data, 143
loading, 143
sLocateByUID() method, 170
SOAP
cookies, adding, 138
cookies, inserting, 125
encryption, 138
envelopes for, 140
errors, 138, 140
extracting authorization data for, 123
headers, adding nonces to, 139
headers for, 138
HttpRequest, using, 127
messages, creating, 136
messages, properties of, 124
namespace constants, defining, 123
output, filtering, 138
requests, extracting properties for, 127
responses, for WSE, 125
signatures, 138
XML created for, 126
SOAP interfaces, 82
Socket connections, 105
Source code
attributelogic example, 66
example files for, 22
FileAuthenticator plugin, 52
SSL
certificates used for, 85
parameter lookup, 113
parameters, 92
properties, 83

SSO, 83, 84
cookie for, 94
Enforcer plugins
SSO, 86
nonce for, 96
nonces, protecting in SOAP response, 125
nonces, sending, 134
nonces for, 116
standards, Select Access, 17
startFolderId parameter, 218
Stateless protocols, HTTP *See* HTTP
String manipulation, 49, 106
Strings
decider_branch_path, 70
DN parameters, 213
String utilities, 49
Struts
HTML generated by, 216
introducing, 213
Subject Editor API
GUI for *See* Subject Editor plugins
methods, summary of, 240
overriding methods, 241
overview, 239
Subject Editor plugins
attributes, updating, 245
classpath for, 246
configuration for, 239, 240
configuring, 239
constructor for, 168
creating, 240
data, validating, 244
description for, 239
help, displaying, 246
installing, 247
interface, building, 241
interface, closing, 245
interface, loading, 243
limitations, 240
overview, 239
Policy Builder, ending session with, 240
SubjectEditorPluginScreen class, 239, 240
Subrules
adding, 32
plugins, 46
Sun ONE (iPlanet) Enforcer plugins, APIs used with, 81
syslog, 233, 234
system.xml library class, 126

T

Tables
filter rows, 73
null entry, 47
TableSorter class, 34
TableUtil class, 34
Templates, 21
Templates, Enforcer plugins, 20
Terminal Point plugins, 46
types of, 48
using, 48
Terminal points, rules, adding terminal points in, 32
Timeouts, 83
Toggle, 22
Tokens, 86
Trace flags, 62, 70
Transient users
creating directory entry for, 161
creating permanent entry for, 170
creating *See* User API
entry, searching for, 170
new entry, creating, 171
Trees, creating, 49
TYPE attribute, 37

U

Uninstalling plugins, 40
UNIX
CGI applications and personalization, 151
considerations, 121
so files, 122
Uploading plugins *See* Plugins
URLs
encoding personalization properties with, 145, 147, 149
invalid characters, 93
mapping, 214
obtaining, 83
parsing, 90
query data, 110
redirect, 99
redirecting, 86, 93, 114

User API
attributes, extracting, 162
entry, creating, 162
LDIF standards, 164
overview, 161
passwords, 162
personalization data, obtaining, 162
vector, of attributes, 168
XML, updating, 162
User cache, 72
UserCacheAttributes class, 162
UserCache class, 48, 67, 72, 161
User class, 48, 67
User credentials
login, 114
obtaining, 50
User data, listing, 49
User entries, synthesizing, 34
User interfaces
customizing Select Access with, 30
See Configuration editors
User roles, 67
Users
authentication, 82
data for, 213
logging in, 86
personalization data for, 20
prompting for data, 50
transient *See* Transient users
UserSource class, 49, 66, 67, 72
UserSource constructor, 57
UTF-8 encoding, 110
Utilities
C/C++, 105
memory, 49
string, 49

V

validate() method, 235
validator_util.h header file, 52
Validator API
classes, 48
features of, 21
overview, 43, 44
utilities, 48
validateScreenData() method, 244
Validator plugins
FileAuthenticator example, 52
installing, 50, 51
testing, 50
Validator plugins *See* Authentication plugins and Decision Point plugins
Value
decoding, 148
for resources, 156
Values, 25
as part of boolean expression, 66
attributes, decoding, 41
for evaluation, 68
key maps, 162
multiple, for attributes, 241
of attributes, 25, 37
of branch outcome, 66
of properties, 34, 49, 88
returning, 70
Variables
environment, extracting, 150
environment, for personalization, 146, 149
environment, reading, 83
for logs, 233
session, HTTP, 225
Vector, of attributes, 168
Visual Basic
displaying server variables with, 154
multiple resource queries, 159
voidp_func type definition, 54

W

Web Administration
categories, 214
See Form-based Administration, 214
Web Administration interface
classes of, 211
data types, 217
overview, 211
Web browser
cookies, 83, 86, 92, 99, 113, 116
cookies, setting, 97
managing remotely with *See* Delegated Administration
redirecting, 86
Web servers, protecting, 20
Web Service Enhancement *See* WSE
WebTransaction class, 90
winsock, 231

Wizards, Select Access configuration *See* Setup Tool, 20
Workflow
data, obtaining from Web Administration interface, 213
managing *See* Web Administration interface
requests, submitting and approving, 211
WorkflowPolicy
described, 195
parameters, 196
WSE Enforcer plugins
.Net API, connecting to, 123
.Net web service, creating, 125
authentication with, 123
building, 142
COM, importing libraries for, 126
COM, using with, 82
COM classes, 126
cookies, 138
errors, handling, 140
initializing, 127
namespace constants, defining, 123
NATIVE_NONCE property, 127
nonces, 125, 138
nonces, adding, 139
Policy Validator, replying to, 133
registering, 142
SOAP, 124
SOAP, filtering, 138
SOAP headers, 138
SOAP messages, 125
XML properties, extracting for SOAP, 127
XML properties, including, 136
XML requests, building, 127

X

X.509 certificates, 252
XML, 25
classes, 49
communicating with, 44
communication in Select Access, 26
communication with, 43
configuration, loading, 35
configuration file, 84
configuration files, locating, 83
creating document trees, 49
data, returning, 84
data, updating, 162
data access classes, 49
documents, processing, 84
encoding and communicating, 25
enforcer.xml, 83, 107
importance of, 25
log configuration, 231
log entries, 230
m_properties, 33
manipulating, 49
multiple resource request, 156
objects, creating, 110
parsing, 55
passing configuration as argument, 47
properties, adding, 83
properties, including, 136
property extraction, 33
property lists, 62
queries, 81
requests, building, 83, 127
response, generating, 47
responses, 62
SOAP, 126
storing rules with, 25
system.xml, 126
templates, 21
updating, 51
updating responses, 50
using Policy API with, 155
version, 37
XmlElement class, 88
XmlParser class, 49, 105
XMLTreeNode class, 50
XmlTreeNode class, 49, 67, 105
XmlQueryInit() method, 94
XmlQuerySend() method, 95