HP OpenView Select Access SDK

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

Software Version: 6.2

Developer's Reference 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

Section Two: The Java APIs 307

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
Section One: The C/C++ APIs	
Chapter 2, Getting Started with Select Access APIs	Before adding or extending Select Access components, you should be familiar with how each of the principle components works. This chapter introduces you to these components.
Chapter 3, C/C++ Enforcer API	The Enforcer API allows you to integrate Select Access security services with most products and custom components. This chapter is the reference you can use with the Java version of the Enforcer API, so you can provide security services for C/C++ objects.
Chapter 4, Policy Validator API	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. This chapter provides a reference for these functions.
Chapter 5, The User API	The Select Access C/C++ API includes several classes used to manage identity data. These classes are used for creating transient identities, retrieving personalization information, accessing the identity cache, and obtaining an LDAP connection. This chapter provides a reference for these functions.
Chapter 6, C++ XML Manipulation API	The Enforcer plugin communicates with the Policy Validator using XML documents. The Enforcer API includes three classes that allow you to manipulate XML. This chapter provides a reference for methods in these classes in C++.
Chapter 7, The Logger API	The Logger API is a robust logging framework used by both Enforcer plugins and Policy Validator plugins. It allows you to filter log messages and forward them to multiple log destinations. This chapter provides the reference to the functions in this API.

Table 1 Chapters in This Guide (cont'd)

Chapter	Description
Chapter 8, Exception Handling API	The Exception Handling API provides one class which is used to handle exceptions. This chapter provides a reference for functions in this single class.
Chapter 9, C/C++ Constant Strings	The Enforcer plugin uses a large number of constants that are repeatedly copied back and forth. These constants are defined in a separate header file, `EnforcerPredefTable.h`. This chapter provides a reference to C/C++ constant strings.
Section Two: The Java APIs	
Chapter 10, Java Enforcer API	The Enforcer API allows you to integrate Select Access security services with most products and custom components. This chapter is the reference you can use with the Java version of the Enforcer API, so you can provide security services for Java objects.
Chapter 11, 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.
Chapter 12, Java XML Manipulation API	The Enforcer plugin communicates with the Policy Validator using XML documents. The Enforcer API includes three classes that allow you to manipulate XML. This chapter provides a reference for methods in these classes in Java.
Chapter 13, Variable Data Strings	A number of variables are used repeatedly throughout the Select Access code. These common variables are predefined in the `DataStrings.java` file. This chapter provides a reference for these string names and their values in Java.
Appendix A, Authentication Failure Reasons	Policy Validator reply can contain reasons for authentication failure. These reasons are included in an `auth_fail_reason` parameter.

2 Getting Started with Select Access APIs

Before adding or extending Select Access components, you should be familiar with how each of the principle components works. This chapter introduces you to these components.

Chapter Overview

Select Access core components make for a sophisticated and consistent architecture. However, with additional customizable plugins, you can easily adapt to any existing network infrastructure. Contents of this chapter include:

Core Select Access Components

Select Access delivers the core of its authorization and authentication functionality with the following technical components:

- Policy Builder: Allows full or delegated administrators to define the authentication methods and authorization policies with an easy-to-use administration grid.
- Policy Validator: Serves the access decision to the Enforcer plugin after it accepts and evaluates the identity's access request with the policy information retrieved from the directory server that holds your Policy Store.
- Enforcer plugin: Acts as the agent for Select Access on the web/application server. The Enforcer plugin enforces the outcome of the access request that has been evaluated by the Policy Validator.

Customizing Select Access with Plugins

Using the various programming API libraries available, you can create plugins which can tailor Select Access to fit the very specific needs of your organization. Table 2 describes the kinds of plugins you can create.

Table 2 Customizable Plugins Overview

Plugin	Description	Details
Plugins to add additional authentication services to Select Auth		
Policy Validator Authenticator plugin	The Authenticator plugin is developed in C/C++ and added to the Policy Validator. It provides the logic necessary to verify an identity using the new authentication service. Every Authenticator plugin must have a corresponding Authentication plugin on the Policy Builder.	Chapter 4, Policy Validator API
Policy Builder Authentication plugin	The Authentication plugin allows you to create a Java-based property dialog box that is added to the Policy Builder and used to configure a new authentication service. Every Authentication plugin must have a corresponding Authenticator plugin on the Policy Validator.	Chapter 11, Policy Builder API
Plugins to add additional methods of controlling access to resources		
Policy Validator Decider plugin	The Decider plugin is added to the Policy Validator to evaluate its corresponding decision point. Every Decider plugin must have a corresponding Decision Point plugin on the Policy Builder.	Chapter 4, Policy Validator API
Rule Builder Decision Point plugin	Decision Point plugins are added to the Rule Builder utility in the Policy Builder. They define certain criteria which is evaluated by the Decider plugin. Once the Decider plugin determines whether or not the conditions in the Decision Point plugin have been met, the Policy Validator can proceed to evaluate the next node in the rule tree, eventually either allowing or denying access. Every Decision Point plugin must have a corresponding Decider plugin on the Policy Validator.	Chapter 11, Policy Builder API

Table 2 Customizable Plugins Overview (cont'd)

Plugin	Description	Details
Plugins to add protection for specific network resources		
Enforcer plugin	An Enforcer plugin is used to control access to a network resource. Each Enforcer plugin is unique depending on the service being secured and the server APIs available. There are currently three implementations of the Enforcer API: • A C/C++ implementation allows you to create plugins for Solaris, Windows 2000/XP, and Linux. • A Java implementation provides support for EJB applications and servlets. A COM implementation provides support for interoperability between the Enforcer C/C++ API and Windows scripting languages, such as VBScript or JScript.	Chapter 3, C/C++ Enforcer API

The Programming APIs

Select Access provides API libraries in the following languages:

- C/C++: Used to create Enforcer plugins, as well as to create Authenticator and Decider plugins for the Policy Validator. See Table 3 on page 16 for an overview of these libraries.
- Java: Used to create Enforcer plugins, as well as the Authentication and Decision Point plugins for the Policy Builder. See Table 4 on page 18 for an overview of these libraries.

Table 3 C/C++ API Libraries

Library	Description	Details
C/C++ Enforcer API	A collection of standalone C functions that provide core Enforcer plugin functionality, including: • Initializing and configuring an Enforcer object. • Initializing and sending the query. • Parsing of URL request data.	Chapter 3, C/C++ Enforcer API.
Policy Validator API	Comprised of two C++ classes: • `AuthPlugin`: The base class for creating Authenticator plugins. Each Authenticator plugin corresponds to a Policy Builder Authentication plugin. • `Decider`: The base class for creating Decider plugins. Each Decider plugin corresponds to a Policy Builder Decision Point plugin.	Chapter 4, Policy Validator API
User API	Comprised of three C++ classes used by the Authenticator and Decider plugins to manage identity data: • `User`: Contains information such as what dynamic group and group memberships an identity possesses. You can use this class to determine if an identity is a transient identity, obtain the `UserSource` where the identity is located, and so on. • `UserCache`: Contains all cached identity data, and stores all transient identities. • `UserSource`: Represents a directory server and a location where identities are located. You can use this class to get a list of all the currently configured Identity Sources, locate the specific Identity Source that stores the identity data, or obtain an LDAP connection.	Chapter 5, The User API

Table 3 C/C++ API Libraries (cont'd)

Library	Description	Details
C/C++ XML manipulation API	Comprised of three C++ classes specifically designed to allow Enforcer, Authenticator and Decider plugins to manipulate the XML in Select Access XML documents: • `XmlTreeNode`: Stores the XML nodes of a parsed XML document. This class provides low-level access to XML attributes. • `PropertyElement`: Stores a name/value pair. • `PropertyListElement`: Stores an XML node that contains a list of zero or more properties and/or nested property lists. For more information about XML as it is used in Select Access, see Understanding the Importance of XML in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.	Chapter 6, C++ XML Manipulation API
Logger API	Comprised of eight C++ classes that enable Enforcer, Authenticator and Decider plugins to log messages: • `Logger`: A singleton that allows you to filter and forward messages to multiple log destinations. • `LogFilter`: A singleton that holds a list of channel/level pairs and a destination. • `LogDestination`: An abstract class that defines the destination for a log message. `LogDestination` is the parent class for a number of classes which represent specific destinations: `LogServer`; `LogStdio`, parent to `LogFile` and `LogStderr`; and *`LogSystem`*.	Chapter 7, The Logger API
Exception Handling API	Comprised of one C++ class: • `EnforcerException`: Manages errors generated while using the Enforcer plugin. In addition, this class is the parent class for Decider plugin exceptions.	Chapter 8, Exception Handling API
Constant strings	A list of commonly used constants. To improve performance and minimize memory allocation problems, these constants have been treated as lightweight functions. As a result, plugins using this code are able to use the same memory for all of the constants.	Chapter 9, C/C++ Constant Strings

Table 4 Java API libraries

Library	Description	Details
Java Enforcer API	Comprised of two classes which provide core Enforcer plugin functionality: • `WebTransaction`: Provides a number of abstract methods which must be over-written in application-specific code. Note: The class also contains a number of public methods which are called by `isAuthorized()`, the method responsible for communicating with the Policy Validator and containing most of the Enforcer plugin logic. These methods, including `isAuthorized()`, should never need to be called, and are therefore not documented in this guide. • `Enforcer`: Provides a number of utility methods which support the Enforcer plugin logic, allowing you to, for example, retrieve the Policy Validator handle object, construct XML documents, extract data from the request URL, etc.	Chapter 9, Administration Servlets: the Web Administration API in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*
Policy Builder plugin API	Comprised of the `RuleComponentPanel` class. This class allows you to create the Java property dialog boxes for each new Authentication and Decision Point plugin.	Policy Builder API on page 371
Java XML manipulation API	Comprised of three classes specifically designed to help Enforcer, Authentication and Decision Point plugins manipulate the XML in Select Access XML documents: • `XmlElement`: Stores the XML nodes of a parsed XML document. This class provides low-level access to XML attributes. • `PropertyElement`: Stores a name/value pair. • `PropertyListElement`: Stores an XML node that contains a list of zero or more properties and/or nested property lists. For more information about XML as it is used in Select Access, see Understanding the Importance of XML in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.	Java XML Manipulation API on page 389
Java data strings	A list of commonly used constants.	Variable Data Strings on page 461

Locating the Select Access Code Samples and Libraries

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 5 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.

Section One: The C/C++ APIs

This section describes the Select Access C/C++ API. The Select Access C/C++ API is a collection of libraries used to build Enforcer plugins as well as Policy Validator plugins. Table 6 on page 22 provides a brief description of the functions and classes required to build each plugin type.

Select Access libraries can be used to construct:

- C/C++-based Enforcers: An access control component added to a web server, a Java servlet, an EJB, or any other resource. The Enforcer plugin intercepts access requests, then sends queries to the Policy Validator to enforce access control.
- Policy Validator plugins: Policy Validator plugins come in two kinds:
  - Authenticator plugins: Authenticator plugins allow you to create new ways to verify identities. Using the Policy Builder, you can configure SelectAuth to allow one or more authentication services. When the Policy Validator receives a request to access a protected resource, the Policy Validator determines which authentication services to use, based on SelectAuth configured for the resource. Each authentication method in SelectAuth corresponds to a Policy Validator Authenticator plugin.
  - Decider plugins: Decider 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. Decider plugins comprise the evaluation side of those rules. Each component, or decision point, in a conditional rule corresponds to a Decider plugin.

For a detailed, step-by-step tutorial on constructing these plugins, see Chapter 3, Custom Configuration GUIs: the Policy Builder API in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.

Table 6 C/C++ API Libraries

Class/Utility	Description	Used By: Authenticator plugins	Used By: Decider plugins	Used By: Enforcer plugins	Page
Enforcer plugin API					
Enforcer Plugin Core C Functions	Standalone C functions that provide core Enforcer plugin functionality, such as initializing and configuring an Enforcer object, initializing and sending the query, etc.			•	28
Enforcer Plugin Web Communications Functions	Standalone C functions that provide web communication functionality, such as parsing of URL request data, handling requests from multi-domain single sign-on (MD-SSO) environments, testing URL safety, etc.			•	61
Policy Validator API					
`AuthPlugin`	Defines the base class for creating Authenticator plugins.	•	•		92
`Decider`	Defines the base class for creating Decider plugins.	•	•		112
XML manipulation API					
`XmlTreeNode`	Defines the methods which handle the XML nodes of a parsed XML document.	•	•	•	171
`PropertyElement`	Defines the methods which create a name/value pair.	•	•	•	228
`PropertyListElement`	Defines the methods the create and hand XML nodes that contains a list of zero or more properties and/or nested property lists.	•	•	•	234

Table 6 C/C++ API Libraries (cont'd)

Class/Utility	Description	Used By: Authenticator plugins	Used By: Decider plugins	Used By: Enforcer plugins	Page
User API					
`User`	Defines methods which handle identity information such as what dynamic group and group memberships an identity possesses, whether an identity is a transient identity, where the identity is located, and so on.	•	•		118
`UserCache`	Defines methods which handle all cached identity data, and stores all transient identities.	•	•		145
`UserSource`	Defines methods which handle directory servers and locations where identities are located. You can use this class to get a list of all the currently configured Identity Sources, locate the specific Identity Source that stores the identity data, or obtain an LDAP connection.	•	•		155
Logger API					
`Logger`	Defines methods that allow you to filter and forward messages to multiple log destinations.	•	•	•	252
`LogFilter`	Defines methods which create and handle a list of channel/level pairs and a destination.	•	•	•	262
`LogDestination`	Defines the destination for a log message. Overridden by instances of its child classes, listed below.	•	•	•	267
`LogServer`	Defines the SOAP audit server as the destination for a log message.	•	•	•	269
`LogStdio`	Defines a C standard I/O file pointer as the destination for a log message. Used as a parent place holder for `LogServer` and `LogStderr`.	•	•	•	272

Table 6 C/C++ API Libraries (cont'd)

Class/Utility	Description	Used By: Authenticator plugins	Used By: Decider plugins	Used By: Enforcer plugins	Page
`LogServer`	Defines a given file as the destination for a log message.	•	•	•	269
`LogStderr`	Defines the standard error output as the destination for a log message.	•	•	•	278
`LogSystem`	Defines the system log (syslog on UNIX, Event Viewer on Windows) as the destination for a log message.	•	•	•	281
Exception Handling API					
`EnforcerException`	Manages errors generated while using the Enforcer plugin. In addition, this class is the parent class for decider plugin exceptions.		•	•	285
Constant strings					
`C/C++ Constant Strings`	A list of commonly used constants. To improve performance and minimize memory allocation problems, these constants are treated as lightweight functions. As a result, plugins using this code are able to use the same memory for all of the constants.	•	•	•	297

3 C/C++ Enforcer API

The Enforcer API allows you to integrate Select Access security services with most products and custom components. This chapter is the reference you can use with the Java version of the Enforcer API, so you can provide security services for C/C++ objects.

While the libraries used to develop Enforcer plugins are a blend of C and C++, the infrastructure of the Enforcer plugin is built upon a framework of stand-alone C functions.

Chapter Overview

Topics in this chapter include subjects that describe how to use the Enforcer API to create custom plugins:

Including the Correct Header Files

The functions upon which the Enforcer plugin framework is built are defined in two header files: `enforcer.h` and `enforcer_web.h`. These files must be included in all C/C++ Enforcer plugins.

Table 7 Enforcer API Header Files

Header File	Description	Page
`enforcer.h`	Functions defined in this file are responsible for: • Initializing the `EnforcerHandle` object. • Managing the connections to one or more Policy Validators. • Initializing and sending the query document. • Waiting for the response from the Policy Validator. This file also defines a number of constants which set system and programming language-dependent values.	28
`enforcer_web.h`	Functions defined in this file are concerned with managing the incoming HTTP request. These functions allow you to: • Extract query data from the URL. • Handle requests from multi-domain single sign-on environments. • Test URLs to determine whether they should be handled in a special way, and to ensure they are safe.	61

Enforcer Plugin Core C Functions

The Enforcer plugin core functions are defined in the `enforcer.h` file. These are standalone C routines that perform the base Enforcer plugin tasks, such as:

- Initializing and freeing the `EnforcerHandle` object, which stores, among other things, information about the Enforcer plugin's connections to one or more Policy Validators.
- Initializing and creating skeleton XML query documents.
- Sending queries to the Policy Validator, and determining the action that the Enforcer plugin should take based on the response it receives.
- Initializing the logging framework.
- Getting configuration information, such as the query and debug levels the Enforcer plugin is configured to use.

Definitions

A number of definitions have been included to set system- and programming language-dependent values. The `enforcer.h` file defines constant values for the following:

Table 8 Definition Overview

<table>
<tr><th>Definition Type</th><th>Description</th></tr>
<tr><td>Enforcer actions</td><td>The <code>enforcer.h</code> file defines an enumerated data type called <code>EnforcerAction</code>, that identifies the four possible actions the Policy Validator can return for any query the Enforcer plugin sends. Defined values are listed below:
<table>
<tr><th>Name</th><th>Value</th><th>Description</th></tr>
<tr><td><code>ENFORCER_ALLOW_ACTION</code></td><td><code>0</code></td><td>Access is granted.</td></tr>
<tr><td><code>ENFORCER_DENY_ACTION</code></td><td><code>1</code></td><td>Access is denied.</td></tr>
<tr><td><code>ENFORCER_ERROR_ACTION</code></td><td><code>2</code></td><td>Access could not be granted due to an error during processing. If any of the arguments are <code>NULL</code>, this value is returned.</td></tr>
<tr><td><code>ENFORCER_USER_DEFINED</code></td><td><code>3</code></td><td>Signals a site-specific request for further information.</td></tr>
</table>
</td></tr>
</table>

Table 8 Definition Overview (cont'd)

<table>
<tr><th>Definition Type</th><th>Description</th></tr>
<tr><td>HTML form names</td><td>Based on the reply received from the Policy Validator, the Enforcer plugin will present the end-user with one of a number of HTML forms. These forms can request additional information from the end-user, inform the identity of a login or authentication error, confirm an identity action, etc. Defined values are listed below:
<table>
<tr><th>Name</th><th>Value</th><th>Represents the form sent when...</th></tr>
<tr><td>ENFORCER_FORM_PROFILE_MGMT_OK_FORM</td><td>profile_ok_form.html</td><td>Updates to an identity's profile are successful.</td></tr>
<tr><td>ENFORCER_FORM_PROFILE_MGMT_ERROR_FORM</td><td>profile_error_form.html</td><td>An error occurred during an identity profile update.</td></tr>
<tr><td>ENFORCER_FORM_PROFILE_MGMT_NO_USER_FORM</td><td>profile_no_user_form.html</td><td>No profile exists for the identity making the request.</td></tr>
<tr><td>ENFORCER_FORM_PASSWORD_MGMT_ACCOUNT_DISABLED</td><td>password_account_disabled_form.html</td><td>A password change is denied because the profile has been disabled.</td></tr>
<tr><td>ENFORCER_FORM_PASSWORD_CHANGED</td><td>password_changed_form.html</td><td>The password has been changed.</td></tr>
<tr><td>ENFORCER_DENY_FORM</td><td>deny.html</td><td>Authentication fails.</td></tr>
<tr><td>ENFORCER_ACCEPTED_FORM</td><td>accepted.html</td><td>Authentication is successful.</td></tr>
<tr><td>ENFORCER_MULTIAUTH_FORM</td><td>multiauth_form.html</td><td>The requested resource is protected by multiple authentication services.</td></tr>
<tr><td>ENFORCER_SUBMIT_BUTTON</td><td>.submit</td><td>Represents the button used to submit form data.</td></tr>
</table></td></tr>
</table>

Table 8 Definition Overview (cont'd)

<table>
<tr><th>Definition Type</th><th>Description</th></tr>
<tr><td>Logging Levels</td><td>Used to translate the severity level of errors generated by the Enforcer plugin into a value understood by the system log (syslog on UNIX; Event Viewer on Windows). Defined values are listed below:
<table>
<tr><th>Name</th><th>UNIX</th><th>Windows</th><th>Description</th></tr>
<tr><td>ENFORCER_LOG_FATAL</td><td>LOG_EMERG</td><td>0</td><td>Fatal errors.</td></tr>
<tr><td>ENFORCER_LOG_ERROR</td><td>LOG_ERR</td><td>1</td><td>Non-fatal errors.</td></tr>
<tr><td>ENFORCER_LOG_WARNING</td><td>LOG_WARNING</td><td>2</td><td>Errors.</td></tr>
<tr><td>ENFORCER_LOG_INFO</td><td>LOG_INFO</td><td>3</td><td>Information only.</td></tr>
<tr><td>ENFORCER_LOG_DEBUG</td><td>LOG_DEBUG</td><td>4</td><td>Debugging information.</td></tr>
</table></td></tr>
<tr><td>Personalization prefix</td><td>Select Access allows you to define identity-specific attributes for any identity and store it in the Policy store. When personalization has been enabled, the Enforcer plugin can export these identity-specific attributes as environment variables through HTTP headers.
Select Access identifies personalization attributes by attaching a prefix to the attribute values. The constant ENFORCER_P13NPREFIX represents this prefix, defined as SA.</td></tr>
</table>

Table 8 Definition Overview (cont'd)

<table>
<tr><th>Definition Type</th><th>Description</th></tr>
<tr><td>Query levels</td><td>The Enforcer plugin allows you to specify how much information will be included in a query.
<table>
<tr><th>Name</th><th>Value</th><th>Description</th></tr>
<tr><td>QUERY_MINIMAL</td><td>0</td><td>Sends the least amount of data to the Policy Validator. This data includes:
site_data
service
path
All related authentication elements</td></tr>
<tr><td>QUERY_REGULAR</td><td>1</td><td>Sends standard query data, which includes:
all of the minimal elements
http_query
http_query_list
method
dstIP and srcIP
dstPort and srcPort
dstHost
protocol</td></tr>
<tr><td>QUERY_MAXIMAL</td><td>2</td><td>Sends all available data, which includes:
all of the minimal and regular elements
http_header_list
server
srcHost</td></tr>
</table></td></tr>
<tr><td>Return Values</td><td>Two return variables, SUCCESS and FAILURE, have been defined as 1 and 0, respectively.</td></tr>
</table>

Table 8 Definition Overview (cont'd)

Definition Type	Description
XML node naming	The Enforcer C/C++ API uses pointers to various objects throughout the API code. Object pointers are provided with both a C and C++ definitions to ensure that they can be understood by either language. Defined values are listed below:

Name	C Definition	C++ Definition
`XMLnode`	`struct XmlTreeNode *`	`class XmlTreeNode *`
`const ConstXMLnode`	`const struct XmlTreeNode *`	`const class XmlTreeNode *`
`EnforcerQueue`	`struct Enforcer ThreadQueue *`	`class Enforcer ThreadQueue *`
`const ConstEnforcer Queue`	`const struct EnforcerThread Queue *`	`const class EnforcerThread Queue *`

Functions

The Enforcer core functions defined in the `enforcer.h` file are listed below. They are described in the sections that follow..

Table 9 Functions Overview

Function	Description	Page
`EnforcerAddrsQueryInit()`	Given the source and destination endpoint addresses, this function allocates and initializes a query XMLnode.	35
`EnforcerConfigInit()`	Initializes and configures the `EnforcerHandle` object using the given configuration file. This function returns the `EnforcerHandle` object that it creates.	37
`EnforcerDocParseFile()`	Creates an XML document by parsing the contents of the given file.	38
`EnforcerDocParseMem()`	Creates an XML document by parsing the contents of a memory buffer.	39
`EnforcerFree()`	Releases all the global resources as well as those associated with the `EnforcerHandle` object.	39
`EnforcerGetDebugLevel()`	Gets the debug level setting. The debug level specifies how much debug information will be included.	41
`EnforcerGetEnableCookies()`	Determines whether Enforcer plugin cookie support is enabled.	42

Table 9 Functions Overview (cont'd)

Function	Description	Page
`EnforcerGetLastError()`	Returns a pointer to the thread-specific `EnforcerErrorData` object, which holds the details about the last Enforcer function that failed.	43
`EnforcerGetP13Ncompat()`	Determines whether old-style personalization will be used.	44
`EnforcerGetQueryLevel()`	Gets the query level setting. The query level specifies how much information will go into the query.	45
`EnforcerInit()`	Initializes the global Enforcer plugin resources and libraries and allocates thread-local storage.	46
`EnforcerLog()`	Logs a message to the Enforcer logging system.	47
`EnforcerLogClear()`	Releases resources associated with the Enforcer plugin logging system.	48
`EnforcerLogInit()`	Initializes the Enforcer plugin logging system.	49
`EnforcerLogTo()`	Initializes the logging system and specifies where messages should be logged to.	50
`EnforcerQueryInit()`	Allocates and creates a skeleton XML document, adding the given parameters as property values.	51
`EnforcerQuerySend()`	Converts the given XML node into an XML string and sends the query to the Policy Validator. It then waits for and parses the reply, returning the action that the Enforcer plugin should take.	53
`EnforcerReplyNeedsAuth()`	Parses the XML reply document from the Policy Validator to determine whether further authentication is required.	54
`EnforcerSimpleQueryInit()`	Given the endpoint IP addresses in string format, allocates and creates a skeleton XML document.	55
`EnforcerSocketCleanup()`	Finalizes the socket library and closes all sockets.	57
`EnforcerSocketInit()`	Initializes the Socket library.	58
`EnforcerTcpQueryInit()`	Given a TCP socket, this function allocates and creates a skeleton XML document.	59

EnforcerAddrsQueryInit()

Description Given the source and destination endpoint addresses, this function allocates and initializes a query XMLnode.

Include File `enforcer.h`

Syntax

```
int EnforcerSimpleQueryInit(
     struct sockaddr_in *   src,
     struct sockaddr_in *   dst,
     const char *          proto,
     const char *          host,
     const char *          path,
     const int             query_level,
     XMLnode *             query_p
);
```

Parameters

- `src` — A pointer to the socket address used by the machine from which the query originated.
- `dst` — A pointer to the socket address used by the machine on which the resource resides.
- `proto` — The protocol used to send the query (for example, `http` or `ftp`).
- `host` — The hostname of the resource. If `NULL`, the function will try to determine the hostname. If the hostname cannot be ascertained, the function will fail.
- `path` — The unescaped path to the resource for which access is being requested.
- `query_level` — The amount of information that will be included in the query. This parameter can have a value of 0, 1, or 2. For information on what data is included with each level, see Query levels on page 32.
- `query_p` — A pointer to the new XML query document.

Return Values

- `1` — The XML document was successfully created.
- `0` — The XML document could not be created.

Remarks This function will create a document similar to the following, if `query_level` is set to `1` (regular):

```
<PolicyValidatorQuery>
    <PROPERTY NAME="service">http://mymenu.food.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="host">mymenu.food.com</PROPERTY>
    <PROPERTY NAME="proto">http</PROPERTY>
</PolicyValidatorQuery>
```

Example The following code sample determines whether the basename of the given URL can be resolved. If not, the request is immediately denied. If the basename can be resolved, it then attempts to initialize the query XML document. If the query cannot be initialized, the request is immediately denied. Otherwise, the code proceeds to add data to the query.

```
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 (cookies != NULL) {
   ap_table_do(add_auth_cookies, *query, cookies, NULL);
}
if (EnforcerGetEnableCookies(Enforcer_Handle)) {
   (*query)->appendChildString(ENFORCER_NATIVE_NONCE,
         SYS_STRDUP(ENFORCER_ENABLE));
}
```

See Also

- `EnforcerQueryInit()` on page 51
- `EnforcerSimpleQueryInit()` on page 55
- `EnforcerTcpQueryInit()` on page 59

EnforcerConfigInit()

Description Initializes and configures the `EnforcerHandle` object using the given configuration file. This function returns the `EnforcerHandle` object that it creates.

Include File

```
enforcer.h
```

Syntax

```
EnforcerHandle *  EnforcerConfigInit(
     const char * configfile,
     const char * loggingName,
     const char * type
);
```

Parameters

- `configfile` — The name of the configuration file to be read. If this parameter is `NULL`, the default configuration file is used.
- `loggingName` — The name used to identify the application when messages are logged.
- `type` — The Enforcer plugin type, for example `apache`, `iis`, `domino`.

Return Values The `EnforcerHandle` object, or `NULL`, if this function fails.

Remarks

- This function should be called after `EnforcerInit()` has been called to initialize the Enforcer plugin libraries.
- The caller must release resources used by an `EnforcerHandle` using `EnforcerFree()`.
- This function will fail if:
  - Memory was not allocated for the `EnforcerHandle`.
  - The Enforcer was not properly configured using the Setup Tool or Policy Builder.

Example The following code sample attempts to initialize the `EnforcerHandle` object. If it can't, it logs an error message and returns `NULL`. If it is successful, it attempts to initialize the `EnforcerHandle` configuration using the default configuration file.

```
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;
}
```

See Also

- `EnforcerFree()` on page 40
- `EnforcerInit()` on page 46

EnforcerDocParseFile()

Description Creates an XML document by parsing the contents of the given file.

Include File `enforcer.h`

Syntax

```
int EnforcerDocParseFile(
     const char * fname,
     XMLnode *    result_p
);
```

Parameters

- `fname` — The file to parse. Must be non-`NULL`.
- `result_p` — A pointer to the result node.

Return Values

- `1` — The file was successfully parsed.
- `0` — The function was either unable to find or unable to parse the file.

Remarks If the file being parsed contains improperly structured XML, this function will parse as much of the file as possible and return a properly formatted XML document.

Memory considerations:

You must `delete[]` a non-`NULL` result.

Example The following code sample determines whether the given XML document can be read. If not, an error message is displayed.

```
XmlTreeNode * content = NULL;
if (! EnforcerDocParseFile(filename, &content)) {
   cerr << "Unable to parse XML file '"
      << filename
      << "'";
   cerr << endl;
   exit(1);
}
```

See Also

- `EnforcerDocParseMem()` on page 39

EnforcerDocParseMem()

Description Creates an XML document by parsing the contents of a memory buffer.

Include File

```
enforcer.h
```

Syntax

```
int EnforcerDocParseMem(
     const char * buff,
     int          len,
     XMLnode *    result_p
);
```

Parameters

- `buff` — The location of the memory buffer to be parsed. Must be non-`NULL`.
- `len` — The number of bytes the XML document is comprised of.
- `result_p` — A pointer to the result node.

Return Values

- `1` — The memory buffer was successfully parsed.
- `0` — The function was either unable to find or unable to parse the memory buffer.

Remarks If the memory buffer contains improperly structured XML, this function will parse as much of the memory as possible and return a properly formatted XML document.

Memory considerations:

You must `delete[]` a non-`NULL` result.

Example The following code sample determines whether the given buffer can be read. If not, an error message is displayed.

```
XmlTreeNode * content = NULL;
if (! EnforcerDocParseMem(buffer, 54, &content)) {
  cerr << "Unable to parse XML file '"
     << filename
     << "'";
  cerr << endl;
  exit(1);
}
```

See Also

- `EnforcerDocParseFile()` on page 38

EnforcerFree()

Description Releases all the global resources as well as those associated with the `EnforcerHandle` object.

Include File

```
enforcer.h
```

Syntax

```
int EnforcerFree(
     EnforcerHandle * eh
);
```

Parameters

- `eh` — A non-null pointer to a previously-initialized `EnforcerHandle` object.

Return Values

- `1` — The `EnforcerHandle` object was successfully freed.
- `0` — The `EnforcerHandle` object was not freed.

Example The following code sample frees all resources associated with an `EnforcerHandle` object.

```
EnforcerFree(Enforcer_Handle);
```

EnforcerGetDebugLevel()

Description

Gets the debug level setting. The debug level specifies how much debug information will be included.

Include File

`enforcer.h`

Syntax

```
int EnforcerGetDebugLevel(
     EnforcerHandle * eh
);
```

Parameters

- `eh` — A non-null pointer to a previously-initialized `EnforcerHandle` object.

Return Values

The debug level.

Example

The following code sample retrieves the debug level and, if the level is greater than 9, it adds a property to the query document which enables tracing.

```
if (EnforcerGetDebugLevel(Enforcer_Handle) > 9)
   (*query)->appendChildString(ENFORCER_TRACE,
    SYS_STRDUP(ENFORCER_ENABLE));
```

EnforcerGetEnableCookies()

Description Determines whether Enforcer plugin cookie support is enabled.

Include File

```
enforcer.h
```

Syntax

```
int EnforcerGetEnableCookies(
     EnforcerHandle * eh
);
```

Parameters

- `eh` — A non-null pointer to a previously-initialized `EnforcerHandle` object.

Return Values

- `1` — Cookies are enabled.
- `0` — Cookies are not enabled.

Example The following code sample determines whether cookies should be enabled. If so, it adds a property to the query document which enables web session cookies.

```
   if (EnforcerGetEnableCookies(Enforcer_Handle)) {
     (*query)->appendChildString(ENFORCER_NATIVE_NONCE,
SYS_STRDUP(ENFORCER_ENABLE));
   }
```

EnforcerGetLastError()

Description Returns a pointer to the thread-specific `EnforcerErrorData` object, which holds the details about the last Enforcer function that failed.

Include File `enforcer.h`

Syntax

```
const EnforcerErrorData * EnforcerGetLastError(void);
```

Parameters None.

Return Values A pointer to the error in `EnforcerErrorData`, or `&noError`, if no error is found.

Example The following code sample gets the details about the last Enforcer function that failed, then logs an error message which specifies the reason why a given string could not be converted to UTF-8.

```
if (dwNumChars == 0) {
   DWORD errcode = GetLastError();
   const char *errptr = "Unknown";
   switch(errcode) {
   case ERROR_INSUFFICIENT_BUFFER:
      errptr = "Insufficient buffer space";
      break;
   case ERROR_INVALID_FLAGS:
      errptr = "Invalid Flags";
      break;
   case ERROR_INVALID_PARAMETER:
      errptr = "Invalid Parameter";
      break;
   }
   EnforcerLog(ENFORCER_LOG_ERROR, "MakeUTF8: error converting string:
      %s", errptr);
   return NULL;
}
```

EnforcerGetP13Ncompat()

Description Determines whether old-style personalization will be used.

Include File `enforcer.h`

Syntax

```
int EnforcerGetp13Ncompat(
     EnforcerHandle * eh
);
```

Parameters

- `eh` — A non-null pointer to a previously-initialized `EnforcerHandle` object.

Return Values

- `0` — Old-style personalization will be used.
- `1` — Old-style personalization will not be used.

Example The following code sample determines which personalization format should be used, then adds the personalization parameters accordingly.

```
if(EnforcerGetP13Ncompat(Enforcer_Handle)) {
   ap_table_add(r->headers_in, name, (char*) val);
}
```

EnforcerGetQueryLevel()

Description Gets the query level setting. The query level specifies how much information will go into the query.

Include File `enforcer.h`

Syntax

```
int EnforcerGetQueryLevel(
     EnforcerHandle * eh
);
```

Parameters

- `eh` — A non-null pointer to a previously-initialized `EnforcerHandle` object.

Return Values The query level, which may be one of the following values:

- `0` — Minimal.
- `1` — Regular.
- `2` — Maximal.

Remarks For more information on what data is included in a query for each query level, see Query levels on page 32.

Example The following code sample retrieves the query level setting, then determines whether the level has been set to `MAXIMAL`. If so, it creates a new XML property list, adds it to the query, and populates it with header list data, then appends a property specifying the version of the Enforcer plugin.

```
int query_level = EnforcerGetQueryLevel(Enforcer_Handle);
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));
}
```

See Also

- Query levels on page 32

EnforcerInit()

Description Initializes the global Enforcer plugin resources and libraries and allocates thread-local storage.

Include File `enforcer.h`

Syntax

```
int EnforcerInit();
```

Parameters None.

Return Values

- `1` — Initialization was successful.
- `0` — An initialization error occurred. This function will fail if the Socket library has not previously been initialized.

Remarks

- This function should be called before calling `EnforcerConfigInit()`.
- The caller must release resources using `EnforcerFree()`.

Example The following code sample attempts to initialize the `EnforcerHandle` object. If it is unsuccessful, it logs an error message and returns `NULL`. If it is successful, it attempts to initialize the `EnforcerHandle` configuration using the default configuration file.

```
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;
}
```

See Also

- `EnforcerConfigInit()` on page 37
- `EnforcerFree()` on page 40
- `EnforcerSocketInit()` on page 58

EnforcerLog()

Description Logs a message to the Enforcer logging system.

Include File

```
enforcer.h
```

Syntax

```
void EnforcerLog(
     int          level,
     const char * fmt,
     .
     .
     .
);
```

Parameters

- `level` — The severity level of the event to be logged. See Logging Levels on page 31 for valid values for this parameter.
- `fmt` — The format of the log entry. This parameter and any following arguments are interpreted as in the `scanf()` and related functions. Refer to documentation on the `scanf()` function for more information.

Return Values None.

Remarks

- If the Logging system has been initialized using either `EnforcerLogInit()` or `EnforcerLogTo()`, the defaults will be used to initialize logging.

Example The following code sample determines whether the given URL contains characters that could be unsafe. If so, it logs a warning message and rejects the request.

```
if(EnforcerIsEvilURL(r->uri, Enforcer_Handle)) {
  EnforcerLog(ENFORCER_LOG_WARNING, "rejecting suspicious url '%s'",
    r->uri);
  return HTTP_FORBIDDEN;
}
```

See Also

- `EnforcerLogInit()` on page 49
- `EnforcerLogTo()` on page 50
- `::vlog()` on page 260

EnforcerLogClear()

Description Releases resources associated with the Enforcer plugin logging system.

Include File `enforcer.h`

Syntax

```
void EnforcerLogClear();
```

Parameters None.

Return Values None.

Remarks This function is called by `EnforcerFree()`.

Example The following code sample initializes the logging system and sets the default log location as the system log (syslog in UNIX, Windows Event Viewer).

```
EnforcerLogInit(logtag, debug);
EnforcerLogTo(ENFORCER_LOGTO_SYSTEM, 0);
.
.
.
EnforcerLogClear()
```

See Also `EnforcerFree()` on page 40

EnforcerLogInit()

Description Initializes the Enforcer plugin logging system.

Include File

```
enforcer.h
```

Syntax

```
int EnforcerLogInit(
     char *     progname,
     int        debug
);
```

Parameters

- `progname` — Identifies the program that is logging the messages. If it is `NULL`, the default logging name, HP OpenView Select Access Enforcer, will be used.
- `debug` — Specifies whether or not debug messages will be printed to the log. This parameter can have one of the following values:
  - `1`—Debugging messages are printed to the log.
  - `0`—Debugging messages are not printed to the log.

Return Values

- `1` — Initialization is successful.
- `0` — Initialization was unsuccessful due to an error.

Remarks Messages in the system log (syslog in UNIX; Event Log in Windows) will contain `progname`.

Example The following code sample initializes the logging system and sets the default log location as the system log (syslog in UNIX, Windows Event Viewer).

```
EnforcerLogInit(logtag, debug);
EnforcerLogTo(ENFORCER_LOGTO_SYSTEM, 0);
.
.
.
EnforcerLogClear()
```

See Also

- `EnforcerLogClear()` on page 48

EnforcerLogTo()

Description Initializes the logging system and specifies where messages should be logged to.

Include File

```
enforcer.h
```

Syntax

```
int EnforcerLogTo(
     int          destcode,
     const char * filename
);
```

Parameters

- `destcode` — A code which specifies where messages should be logged. This value should be a bitwise OR one of the following values:
  - `ENFORCER_LOGTO_STDERR` — Write log to standard error.
  - `ENFORCER_LOGTO_SYSTEM` — Write to syslog on UNIX, or Event Log on Windows.
  - `ENFRORCER_LOGTO_FILENAME` — Write to the log defined by the `filename` parameter.
  - `0` — Disable logging.
- `filename` — The name of the file that log messages will be saved to. Only used if `destcode` has a value of `ENFORCER_LOGTO_FILENAME`.

Return Values

- `1` — Initialization was successful.
- `0` — Initialization was unsuccessful due to an error.

Remarks This call sets up a new log filter based on the specified destination.

Example The following code sample initializes the logging system and sets the default log location as the system log (syslog in UNIX, Windows Event Viewer).

```
EnforcerLogInit(logtag, debug);
EnforcerLogTo(ENFORCER_LOGTO_SYSTEM, 0);.
.
.
EnforcerLogClear()
```

See Also

- `EnforcerLogClear()` on page 48
- `EnforcerLogInit()` on page 49

EnforcerQueryInit()

Description Allocates and creates a skeleton XML document, adding the given parameters as property values.

Include File `enforcer.h`

Syntax

```
int EnforcerQueryInit(
     const char * queried,
     const char * service
     const char * path,
     XMLnode *    query_p
);
```

Parameters

- `queryID` — An arbitrary label used to coordinate a response with a query. This value can be any arbitrary string. The Policy Validator passes it back in the reply.
- `service` — The name of the service for which access is being requested. For example, `http://mymenu.foodcompany.com:8070`.
- `path` — The path to the resource for which access is being requested. For example, `/index.html`.
- `query_p` — A pointer to the new XML query document.

Return Values

- `1` — The XML document was successfully created.
- `0` — The XML document could not be created.

Remarks This function will create a document similar to the following:

```
<PolicyValidatorQuery>
  <PROPERTY NAME="service">http://mymenu.food.com:8070</PROPERTY>
  <PROPERTY NAME="path">/test/auth/submit.cgi</PROPERTY>
  <PROPERTY NAME="queryID">2</PROPERTY>
</PolicyValidatorQuery>
```

Example The following code sample determines whether the basename of the given URL can be resolved. If not, the request is immediately denied. If the basename can be resolved, it then attempts to initialize the query XML document. If the query cannot be initialized, the request is immediately denied. Otherwise, the code proceeds to add data to the query.

```
if (!EnforcerGetBasenameOfURL(uri, &base_url, &url_changed))
  return 0;
if (!EnforcerQueryInit(id, service, base_url, query)) {
  if (url_changed)
    delete base_url;
  return 0;
}
if (url_changed)
  delete base_url;
if (strcmp(auth_type, "Basic") == 0) {
  add_basic_auth_data(&request_info, query);
}
if (cookies != NULL) {
  add_cookie_data(cookies, query);
if (EnforcerGetEnableCookies(Enforcer_Handle)) {
  /* enable web session cookies */
  (*query)->appendChildString(ENFORCER_NATIVE_NONCE,
    SYS_STRDUP(ENFORCER_ENABLE));
```

```
}
```

See Also

- `EnforcerAddrsQueryInit()` on page 35
- `EnforcerSimpleQueryInit()` on page 55
- `EnforcerTcpQueryInit()` on page 59

EnforcerQuerySend()

Description Converts the given XML node into an XML string and sends the query to the Policy Validator. It then waits for and parses the reply, returning the action that the Enforcer plugin should take.

Include File `enforcer.h`

Syntax

```
EnforcerAction EnforcerQuerySend(
     EnforcerHandle * eh,
     XMLnode          query,
     XMLnode *        reply
);
```

Parameters

- `eh` — A non-null pointer to a previously-initialized `EnforcerHandle` object.
- `query` — A pointer to the query node.
- `*reply` — A pointer to the reply node.

Return Values The Enforcer action to be taken. The returned value can be one of:

- `ENFORCER_ALLOW_ACTION` — Access is granted.
- `ENFORCER_DENY_ACTION` — Access is denied.
- `ENFORCER_ERROR_ACTION` — Access could not be granted due to an error during processing. If any of the arguments are `NULL`, this value is returned.
- `ENFORCER_USER_DEFINED` — Signals a site-specific request for further information.

Remarks If `reply` returns `NULL`, then a communication or other error occurred.

Example The following code sample sends the query, then refuses access to any request that doesn't return an Allow action from the Policy Validator.

```
returnAction = EnforcerQuerySend(eh, query, &reply);
delete query;
delete reply;
EnforcerFree(eh);

if (returnAction != ENFORCER_ALLOW_ACTION) {
  EnforcerLog(LOGTYPE, "denied");
  return 0;
}
```

EnforcerReplyNeedsAuth()

Description Parses the XML reply document from the Policy Validator to determine whether further authentication is required.

Include File

```
enforcer.h
```

Syntax

```
int EnforcerReplyNeedsAuth(
     XMLnode reply
);
```

Parameters

- `reply` — A pointer to the reply node.

Return Values

- `1` — Further authentication is required.
- `0` — No authentication is required.

Example The following code sample checks the Policy Validator reply to determine whether additional authentication is needed.

```
if (EnforcerReplyNeedsAuth(reply)) {
  return SUCCESS;
}
```

EnforcerSimpleQueryInit()

Description Given the endpoint IP addresses in string format, allocates and creates a skeleton XML document.

Include File `enforcer.h`

Syntax

```
int EnforcerSimpleQueryInit(
     char *          src_ip,
     char *          dst_ip,
     int             src_port,
     int             dst_port,
     const char *    proto,
     const char *    host,
     const char *    path,
     const int       query_level,
     XMLnode *       query_p
);
```

Parameters

- `src_ip` — The IP address of the machine from which the query originated.
- `dst_ip` — The IP address of the machine on which the resource resides.
- `src_port` — The port number being used by the machine from which the query originated.
- `dst_port` — The port number being used by the resource.
- `proto` — The protocol used to send the query. For example, `http` or `ftp`.
- `host` — The hostname of the resource. If `NULL`, the function will try to determine the hostname. If the hostname cannot be ascertained, the function will fail.
- `path` — The path to the resource for which access is being requested.
- `query_level` — The amount of information that will be included in the query. This parameter may be one of the following values:
  - `0` — Minimal.
  - `1` — Regular.
  - `2` — Maximal.
- `query_p` — A pointer to the new XML query document.

Return Values

- `1` — The XML document was successfully created.
- `0` — The XML document could not be created.

Remarks

- The value for the service being requested is constructed from the `proto`, `host`, and `dst_port` values.
- For detailed information on what data is included for each query level, see Query levels on page 32.
- This function will create a document similar to the following, if `query_level` is set to `1` (regular):

```
<PolicyValidatorQuery>
  <PROPERTY NAME="service">http://mymenu.food.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="host">mymenu.food.com</PROPERTY>
    <PROPERTY NAME="proto">http</PROPERTY>
  </PolicyValidatorQuery>
```

Example The following code sample determines whether the basename of the given URL can be resolved. If not, the request is immediately denied. If the basename can be resolved, it then attempts to initialize the query XML document. If the query cannot be initialized, the request is immediately denied. Otherwise, the code proceeds to add data to the query.

```
if (!EnforcerGetBasenameOfURL(uri, &base_url, &url_changed))
  return 0;
if (!EnforcerSimpleQueryInit(src_ip, dst_ip, src_port, dst_port,
    proto, host, base_url, query_level, query)) {
  if (url_changed)
    delete base_url;
  return 0;
}
if (url_changed)
  delete base_url;
if (strcmp(auth_type, "Basic") == 0) {
  add_basic_auth_data(&request_info, query);
}
if (cookies != NULL) {
  add_cookie_data(cookies, query);
if (EnforcerGetEnableCookies(Enforcer_Handle)) {
  /* enable web session cookies */
  (*query)->appendChildString(ENFORCER_NATIVE_NONCE,
    SYS_STRDUP(ENFORCER_ENABLE));
} }
```

See Also

- `EnforcerAddrsQueryInit()` on page 35
- `EnforcerQueryInit()` on page 51
- `EnforcerTcpQueryInit()` on page 59
- Query levels on page 32

EnforcerSocketCleanup()

Description Finalizes the socket library and closes all sockets.

Include File `enforcer.h`

Syntax
```
int EnforcerSocketCleanup(void);
```

Parameters None.

Return Values
- `1` — If successful.
- `0` — If unsuccessful.

Remarks This function is called from `EnforcerFree()`.

Example The following code sample gets rid of the `enforcerhandle` object then clears the log and socket library.

```
free((void *)eh);
EnforcerLogClear();
curl_global_cleanup();
EnforcerSocketCleanup();
```

See Also `EnforcerFree()` on page 40

EnforcerSocketInit()

Description Initializes the Socket library.

Include File `enforcer.h`

Syntax

```
int EnforcerSocketInit(void);
```

Parameters None.

Return Values

- `1` — If successful.
- `0` — If unsuccessful.

Remarks This function is called from `EnforcerInit()`.

Example The following code attempts to initialize the socket library and logs an error if unsuccessful.

```
if () {
     EnforcerLog(ENFORCER_LOG_ERROR, "EnforcerSocketInit failed");
     return FAILURE;
  }
```

See Also

- `EnforcerInit()` on page 46

EnforcerTcpQueryInit()

Description Given a TCP socket, this function allocates and creates a skeleton XML document.

Include File `enforcer.h`

Syntax

```
int EnforcerTcpQueryInit(
     int sock,
     const char * proto,
     const char * host,
     const char * path,
     const int    query_level,
     XMLnode *    query_p
);
```

Parameters

- `sock` — The TCP socket being used.
- `proto` — The protocol used to send the query. For example, `http` or `ftp`.
- `host` — The virtual hostname of the resource. If `NULL`, the function will try to determine the hostname. If the hostname cannot be ascertained, the function will fail.
- `path` — The path to the resource for which access is being requested.
- `query_level` — The amount of information that will be included in the query. This parameter may be one of the following values:
  - `0` — Minimal.
  - `1` — Regular.
  - `2` — Maximal.
- `query_p` — A pointer to the new XML query document.

Return Values

- `1` — The XML document was successfully created.
- `0` — The XML document could not be created.

Remarks This function will create a document similar to the following, if `query_level` is set to `1` (regular):

```
<PolicyValidatorQuery>
  <PROPERTY NAME="service">http://mymenu.food.com:8070</PROPERTY>
  <PROPERTY NAME="path">/test/auth/submit.cgi</PROPERTY>
  <PROPERTY NAME="socket">??????????</PROPERTY>
  <PROPERTY NAME="host">mymenu.food.com</PROPERTY>
  <PROPERTY NAME="proto">http</PROPERTY>
</PolicyValidatorQuery>
```

Example The following code sample determines whether the basename of the given URL can be extracted. If not, the request is immediately denied. If the basename can be resolved, it then attempts to initialize the query XML document. If the query cannot be initialized, the request is immediately denied. Otherwise, the code proceeds to add data to the query.

```
if (!EnforcerGetBasenameOfURL(uri, &base_url, &url_changed))
  return REQ_ABORTED;
if (keysize != NULL) {
  *proto = "https";
}
if (!EnforcerTcpQueryInit(sock, *proto, server_hostname, base_url,
     query_level, query)) {
```

```
    if (url_changed)
      delete base_url;
    return REQ_ABORTED;
  }
  if (cookies != NULL) {
    add_cookie_data(sn, rq, cookies, query);
  }
  if (EnforcerGetEnableCookies(Enforcer_Handle)) {
    (*query)->appendChildString(ENFORCER_NATIVE_NONCE,
      SYS_STRDUP(ENFORCER_ENABLE));
  }
```

See Also

- `EnforcerAddrsQueryInit()` on page 35
- `EnforcerQueryInit()` on page 51
- `EnforcerSimpleQueryInit()` on page 55

Enforcer Plugin Web Communications Functions

The `enforcer_web.h` file defines a number of stand-alone routines that allow the Enforcer plugin to handle HTTP requests. These functions include:

- Handling HTTP requests from domains that are part of a single sign-on environment.
- Testing URLs to determine whether they should be handled in a special way, as well as ensuring they are safe.
- Setting the Enforcer plugin to function as a proxy.

Functions

The Enforcer web communication functions are defined in the `enforcer_web.h` file and are listed below. They are described in the sections that follow.

Table 10 Functions Overview

Function	Description	Page
`EnforcerAddEncoded()`	Converts the value of the given `val` parameter to UTF-8 and adds it to the specified XML object.	64
`EnforcerAddFormData()`	Adds POSTed form data to a nested property list in the XML query document.	65
`EnforcerAddMultiResource()`	Adds the given resource to multi-resource property list in the given XML query document. If the query does not already contain a multi-resource list, one will be added.	66
`EnforcerAddQueryData()`	Adds HTTP query data to a nested property list in the XML query document.	67
`EnforcerAddUrlEncoded()`	Parses the given URL-encoded string, and adds the attribute/value pairs it contains to a new nested property list in the given query.	68
`EnforcerAddWebHints()`	Adds web server-specific hints to an XML query.	69
`EnforcerAuthFailed()`	Determines whether the Policy Validator reply indicates that authentication failed.	70
`EnforcerConstructSSOAuthURL()`	In a single sign-on environment, constructs the second redirect URL, which contains a nonce.	71

Table 10 Functions Overview (cont'd)

Function	Description	Page
`EnforcerConstructSSOOrigURL()`	In a single sign-on environment, constructs the third redirect URL, which sends the browser back to the originating URL.	72
`EnforcerConstructSSORedirectURL()`	In a single sign-on environment, constructs the first redirect URL, which requests that the next server sends back a nonce.	73
`EnforcerCouldSSOHelp()`	Tests the given URL to determine whether multi-domain single-sign on (MD-SSO) should be attempted to provide the needed authentication.	74
`EnforcerGetAuthHint()`	Extracts the authentication hints, if any, out of a Policy Validator reply object. If only one hint is present in the reply, it is returned. If more than one hint is present, this function merges them into a list.	75
`EnforcerGetBasenameOfURL()`	Extracts the basename of the given URL by terminating it prior to an anchor symbol (`#`) or `jsessionid` tag.	76
`EnforcerGetDisableCaching()`	Determines whether the given Enforcer plugin is configured to disable caching of Select Access-protected web pages.	77
`EnforcerGetLoginViaForm()`	Determines whether the given Enforcer plugin is configured to use a web form to perform login operations.	78
`EnforcerIsDomainProtected()`	Tests the referring site to determine whether it is in the multi-domain single sign-on (MD-SSO) protected sites list.	79
`EnforcerIsEvilURL()`	Determines whether the URL path is safe.	80
`EnforcerIsIgnoredFile()`	Tests the given filename to determine whether it is in the list of ignored filenames. Files listed in the `ignoredFile` list are unconditionally allowed.	81
`EnforcerIsPassthroughDomain()`	Tests the given virtual hostname to determine whether it is in the list of pass-though domains. Domains listed in the `passthroughDomain` list are unconditionally allowed.	82

Table 10 Functions Overview (cont'd)

Function	Description	Page
`EnforcerIsSSOAuthURL()`	Tests the given URL to determine whether it as an authentication URL, that is, whether it contains a nonce. If so, the `noncePtr` parameter will be pointed to the value of the nonce.	83
`EnforcerIsSSORedirectURL()`	Tests the given URL to determine whether it contains an SSO redirect which requests a nonce. If so, the `urlPtr` parameter will be pointed to the value of the return URL.	84
`EnforcerNeedsRefresh()`	Determines whether the given request needs to trigger a refresh (GET) of the current URL.	85
`EnforcerProxyMode()`	Tests to determine whether the given Enforcer plugin should function as a proxy.	86
`EnforcerTransformPage()`	Loads and transforms an HTML form page.	87
`EnforcerUnescapeUrl()`	Removes all escape sequences from the given URL and replaces them with their single-character, Latin-1 character-set equivalent.	88
`EnforcerWebAuthDecode()`	Decodes the username and password of an authentication header.	89

EnforcerAddEncoded()

Description

Converts the value of the given `val` parameter to UTF-8 and adds it to the specified XML object.

Include File

`enforcer_web.h`

Syntax

```
int EnforcerAddEncoded(
     XMLnode parent,
     const char *     name,
     const char *     val,
     EnforcerHandle * eh,
);
```

Parameters

- `parent` — A pointer to the XML node to which the encoded value will be added.
- `name` — The attribute name.
- `val` — The attribute value to be encoded.
- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.

Return Values

- `1` — An encoded value was successfully added to the XML tree.
- `0` — The value could not be added to the XML tree or could not be encoded.

Remarks

The `name` parameter is intentionally passed on without being encoded.

Example

The following example determines whether the request included a header containing HTTP basic authentication data. If so, it then attempts to extract the name and password parameters from the request, then converts them to UTF-8 and adds them to the XML query document.

```
if ((hvalue = (char *) ap_table_get(r->headers_in, "Authorization"))
        != NULL) {
  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);
  }
}
```

EnforcerAddFormData()

Description Adds POSTed form data to a nested property list in the XML query document.

Include File `enforcer_web.h`

Syntax

```
int EnforcerAddFormData(
     XMLnode query,
     char *           buff,
     EnforcerHandle * eh
);
```

Parameters

- `query` — A pointer to the query node to which the form data will be added.
- `buff` — The location of the memory buffer in which the form data is being held. Must be non-`NULL`.
- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.

Return Values

- `1` — Form data was successfully added to the query.
- `0` — Form data could not be added to the query.

Remarks Once completed successfully, this function indicates that the Enforcer plugin should be refreshed.

Example The following code sample determines whether the query is a magic query. If so, it reads the POSTed data and adds it to the XML query document.

```
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);
}
```

EnforcerAddMultiResource()

Description

Adds the given resource to multi-resource property list in the given XML query document. If the query does not already contain a multi-resource list, one will be added.

Include File

```
enforcer_web.h
```

Syntax

```
int EnforcerAddMultiResource(
     XMLnode query,
     char * resource
);
```

Parameters

- `query` — A pointer to the query node.
- `resource` — The resource URL.

Return Values

- `1` — The resource was successfully appended to a multi-resource list.
- `0` — The resource could not be added to a multi-resource list.

EnforcerAddQueryData()

Description Adds HTTP query data to a nested property list in the XML query document.

Include File

```
enforcer_web.h
```

Syntax

```
void EnforcerAddQueryData(
     XMLnode query,
     char *           buff,
     EnforcerHandle * eh
);
```

Parameters

- `query` — A pointer to the query node to which the query data is being added.
- `buff` — The location of the memory buffer in which the HTTP query data is being held. Must be non-`NULL`.
- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.

Return Values

- `1` — The HTTP query data was successfully added to the query.
- `0` — The HTTP query data could not be added to the query.

Example The following code sample determines whether the query level is greater than `MINIMAL`. If so, it adds query data from the HTTP query string into the XML query document.

```
if (query_level > QUERY_MINIMAL) {
  if (http_query != NULL) {
    EnforcerAddQueryData(*query, http_query, Enforcer_Handle);
  }
}
```

EnforcerAddUrlEncoded()

Description Parses the given URL-encoded string, and adds the attribute/value pairs it contains to a new nested property list in the given query.

Include File

```
enforcer_web.h
```

Syntax

```
int EnforcerAddUrlEncoded(
     const char *     type
     XMLnode          query,
     char *           buff,
     EnforcerHandle * eh
);
```

Parameters

- `type` — The element type of the new node that will be added to the query.
- `query` — A pointer to the query node to which the attribute/value pairs will be added. Must be non-`NULL`.
- `buff` — The URL-encoded buffer, typically either an http query or post data.
- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.

Return Values

- `1` — The attribute/value pairs were successfully added to the query.
- `0` — The attribute/value pairs could not be added to the query.

Example The following code sample adds a property list which containing the properties for each query parameter to the query document.

```
EnforcerAddUrlEncoded(ENFORCER_HTTP_QUERY_LIST, query, buf, eh);
```

EnforcerAddWebHints()

Description Adds web server-specific hints to an XML query.

Include File

```
enforcer_web.h
```

Syntax

```
int EnforcerAddWebHints(
     XMLnode query,
     int     flags,
);
```

Parameters

- `query` — A pointer to the query node to which the web hints will be added.
- `flags` — An integer which indicates whether SSL with client certificates is supported.

Return Values

- `1` — Web hints were successfully added to the query.
- `0` — No web hints were added to the XML query.

EnforcerAuthFailed()

Description Determines whether the Policy Validator reply indicates that authentication failed.

Include File

```
enforcer_web.h
```

Syntax

```
int EnforcerAuthFailed(
     XMLnode query,
     XMLnode reply,
     int     auth_form
);
```

Parameters

- `query` — A pointer to the query node.
- `reply` — A pointer to the reply node.
- `auth_form` — A value which indicates whether or not the request is the result of a Select Access authentication form.

Return Values

- `1` — Authentication failure was detected.
- `0` — No authentication failure was detected.

Example The following code sample determines whether a redirect URL exists and whether the Policy Validator indicated that authentication failed. If so, the client browser is redirected to the existing redirect URL.

```
if(redirect_URL != NULL && EnforcerAuthFailed(query, reply,
     selectaccess_needs_refresh)) {
   EnforcerLog(ENFORCER_LOG_DEBUG, "doing auth_fail redirect to %s",
     redirect_URL);
   response = send_redirect(r, (char *)redirect_URL);
}
```

See Also Authentication Failure Reasons on page 479

EnforcerConstructSSOAuthURL()

Description In a single sign-on environment, constructs the second redirect URL, which contains a nonce.

Include File `enforcer_web.h`

Syntax

```
int EnforcerConstructSSOAuthURL(
     EnforcerHandle * eh,
     char *           currentURL,
     char *           nonce,
     char **          new_url
);
```

Parameters

- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.
- `currentURL` — The current URL.
- `nonce` — The current cookie, an unforgeable string verifying that the identity has been authorized.
- `new_url` — Return parameter containing the address of the buffer where the new URL is stored.

Return Values

- `1` — A valid URL was constructed successfully.
- `0` — The URL could not be constructed or was invalid.

Remarks The new URL is constructed by appending a nonce tag and the value of `nonce` to `currentURL`.

The caller must `delete[] new-url`.

Example The following code sample determines whether the given URL is a multi-domain single sign-on (MD-SSO) redirect URL, which is requesting a nonce. If so, and if a cookie exists, an authentication URL is constructed containing a nonce.

```
if (EnforcerIsSSORedirectURL(http_query, &old_url)) {
  char *sso_authURL;
  if (cookies != NULL) {
    nonce = (char *) ap_table_get(cookies, AUTH_COOKIE_NAME);
  }
  EnforcerConstructSSOAuthURL(Enforcer_Handle, old_url, nonce,
    &sso_authURL);
  EnforcerLog(ENFORCER_LOG_DEBUG, "doing SSO redirect 2 to %s",
    sso_authURL);
  send_redirect(r, sso_authURL);
  delete[] sso_authURL;
  return HTTP_MOVED_TEMPORARILY;
}
```

See Also

- `EnforcerConstructSSOOrigURL()` on page 72
- `EnforcerConstructSSORedirectURL()` on page 73

EnforcerConstructSSOOrigURL()

Description In a single sign-on environment, constructs the third redirect URL, which sends the browser back to the originating URL.

Include File `enforcer_web.h`

Syntax

```
int EnforcerConstructSSOOrigURL(
     char *  file,
     char *  http_query,
     char ** new_url
);
```

Parameters

- `file` — The filename component of the current URL.
- `http_query` — The HTTP query sent to the Policy Validator.
- `new_url` — Return parameter containing the address of the buffer where the new URL is stored.

Return Values

- `1` — A valid URL was constructed successfully.
- `0` — The URL could not be constructed or was invalid.

Remarks The caller must `delete[] new-url`.

Example The following code sample determines whether the given URL is a multi-domain single sign-on (MD-SSO) authentication URL containing a nonce. If so, authentication is successful. The original URL is reconstructed and the client browser is refreshed.

```
if (EnforcerIsSSOAuthURL(http_query, &nonce)) {
  char *sso_origURL;
  EnforcerConstructSSOOrigURL(r->uri, http_query, &sso_origURL);
  EnforcerLog(ENFORCER_LOG_DEBUG, "doing SSO redirect 3 to %s",
    sso_origURL);
  set_cookie(Enforcer_Handle, r, AUTH_COOKIE_NAME, nonce,
    COOKIE_LIFE, 0);
  send_refresh(r, 0, sso_origURL);
  response = DONE;
  delete[] sso_origURL;
}
```

See Also

- `EnforcerConstructSSOAuthURL()` on page 71
- `EnforcerConstructSSORedirectURL()` on page 73

EnforcerConstructSSORedirectURL()

Description

In a single sign-on environment, constructs the first redirect URL, which requests that the next server sends back a nonce.

Include File

`enforcer_web.h`

Syntax

```
int EnforcerConstructSSORedirectURL(
     char *  referer,
     char *  currentURL,
     char ** new_url
);
```

Parameters

- `referer` — The URL of the site where the link to the resource was clicked.
- `currentURL` — The current URL.
- `new_url` — Return parameter containing the address of the buffer where the new URL is stored.

Return Values

- `1` — Returned if the new URL is valid.
- `0` — Returned if the new URL is invalid.

Remarks

The caller must `delete[] new-url`.

Example

The following code sample determines whether multi-domain single sign-on (MD-SSO) will provide the needed authentication, so that the identity need not be prompted. If so, a redirect URL is constructed requesting a nonce.

```
if (EnforcerCouldSSOHelp(reply, referer, full_url, Enforcer_Handle)) {
  char *sso_redirectURL;
  EnforcerConstructSSORedirectURL(referer, full_url,
    &sso_redirectURL);
  EnforcerLog(ENFORCER_LOG_DEBUG, "doing SSO redirect 1 to %s",
    sso_redirectURL);
  response = send_redirect(r, sso_redirectURL);
  delete[] sso_redirectURL;
}
```

See Also

- `EnforcerConstructSSOAuthURL()` on page 71
- `EnforcerConstructSSOOrigURL()` on page 72

EnforcerCouldSSOHelp()

Description

Tests the given URL to determine whether multi-domain single sign-on (MD-SSO) should be attempted to provide the needed authentication.

Include File

```
enforcer_web.h
```

Syntax

```
int EnforcerCouldSSOHelp(
     XMLnode reply,
     char *           referer,
     char *           full_url,
     EnforcerHandle * eh
);
```

Parameters

- `reply` — A pointer to the reply node.
- `referer` — The URL of the site where the link to the resource was clicked.
- `full_url` — The current URL.
- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.

Return Values

- `1` — MD-SSO could provide the needed authentication.
- `0` — Authentication failed for reasons which MD-SSO cannot overcome.

Example

The following code sample determines whether multi-domain single sign-on (MD-SSO) will provide the needed authentication, so that the identity need not be prompted. If so, a redirect URL is constructed requesting a nonce.

```
if (EnforcerCouldSSOHelp(reply, referer, full_url, Enforcer_Handle)) {
  char *sso_redirectURL;
  EnforcerConstructSSORedirectURL(referer, full_url,
    &sso_redirectURL);
  EnforcerLog(ENFORCER_LOG_DEBUG, "doing SSO redirect 1 to %s",
    sso_redirectURL);
  response = send_redirect(r, sso_redirectURL);
  delete[] sso_redirectURL;
}
```

EnforcerGetAuthHint()

Description Extracts the authentication hints, if any, out of a Policy Validator reply object. If only one hint is present in the reply, it is returned. If more than one hint is present, this function merges them into a list.

Include File

```
enforcer_web.h
```

Syntax

```
int EnforcerGetAuthHint(
     XMLnode reply,
     XMLnode *     auth_hint,
     int *         hint_flags
);
```

Parameters

- `reply` — A pointer to the reply node.
- `auth_hint` — Return parameter which contains the hint.
- `hint_flags` — Return parameter containing the type of hint returned.

Return Values

- `1` — The authentication hints were successfully extracted from the Policy Validator reply contained a.
- `0` — No authentication hints were extracted.

Example The following code sample retrieves authentication hints from the Policy Validator reply, then determines whether the `Hint_Flags` parameter contains a password and whether the Enforcer plugin is configured to perform login operations via a web form. If so, it sends an HTTP basic authentication page to the web client.

```
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);
  }
}
```

EnforcerGetBasenameOfURL()

Description

Extracts the basename of the given URL by terminating it prior to an anchor symbol (`#`) or `jsessionid` tag.

Include File

`enforcer_web.h`

Syntax

```
int EnforcerGetBasenameOfURL(
     char *          url
     char **         new_url,
     int *           url_changed
);
```

Parameters

- `url` — The original URL.
- `new_url` — Return parameter containing the address of the buffer where the newly-created URL is stored.
- `url_changed` — Return parameter indicating whether or not `*url` has changed.

Return Values

- `1` — The basename was successfully extracted and is valid.
- `0` — The basename could not be successfully extracted.

Remarks

This method only removes non-basename components from the provided string. It does not attempt to resolve the URL nor does it ensure that the URL is valid.

Example

The following code sample determines whether the basename of the given URL can be extracted. If not, the request is immediately denied.

```
if (!EnforcerGetBasenameOfURL(r->uri, &base_url, &url_changed))
  return HTTP_FORBIDDEN;
```

EnforcerGetDisableCaching()

Description Determines whether the given Enforcer plugin is configured to disable caching of Select Access-protected web pages.

Include File

```
enforcer_web.h
```

Syntax

```
int EnforcerGetDisableCaching(
     EnforcerHandle * eh
);
```

Parameters

- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.

Return Values

- `1` — The Enforcer plugin disables caching for protected web pages.
- `0` — The Enforcer plugin does not disable caching for protected web pages.

Example The following code sample determines whether the Enforcer plugin is configured to not cache Select Access-protected web pages. If so, it disables caching for all requests that receive an Allow action and logs a message.

```
if (EnforcerGetDisableCaching(Enforcer_Handle)) {
  EnforcerLog(ENFORCER_LOG_DEBUG, "disable_caching for allow");
  disable_caching(&header_text);
}
```

EnforcerGetLoginViaForm()

Description Determines whether the given Enforcer plugin is configured to use a web form to perform login operations.

Include File `enforcer_web.h`

Syntax

```
int EnforcerGetLoginViaForm(
     EnforcerHandle * eh
);
```

Parameters

- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.

Return Values

- `1` — The Enforcer plugin uses web forms for login operations.
- `0` — The Enforcer plugin does not use web forms for login operations.

Example The following code sample retrieves authentication hints from the Policy Validator reply, then determines whether the `Hint_Flags` parameter contains a password and whether the Enforcer plugin is configured to perform login operations via a web form. If so, it sends an HTTP basic authentication page to the web client.

```
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);
  }
}
```

EnforcerIsDomainProtected()

Description Tests the referring site to determine whether it is in the multi-domain single sign-on (MD-SSO) protected sites list.

Include File

```
enforcer_web.h
```

Syntax

```
int EnforcerIsDomainProtected(
     EnforcerHandle * eh,
     char *           referer
);
```

Parameters

- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.
- `referer` — The URL of the referring site.

Return Values

- `1` — The referring site matches one of the domains listed in the MD-SSO protected sites list.
- `0` — The referring site does not match one of the domains listed in the MD-SSO protected sites list.

Example The following code sample determines whether the referring site is in a protected domain. If so, constructs a redirect URL and

```
FInfo *fi = (FInfo *)pFC->pFilterContext;
if ( (fi->referrer) && (EnforcerIsDomainProtected(Enforcer_Handle,
     fi->referrer)) )
{
  char * sso_redirectURL;
  EnforcerConstructSSORedirectURL(fi->referrer, fi->completeURL,
    &sso_redirectURL);
  EnforcerLog(ENFORCER_LOG_DEBUG, "doing redirect 1 to : %s",
    sso_redirectURL);
  doRedirect(pFC, sso_redirectURL);
  delete[] sso_redirectURL;
  delete query;
  delete reply;
  return close_mmFile(pFC);
}
```

EnforcerIsEvilURL()

Description Determines whether the URL path is safe.

Include File

```
enforcer_web.h
```

Syntax

```
int EnforcerIsEvilURL(
     const char *     url,
     EnforcerHandle * eh
);
```

Parameters

- `url` — The URL to be tested.
- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.

Return Values

- `1` — The given URL contains characters that could be unsafe.
- `0` — The given URL is safe.

Remarks The `EnforcerUnescapeUrl()` function should be executed on the given URL to remove its URL encoding before calling this function.

Example The following code sample determines whether the given URL contains characters that could be unsafe, then logs a warning message and rejects the request.

```
   if(EnforcerIsEvilURL(r->uri, Enforcer_Handle)) {
     EnforcerLog(ENFORCER_LOG_WARNING, "rejecting suspicious url '%s'",
r->uri);
     return HTTP_FORBIDDEN;
   }
```

See Also

- `EnforcerUnescapeUrl()` on page 88

EnforcerIsIgnoredFile()

Description Tests the given filename to determine whether it is in the list of ignored filenames. Files listed in the `ignoredFile` list are unconditionally allowed.

Include File `enforcer_web.h`

Syntax

```
int EnforcerIsIgnoredFile(
     EnforcerHandle * eh,
     char *           filemane
);
```

Parameters

- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.
- `filename` — The filename to be compared.

Return Values

- `1` — The given filename is on the `ignoredFile` list.
- `0` — The given filename is not on the `ignoredFile` list.

Remarks All filenames are compared without regard to case.

Example The following code sample determines whether the given file is on the list of ignored filenames. If so, access is granted.

```
if (EnforcerIsIgnoredFile(Enforcer_Handle, r->uri)) {
  EnforcerLog(ENFORCER_LOG_DEBUG, "Ignore: %s", r->uri);
  return ALLOW;
}
```

EnforcerIsPassthroughDomain()

Description Tests the given virtual hostname to determine whether it is in the list of pass-though domains. Domains listed in the `passthroughDomain` list are unconditionally allowed.

Include File `enforcer_web.h`

Syntax

```
int EnforcerIsPassthroughDomain(
     EnforcerHandle * eh,
     const char *     host
);
```

Parameters

- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.
- `host` — The name of the domain host to be tested.

Return Values

- `1` — The given virtual hostname matches one of the domains listed in the `passthroughDomain` list.
- `0` — The given virtual hostname does not match any of the domains listed in the `passthroughDomain` list.

Remarks All domains are compared without regard to case.

Example The following code sample determines whether the given hostname is on the list of passthrough domains. If so, access is granted.

```
virt_domain = (char*) ap_get_server_name(r);
if (EnforcerIsPassthroughDomain(Enforcer_Handle, virt_domain)) {
   EnforcerLog(ENFORCER_LOG_DEBUG, "unsecured_domain: %s",
     virt_domain);
   return ALLOW;
}
```

EnforcerIsSSOAuthURL()

Description Tests the given URL to determine whether it as an authentication URL, that is, whether it contains a nonce. If so, the `noncePtr` parameter will be pointed to the value of the nonce.

Include File

```
enforcer_web.h
```

Syntax

```
int EnforcerIsSSOAuthURL(
     char *  url,
     char ** noncePtr
);
```

Parameters

- `url` — The URL to be tested.
- `urlPtr` — Return parameter containing the value of the nonce.

Return Values

- `1` — The given URL is an authentication URL which contains a nonce.
- `0` — The given URL does not contain a nonce.

Example The following code sample tests to see whether the given URL is a multi-domain single sign-on (MD-SSO) authentication URL containing a nonce. If so, authentication is successful. The original URL is reconstructed and the client browser is refreshed.

```
if (EnforcerIsSSOAuthURL(http_query, &nonce)) {
  char *sso_origURL;
  EnforcerConstructSSOOrigURL(r->uri, http_query, &sso_origURL);
  EnforcerLog(ENFORCER_LOG_DEBUG, "doing SSO redirect 3 to %s",
    sso_origURL);
  set_cookie(Enforcer_Handle, r, AUTH_COOKIE_NAME, nonce,
    COOKIE_LIFE, 0);
  send_refresh(r, 0, sso_origURL);
  response = DONE;
  delete[] sso_origURL;
}
```

EnforcerIsSSORedirectURL()

Description Tests the given URL to determine whether it contains an SSO redirect which requests a nonce. If so, the `urlPtr` parameter will be pointed to the value of the return URL.

Include File

```
enforcer_web.h
```

Syntax

```
int EnforcerIsSSORedirectURL(
     char *  url,
     char ** urlPtr
);
```

Parameters

- `url` — The URL to be tested.
- `urlPtr` — Return parameter containing the address of the buffer where the new URL is stored.

Return Values

- `1` — The given URL requests a nonce.
- `0` — The given URL does not request a nonce.

Example The following code sample tests to see whether the given URL is a multi-domain single sign-on (MD-SSO) redirect URL, which is requesting a nonce. If so, and if a cookie exists, an authentication URL is constructed containing a nonce.

```
if (EnforcerIsSSORedirectURL(http_query, &old_url)) {
  char *sso_authURL;
  if (cookies != NULL) {
    nonce = (char *) ap_table_get(cookies, AUTH_COOKIE_NAME);
  }
  EnforcerConstructSSOAuthURL(Enforcer_Handle, old_url, nonce,
    &sso_authURL);
  EnforcerLog(ENFORCER_LOG_DEBUG, "doing SSO redirect 2 to %s",
    sso_authURL);
  send_redirect(r, sso_authURL);
  delete[] sso_authURL;
  return HTTP_MOVED_TEMPORARILY;
}
```

EnforcerNeedsRefresh()

Description Determines whether the given request needs to trigger a refresh (GET) of the current URL.

Include File `enforcer_web.h`

Syntax

```
int EnforcerNeedsRefresh(
     XMLnode post_data_list
);
```

Parameters

- `post_data_list` — A pointer to the XML node containing the POST data.

Return Values

- `1` — `ENFORCER_SEND_REFRESH` is found.
- `0` — `ENFORCER_SEND_REFRESH` is not found.

Example The following code sample takes a query string and splits it its component name/value pairs, then stored them in parallel arrays of names and values. It then removes any URL encoding, encodes the contents of the arrays in UTF-8 and adds them to a property list, before triggering a refresh.

```
nvcount = EnforcerQstr2nv(buf2.get(), name, value);
for ( i=0; i < nvcount; i++) {
  if ( name[i] == NULL ) {
     free(name);
     free(value);
     return 0;
  }
  EnforcerUnescapeUrl(name[i]);
  EnforcerUnescapeUrl(value[i]);
}
for (i = 0; i < nvcount; i++) {
  EnforcerAddEncoded(property_list, name[i], value[i], eh);
}
free(name);
free(value);
return EnforcerNeedsRefresh(property_list);
```

EnforcerProxyMode()

Description Tests to determine whether the given Enforcer plugin should function as a proxy.

Include File

```
enforcer_web.h
```

Syntax

```
int EnforcerProxyMode(
     EnforcerHandle * eh
);
```

Parameters

- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.

Return Values

- `1` — The Enforcer plugin should function as a proxy.
- `0` — The Enforcer plugin should not function as a proxy.

Example The following code sample tests the URL to see if it is a proxy request.

```
if (!EnforcerProxyMode(eh) && strstr(url, "//"))
  return 1;
```

EnforcerTransformPage()

Description Loads and transforms an HTML form page.

Include File `enforcer_web.h`

Syntax

```
int EnforcerTransformPage(
     EnforcerHandle * eh,
     XMLnode          reply,
     const char *     extraValues[][2],
     const char **    contentTypeRef
);
```

Parameters

- `eh` — A non-`NULL` pointer to a previously-initialized `EnforcerHandle` object.
- `reply` — A pointer to the reply node.
- `extraValues` — An array that can be used to store extra form substitutions.
- `contentTypeRef` — A string pointing to the appropriate MIME type for the newly-generated page.

Return Values

- If successful, returns the transformed page data.
- If unsuccessful, returns `NULL`.

Remarks Memory considerations:

- You must `delete[]` the returned transformed file to free its memory.
- `contentTypeRef` is static, and it should not be freed.

Example

```
auto_ptarray<char> page(EnforcerTransformPage(Enforcer_Handle,
     auth_plist, extravalues, &contentType));
if (page.get() != NULL) {
   r->content_type = contentType;
   r->status = 200;
   r->status_line = "200 OK, page follows";
   if (reload_url != NULL) {
      snprintf(tmp, BUFSIZ, "%d; URL=%s", RELOAD_DELAY, reload_url);
      ap_table_set(r->headers_out, "Refresh", tmp);
   }
   disable_caching(r, TRUE);

   ap_rputs(page.get(), r);
}
else {
   return no_forms_error(r, 1);
}
return OK;
```

EnforcerUnescapeUrl()

Description Removes all escape sequences from the given URL and replaces them with their single-character, Latin-1 character-set equivalent.

Include File `enforcer_web.h`

Syntax

```
int EnforcerUnescapeUrl(
     char * url
);
```

Parameters

- `url` — The URL to be decoded.

Return Values

- `0` — All escape sequences in the URL were replaced with their Latin-1 equivalents, resulting in a valid URL.
- any non-zero value — The URL could not be successfully decoded.

Remarks This function should called before calling `EnforcerIsEvilURL()`.

Example The following code sample takes a query string and splits it its component name/value pairs, then stored them in parallel arrays of names and values. It then removes any URL encoding, encodes the contents of the arrays in UTF-8 and adds them to a property list, before triggering a refresh.

```
nvcount = EnforcerQstr2nv(buf2.get(), name, value);
for ( i=0; i < nvcount; i++) {
   if ( name[i] == NULL ) {
      free(name);
      free(value);
      return 0;
   }
   EnforcerUnescapeUrl(name[i]);
   EnforcerUnescapeUrl(value[i]);
}
for (i = 0; i < nvcount; i++) {
   EnforcerAddEncoded(property_list, name[i], value[i], eh);
}
free(name);
free(value);
return EnforcerNeedsRefresh(property_list);
```

See Also

- `EnforcerIsEvilURL()` on page 80

EnforcerWebAuthDecode()

Description Decodes the username and password of an authentication header.

Include File `enforcer_web.h`

Syntax

```
int EnforcerWebAuthDecode(
     char * auth_header,
     char * user,
     char * password
);
```

Parameters

- `auth_header` — The authentication header to be decoded.
- `user` — The buffer where the encoded username will be copied.
- `password` — The buffer where the encoded password will be copied.

Return Values

- `1` — The username and password were decoded successfully.
- `0` — The username and password could not be decoded. This could be because:
  - No basic authentication header exists.
  - The decoded header had a length of 0.
  - The header contained no password.

Remarks This method decodes username and password using the `Base64_DecodeBlock()` function.

Example The following code sample determines whether the given authentication header can be decoded. If not, the request is immediately aborted. If it can be decoded, the name and password parameter are then encoded into UTF-8 and added to the XML query document.

```
  if (!EnforcerWebAuthDecode(auth_header, user, password)) {
    return REQ_ABORTED;
  }
  EnforcerAddEncoded(*query, ENFORCER_USER, user, Enforcer_Handle);
  EnforcerAddEncoded(*query, ENFORCER_PASSWD, password,
Enforcer_Handle);
  return REQ_PROCEED;
```

4 Policy Validator API

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. This chapter provides a reference for these functions.

Understanding the Plugin Types

There are two types of Policy Validator plugins:

- Authenticators: Authenticator plugins allow you to create new ways to verify identities.
- Deciders: Decider/Evaluator plugins allow you to create new methods of controlling access to resources.

The Policy Validator API includes a base class for each plugin type: AuthPlugin and Decider. All Policy Validator plugins must be derived from one of these classes. Table 11 describes the two plugin base classes in further detail.

Table 11 Policy Validator Plugin Base Classes

Base class	Description	Page
`AuthPlugin`	`AuthPlugin` is an abstract class which provides the template for Policy Validator Authentication plugins. This class defines: • The authenticator result codes. • A `factory()` method which your plugin should define. • The virtual `authenticate()` which you must override in your derived class. For a detailed description of the member methods of this class, see AuthPlugin on page 92.	92
`Decider`	`Decider` is an abstract class which provides the template for rule deciders. Each rule decider implements a single type of decision node. This class defines: • The evaluator result codes. • A `factory()` method which your plugin should define. • The virtual `decide()` which you must override in your derived class. For a detailed description of the member methods of this class, see Decider on page 112.	112

AuthPlugin

The `AuthPlugin` class is an abstract class which provides the template for Policy Validator Authentication plugin subclasses. This class defines:

- the authenticator result codes
- a `factory()` method, which your subclass must define
- the virtual method `authenticate()`, which you must override to perform identity authentication
- the virtual method `handleUserInfo()`, which automates processing of transient identities and populating personalization data. While it is possible to override the `handleUserInfo()` method, you should not.

Class Summary

The `AuthPlugin` class is abstract and should never be instantiated. To create an instance of your plugin, you must define the `factory()` method to call a constructor defined in your subclass.

The `AuthPlugin` class defines the methods listed in Table 12. They are described in the sections that follow.

Table 12 Class Overview

Method	Description	Page
`addAuthHint2Response()`	Adds standard authentication hints to an XML response, which inform the Enforcer plugin to request identity authentication. Plugins can redefine it.	95
`authenticate()`	Attempts to authenticate an identity. This method must be overridden by the `AuthPlugin` subclass.	96
`factory()`	Creates an instance based on the given XML properties.	98
`handleUserInfo()`	Manages identities who were authenticated using `SecurID`, but for whom there is no directory server record.	100
`inspect()`	Examines an identity's password to ensure that it is valid.	101
`name()`	Gets the name of the current authentication node.	102
`sExtractFormName()`	Extracts the name of the form used.	103
`sExtractSyntheticLocation()`	Helpers for IWA and trusted plugins, other plugins could be refactored to use them.	104

Table 12 Class Overview (cont'd)

Method	Description	Page
`sExtractUserSource()`	Extracts the name of an identity source from an Authentication plugin's XML configuration data. This assumes the identity source is stored in an XML property called `user_source`. If the value of this property is "Known Identities", this method returns NULL, which signals the validator that it should search all identity sources.	105
`sFindAuthHintCreatedByAuthServer()`	Looks through the response XML that is going to be sent to the Enforcer plugin to find an authentication hint created by the specified authentication service.	106
`sNameToUserSource()`	Converts a `UserSourceName` to an identity source or `NULL` (signaling, "Known Identities").	107
`sValidateSyntheticUserLocation()`	Check that the location where transient identities are going to be created is below the specified identity location.	108
`type()`	Gets the type of the Authentication plugin.	109
`type(char)`	Sets the type of the Authentication plugin.	110
`user_source()`	Retrieves the location where this plugin looks for identities.	111

Definitions

`AuthPlugin` has one member type definition.

Table 13 Definitions Overview

Definition Type	Description
Result codes	The enumerated type `Result` defines the authenticator result codes. Each Authenticator plugin must return one of the following Result codes, which informs the Enforcer plugin how it should proceed. Defined values are listed below:

Result Code	Description
`R_PERMIT`	The identity has been successfully authenticated. Any remaining authenticators are ignored.
`R_DENY`	The identity has been denied access. Any remaining authenticators are ignored
`R_CONTINUE`	The identity could not be authenticated with this plugin. The next authenticator listed in Select Auth is processed.
`R_RESTART`	The identity must be reauthenticated using the first authenticator listed in Select Auth. This usually occurs when new information about the identity has become available.
`R_ERROR`	An error occurred while the Policy Validator was authenticating the identity. Authentication is stopped and the identity is denied access. Your plugin should log an error level message explaining the problem.

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

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

addAuthHint2Response()

Description Adds standard authentication hints to an XML response, which inform the Enforcer plugin to request identity authentication. Plugins can redefine it.

Class `AuthPlugin`

Include File `AuthPlugin.h`

Syntax

```
PropertyListElement * addAuthHint2Response(
     const PropertyListElement & response,
     const string & formName
);
```

Parameters

- `response` — A pointer to the XML response document that is returned to the Enforcer plugin.
- `formName` — The name of the HTML form the Enforcer plugin should display to request further identity authentication.

Return Values A property list containing authentication hints.

Example The following code sample triggers the Enforcer plugin to display a login form for the identity.

```
void radius::AddInitialForm(PropertyListElement *response){
        addAuthHint2Response(*response, login_formName_.get());
}
```

authenticate()

Description Attempts to authenticate an identity. This method must be overridden by the `AuthPlugin` subclass.

Class `AuthPlugin`

Include File `AuthPlugin.h`

Syntax

```
Result authenticate(
     PropertyListElement *       data,
     PropertyListElement *       result,
     const PropertyListElement * const personalizer,
     const bool trace
) = 0;
```

Parameters

- `data` — A pointer to the request data. Because a request may pass through m ore than one Authentication plugins, each authenticator may embed additional data to the original XML query document.
- `result` — A pointer to the XML response. This response contains any existing XML properties from authenticators that have already been processed.
- `personalizer` — A pointer to an XML document that contains personalization information.
- `trace` — A flag which indicates whether the authenticator should include detailed information about the decision. Used primarily for debugging purposes.

Return Values The result code. See Definitions on page 94 for more information on the possible result codes.

Example The following code is a sample subclass implementation of the `authenticate()` method.

```
AuthPlugin::Result radius::authenticate(
PropertyListElement *     data,
  PropertyListElement *  response,
  const PropertyListElement * const personalizer,
  const bool     trace ){
  Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG, "Invoking radius
    plugin");
  AuthPlugin::Result result = R_DENY;
  const char *prophylaxis =
data->getChildStringValue(VALIDATOR_REAUTH_PROPHYLAXIS);
  if ( prophylaxis && STREQ(prophylaxis, name_.get()) ) {
    result = R_PERMIT;
    if ( canCacheUser() ) {
      PropertyListElement * post_data_list =
ValidatorGetPostDataList(data);
      if (! post_data_list) {
        Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
          "RADIUS authentication query does not contain post data
            list");
        return R_ERROR;
      }
      const char *user = post_data_list->getChildStringValue
        (ENFORCER_USER);
      if (! user) {
```

```
            Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR,
                   "RADIUS authentication query's post data list does not
                   contain user");
            return R_ERROR;
          }
          result = handleUserInfo(data, response, personalizer, user,
               synthetic_loc_.get());
        }
      } else {
        try {
          radiusRequest request(data, ValidatorGetPostDataList(data));
          result = processRequest(request, data, response, personalizer);
        }
        catch (const char * error) {
          if ( STREQ(error, ENFORCER_NATIVE_PASSWD) ) {
            AddInitialForm(response);
            result = R_DENY;
          }
          else {
            throw;
          }
        }
        catch (string & message) {
          ValidatorError(response, SYS_STRDUP(message.c_str()));
          result = R_ERROR;
        }
      }
      setResponseAction(result, response);
      return result;
    }
```

factory()

Description Creates an instance based on the given XML properties.

Class `AuthPlugin`

Include File `AuthPlugin.h`

Syntax

```
Authplugin * factory(
    const PropertyListElement * props
);
```

Parameters

- `props` — A pointer to the property list in the plugin's `component.xml` file that contains the configuration properties for the plugin.

Return Values An instance of the Policy Validator Authenticator plugin object.

Remarks This method must be defined by the `AuthPlugin` subclass.

Example The following code is a sample subclass implementation of this method which calls a subclass constructor called `attributelogic`.

```
AuthPlugin * radius::factory(
const string &                name,
const XmlTreeNode *           props)
   {
   if (! props) {
     Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR, INVALID_PROPS);
     return NULL;
   }
   const char * synthetic_loc = props->getChildStringValue
     (SURROGATE_LOCATION_TAG);
   const char * login_form_name = props->getChildStringValue
     (FORM_FILENAME_TAG);
   const char * challenge_form_name =
props->getChildStringValue(CHALLENGE_LOGIN_FORMNAME);
   const char * user_source_name = props->getChildStringValue
     (USER_SOURCE_TAG);
   const UserSource * user_source = NULL;
   if (! sNameToUserSource(name.c_str(), user_source_name, user_source))
{
     return NULL;
   }
   if ( ! synthetic_loc || ! sValidateSyntheticUserLocation
        (synthetic_loc, user_source) ) {
     return NULL;
   }
   if (! login_form_name) {
     login_form_name = DEFAULT_LOGIN_FORMNAME;
   }
   if (! challenge_form_name) {
     challenge_form_name = DEFAULT_CHALLENGE_LOGIN_FORMNAME;
   }
   radius *radPlugin = NULL;
   try {
```

```
        radPlugin = new radius(name.c_str(), user_source, synthetic_loc,
            login_form_name, challenge_form_name);
    }
    catch (...) {
      Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR, ERROR_NEW_RADIUS);
        return NULL;
    }
    try {
      if ( radPlugin->fetchServersInfo(props) < 1 ) {
        throw INVALID_SERVER_CONF;
      }
    }
    catch (...) {
      delete radPlugin;
      Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR, INVALID_SERVER_CONF);
      return NULL;
    }
    return radPlugin;
}
```

handleUserInfo()

Description Manages identities who were authenticated using `SecurID`, but for whom there is no directory server record.

Class `AuthPlugin`

Include File `AuthPlugin.h`

Syntax

```
Result handleUserInfo(
     PropertyListElement *           data,
     PropertyListElement *           response,
     const PropertyListElement *     personalizer,
     const char *                    username,
     const char *                    synthetic_loc
);
```

Parameters

- `data` — A pointer to the XML request data.
- `response` — A pointer to the XML response. This response contains any existing XML properties from authenticators that have already been processed.
- `personalizer` — A pointer to an XML document that contains personalization information.
- `username` — The username given to the transient identity.
- `synthetic_loc` — The identity location in which the transient identity is stored.

Return Values The result code. See Definitions on page 94 for more information on the possible result codes.

Remarks Plugins may redefine this method, however it is not recommended.

Example The following code sample determines how an unknown identity will be handled if he/she is granted access.

```
case RAD_ACCESS_ACCEPT: {
  if ( request.getReplyMessage(buffer, RAD_MSGSIZE) >= 0 ) {
    addMessage2Response(buffer, response);
    data->setChildStringValue(VALIDATOR_REAUTH_PROPHYLAXIS,
       SYS_STRDUP(name_.get()));
    result = R_PERMIT;
    if ( canCacheUser() ) {
      result = handleUserInfo(data, response, personalizer,
           request.getUser().c_str(), synthetic_loc_.get());
    }
    break;
  }
}
```

See Also

- See Definitions on page 94

inspect()

Description Examines an identity's password to ensure that it is valid.

Class `AuthPlugin`

Include File `AuthPlugin.h`

Syntax

```
Result inspect(
     const UserSource *    userSource,
     PropertyListElement * data,
     PropertyListElement * response,
);
```

Parameters

- `userSource` — A pointer to a known identity source.
- `data` — A pointer to the XML request data.
- `result` — A pointer to the XML response. This response contains any existing XML properties from authenticators that have already been processed.

Return Values The result code. See Definitions on page 94 for more information on the possible result codes.

Example The following code is a sample subclass implementation of the `inspect()` method.

```
AuthPlugin::Result password::inspect(
const UserSource *          userSource,
PropertyListElement *      data,
PropertyListElement *response )
{
   Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG, "password::inspect
     called");
   auto_mem<ACE_Read_Guard<ACE_RW_Thread_Mutex> > configMgrGuard;
   configMgrGuard = ConfigMgr::acquireReadLock();
   PasswordPolicy *pp = ConfigMgr::getPasswordPolicy
     (configMgrGuard.get());
   UserRefPtr dummy;
   Result result = pp->changePassword(data, response, name_.get(),
      ENFORCER_FORM_AUTHENTICATOR_TYPE, userSource, dummy, true);
   if ( result == R_DENY_USER_NOT_FOUND_FORM ) {
     result = R_DENY_FORM;
   }
   return result;
}
```

name()

Description	Gets the name of the current authentication node.
Class	`AuthPlugin`
Include File	`AuthPlugin.h`
Syntax	`const char * const name() const;`
Parameters	None.
Return Values	The name of the authentication node.

sExtractFormName()

Description Extracts the name of the form used.

Class `AuthPlugin`

Include File `AuthPlugin.h`

Syntax

```
const char *                sExtractFormName(
     const XmlTreeNode &    props,
);
```

Parameters

- `props` — A pointer to the property list in the plugin's `component.xml` file that contains the configuration properties for the plugin.

sExtractSyntheticLocation()

Description Helpers for IWA and trusted plugins, other plugins could be refactored to use them.

Class `AuthPlugin`

Include File `AuthPlugin.h`

Syntax

```
const char *                sExtractSyntheticLocation(
     const XmlTreeNode &    props,
     const UserSource *     userSource
);
```

Parameters

- `props` — A pointer to the property list in the plugin's `component.xml` file that contains the configuration properties for the plugin.
- `userSource` — A pointer to a known identity source.

sExtractUserSource()

Description Extracts the name of an identity source from an Authentication plugin's XML configuration data. This assumes the identity source is stored in an XML property called `user_source`. If the value of this property is "Known Identities", this method returns NULL, which signals the validator that it should search all identity sources.

Class `AuthPlugin`

Include File `AuthPlugin.h`

Syntax

```
const UserSource *          sExtractUserSource(
     const XmlTreeNode &    props,
     const string &         authServer
);
```

Parameters

- `props` — A pointer to the property list in the plugin's `component.xml` file that contains the configuration properties for the plugin.
- `authServer` — The name of the authentication service.

Return Values

- The `UserSource` associated with the Authentication plugin.
- `NULL` — search all identity sources.

sFindAuthHintCreatedByAuthServer()

Description Looks through the response XML that is going to be sent to the Enforcer plugin to find an authentication hint created by the specified authentication service.

Class `AuthPlugin`

Include File `AuthPlugin.h`

Syntax

```
const PropertyListElement *          sFindAuthHintsCreatedByAuthServer(
     const PropertyListElement &     response,
     const char *                    authServer
);
```

Parameters

- `response` — A pointer to the XML response document that is returned to the Enforcer plugin.
- `authServer` — The authentication server which created the authentication hint.

Return Values

- The hint within an XML property list.
- `NULL` — Searched all identity sources and no hint was found.

Example The following code sample defines a property list element

```
const PropertyListElement * hint = sFindAuthHintCreatedByAuthServer
     (response, name_.get());
```

sNameToUserSource()

Description Converts a `UserSourceName` to an identity source or `NULL` (signaling, “Known Identities”).

Class `AuthPlugin`

Include File `AuthPlugin.h`

Syntax

```
static const bool sNameToUserSource(
     const char *          caller,
     const char *          name,
     const UserSource *&   userSource
);
```

Parameters

- `caller` — The name of the caller function used in log messages.
- `name` — The `UserSource` name to be converted.
- `&userSource` — A pointer to a known identity source.

Return Values

- `true` — The `UserSource` name was successfully converted to a `UserSource`.
- `false` — The `UserSource` Name could not be converted because an error occurred.
- `NULL` — The Identity location is “Known Identities”.

Example The following code sample tests a given transient location to ensure that it lies below a known `UserSource`.

```
const UserSource * user_source = NULL;
   if (! sNameToUserSource(name.c_str(), user_source_name, user_source))
{
      return NULL;
   }
   if ( ! synthetic_loc || ! sValidateSyntheticUserLocation
        (synthetic_loc, user_source) ) {
      return NULL;
   }
```

sValidateSyntheticUserLocation()

Description Check that the location where transient identities are going to be created is below the specified identity location.

Class `AuthPlugin`

Include File `AuthPlugin.h`

Syntax

```
bool sValidateSyntheticUserLocation(
     const string &     syntheticLoc,
     const UserSource * userSource,
);
```

Parameters

- `syntheticLoc` — The location of the transient identity location within the source.
- `userSource` — A pointer to a known identity location.

Return Values

- `true` — The location lies within the specified data location.
- `false` — The does not lie with in the specified data location.

Remarks

- If the specified identity data location is `NULL`, this method treats it as meaning “Known Identities”, and checks that the specified location is below some other location.
- Identity data locations are checked in the order in which they are shown in the Identities Tree in the policy builder.

Example The following code sample tests a given transient location to ensure that it lies below a known identity data location.

```
const UserSource * user_source = NULL;
  if (! sNameToUserSource(name.c_str(), user_source_name, user_source))
{
     return NULL;
  }
  if ( ! synthetic_loc || ! sValidateSyntheticUserLocation
       (synthetic_loc, user_source) ) {
     return NULL;
  }
```

type()

Description	Gets the type of the Authentication plugin.
Class	`AuthPlugin`
Include File	`AuthPlugin.h`
Syntax	`const char * type() const;`
Parameters	None.
Return Values	The Authentication plugin type.

type(char)

Description Sets the type of the Authentication plugin.

Class `AuthPlugin`

Include File `AuthPlugin.h`

Syntax

```
void type(
     const char * type
);
```

Parameters

- `type` — The Authentication plugin type.

Return Values None.

user_source()

Description	Retrieves the location where this plugin looks for identities.
Class	`AuthPlugin`
Include File	`AuthPlugin.h`
Syntax	`const UserSource * user_source() const;`
Parameters	None.
Return Values	The Identity Data Location in which the plugin searches for identities.

Decider

The `Decider` class is an abstract class that provides the template for Policy Validator Decider plugin subclasses. Each Decider plugin implements a single type of decision node. The `DeciderMgr` class provides a factory to create instances of the actual deciders, including tracking what deciders are available and dynamically loading them if required.

This class defines:

- The decider result codes.
- A `factory()` method, which your plugin should define and which you must register with `DeciderMgr`
- The virtual `decide()` which you must override in your derived class.

Class Summary

The decider class is abstract and therefore should never be instantiate. To create an instance of your plugin, you must define the `factory()` method to call a constructor defined in your subclass.

The `Decider` class defines the methods listed in Table 14.

Table 14 Class Summary

Method	Description	Page
`decide()`	Makes a decision. This method is a virtual method that must be overloaded by your plugin.	114
`factory()`	Create an instance based on the given XML properties; must be defined by the decider subclass.	115
`init()`	Registers a Decider plugin with the `DeciderMgr` class.	116

Definitions

`Decider` has one member type definition, described in detail in Table 15:

Table 15 Definitions Overview

Definition	Description	
`Result Codes`	The enumerated type `Result` defines the decider result codes. Each Decider plugin must return one of the following Result codes, which informs the Policy Builder how it should proceed.	
	Code	**Description**
	`CONTINUE`	Processes the next the evaluator in the conditional rule, as determined by the response variable (i.e. `DECIDER_BRANCH_TRUE` or `DECIDER_BRANCH_FALSE`). Most evaluators return this value under normal conditions.
	`DONE`	Stops processing evaluators and returns the XML response to the Policy Enforcer. A decider 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.
	`ALLOW`	Stops processing and allows access. This is a reserved value for Select Access terminators. Your evaluator should not return this value.
	`FAILED`	Stops processing due to an error.
	`R_RESTART`	Reevaluates the conditional rule from the first decision point. An evaluator usually returns this code after requesting that the identity authenticate.
	`SEND_FORM`	Sends the the requested authentication form.

decide()

Description Makes a decision. This method is a virtual method that must be overloaded by your plugin.

Class `Decider`

Include File `Decider.h`

Syntax

```
Result decide(
     PropertyListElement * request,
     PropertyListElement * response,
     const bool trace,
     string & decider_branch_path
);
```

Parameters

- `request` — A pointer to the Enforcer XML request data. This XML node contains the resource access request sent by a Policy Enforcer, and may contain additional data embedded by other evaluators or authenticators.
- `response` — A pointer to the XML response that will be sent back to the Enforcer plugin. This XML node contains any existing XML responses from evaluators that have already been processed as part of the rule.
- `trace` — Specifies whether or not debugging is enabled.
- `decider_branch_path` — The branch that the Policy Validator should evaluate next. The two valid settings for the response string are the predefined strings `DECIDER_BRANCH_TRUE` and `DECIDER_BRANCH_FALSE`.

Return Values The result code.

Example The following code is a sample subclass implementation of this method.

```
Decider::Result attributelogic::decide
   (
     PropertyListElement *request,
     PropertyListElement *response,
     const bool trace,
     string& decider_branch_path
   ) {
   Logger::log(VAL_CHAN_OP, ENFORCER_LOG_DEBUG, "Invoking
     attributelogic plugin");
   decider_branch_path = DECIDER_BRANCH_FALSE;
   if ( evaluate(*request) ) {
     decider_branch_path = DECIDER_BRANCH_TRUE;
   }
   return Decider::CONTINUE;
}
```

See Also

- Overriding the decide() Method in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*
- Evaluating a Policy Node in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*

factory()

Description Create an instance based on the given XML properties; must be defined by the decider subclass.

Class `Decider`

Include File `Decider.h`

Syntax

```
Decider *                       factory(
   const PropertyListElement * props
);
```

Parameters

- `props` — A pointer to the property list in the plugin's `component.xml` file that contains the configuration properties for the plugin.

Return Values An instance of the Policy Validator decider plugin object.

Example The following code is a sample subcless implementation of this method, which calls a subclass constructor called `attributelogic`:

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

See Also

- Creating Decision Point Plugin Instances in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*

init()

Description Registers a Decider plugin with the `DeciderMgr` class.

Class `Decider`

Include File `Decider.h`

Syntax

```
PluginEntry * init();
```

Parameters None.

Remarks The `init()` function 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 an evaluator 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 in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.
- The plugin type: Decider plugins are registered as type `DECIDER`.
- A pointer to a `factory()` method: The `factory()` method provides the Policy Validator with a pointer to an Evaluator plugin instance.

  The table of plugin entries is terminated with a NULL plugin entry.

Example The following code is a sample subclass implementation of this method.

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

See Also

- `factory()` on page 115
- Creating a component.xml File in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*
- Registering an Authentication Plugin in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*

5 The User API

The Select Access C/C++ API includes several classes used to manage identity data. These classes are used for creating transient identities, retrieving personalization information, accessing the identity cache, and obtaining an LDAP connection. This chapter provides a reference for these functions.

For details on what transient identities are and how to create them, see Chapter 8, Transient Directory Profiles: the User API, in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.

Classes in this API

The identity management API contains the following three classes:

Table 16 Class Overview

Class	Description	Page
`User`	This class stores all pertinent identity data, such as what dynamic group and group memberships an identity possesses.	118
`UserCache`	The Policy Validator caches identity data to increase performance. This enables the Policy Validator to access the local cache instead of performing an LDAP search. Additionally, transient identities are not stored in a directory server, but instead are added directly to and stored in the cache. This class allows you to access and manage the cache.	145
`UserSource`	The `UserSource` class is both a singleton that manages a pool of identity data location records, and the identity data location records themselves. Each `UserSource` record keeps track of the correspondence between a base DN and a directory server.	155

User

This class stores all pertinent identity data, such as what dynamic group and group memberships an identity possesses.

You can use this class to:

- Determine if an identity is a transient identity.
- Obtain the `UserSource` in which the identity is located.
- Get information about the identity.

Class Summary

Table 17 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 17 User Class Methods

Method	Description	Page
Construction and destruction		
`User(UserSource, char, char)`	Constructor. Creates a new LDAP record object for the identity.	121
`~User()`	Destructor. Releases identity's LDAP record objects.	NA
Member methods		
`buildGroupsAndRoles()`	Gets a list of both the groups and roles to which the current identity belongs.	118
`copyUserInfo(PropertyListElement, bool)`	Copies information from the current identity's LDAP record into an XML property list, typically the response.	123
`createStandardKeys()`	Creates standard keys for the current identity's LDAP record.	124
`exists()`	Determines whether the current identity is a real cache entry, that is, whether such an identity actually exists. This type of entry contrasts against a transient identity entry. For details on this type of identity entry, see Chapter 8, Transient Directory Profiles: the User API, in the HP OpenView Select Access 6.2 Developer's Tutorial Guide.	125
`fillResponse(PropertyListElement, PropertyListElement)`	Copies information from an identity's LDAP record into a property list in the Policy Validator response.	126

Table 17 User Class Methods (cont'd)

Method	Description	Page
`getDN()`	Retrieves the current identity's distinguished name.	127
`getInfo()`	Gets all known information about the current identity.	128
`getKeys()`	Gets all of the cache keys for the current identity's LDAP record.	129
`getMemberships()`	Gets all roles and groups that the current identity is a member of.	130
`getName()`	Retrieves the name of the current identity.	131
`getQuery()`	Gets the query that is used to locate the current identity in the Identity Data Location.	132
`getUids()`	Gets all the IDs corresponding to the current identity.	133
`getUserSource()`	Retrieves the identity data location where the current identity came from.	134
`infoAdd(char, Bytes)`	Add an item to the identity's information.	135
`infoFindAll(char)`	Finds and returns all matches for a key in the current identity's LDAP record.	136
`infoIsEmpty()`	Determines whether there is any information in the identity's LDAP record.	137
`isSynthetic()`	Determines whether the current identity's LDAP record is the record of a transient identity.	138
`isUpToDate(auto_LDAPValue)`	Determines whether the current identity's LDAP record (including groups and roles) is completely up to date.	139
`keyAdd(char)`	Adds a key to an identity's LDAP record.	140
`keyClear()`	Clears the identity's keys.	141

Table 17 User Class Methods (cont'd)

Method	Description	Page
`sCreateKey(char, char)`	Creates a lowercase key for indexing the cache. Keys take the form of `uid=dn`.	142
`sGetCopyUserInfo()`	Determines whether the identity information is being copied to the replies.	143
`sSetCopyUserInfo(bool)`	Specifies whether or not identity information is to be copied.	144

Definitions

`User` has one member type definition.

Table 18 Definitions Overview

Definition Type	Description
Information matches	A vector, named `InfoMatch`, has been defined to contain the results of matched information found.

User(UserSource, char, char)

Description Constructor. Creates a new LDAP record object for the identity.

Class `User`

Include File `User.h`

Syntax

```
User(
     const UserSource *    source,
     const char *          const dn,
     const char *          const query
);
```

Parameters

- `source` — The location in which the identity was found.
- `dn` — The identity.
- `query` — The query used to locate the identity.

Return Values None.

buildGroupsAndRoles()

Description	Gets a list of both the groups and roles to which the current identity belongs.
Class	`User`
Include File	`User.h`
Syntax	`void buildGroupsAndRoles();`
Parameters	None.
Return Values	None.
Remarks	These are stored in the identity's LDAP object.

copyUserInfo(PropertyListElement, bool)

Description Copies information from the current identity's LDAP record into an XML property list, typically the response.

Include File `User.h`

Syntax

```
const bool copyUserInfo(
     PropertyListElement * data,
     const bool isUser=true
):
```

Parameters

- `data` — A pointer to the identity information from the data record.
- `isUser` — A boolean which specifies whether the record corresponds to a real identity.

Return Values

- `true` — The DN in the identity's LDAP record was modified.
- `false` — No data was modified.

Example The following code sample determines whether an identity exists in the cache, then copies the information the newly-created identity's LDAP record information and adds that information to a property list in the Policy Validator response.

```
if ((userp != NULL) && userp->exists()) {
  copyInfo = true;}
.
.
.
if (copyInfo) {
  if (userp->copyUserInfo(data)) {
    result = R_RESTART;
  }
  userp->fillResponse(response, personalizer);
}
```

See Also

- `fillResponse(PropertyListElement, PropertyListElement)` on page 126

createStandardKeys()

Description	Creates standard keys for the current identity's LDAP record.
Parent Class	`User`
Include File	`User.h`
Syntax	`void createStandardKeys();`
Parameters	None.
Return Values	None.

exists()

Description Determines whether the current identity is a real cache entry, that is, whether such an identity actually exists. This type of entry contrasts against a transient identity entry. For details on this type of identity entry, see Chapter 8, Transient Directory Profiles: the User API, in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.

Class `User`

Include File `User.h`

Syntax

```
const bool exists();
```

Parameters None.

Return Values

- `true` — The current identity actually exists.
- `false` — The current identity has no entry in the directory and therefore is considered transient.

Remarks This method tests the current identity using the `infoIsEmpty()` method, returning the opposite result.

Example The following code sample determines whether an identity exists in the cache, then copies the information the newly-created LDAP record information and adds that information to a property list in the Policy Validator response.

```
if ((userp != NULL) && userp->exists()) {
   copyInfo = true;}
.
.
.
if (copyInfo) {
   if (userp->copyUserInfo(data)) {
      result = R_RESTART;
   }
   userp->fillResponse(response, personalizer);
}
```

See Also

- `infoIsEmpty()` on page 137

fillResponse(PropertyListElement, PropertyListElement)

Description Copies information from an identity's LDAP record into a property list in the Policy Validator response.

Class `User`

Include File `User.h`

Syntax

```
void fillResponse(
     PropertyListElement *           data,
     const PropertyListElement *     const personalizer
);
```

Parameters

- `data` — A pointer to a property list containing all non-personalization data for the identity.
- `personalizer` — A pointer to a property list containing the personalization information for the identity.

Return Values None.

Example The following code sample determines whether an identity exists in the cache, then copies the information the newly-created LDAP record information and adds that information to a property list in the Policy Validator response.

```
if ((userp != NULL) && userp->exists()) {
  copyInfo = true;}
.
.
.
if (copyInfo) {
  if (userp->copyUserInfo(data)) {
    result = R_RESTART;
  }
  userp->fillResponse(response, personalizer);
}
```

See Also

- `copyUserInfo(PropertyListElement, bool)` on page 123

getDN()

Description Retrieves the current identity's distinguished name.

Class `User`

Include File `User.h`

Syntax

```
const char * const getDN() const;
```

Parameters None.

Return Values The distinguished name of the current user.

Example The following code sample retrieves the current identity's DN, then gets all IDs associated with the identity and creates keys for each of them.

```
keyAdd(getDN());
const TextVector uids = getUids();
for (TextVector::const_iterator i = uids.begin(); i != uids.end();
     ++i) {
   const string key = sCreateKey(*i, m_userSource->getBaseDN());
   keyAdd(key.c_str());
}
```

getInfo()

Description Gets all known information about the current identity.

Class `User`

Include File `User.h`

Syntax

```
const UserInfo & getInfo() const;
```

Parameters None.

Return Values A `UserInfo` object which contains all known information about the current identity; a multimap of keys to values.

Example The following code sample loops through the identity cache.

```
     for (UserRefPtrVector::const_iterator ie(entries.begin());
          ie != entries.end(); ie++) {
       UserRefPtr entry = *ie;

       bool firstValue = true;
       typedef UserInfo::const_iterator ICI;
       pair<ICI, ICI> ier = entry->getInfo().equal_range(attrName);
       for (ICI ii = ier.first; ii != ier.second; ii++) {
         const char *data = ii->second.data();
//continue
```

getKeys()

Description Gets all of the cache keys for the current identity's LDAP record.

Class `User`

Include File `User.h`

Syntax

```
const UserKeys & getKeys() const;
```

Parameters None.

Return Values A `UserKeys` object containing the cache keys for the current identity's LDAP record.

Remarks Keys are based on DN and UIDs (possibly several).

See Also

- `getMemberships()` on page 130

getMemberships()

Description Gets all roles and groups that the current identity is a member of.

Class `User`

Include File `User.h`

Syntax

```
const UserMemberships ** getMemberships();
```

Parameters None.

Return Values A vector of all the roles and groups to which the identity belongs. The vector is returned as an object of type `UserMemberships`.

Example The following code sample report all memberships for this identity, including groups and roles.

```
VALIDATOR_API const UserMemberships **
User::getMemberships()
{
   return (const UserMemberships **)m_memberships;
}
```

See Also

- `getKeys()` on page 129

getName()

Description Retrieves the name of the current identity.

Class `User`

Include File `User.h`

Syntax

```
const char * const getName() const;
```

Parameters None.

Return Values The name of the current identity.

Example The following code sample checks to see what the identity's name is.

```
VALIDATOR_API const char * const
User::getName() const
{
   return m_name.get();
}
```

getQuery()

Description Gets the query that is used to locate the current identity in the Identity Data Location.

Class `User`

Include File `User.h`

Syntax

```
const char * const getQuery() const;
```

Parameters None.

Return Values The query used to find the current identity.

Remarks This function was previously used for debugging purposes.

getUids()

Description Gets all the IDs corresponding to the current identity.

Class `User`

Include File `User.h`

Syntax

```
TextVector getUids() const;
```

Parameters None.

Return Values A vector containing the UID strings.

Remarks This method creates a new `TextVector` object, but aliases the strings inside that vector.

Example The following code sample retrieves the current identity's DN, then gets all IDs associated with the identity and creates keys for each of them.

```
keyAdd(getDN());
const TextVector uids = getUids();
for (TextVector::const_iterator i = uids.begin(); i != uids.end();
  ++i) {
   const string key = sCreateKey(*i, m_userSource->getBaseDN());
   keyAdd(key.c_str());
}
```

See Also

- `getKeys()` on page 129

getUserSource()

Description	Retrieves the identity data location where the current identity came from.
Class	`User`
Include File	`User.h`
Syntax	`const UserSource * getUserSource() const;`
Parameters	None.
Return Values	The Identity Data Location from which the current identity came.
Remarks	This function is used to access things shared by all identities from a source, for example, certificate verification.

infoAdd(char, Bytes)

Description Add an item to the identity's information.

Class `User`

Include File `User.h`

Syntax

```
void infoAdd(
     const char *     key,
     const Bytes &    value
);
```

Parameters

- `key` — The name of the key to be added to the identity's LDAP record.
- `value` — The value of the key to be added to the identity's LDAP record.

Return Values None.

Example The following code sample checks to see if the identity is a transient identity. If so, it adds the identity's DN to the identity's LDAP record. Then, if the UID in the certificate is empty, it also adds a UID to the record as well.

```
else if (synthetic) {
  user->infoAdd("dn", Bytes(dn));

  if (! strEmpty(uid)) {
    user->infoAdd("uid", Bytes(uid));
```

infoFindAll(char)

Description Finds and returns all matches for a key in the current identity's LDAP record.

Class `User`

Include File `User.h`

Syntax

```
InfoMatch infoFindAll(
     const char * const key
);
```

Parameters

- `key` — The key to be matched.

Return Values A vector containing all matches.

Example The following code sample looks to find all instances where an identity password exists during the Policy Validator authentication process. If it does not, to then generate an error stating the password was not found.

```
User::InfoMatch match = userp->infoFindAll("userpassword");
       if (match.empty()) {
         Logger::log(VAL_CHAN_AUTH, ENFORCER_LOG_DEBUG,
           "password::authenticate entry has no password\n");
         result = PW_ERR_PASSWORD_NOT_FOUND;
```

infoIsEmpty()

Description Determines whether there is any information in the identity's LDAP record.

Class `User`

Include File `User.h`

Syntax

```
const bool infoIsEmpty();
```

Parameters None.

Return Values

- `true` — No information exists in the identity's LDAP record.
- `false` — The identity's LDAP record contains information.

Remarks This function is only empty for non-existent identities.

See Also

- `exists()` on page 125

isSynthetic()

Description Determines whether the current identity's LDAP record is the record of a transient identity.

Class `User`

Include File `User.h`

Syntax

```
const bool isSynthetic() const;
```

Parameters None.

Return Values

- `true` — The current identity's LDAP record is for a transient identity.
- `false` — The current identity's LDAP record is not for a transient identity.

Example The following code sample tries to locate the identity's LDAP record for the given UID. If the identity is found, it determines whether or not the identity's LDAP record is transient, and if so, whether the current plugin created the entry.

```
userp = UserCache::sLocateByUID(*iter, user);
if ((userp != NULL) && userp->exists() )
{
   if
((!userp->isSynthetic())||(userp->getUserSource()==syntheticSource))
   {
   return true;
   }
}
```

isUpToDate(auto_LDAPValue)

Description Determines whether the current identity's LDAP record (including groups and roles) is completely up to date.

Class `User`

Include File `User.h`

Syntax

```
const bool isUpToDate(
     auto_LDAPValue & mod
);
```

Parameters

- `mod` — A pointer to the last modification date of the identity's LDAP record.

Return Values

- `true` — The identity's LDAP record is up to date.
- `false` — The identity's LDAP record is not currently up to date.

Remarks You are not expected to need this function.

keyAdd(char)

Description Adds a key to an identity's LDAP record.

Class `User`

Include File `User.h`

Syntax

```
void keyAdd(
    const char *   key
);
```

Parameters

- `key` — The key to be added to the identity's LDAP record.

Return Values None.

Example The following code sample retrieves the current identity's DN, then gets all UIDs associated with the identity and creates keys for each of them.

```
keyAdd(getDN());
const TextVector uids = getUids();
for (TextVector::const_iterator i = uids.begin(); i != uids.end();
     ++i) {
   const string key = sCreateKey(*i, m_userSource->getBaseDN());
   keyAdd(key.c_str());
}
```

See Also

- `getUids()` on page 133

keyClear()

Description Clears the identity's keys.

Class `User`

Include File `User.h`

Syntax

```
void keyClear();
```

Parameters None.

Return Values None.

Remarks You are not expected to need this function.

Example The following code sample erases an identity's LDAP record's current keys.

```
user->keyClear();
```

sCreateKey(char, char)

Description Creates a lowercase key for indexing the cache. Keys take the form of `uid=dn`.

Class `User`

Include File `User.h`

Syntax

```
const String sCreateKey(
     const char *     uid,
     const char *     baseDN
);
```

Parameters

- `uid` — The UID from which the index key will be created.
- `baseDN` — The search base.

Return Values An index key.

Example The following code sample retrieves the current identity's DN, then gets all UIDs associated with the identity and creates keys for each of them.

```
   keyAdd(getDN());
   const TextVector uids = getUids();
   for (TextVector::const_iterator i = uids.begin(); i != uids.end();
++i) {
     const string key = sCreateKey(*i, m_userSource->getBaseDN());
     keyAdd(key.c_str());
   }
```

sGetCopyUserInfo()

Description Determines whether the identity information is being copied to the replies.

Class `User`

Include File `User.h`

Syntax

```
const bool sGetCopyUserInfo();
```

Parameters None.

Return Values

- `true` — The identity information is being copied.
- `false` — The identity information is not being copied.

Remarks You are not expected to need this function.

See Also

- `sSetCopyUserInfo(bool)` on page 144

sSetCopyUserInfo(bool)

Description Specifies whether or not identity information is to be copied.

Class `User`

Include File `User.h`

Syntax

```
void sSetCopyUserInfo(
     const bool setting
);
```

Parameters

- `setting` — A value indicating whether the identity information is to be copied.

See Also

- `sGetCopyUserInfo()` on page 143

UserCache

The Policy Validator caches identity data to increase performance. This enables the Policy Validator to access the local cache instead of performing an LDAP search. Additionally, transient identities are not stored in a directory server, but instead are added directly to and stored in the cache. This class allows you to access and manage the cache.

Class Summary

The constructor for `UserCache` is protected. Select Access automatically creates one instance at startup. Do not delete this constructor.

The `UserCache` class contains the following methods.

Table 19 Class Overview

Method	Description	Page
`sCleanup(int, int)`	Removes old records from a configuration-specified fraction of the whole cache.	146
`sClear()`	Deletes all records in the identity cache.	147
`sCreateSynthetic(UserSource, char, char)`	Creates a transient identity. If the transient identity already exists, this method returns it. Otherwise it throws a cache exception to cause the identity to be created.	148
`sCreateTemporary(UserSource, char, char)`	Creates a temporary identity in the directory server as well as in the cache.	149
`sInvalidate()`	Marks all entries in the identity cache as invalid, so that they will be updated the next time they are accessed.	150
`sLocateByDN(char)`	Finds an identity's LDAP record by the given DN.	151
`sLocateByDN(UserSource, char)`	Finds an identity's LDAP record by the given DN.	152
`sLocateByUID(UserSource, char)`	Finds an identity's LDAP record by the given ID below a specified identity data location. The identity's LDAP record is loaded into the cache if necessary.	153
`sRefresh(UserRefPtr)`	Refreshes the given identity's LDAP record in the cache.	154

sCleanup(int, int)

Description Removes old records from a configuration-specified fraction of the whole cache.

Class `UserCache`

Include File `UserCache.h`

Syntax

```
void sCleanup(
     const int percent,
     const int trace
);
```

Parameters

- `percent` — The amount of the cache, as a percentage, that will be cleaned.
- `trace` — Indicates whether a report will be produced for the action.

Return Values None.

Remarks You are not expected to need this function.

sClear()

Description	Deletes all records in the identity cache.
Class	`UserCache`
Include File	`UserCache.h`
Syntax	`void sClear();`
Parameters	None.
Return Values	None.
Remarks	You are not expected to need this function.

sCreateSynthetic(UserSource, char, char)

Description

Creates a transient identity. If the transient identity already exists, this method returns it. Otherwise it throws a cache exception to cause the identity to be created.

A transient/synthetic identity is someone who was authenticated using SecurID or RADIUS, but for whom there is no LDAP record. Transient identities never go stale but stay in the cache forever.

Class

`UserCache`

Include File

`UserCache.h`

Syntax

```
UserRefPtr sCreateSynthetic(
     const UserSource *     userSource,
     const char *           const dn,
     const char *           const uid
);
```

Parameters

- `userSource` — The apparent identity location.
- `dn` — The identity's distinguished name.
- `uid` — The identity's UID.

Return Values

The transient identity.

Example

The following code example indicates that no record was found, but one can be synthesized.

```
else if (synthetic_loc){
  string syntheticDN("uid=");
  syntheticDN += username;
  syntheticDN += ",";
  syntheticDN += synthetic_loc;
  userp = UserCache::sCreateSynthetic(syntheticSource,
    syntheticDN.c_str(), username);
  copyInfo = true;
```

See Also

- `sCreateTemporary(UserSource, char, char)` on page 149

sCreateTemporary(UserSource, char, char)

Description Creates a temporary identity in the directory server as well as in the cache.

Class `UserCache`

Include File `UserCache.h`

Syntax

```
UserRefPtr sCreateTemporary(
     const UserSource *    userSource,
     const char *          const  subject_name,
     const char *          const  base_dn
     UserCacheAttributes & attributes
);
```

Parameters

- `userSource` — The location of the temporary identity.
- `subject_name` — The subject name of the temporary identity.
- `base_dn` — The base DN of the temporary identity.
- `attributes` — The attributes the temporary identity has.

Return Values The temporary identity.

Example The following code sample creates a temporary record on the directory server along with the identity-specific attributes. This example returns NULL if the record could not be created for any reason.

```
VALIDATOR_API UserRefPtr
UserCache::sCreateTemporary(
   const UserSource *userSource,
   const char * const subject_name,
   const char * const base_dn,
   UserCacheAttributes &attributes
){
   auto_ptarray<char> dn(build_DN(subject_name, base_dn));

   LdapConnection *ldapCnxn = userSource->getLdapConnection();
```

See Also

- `sCreateSynthetic(UserSource, char, char)` on page 148

sInvalidate()

Description	Marks all entries in the identity cache as invalid, so that they will be updated the next time they are accessed.
Class	`UserCache`
Include File	`UserCache.h`
Syntax	`void sInvalidate();`
Parameters	None.
Return Values	None.
Remarks	You are not expected to need this function.

sLocateByDN(char)

Description Finds an identity's LDAP record by the given DN.

Class `UserCache`

Include File `UserCache.h`

Syntax

```
UserRefPtr *          sLocateByDN(
     const char *     const dn
);
```

Parameters

- `dn` — The identity's distinguished name.

Return Values The LDAP record for identity, or `NULL` if the identity cannot be located.

Remarks This function searches all identity locations. Typically you should use `sLocateByUID`.

See Also

- `sLocateByUID(UserSource, char)` on page 153

sLocateByDN(UserSource, char)

Description Finds an identity's LDAP record by the given DN.

Class `UserCache`

Include File `UserCache.h`

Syntax

```
UserRefPtr *                sLocateByDN(
     const UserSource *     userSource,
     const char *           const dn
);
```

Parameters

- `userSource` — The base for the identity search.
- `dn` — The identity's distinguished name.

Return Values The identity's LDAP record, or `NULL` if the identity cannot be located.

Remarks This function searches all identity locations. Typically you should use `sLocateByUID`.

Example The following code sample attempts to locate an identity's LDAP record in the cache. If it is unsuccessful, an error is generated.

```
User *user = UserCache::sLocateByDN(userSrc, dn);
if ( ! user ) {
   throw EvaluatorInternal(EnforcerException::E_INVALID_ARG, LOCATION,
     "User cannot be found in User Cache");
}
```

See Also

- `sLocateByUID(UserSource, char)` on page 153

sLocateByUID(UserSource, char)

Description Finds an identity's LDAP record by the given ID below a specified identity data location. The identity's LDAP record is loaded into the cache if necessary.

Class `UserCache`

Include File `UserCache.h`

Syntax

```
UserRefPtr *                sLocateByUID(
     const UserSource *     UserSource
     const char *           const uid,
);
```

Parameters

- `UserSource` — The base for the identity search.
- `uid` — The ID of the identity's LDAP record.

Return Values The identity's LDAP record, or `NULL` if the identity cannot be located.

Remarks This function is the preferred method of retrieving identity information.

Example The following code sample tries to locate the identity's LDAP record for the given ID. If the identity is found, it determines whether or not the identity's LDAP record is transient, and if so, whether the current plugin created the entry.

```
userp = UserCache::sLocateByUID(*iter, user);
if ((userp != NULL) && userp->exists() )
{
  if
((!userp->isSynthetic())||(userp->getUserSource()==syntheticSource))
  {
    return true;
  }
}
```

sRefresh(UserRefPtr)

Description Refreshes the given identity's LDAP record in the cache.

Class `UserCache`

Include File `UserCache.h`

Syntax

```
UserRefPtr sRefresh(
     UserREfPtr & user
);
```

Parameters

- `user` — The identity's LDAP record to be refreshed.

Return Values The most recently updated version of `user`.

Remarks You are not expected to need this function.

UserSource

The `UserSource` class is both a singleton that manages a pool of identity data location records, and the identity data location records themselves. Each `UserSource` record keeps track of the correspondence between a base DN and a directory server.

This 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 LDAP server that stores the identity data. This is useful when you want to query the LDAP server for additional identity data.

Class Summary

Identity data location records can only be created by Select Access. They must not be created or destroyed by identities.

The `UserSource` class contains the methods listed in Table 20.

Table 20 Class Overview

Method	Description	Page
`getBaseDn()`	Returns the base DN of the current identity data location. This is the DN below which all identities in that identity data location are stored.	156
`getLdapConnection()`	Returns the current LDAP connection handle for the identity data locations.	157
`getName()`	Returns the name of the current identity data location.	158
`getUseServerAuth()`	Determines whether server-side authentication will be used. Some directories require this. Administrators can set this ability in Select Access.	159
`sAdd(char, LDAPConnection, char, bool)`	Creates an identity's LDAP record and adds it to the singleton. The LDAP connection is closed and freed when the identity data location is destroyed.	160
`sAddFromXml(XmlTree Node)`	Creates an identity source and adds it to the singleton based on a configuration in an XML property list.	161
`sClear()`	Deletes all identity data locations from the singleton.	162
`sDelete(UserSource)`	Deletes the given record from the identity data location.	163
`sGetByBaseDN(char)`	Finds the identity data location that contains the given base DN.	164
`sGetByDN(char)`	Finds the identity data location for the given DN.	165
`sGetByName(char)`	Finds an identity source given its name, as configured by the administrator.	166
`sGetList()`	Gets the list of all known identity data locations.	167

getBaseDn()

Description Returns the base DN of the current identity data location. This is the DN below which all identities in that identity data location are stored.

Class `UserSource`

Include File `UserSource.h`

Syntax

```
const char * const getBaseDN() const;
```

Parameters None.

Return Values The base DN of the current identity data location.

Example The following code sample retrieves the current identity's DN, then gets all IDs associated with the identity and creates keys for each of them.

```
keyAdd(getDN());
const TextVector uids = getUids();
for (TextVector::const_iterator i = uids.begin(); i != uids.end();
     ++i) {
  const string key = sCreateKey(*i, m_userSource->getBaseDN());
  keyAdd(key.c_str());
}
```

getLdapConnection()

Description Returns the current LDAP connection handle for the identity data locations.

Class `UserSource`

Include File `UserSource.h`

Syntax

```
LdapConnection * getLdapConnection() const;
```

Parameters None.

Return Values The LDAP connection handle.

Remarks Can be used to get information about the directory type. For example, Active Directory.

Example The following code sample attempts to get the current LDAP connection handle. If it is unsuccessful, an error is generated.

```
LdapConnection *ldapConn = userSrc->getLdapConnection();
if ( ! ldapConn ) {
   throw EvaluatorInternal(EnforcerException::E_INVALID_ARG, LOCATION,
     "User source has an invalid LdapConnection");
}
```

getName()

Description Returns the name of the current identity data location.

Class `UserSource`

Include File `UserSource.h`

Syntax

```
const char * const getName() const;
```

Parameters None.

Return Values The name of the current identity data location, as configured by the administrator.

Example The following code sample attempts to determine the source for the given DN. If it can't locate the identity data location, an error is logged.

```
   const UserSource * tmpSource = sGetByDN(syntheticLoc.c_str());
   if ( userSource && tmpSource != userSource ) {
      Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR, "Synthetic user
        location %s does not lie below user source %s",
        syntheticLoc.c_str(), userSource->getName());
      return false;
}
```

getUseServerAuth()

Description Determines whether server-side authentication will be used. Some directories require this. Administrators can set this ability in Select Access.

Class `UserSource`

Include File `UserSource.h`

Syntax

```
const bool getUseServerAuth() const;
```

Parameters None.

Return Values

- `true` — Server-side authentication should be used.
- `false` — Server-side authentication should not be used.

Remarks You are not expected to need this function.

sAdd(char, LDAPConnection, char, bool)

Description Creates an identity's LDAP record and adds it to the singleton. The LDAP connection is closed and freed when the identity data location is destroyed.

Class `UserSource`

Include File `UserSource.h`

Syntax

```
const bool sAdd(
     const char *          const name,
     LdapConnection *      ldapCnxn,
     const char *          const baseDN,
     const bool            useServerAuth
);
```

Parameters

- `name` — The name of the identity data location you are adding the record to.
- `ldapCnxn` — The LDAP connection handle.
- `baseDN` — The base DN of the identity data location.
- `useServerAuth` — Specifies whether server-side authentication will be used.

Return Values

- `SUCCESS` — The record was successfully added.
- `FAILURE` — The record could not be added.

Remarks The LDAP connection is closed and freed when the identity data location is destroyed. Developers should not call this function.

sAddFromXml(XmlTreeNode)

Description Creates an identity source and adds it to the singleton based on a configuration in an XML property list.

Class `UserSource`

Include File `UserSource.h`

Syntax

```
const bool sAddFromXml(
     XmlTreeNode * config
);
```

Parameters

- `config` — A pointer to a property list containing the description of the record.

Return Values

- `SUCCESS` — The record was successfully created and added.
- `FAILURE` — The record could not be added.

Remarks You are not expected to need this function.

sClear()

Description	Deletes all identity data locations from the singleton.
Class	`UserSource`
Include File	`UserSource.h`
Syntax	`void sClear();`
Parameters	None.
Return Values	None.
Remarks	You are not expected to need this function.

sDelete(UserSource)

Description Deletes the given record from the identity data location.

Class `UserSource`

Include File `UserSource.h`

Syntax

```
void sDelete(
     const UserSource * record
);
```

Parameters

- `record` — A pointer to the record to be removed. The given record must exist in the identity data location singleton.

Return Values None.

Remarks You are not expected to need this function.

sGetByBaseDN(char)

Description Finds the identity data location that contains the given base DN.

Class `UserSource`

Include File `UserSource.h`

Syntax

```
const UserSource *      sGetByBaseDN(
     const char *       const  baseDN
);
```

Parameters

- `baseDN` — The base DN of the identity data location you are trying to locate.

Return Values The specified identity data location, or `NULL` if the base DN cannot be found.

Remarks This function performs a reverse lookup instead of the base DN for identity data locations.

See Also

- `getBaseDn()` on page 156

sGetByDN(char)

Description Finds the identity data location for the given DN.

Class `UserSource`

Include File `UserSource.h`

Syntax

```
const UserSource *      sGetByDN(
     const char *       const dn
);
```

Parameters

- `dn` — The distinguished name of the identity data location you are trying to locate.

Return Values The specified identity data location, or `NULL` if the given DN cannot be found.

Remarks Given an identity DN, this function finds the first identity data location that could possibly contain it.

Example The following code sample attempts to determine the source for the given DN. If it can't locate the identity data location, an error is logged.

```
  const UserSource * tmpSource = sGetByDN(syntheticLoc.c_str());
  if ( userSource && tmpSource != userSource ) {
    Logger::log(VAL_CHAN_OP, ENFORCER_LOG_ERROR, "Synthetic user
      location %s does not lie below user source %s",
      syntheticLoc.c_str(), userSource->getName());
    return false;
}
```

See Also

- `sGetByBaseDN(char)` on page 164

sGetByName(char)

Description Finds an identity source given its name, as configured by the administrator.

Class `UserSource`

Include File `UserSource.h`

Syntax

```
const UserSource *      sGetByName(
     const char *       const  name
);
```

Parameters

- `*name` — The name of the identity data location you want to locate.

Return Values The specified identity data location, or `NULL` if no record exists with the given name.

Remarks Instead of looking up by contents of the DN, the lookup is performed by the configured name of the identity data location.

sGetList()

Description	Gets the list of all known identity data locations.
Class	`UserSource`
Include File	`UserSource.h`
Syntax	`UserSourceVector & sGetList();`
Parameters	None.
Return Values	A vector containing a list of all known identity data locations.
Remarks	This function is used to iterate over the list. You are not expected to need this function.

6 C++ XML Manipulation API

The Enforcer plugin communicates with the Policy Validator using XML documents. The Enforcer API includes three classes that allow you to manipulate XML. This chapter provides a reference for methods in these classes in C++.

Understanding XML Elements

The XML used by Select Access is extremely simple. An XML tree in Select Access can be composed of only two type of elements:

- Property: A property is a name/value pair. That is, 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>.
  ```

- Property list: A property list is an element which contains a list of zero or more properties, as well as zero or more nested property lists. An example of a `PropertyListElement` is:

  ```
  <PROPERTYLIST NAME="setting">
     <PROPERTY NAME="color">red</PROPERTY>
  </PROPERTYLIST>.
  ```

Classes in this API

The XML manipulation API contains a class for each of the element types (`PropertyElement` and `PropertyListElement`) as well as a parent class, `XmlTreeNode`, which contains methods that can be used to handle nodes of either kind.

Table 21 Class Overview

Class	Description	Page
`XmlTreeNode`	This class stores the XML nodes of a parsed XML document. It provides low-level access to XML attributes. Typically, you will use the `PropertyListElement` and `PropertyElement` classes to access XML data.	171
`PropertyElement`	Extends `XmlTreeNode`. 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.	228
`PropertyListElement`	Extends `XmlTreeNode`. This class stores an XML node that contains zero or more `PropertyElement` tags. Property list themselves may be nested; a `PropertyListElement` may contain zero or more property lists.	234

While several third-party APIs allow you to manipulate XML, they are overly complex for the needs of Select Access, and therefore possess unnecessary overhead. Because it has been designed to meet the specific needs of Select Access, the Enforcer XML API dramatically improves performance over these third-party libraries. We therefore strongly recommend using these libraries when building Enforcer plugins.

XmlTreeNode

This class stores the XML nodes of a parsed XML document. It provides low-level access to XML attributes. Typically, you will use the `PropertyListElement` and `PropertyElement` classes to access XML data.

Class Summary

The constructor for `XmlTreeNode` is protected, and therefore is not used directly to create new XML nodes. To create a new node of a specified element type, use the `sNew()` method. For more information, see `sNew()` on page 225.

Table 22 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 22 XmlTreeNode Class Summary

Method Name	Description	Page
Construction and Destruction		
`~XmlTreeNode`	Destructor. Releases XML nodes and their contents.	NA
Member Methods		
`appendChild()`	Appends a new child node to the current property list.	176
`appendChildInt()`	Appends a new child containing a stringified integer to the current property list.	177
`appendChildrenFromNode()`	Appends all children of the given source node that have the given element type to the current property list.	178
`appendChildString()`	Appends a new child node with the given `NAME` attribute and string value to the current property list.	179
`appendMultiResource()`	Appends the given resource path to an existing multi-resource property list.	180
`appendStringValue(char)`	Appends the given string to an existing string value. If no string currently exists, this function creates one.	181
`appendStringValue(char, int)`	Appends a string of the given length to an existing string value. If no destination string exists, this function creates one.	182
`deleteAttribute()`	Deletes the attribute with the given name.	183
`deleteChild(char)`	Deletes the child with the given name from the current property list.	184

Table 22 XmlTreeNode Class Summary (cont'd)

Method Name	Description	Page
deleteChild(int)	Deletes the child found at the given index from the current property list.	185
deleteChild(XmlTreeNode)	Deletes the specified child from the current property list.	186
deleteChildrenByElementType()	Deletes all the children with the given element type from the current property list.	187
deleteChildrenByName)	Deletes all the children with the given name attribute from the current property list.	188
duplicateTreeNode()	Duplicates the given tree node, including all children (if any), attributes and string values.	189
elementTypeIs()	Compares a node's element type to the given tag.	190
getAttributeName()	Gets the name of the attribute at the given index in the current property or property list.	191
getAttributeValue(char)	Gets the string value of the attribute with the given name.	192
getAttributeValue(int)	Gets the string value of the attribute at a given index in the current property or property list.	193
getChild(char)	Gets the child node with the given NAME attribute from the current property list.	194
getChild(int)	Gets the child node found at the given index within the current property list.	195
getChildElement()	Gets the first element with the given element tag contained within the current property list.	196
getChildLongValue()	Translates the string value of the given child node of the current property list into a long integer.	197
getChildNoCase()	Gets the child node with the given name from the current property list. The name-matching performed is case-insensitive.	198
getChildNoCaseStringValue()	Gets the string value of the child node with the given name in the current property list. The name-matching performed is case-insensitive.	199

Table 22 XmlTreeNode Class Summary (cont'd)

Method Name	Description	Page
`getChildren()`	Gets all children of the current property list that have the given name. If `childName` is `NULL`, this function retrieves all children.	200
`getChildStringValue()`	Gets the pointer to a COM implementation of XML manipulation API.	201
`getComFacade()`	Gets the string value associated with the child node whose `NAME` attribute has the given value.	202
`getElementStringValue()`	Gets the string value of the child with the given element tag.	203
`getElementType()`	Gets the element type of the current node.	204
`getLongValue()`	Translates the string value of the current node into a long integer.	205
`getName()`	Gets the value associated with a node's `NAME` attribute. If no name has ever been specified for the node, an empty string is returned.	206
`getNumAttributes()`	Counts the number of attributes the given node has.	207
`getNumChildren()`	Counts the number of children the current node has.	208
`getStringValue()`	Gets the current node's contained text. This value may be `NULL`.	209
`getStringValueLen()`	Gets the length of the current node's contained text. This value may be zero.	210
`isPropertyList()`	Determines whether the current node is actually a list of other properties, as opposed to a name/value pair.	211
`isPropertyValue()`	See `propertyElement::isPropertyValue()` for full details.	212
`mergeNode()`	Merges the children of the given source node into an existing node based on the element type of the children, if the node supports it.	213
`removeChild(char)`	Removes the child with the given name from the current property list.	214
`removeChild(int)`	Removes the child found at the given index from the current property list.	215

Table 22 XmlTreeNode Class Summary (cont'd)

Method Name	Description	Page
`removeChild(XmlTreeNode)`	Removes the specified child from a `PropertyListElement`.	216
`setAttributeValue()`	Sets the given attribute to the given value.	217
`setChildStringValue()`	Sets the string value of the child of the given node with the given name. If no such property exists, one is created.	218
`setComFacade()`	Sets a pointer to `CEnforcerXMLTree`, a COM implementation of XML manipulation API.	219
`setElementStringValue()`	Sets the value of the child with the given element tag. If no child with the given element tag exists, one is created.	220
`setElementType()`	Sets a node's element type.	221
`setName()`	Sets the `NAME` attribute for a node. If no attribute is present, this function creates one.	222
`setStringValue(char)`	Sets or changes the given node's contained text provided. `NULL` or `0` clears the string value.	223
`setStringValue(char, int)`	Sets or changes the given node's contained text provided. `NULL` or `0` clears the string value.	224
`sNew()`	Creates a new node of the type specified by the first tag.	225
`sToString(XmlTreeNode, bool)`	Converts the value of the given node into its equivalent string representation.	226
`toString()`	Converts the value of the given node into a printable string.	227

Unimplemented methods

The following methods are not implemented for the `XmlTreeNode` class.

- `appendNestedProperty()`
- `appendPropertyValue()`
- `getPropertyValueCount()`
- `getPropertyValues()`
- `getPropertyValuesIgnoreCase()`
- `getNestedPropertyCount()`
- `getNestedProperties()`
- `getNestedPropertiesIgnoreCase`

Definitions

Table 23 describes the constants defined by the `XmlTreeNode` class.

Table 23 XmlTreeNode Type Definitions

<table>
<tr><th>Definition Type</th><th>Description</th></tr>
<tr><td>Containers</td><td>Several vectors have been defined to contain sets of XML nodes. Defined vectors are listed below:
<table>
<tr><th>Name[a]</th><th>Value[a]</th><th>Description</th></tr>
<tr><td>XmlTreeNode Vector</td><td>vector<XmlTree Node*></td><td>Used to store vector sets of instances of both the PropertyElement and PropertyListElement derived classes.</td></tr>
<tr><td>PropertyElement Vector</td><td>vector<Property Element*></td><td>Used to store vector sets of instances of the PropertyElement derived class.</td></tr>
<tr><td>PropertyList ElementVector</td><td>vector<Property ListElement*></td><td>Used to store vector sets of instances of the PropertyListElement derived class.</td></tr>
</table></td></tr>
<tr><td>Node identification</td><td>Attribute_t has been defined to represent the NAME attribute of a node together with its value. It is defined as:
pair<const char *, const char *></td></tr>
<tr><td>Iterators</td><td>Two iterators have been defined. Defined iterators are listed below:
<table>
<tr><th>Name</th><th>Value</th><th>Description</th></tr>
<tr><td>AttrIter_t</td><td>vector<Attribute_ t>::iterator</td><td>Iterates over a vector of attribute name value pairs.</td></tr>
<tr><td>ChildIter_t</td><td>vector<XmlTreeNode *>::iterator</td><td>Iterates over a vector of XML nodes.</td></tr>
</table></td></tr>
</table>

a. Do not type spaces in Names or Values. Due to space restrictions in this table, these elements have been wrapped to avoid awkward breaks.

appendChild()

Description Appends a new child node to the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void appendChild(
  XmlTreeNode * newChild
);
```

Parameters

- `newChild` — An XML pointer to the new unnamed child node.

Return Values None.

Remarks The given node may only be the child of one parent.

Example The following code sample retrieves the query level setting, then test to see if the level has been set to `MAXIMAL`. If so, it creates a new XML property list, adds it to the query, and populates it with header list data, then appends a property specifying the version of the Enforcer plugin.

```
int query_level = EnforcerGetQueryLevel(Enforcer_Handle);
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));
}
```

See Also

- `appendChildInt()` on page 177
- `appendChildString()` on page 179

appendChildInt()

Description Appends a new child containing a stringified integer to the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void appendChildInt(
   const char *     name,
   const char *     value
);
```

Parameters

- `name` — The `NAME` attribute of the child to be created.
- `value` — A stringified integer containing the value of the new child to be created.

Return Values None.

Remarks The given node may only be the child of one parent.

Example The following code example allocates and initializes an XML query if an administrator configures the Enforcer plugin to have a minimal query level, given the endpoint addresses.

```
if (query_level > QUERY_MINIMAL) {

     (*query_p)->appendChildString(ENFORCER_DSTIP, SYS_STRDUP(dst_ip));
     if (strcmp(src_ip, "0.0.0.0")) {
        (*query_p)->appendChildString(ENFORCER_SRCIP,
          SYS_STRDUP(src_ip));
     }
     (*query_p)->appendChildInt(ENFORCER_DSTPORT, dst_port);
     if (src_port) {
        (*query_p)->appendChildInt(ENFORCER_SRCPORT, src_port);
     }

     (*query_p)->appendChildString(ENFORCER_DSTHOSTNAME,
          SYS_STRDUP(host));
     (*query_p)->appendChildString(ENFORCER_PROTOCOL,
          SYS_STRDUP(proto));
  }
```

See Also

- `appendChild()` on page 176
- `appendChildString()` on page 179

appendChildrenFromNode()

Description Appends all children of the given source node that have the given element type to the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void appendChildrenFromNode(
     const XmlTreeNode *     srcNode,
     const char *            elementType
);
```

Parameters

- `srcNode` — An XML pointer to the parent node containing the children that will be copied.
- `elementType` — The element type of the children to be copied. The value can be either `ENFORCER_TAG_PROPERTY` or `ENFORCER_TAG_PROPERTYLIST`.

Return Values None.

Remarks The children of the node are copied, not moved.

appendChildString()

Description Appends a new child node with the given `NAME` attribute and string value to the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void appendChildString(
     const char *     name,
     const char *     value
);
```

Parameters

- `name` — The `NAME` attribute of the child to be created.
- `value` — The value of the new child to be created.

Return Values None.

Remarks The given node may only be the child of one parent.

The XML tree takes ownership of `value`.

Example The following code sample retrieves the query level setting, then test to see if the level has been set to `MAXIMAL`. If so, it creates a new XML property list, adds it to the query, and populates it with header list data, then appends a property specifying the version of the Enforcer plugin.

```
int query_level = EnforcerGetQueryLevel(Enforcer_Handle);
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));
}
```

See Also

- `appendChild()` on page 176
- `appendChildInt()` on page 177

appendMultiResource()

Description Appends the given resource path to an existing multi-resource property list.

If no multi-resource property list currently exists, this method creates one and adds the resource.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void appendMultiResource(
     const char *     resource
);
```

Parameters

- `resource` — The path to the resource to be added to the property list.

Return Values None.

appendStringValue(char)

Description Appends the given string to an existing string value. If no string currently exists, this function creates one.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void appendStringValue(
     const char * newStringValue
);
```

Parameters

- `newStringValue` — The text that will be copied and appended to an existing string.

Return Values None.

Remarks Used when parsing XML. You are not expected to need this function.

See Also

- `appendStringValue(char, int)` on page 243
- `propertyListElement::appendStringValue(char)` on page 242
- `propertyListElement::appendStringValue(char, int)` on page 243

appendStringValue(char, int)

Description Appends a string of the given length to an existing string value. If no destination string exists, this function creates one.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void appendStringValue(
     const char *       newStringValue,
     const int          newStringValueLen
);
```

Parameters

- `newStringValue` — The text that will be copied and appended to an existing string.
- `newStringValueLen` — The length of the text to be appended. Must be a non-negative integer.

Return Values None.

Remarks Used when parsing XML. You are not expected to need this function.

See Also

- `appendStringValue(char)` on page 242
- `propertyListElement::appendStringValue(char)` on page 242
- `propertyListElement::appendStringValue(char, int)` on page 243

deleteAttribute()

Description Deletes the attribute with the given name.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void deleteAttribute(
     const char * attributeName
);
```

Parameters

- `attributeName` — The name of the attribute to be deleted. This parameter cannot have a value of `NULL`.

Return Values None.

Remarks Used when parsing XML. You are not expected to need this function.

deleteChild(char)

Description Deletes the child with the given name from the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void deleteChild(
     const char * childName
);
```

Parameters

- `childName` — The `NAME` attribute of the child to be deleted.

Return Values None.

Example The following code example removes an old XML entry if it exists.

```
PropertyListElement * userInfo = data;
if (isUser) {
   data->deleteChild(ENFORCER_USERINFO);
   userInfo = new PropertyListElement(ENFORCER_TAG_PROPERTYLIST,
     ENFORCER_USERINFO);
   data->appendChild(userInfo);
}
else {
   data->deleteChild(ENFORCER_ATTRINFO);
```

See Also

- `deleteChild(int)` on page 185
- `deleteChild(XmlTreeNode)` on page 186

deleteChild(int)

Description Deletes the child found at the given index from the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void deleteChild(
     const int childIndex
);
```

Parameters

- `childIndex` — The position in a property list of the child to be deleted. Must be a non-negative integer. The range for this value is `0` to (the *`number of children`*`-1`).

Return Values None.

Remarks You would more commonly want to delete by name.

See Also

- `deleteChild(char)` on page 184
- `deleteChild(XmlTreeNode)` on page 186

deleteChild(XmlTreeNode)

Description Deletes the specified child from the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void deleteChild(
     const XmlTreeNode * child
);
```

Parameters

- `child` — A pointer to the child node. The node must be a child of the object on which the method will be called.

Return Values None.

Remarks You would more commonly want to delete by name.

See Also

- `deleteChild(char)` on page 184
- `deleteChild(int)` on page 185

deleteChildrenByElementType()

Description Deletes all the children with the given element type from the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void deleteChildrenByElementType(
     const char * elementType|
);
```

Parameters

- `elementType` — The element type of the children. The value can be either `ENFORCER_TAG_PROPERTY` or `ENFORCER_TAG_PROPERTYLIST`.

Return Values None.

Remarks This is a function that you will likely not need to use. It is typically an internal function.

See Also

- `deleteChild(char)` on page 184
- `deleteChild(int)` on page 185
- `deleteChild(XmlTreeNode)` on page 186
- `deleteChildrenByName)` on page 188

deleteChildrenByName)

Description Deletes all the children with the given `name` attribute from the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void deleteChildrenByName(
     const char * name|
);
```

Parameters

- `name` — The `NAME` attribute of the children to be deleted.

Return Values None.

See Also

- `deleteChild(char)` on page 184
- `deleteChild(int)` on page 185
- `deleteChild(XmlTreeNode)` on page 186
- `deleteChildrenByElementType()` on page 187

duplicateTreeNode()

Description	Duplicates the given tree node, including all children (if any), attributes and string values.
Class	`XmlTreeNode`
Include File	`XmlTreeNode.h`
Syntax	`XmlTreeNode * duplicateTreeNode();`
Parameters	None.
Return Values	A pointer to the new tree node.
Remarks	This function is typically used for debugging.

elementTypeIs()

Description Compares a node's element type to the given tag.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
bool elementTypeIs(
     const char * tag
) const ;
```

Parameters

- `tag` — The element type tag that will be used as a comparison. The value can be either `ENFORCER_TAG_PROPERTY` or `ENFORCER_TAG_PROPERTYLIST`.

Return Values

- `true` — The element type of the given node matches `tag`.
- `false` — The element type of the given node does not match `tag`.

Remarks If `tag` has a value of `NULL`, this method returns `false`.

This method is used to avoid having to rely on runtime type identification (RTTI).

getAttributeName()

Description Gets the name of the attribute at the given index in the current property or property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const char * const getAttributeName(
     const int attributeIndex
) const ;
```

Parameters

- `attributeIndex` — The index value of the attribute in the current property or property list. Must be a non-negative integer. The range for this value is 0 to (the *`number of attributes`*`-1`).

Return Values The name of the attribute.

See Also

- `getAttributeValue(char)` on page 192
- `getAttributeValue(int)` on page 193

getAttributeValue(char)

Description Gets the string value of the attribute with the given name.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const char * const getAttributeValue(
     const char * attributeName
) const ;
```

Parameters

- `attributeName` — The name of the attribute. This parameter is case-sensitive.

Return Values The string value of the given attribute, or `NULL`, if the no attribute named `attributeName` exists.

Example The following code sample gets the string value of an attribute:

```
if ( STREQ(expressionName, attributelogic_tags::LOGICAL_OPERATOR) ) {
   const char *operatorName =
    expression.getAttributeValue(attributelogic_tags::OPERATOR);
   const char **it = find_first_of(logicalOperatorNames,
    logicalOperatorNames + logicalOperatorSize, &operatorName,
    &operatorName + 1, streq);
   for (int i = 0; i < expression.getNumChildren(); ++i) {
      XmlTreeNode *child = expression.getChild(i);
      bool result = evaluateExpressionUsingLdap(*child, ldapConn,
         userSrc, userDn, membership);
      if ( result != logicalOperatorSigns[it - logicalOperatorNames]
   {
         return result;
      }
   }
   return true;
}
```

See Also

- `getAttributeName()` on page 191
- `getAttributeValue(int)` on page 193

getAttributeValue(int)

Description Gets the string value of the attribute at a given index in the current property or property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const char * const getAttributeValue(
     const int attributeIndex
) const ;
```

Parameters

- `attributeIndex` — The index value of the attribute in the current property or property list. Must be a non-negative integer. The range for this value is `0` to (the *`number of attributes`*`-1`).

Return Values The string value of the attribute.

See Also

- `getAttributeName()` on page 191
- `getAttributeValue(char)` on page 192

getChild(char)

Description Gets the child node with the given `NAME` attribute from the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
XmlTreeNode * getChild(
     const char * childName
) const ;
```

Parameters

- `childName` — The name of the child. This parameter is case-sensitive.

Return Values The child node, or `NULL`, if no child with a `NAME` attribute of `childName` exists.

Example The following code sample extracts the property list containing personalization information from the XML query document. It then cycles through each property contained within the list and exports its value as an environment variable to be used for personalization.

```
XmlTreeNode * info = reply->getChild(ENFORCER_P13NINFO);
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 (value == NULL) {
    value = &empty;
  }
  Setenv(name.get(), value, r);
}
```

See Also

- `getChild(int)` on page 195
- `getChildNoCase()` on page 198
- `getChildren()` on page 200

getChild(int)

Description Gets the child node found at the given index within the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
XmlTreeNode * getChild(
     const int childIndex
) const ;
```

Parameters

- `childIndex` — The position of the child in the current property list. Must be a non-negative integer. The range for this value is `0` to (the *`number of children`*`-1`).

Return Values The child node.

Example The following code sample extracts the property list containing personalization information from the XML query document. It then cycles through each property contained within the list and exports its value as an environment variable to be used for personalization.

```
XmlTreeNode * info = reply->getChild(ENFORCER_P13NINFO);
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 (value == NULL) {
      value = &empty;
   }
   Setenv(name.get(), value, r);
}
```

See Also

- `getChild(char)` on page 194
- `getChildNoCase()` on page 198
- `getChildren()` on page 200

getChildElement()

Description Gets the first element with the given element tag contained within the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
XmlTreeNode * getChildElement(
     const char * elementType
) const ;
```

Parameters

- `elementType` — The element type of the child. The value can be either `ENFORCER_TAG_PROPERTY` or `ENFORCER_TAG_PROPERTYLIST`.

Return Values The child node, or `NULL`, if no child of the given element type exists.

Remarks It is unlikely that you will need to call this function. It is typically used internally by Select Access.

getChildLongValue()

Description Translates the string value of the given child node of the current property list into a long integer.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
bool getChildLongValue(
     const char * childName,
     long * longPtr
) const ;
```

Parameters

- `childName` — The `NAME` attribute of the child node. This parameter is case-sensitive.
- `longPtr` — A pointer to the long integer.

Return Values

- `true` — The method was able to translate the value into a long integer.
- `false` — The method could not successfully translate the value.

Example The following code sample determines if the XML reply object had a registration cookie, and if so, adds it to the Enforcer object.

```
  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);
  }
```

See Also

- `getLongValue()` on page 205

getChildNoCase()

Description Gets the child node with the given name from the current property list. The name-matching performed is case-insensitive.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
XmlTreeNode * getChildNoCase(
     const char * childName
) const ;
```

Parameters

- `childName` — The name of the child. This parameter is *not* case-sensitive.

Return Values The child node, or `NULL`, if the child does not exist.

Remarks This function allows you to perform an exhaustive search.

See Also

- `getChild(char)` on page 194
- `getChild(int)` on page 195
- `getChildNoCaseStringValue()` on page 199
- `getChildren()` on page 200

getChildNoCaseStringValue()

Description Gets the string value of the child node with the given name in the current property list. The name-matching performed is case-insensitive.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const char * getChildNoCaseStringValue(
     const char * childName
) const ;
```

Parameters

- `childName` — The name of the child. This parameter is *not* case-sensitive.

Return Values The value of the specified node returned as a string, or `NULL`, if the child does not exist.

Remarks The value returned by this function must be freed using `EnforcerFree()`.

See Also

- `getChildNoCase()` on page 198
- `getChildStringValue()` on page 201
- `getElementStringValue()` on page 203

getChildren()

Description Gets all children of the current property list that have the given name. If `childName` is `NULL`, this function retrieves all children.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
XmlTreeNodeVector * getChildren(
     const char * childName
) const ;
```

Parameters

- `childName` — The `NAME` attribute of the children to be retrieved. This parameter may be `NULL`.

Return Values A list of the child elements of the current node.

Remarks This function allows you to perform an exhaustive search.

Example The following code example adds a cookie to the XML query object. Before doing so, it first looks for an old cookie by calling this function. If it does find an old cookie, it will be removed.

```
ENFORCER_API int
EnforcerQuerySetCookie(XMLnode query, const char *name, const char
*data, time_t lifetime)
{
  XmlTreeNodeVector cookies = query->getChildren(ENFORCER_COOKIE);
  for (size_t ic=0; ic<cookies.size(); ++ic) {
    const char * cookieName = cookies[ic]->getChildStringValue
        ENFORCER_COOKIE_NAME);
    if (STREQ(name, cookieName)) {
      query->removeChild(cookies[ic]);
}
```

getChildStringValue()

Description Gets the string value associated with the child node whose `NAME` attribute has the given value.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const char * getChildStringValue(
     const char * childName
) const ;
```

Parameters

- `childName` — The name of the child. This parameter is case-sensitive.

Return Values The value of the specified node returned as a string, or `NULL`, if the node has no string value.

Remarks If `childName` has a value of `NULL`, this method returns `NULL`.

Example The following code sample determines if the XML reply object had a registration cookie, and if so, adds it to the Enforcer object.

```
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);
}
```

See Also

- `getChildNoCaseStringValue()` on page 199
- `getElementStringValue()` on page 203

getComFacade()

Description Gets the pointer to a COM implementation of XML manipulation API.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
CEnforcerXMLTree * getCOMFacade();
```

Return Values A pointer to `CEnforcerXMLTree` — a COM implementation of XML manipulation API.

Remarks This function is not used by `XmlTreeNode` internally.

See Also

- `setComFacade()` on page 219

getElementStringValue()

Description Gets the string value of the child with the given element tag.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const char * getElementStringValue(
     const char * elementType
) const ;
```

Parameters

- `elementType` — The element tag to be matched. The value can be either `ENFORCER_TAG_PROPERTY` or `ENFORCER_TAG_PROPERTYLIST`.

Return Values The value of the specified node, returned as a string, or `NULL` if no child element with the given tag exists.

Remarks It is unlikely that you will need to call this function. It is more typically used internally by Select Access.

See Also

- `getChildNoCaseStringValue()` on page 199
- `getChildStringValue()` on page 201

getElementType()

Description Gets the element type of the current node.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const char * getElementType() const ;
```

Parameters None.

Return Values The element type of the current node: either `PROPERTY` or `PROPERTYLIST`.

getLongValue()

Description Translates the string value of the current node into a long integer.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
bool getLongValue(
     long * longPtr
) const ;
```

Parameters

- `longPtr` — A pointer to the long integer.

Return Values

- `true` — The value was translated into a long integer.
- `false` — The method could not successfully translate the value.

See Also

- `getChildLongValue()` on page 197

getName()

Description Gets the value associated with a node's NAME attribute. If no name has ever been specified for the node, an empty string is returned.

Class XmlTreeNode

Include File XmlTreeNode.h

Syntax

```
const char * getName() const ;
```

Parameters None.

Return Values The NAME attribute for the current node, or NULL, if the node has no NAME attribute.

Example The following code sample extracts the property list containing personalization information from the XML query document. It then cycles through each property contained within the list and exports its value as an environment variable to be used for personalization.

```
XmlTreeNode * info = reply->getChild(ENFORCER_P13NINFO);
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 (value == NULL) {
      value = &empty;
   }
   Setenv(name.get(), value, r);
}
```

getNumAttributes()

Description	Counts the number of attributes the given node has.
Class	`XmlTreeNode`
Include File	`XmlTreeNode.h`
Syntax	`const int getNumAttributes() const ;`
Parameters	None.

getNumChildren()

Description Counts the number of children the current node has.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const int getNumChildren() const ;
```

Parameters None.

Return Values The number of child nodes.

Remarks If the current node is a `<Property>` element, this method will always return `0`, since by definition a property cannot contain any children.

Example The following code sample extracts the property list containing personalization information from the XML query document. It then cycles through each property contained within the list and exports its value as an environment variable to be used for personalization.

```
XmlTreeNode * info = reply->getChild(ENFORCER_P13NINFO);
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 (value == NULL) {
      value = &empty;
   }
   Setenv(name.get(), value, r);
}
```

See Also

- `propertyElement::PropertyElement()` on page 229
- `propertyListElement::getNestedPropertyCount()` on page 246
- `propertyListElement::getPropertyValueCount()` on page 247

getStringValue()

Description Gets the current node's contained text. This value may be `NULL`.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const char * getStringValue() const ;
```

Parameters None.

Return Values The string value, or `NULL`, if no string value exists.

Remarks Property lists might not contain text.

Example The following code sample extracts the property list containing personalization information from the XML query document. It then cycles through each property contained within the list and exports its value as an environment variable to be used for personalization.

```
XmlTreeNode * info = reply->getChild(ENFORCER_P13NINFO);
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 (value == NULL) {
      value = &empty;
   }
   Setenv(name.get(), value, r);
}
```

See Also

- `getStringValueLen()` on page 210

getStringValueLen()

Description Gets the length of the current node's contained text. This value may be zero.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const int * getStringValueLen() const ;
```

Parameters None.

Return Values The length of the string value. If there is no string value, returns `0`.

Remarks Length does not include the space required for the trailing null byte.

See Also

- `getStringValue()` on page 209

isPropertyList()

Description Determines whether the current node is actually a list of other properties, as opposed to a name/value pair.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const bool isPropertyList() const ;
```

Parameters None.

Return Values

- `false` — The node is a property, which cannot contain other properties and property lists.

Remarks See `propertyListElement::isPropertyList()` for full details.

See Also

- `propertyListElement::isPropertyList()` on page 250

isPropertyValue()

Description See `propertyElement::isPropertyValue()` for full details.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const bool isPropertyValue() const ;
```

Parameters None.

Return Values Returns `false`.

See Also

- `propertyElement::isPropertyValue()` on page 233

mergeNode()

Description Merges the children of the given source node into an existing node based on the element type of the children, if the node supports it.

This method allows you to copy multiple children of the same element at one time.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void mergeNode(
     const XmlTreeNode * srcNode
);
```

Parameters

- `srcNode` — A pointer to the node containing the children that will be copied.

Return Values None.

Remarks It is unlikely that you will need to call this function. It is typically used internally by Select Access.

removeChild(char)

Description Removes the child with the given name from the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
XmlTreeNode * removeChild(
      const char * childName|
);
```

Parameters

- `childName` — The `NAME` attribute of the child node.

Return Values The removed child, if it exists, or `NULL`, if it does not.

Example The following code sample tests to see whether the given cookie contains a nonce. If so, and if it has not expired, the cookie is added to the XML query.

```
void CNvxIISPluginApp::AddNonce(LPSTR pszNonce, XmlTreeNode** pXMLQuery)
{
  XmlTreeNode * oldnonce = (*pXMLQuery)->removeChild(ENFORCER_NONCE);
  if (oldnonce) {
    delete oldnonce;
}
  (*pXMLQuery)->appendChildString(ENFORCER_NONCE,
      SYS_STRDUP(pszNonce));
}
```

See Also

- `removeChild(int)` on page 215
- `removeChild(XmlTreeNode)` on page 216

removeChild(int)

Description Removes the child found at the given index from the current property list.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
XmlTreeNode * removeChild(
     const int  childIndex
);
```

Parameters

- `childIndex` — The position in a property list of the child node. Must be a non-negative integer. The range for this value is `0` to (the *`number of children`*`-1`).

Return Values The removed child.

See Also

- `removeChild(char)` on page 214
- `removeChild(XmlTreeNode)` on page 216

removeChild(XmlTreeNode)

Description Removes the specified child from a `PropertyListElement`.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
XmlTreeNode * removeChild(
     const XmlTreeNode *  child
);
```

Parameters

- `child` — A pointer to the child node. The node must be a child of the object.

Return Values The removed child, if the specified child node exists, or `NULL`, if it does not.

Example The following code sample tests to see whether the given URL is a multi-domain single sign-on (MD-SSO) authentication URL containing a nonce. If so, and if it has not expired, the given nonce is added to the XML query.

```
if (EnforcerIsSSOAuthURL(http_query, &nonce)) {
  XmlTreeNode * oldnonce = query->removeChild(ENFORCER_NONCE);
  if (oldnonce) {
    delete oldnonce;
  }
  query->appendChildString(ENFORCER_NONCE, SYS_STRDUP(nonce));
  EnforcerLog(ENFORCER_LOG_DEBUG, "got SSO nonce from %s",
    http_query);
}
```

See Also

- `removeChild(char)` on page 214
- `removeChild(int)` on page 215

setAttributeValue()

Description Sets the given attribute to the given value.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void setAttributeValue(
     const char * attributeName,
     const char * attributeValue
);
```

Parameters

- `attributeName` — The name of the attribute whose value will be changed.
- `attributeValue` — The new or modified value of `attributeName`.

Return Values None.

Remarks It is unlikely that you would need to call this function.

setChildStringValue()

Description Sets the string value of the child of the given node with the given name. If no such property exists, one is created.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
XmlTreeNode * setChildStringValue(
     const char * childName
     const char * value
);
```

Parameters

- `childName` — The value of the `NAME` attribute of the child node.
- `value` — The new or modified value of the child property.

Return Values

- If successful, the child that was created or modified is returned.
- If there was an error, `NULL` is returned.

Remarks Memory considerations:

- The tree copies the `childName` string if necessary.
- The tree takes ownership of `value`, and will `free()` it.
- Use `strdup()` on a constant string; the old value is `free()`d.

See Also

- `setElementStringValue()` on page 220

setComFacade()

Description Sets a pointer to `CEnforcerXMLTree`, a COM implementation of XML manipulation API.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void setCOMFacade(CEnforcerXMLTree * enforcerXmlTree);
```

Parameters

- `enforcerXmlTree` — A pointer to `CEnforcerXMLTree`.

Return Values Void.

Remarks This function is not used by `XmlTreeNode` internally.

See Also

- `getComFacade()` on page 202

setElementStringValue()

Description Sets the value of the child with the given element tag. If no child with the given element tag exists, one is created.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
XmlTreeNode * setElementStringValue(
     const char * elementType
     const char * stringValue
     bool replace = true
);
```

Parameters

- `elementType` — A string containing the element type of the given node. The value can be either `ENFORCER_TAG_PROPERTY` or `ENFORCER_TAG_PROPERTYLIST`.
- `stringValue` — A string containing the new or modified value of the child node with the `elementType` tag.
- `replace` — A boolean value which indicates whether or not `stringValue` will replace a previous value.

Return Values

- If successful, the child that was created or modified is returned.
- If there was an error, `NULL` is returned.

Remarks It is unlikely that you would need to call this function.

See Also

- `setChildStringValue()` on page 218

setElementType()

Description Sets a node's element type.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void setElementType(
     const char * newElementType
);
```

Parameters

- `newElementType` — A string containing the new element type of the given node. The value can be either `ENFORCER_TAG_PROPERTY` or `ENFORCER_TAG_PROPERTYLIST`.

Return Values None.

Remarks The `newElementType` argument is duplicated.

It is unlikely that you would need to call this function.

setName()

Description Sets the `NAME` attribute for a node. If no attribute is present, this function creates one.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void setName(
     const char * newName
);
```

Parameters

- `newName` — The new value of the `NAME` attribute of the node.

Return Values None.

Remarks The `newName` argument is duplicated and the `NAME` attribute is set in a condition.

It is unlikely that you would need to call this function.

setStringValue(char)

Description Sets or changes the given node's contained text provided. `NULL` or `0` clears the string value.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void setStringValue(
     const char * newStringValue,
     );
```

Parameters

- `newStringValue` — The text that will be copied and appended to an existing string.

Return Values Void.

Remarks This method usse `strlen()` to caculate the length of `newStringValue`, and calls `setStringValue(char, int)` to set the string value.

See Also

- `setChildStringValue()` on page 218
- `setStringValue(char, int)` on page 224

setStringValue(char, int)

Description Sets or changes the given node's contained text provided. `NULL` or `0` clears the string value.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
void setStringValue(
     const char * newStringValue,
     const int newStringValueLen
);
```

Parameters

- `newStringValue` — The text that will be copied and appended to an existing string.
- `newStringValueLen` — The length of the text to be appended. Must be a non-negative integer.

Return Values None.

Remarks The tree takes ownership of the new value.

See Also

- `setChildStringValue()` on page 218
- `setStringValue(char)` on page 223

sNew()

Description Creates a new node of the type specified by the first tag.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
static XmlTreeNode * sNew(
     const char * elementType,
     const char * name
);
```

Parameters

- `elementType` — A string containing the element type of the new node. The value can be either `ENFORCER_TAG_PROPERTY` or `ENFORCER_TAG_PROPERTYLIST`.
- `name` — A string containing the name of the new node.

Return Values The newly-created node.

Remarks This method calls the `propertyListElement::PropertyElement()` and `propertyListElement::PropertyListElement(char)` constructors. The `elementType` argument is duplicated by the constructor.

Example The following code sample retrieves the query level setting, then test to see if the level has been set to `MAXIMAL`. If so, it creates a new XML property list, adds it to the query, and populates it with header list data, then appends a property specifying the version of the Enforcer plugin.

```
int query_level = EnforcerGetQueryLevel(Enforcer_Handle);

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));
}
```

See Also

- `propertyListElement::PropertyElement()` on page 229
- `propertyListElement::PropertyListElement(char)` on page 236

sToString(XmlTreeNode, bool)

Description Converts the value of the given node into its equivalent string representation.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
static char * sToString(
     const XmlTreeNode * node,
     const bool readable = 0
);
```

Parameters

- `node` — A pointer to the node.
- `readable` — A boolean value which indicates whether or not the resultant string is indented and broken across lines. The value can be either `0` (not readable) or `1` (readable).

Return Values The string representation of the current object.

Remarks The resultant string is readable only if specified. This function is normally used for logging and debugging purposes.

toString()

Description Converts the value of the given node into a printable string.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
static char * ToString;
```

Return Values The string representation of the current object.

Remarks The resultant string is readable only if specified. This function is normally used for logging and debugging purposes.

PropertyElement

This class extends `XmlTreeNode`. A `PropertyElement` is an XML element defined by Select Access that contains a name/value pair. The `PropertyElement` tag requires one attribute, `name`, that identifies the name of the name/value pair. The value stored is the data between the element's opening and closing tags.

Class Summary

Table 24 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 24 PropertyElement Class Summary

Method Name	Description	Page
Construction and Destruction		
`PropertyElement()`	Constructor. Creates a new empty property element.	229
`PropertyElement(char, char)`	Constructor. Creates a new property element with the given name and value.	230
`PropertyElement(char, char, int)`	Constructor. Creates a new property element with the given name, value, and value length.	231
`~PropertyElement()`	Destructor. Releases property elements and anything they contain.	NA
Member Methods		
`getNumChildren()`	Counts the number of children the current node has.	232
`isPropertyValue()`	Determines whether this node is actually a name/value pair, as opposed to a list of other properties.	233

Unimplemented methods

The following methods intended for the `XMLTreeNode` and are not implemented for the `PropertyElement` class.

- `appendChild()`
- `deleteChild()`
- `getChild()`
- `getChildStringValue()`
- `removeChild()`
- `setChildStringValue()`

PropertyElement()

Description Constructor. Creates a new empty property element.

Class `PropertyElement`

Include File `PropertyElement.h`

Syntax

```
PropertyElement();
```

Parameters None.

Return Values None.

Remarks Typically, the `::sNew()` method would be used to construct a new empty property element.

Example The following code sample determines whether the new node should be a property element and if so, calls the constructor method to create it.

```
XmlTreeNode * newNode = NULL;
if (STREQ(elementType, ENFORCER_TAG_PROPERTY))
{
   newNode = new PropertyElement();
}
```

See Also

- `PropertyElement(char, char)` on page 230
- `PropertyElement(char, char, int)` on page 231
- `XmlTreeNode::sNew()` on page 225

PropertyElement(char, char)

Description Constructor. Creates a new property element with the given name and value.

Class `PropertyElement`

Include File `PropertyElement.h`

Syntax

```
PropertyElement(
     const char * name,
     const char * value
);
```

Parameters

- `*name` — The `NAME` attribute of the property to be created.
- `*value` — The value of the property to be created.

Return Values None.

Remarks Typically, the `XmlTreeNode::sNew()` method would be used to construct a new empty property element.

See Also

- `PropertyElement()` on page 229
- `PropertyElement(char, char, int)` on page 231
- `XmlTreeNode::sNew()` on page 225

PropertyElement(char, char, int)

Description Constructor. Creates a new property element with the given name, value, and value length.

Class `PropertyElement`

Include File `PropertyElement.h`

Syntax

```
PropertyElement(
     const char * name,
     const char * value,
     const int valueLen
);
```

Parameters

- `*name` — The `NAME` attribute of the property to be created.
- `*value` — The value of the property to be created.
- `valueLen` — The length of the `*value` parameter. Must be a non-negative integer.

Return Values None.

Remarks Typically, the `XmlTreeNode::sNew()` method would be used to construct a new empty property element.

This function is only used by a parser.

See Also

- `PropertyElement()` on page 229
- `PropertyElement(char, char)` on page 230
- `XmlTreeNode::sNew()` on page 225

getNumChildren()

Description Counts the number of children the current node has.

Class `XmlTreeNode`

Include File `XmlTreeNode.h`

Syntax

```
const int getNumChildren() const ;
```

Parameters None.

Return Values

- Always returns `0` — Property elements can contain no children.

See Also

- `XmlTreeNode::getNumChildren()` on page 208
- `PropertyListElement::getNestedPropertyCount()` on page 246
- `PropertyListElement::getPropertyValueCount()` on page 247

isPropertyValue()

Description Determines whether this node is actually a name/value pair, as opposed to a list of other properties.

Class `PropertyElement`

Include File `PropertyElement.h`

Syntax

```
const bool isPropertyValue() const ;
```

Parameters None.

Return Values

- `true` — The node is a property, which may possess a value.

See Also

- `PropertyListElement::isPropertyList()` on page 250

PropertyListElement

The `PropertyListElement` class extends `XMLTreeNode`. It allows you to create and destroy property lists, and parse and manipulate their children.

Class Summary

Table 25 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 25 PropertyListElement Class Summary

Method Name	Description	Page
Construction and destruction		
`PropertyListElement(char)`	Constructor. Creates a new empty property list element.	236
`PropertyListElement(char, char)`	Constructor. Creates a new property element with the given name and value.	237
`~PropertyListElement`	Destructor. Releases property list elements and anything they contain.	NA
Member methods		
`appendNestedProperty(char)`	Creates a new `<PropertyList>` element with the given name and makes it a child of the current property list.	238
`appendNestedProperty(PropertyList Element)`	Takes a previously created `<PropertyList>` element and makes it a child of the current property list.	239
`appendPropertyValue(char, char)`	Creates a new `<Property>` element with the given name and makes it a child of the current property list.	240
`appendPropertyValue(Property Element)`	Takes a previously created `<Property>` element and makes it a child of the current property list.	241
`appendStringValue(char)`	Appends a string to an existing string value. The given text is copied.	242
`appendStringValue(char, int)`	Appends a string of the given length to an existing string value. The given text is copied.	243
`getNestedProperties(const char)`	Gets the property list with the given `NAME` attribute contained within the current property list.	244

Table 25 PropertyListElement Class Summary (cont'd)

Method Name	Description	Page
`getNestedPropertiesIgnoreCase (const char)`	Gets the property list with the given case-insensitive `NAME` attribute contained within the current property list.	245
`getNestedPropertyCount()`	Counts the number of nested property lists the current property list contains.	246
`getPropertyValueCount()`	Counts the number of properties the current property list contains.	247
`getPropertyValues(const char)`	Gets the property with the given `NAME` attribute contained in the current property list.	248
`getPropertyValuesIgnoreCase (const char)`	Gets the property list with the given case-insensitive `NAME` attribute contained within the current property list.	249
`isPropertyList()`	Determines whether this node is actually a list of other properties, as opposed to a name/value pair.	250

PropertyListElement(char)

Description Constructor. Creates a new empty property list element.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
PropertyListElement(
     const char * tag
);
```

Parameters

- `*tag` — A string representing the element type of the node. This parameter must have a value of `ENFORCER_TAG_PROPERTYLIST`.

Return Values None.

Remarks Typically, the `::sNew()` method would be used to construct a new empty property list element.

See Also

- `PropertyListElement(char, char)` on page 237
- `PropertyListElement::sNew()` on page 225

PropertyListElement(char, char)

Description Constructor. Creates a named empty `PropertyListElement`.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
PropertyListElement(
     const char * tag,
     const char * name
);
```

Parameters

- `*tag` — A string containing the element type of the node. This parameter must have a value of `ENFORCER_TAG_PROPERTYLIST`.
- `*name` — A string containing the value of the `NAME` attribute of the new property list.

Return Values None.

Remarks Typically, the `::sNew()` method would be used to construct a new empty property list element.

See Also

- `PropertyListElement(char)` on page 236
- `PropertyListElement::sNew()` on page 225

appendNestedProperty(char)

Description Creates a new `<PropertyList>` element with the given name and makes it a child of the current property list.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
PropertyListElement * appendNestedProperty(
     const char * name
);
```

Parameters

- `*name` — The `NAME` attribute of the property list to be created. If `NULL`, an unnamed property list is added.

Return Values The newly-created `<PropertyList>` element, or `NULL`, if unsuccessful.

See Also

- `appendNestedProperty(PropertyListElement)` on page 239

appendNestedProperty(PropertyListElement)

Description Takes a previously created `<PropertyList>` element and makes it a child of the current property list.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
PropertyListElement * appendNestedProperty(
     PropertyListElement * list
);
```

Parameters

- `*list` — A pointer to a previously created property list.

Return Values The newly appended `<PropertyList>` element, or `NULL`, if unsuccessful.

See Also

- `appendNestedProperty(char)` on page 238

appendPropertyValue(char, char)

Description Creates a new <Property> element with the given name and makes it a child of the current property list.

Class PropertyListElement

Include File PropertyListElement.h

Syntax

```
PropertyElement * appendPropertyValue(
     const char * name,
     const char * value
);
```

Parameters

- *name — The NAME attribute of the property to be created.
- *value — The value of the property to be created.

Return Values The newly-created <Property> element, or NULL, if unsuccessful.

Example

```
if (! strEmpty(challenge)) {
  PropertyListElement * hint = addAuthHint2Response(*response,
           challenge_formName_.get());
  if ( hint ) {
    hint->appendPropertyValue(ENFORCER_NATIVE_CHAL,
           SYS_STRDUP(TRUE_TEXT));
    hint->appendPropertyValue(ENFORCER_FORM_CHALLENGE,
           SYS_STRDUP(challenge));
    hint->appendPropertyValue(ENFORCER_USER,
           SYS_STRDUP(user.c_str()));
    hint->appendPropertyValue(ENFORCER_FORM_RADIUS_STATE,
           SYS_STRDUP(state));
  }
}
```

See Also

- appendPropertyValue(PropertyElement) on page 241

appendPropertyValue(PropertyElement)

Description Takes a previously created `<Property>` element and makes it a child of the current property list.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
PropertyElement * appendPropertyValue(
     PropertyElement * element
);
```

Parameters

- `*element` — An XML pointer to a previously created property.

Return Values The newly appended `<PropertyList>` element, or `NULL`, if unsuccessful.

See Also

- `appendPropertyValue(char, char)` on page 240

appendStringValue(char)

Description Appends a string to an existing string value. The given text is copied.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
void appendStringValue(
     const char * newStringValue,
);
```

Parameters

- `*newStringValue` — A string containing the text that will be copied and appended to an existing string.

Return Values None.

Remarks If no string currently exists, this function creates one.

See Also

- `appendStringValue(char, int)` on page 243

appendStringValue(char, int)

Description Appends a string of the given length to an existing string value. The given text is copied.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
void appendStringValue(
     const char * newStringValue,
     const int newStringValueLen
);
```

Parameters

- `*newStringValue` — The text that will be copied and appended to an existing string.
- `newStringValueLen` — The length of the text to be appended. Must be a non-negative integer.

Return Values None.

Remarks If no string currently exists, this method creates one.

This function is normally called when parsing XML. Typically you would use `appendStringValue(char)`.

See Also

- `appendStringValue(char)` on page 242

getNestedProperties(const char)

Description Gets the property list with the given `NAME` attribute contained within the current property list.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
PropertyListElementVector getNestedProperties(
     const char * name
) const ;
```

Parameters

- `*name` — The `NAME` attribute of the property list being searched for. This parameter is case-sensitive.

Return Values The property list with a `NAME` attribute of `name`, if it exists.

Remarks If `name` is `NULL`, this method retrieves all property lists.

See Also

- `getNestedPropertiesIgnoreCase(const char)` on page 245

getNestedPropertiesIgnoreCase(const char)

Description Gets the property list with the given case-insensitive `NAME` attribute contained within the current property list.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
PropertyListElementVector getNestedPropertiesIgnoreCase(
     const char * name
) const ;
```

Parameters

- `*name` — A string containing the value of the `NAME` attribute of the property list that this method will try to match.

Return Values The property list with a `NAME` attribute of `name`, if it exists.

Remarks If `name` is `NULL`, this method retrieves all nameless property lists.

See Also

- `getNestedProperties(const char)` on page 244

getNestedPropertyCount()

Description Counts the number of nested property lists the current property list contains.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
virtual const int getNestedPropertyCount() const ;
```

Parameters None.

Return Values The number of property lists nested within the current property list, or `NULL`, if unsuccessful.

See Also

- `getNestedProperties(const char)` on page 244
- `XmlTreeNode::getNumChildren()` on page 208
- `PropertyElement::PropertyElement()` on page 229

getPropertyValueCount()

Description Counts the number of properties the current property list contains.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
const int getPropertyValueCount() const ;
```

Parameters None.

Return Values The number of properties, or `NULL`, if unsuccessful.

See Also

- `getNestedPropertyCount()` on page 246
- `XmlTreeNode::getNumChildren()` on page 208
- `PropertyElement::PropertyElement()` on page 229

getPropertyValues(const char)

Description Gets the property with the given `NAME` attribute contained in the current property list.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
PropertyElementVector getPropertyValues(
     const char * name
) const ;
```

Parameters

- `*name` — The `NAME` attribute of the property that this method will try to match. This parameter is case-sensitive.

Return Values A vector of properties with matching names.

Remarks If `name` is `NULL`, this method retrieves all properties.

Example The following code sample returns properties:

```
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");
}
```

See Also

- `getPropertyValuesIgnoreCase(const char)` on page 249

getPropertyValuesIgnoreCase(const char)

Description Gets the property list with the given case-insensitive `NAME` attribute contained within the current property list.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
PropertyElementVector getPropertyValuesIgnoreCase(
     const char * name
) const ;
```

Parameters

- `*name` — The `NAME` attribute value of the property that this method will try to match.

Return Values A vector of properties with matching names.

Remarks If `name` is `NULL`, this method retrieves all properties.

See Also

- `getPropertyValues(const char)` on page 248

isPropertyList()

Description Determines whether this node is actually a list of other properties, as opposed to a name/value pair.

Class `PropertyListElement`

Include File `PropertyListElement.h`

Syntax

```
const bool isPropertyList() const ;
```

Parameters None.

Return Values

- `true` — The node is a property list, which may contain properties and/or nested property lists.

Example The following code sample determines whether the given object is a property list.

```
if ((auth_plist = reply->getChild(ENFORCER_FORM_DATA)) &&
   auth_plist->isPropertyList()) {
   response = DONE;
   if (!magic_query(http_query)) {
      response = send_magic_redirect(r, http_query);
   }
   else {
      send_dynamic_form(r, NULL, auth_plist);
   }
}
```

See Also

- `PropertyElement::isPropertyValue()` on page 233

7 The Logger API

The Logger API is a robust logging framework used by both Enforcer plugins and Policy Validator plugins. It allows you to filter log messages and forward them to multiple log destinations. This chapter provides the reference to the functions in this API.

Classes in this API

The Logger API is composed of the classes listed in Table 26:

Table 26 Class Summary

Method	Description	Page
`Logger`	The `Logger` is a singleton class that allows you to filter and forward messages to multiple log destinations. The `Logger` class allows you to register a log destination factory. If no destinations have been registered, messages are logged to the system log (syslog in UNIX; Event Log in Windows).	252
`LogFilter`	The `LogFilter` class is a singleton class that holds a list of channel/level pairs and a destination.	262
`LogDestination`	The `LogDestination` class is an abstract class that defines the destination for a log message.	267
`LogServer`	The `LogServer` class logs messages to a Select Access SOAP audit server.	269
`LogStdio`	The `LogStdio` is an abstract class that logs messages to a C standard I/O file pointer. Used as the parent place holder for `LogFile` and `LogStderr`.	272
`LogFile`	The `LogFile` class logs messages to the given file, rolling over when the file exceeds the given length. Extends the `LogStdio` class.	269
`LogStderr`	The `LogStderr` class logs messages to the standard error output. Extends the `LogStdio` class.	278
`LogSystem`	The `LogSystem` class logs messages to the system log, either syslog on UNIX, or the Event Log on Windows.	281

Logger

The `Logger` is a singleton class that allows you to filter and forward messages to multiple log destinations. The `Logger` class allows you to register a log destination factory. If no destinations have been registered, messages are logged to the system log (syslog in UNIX; Event Log in Windows).

Class Summary

Table 27 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 27 Class Overview

Method Name	Description	Page
`addFilter()`	Adds the given filter to the existing `Logger` configuration.	253
`configure()`	Configures the `Logger` object based on the given XML input.	254
`fallback()`	Causes the application to fall back to the system log because something has broken.	255
`init()`	Initializes a `Logger` by clearing any existing configuration.	256
`log()`	Logs a message through all possible filters.	257
`reconfigure()`	Clears the existing filters then configures the Logger object, based on the XML input.	258
`registerDest()`	Registers a log destination factory. This is an extension point for the logger system. You can create a class that defines your own logging destination.	259
`vlog()`	Logs a message to the log destination. This method is a `vprintf` version of `log()`. It includes the extended interface.	260
`xlog()`	Logs a message to the log destination. This method is an extended interface version of `log()`.	261

addFilter()

Description Adds the given filter to the existing `Logger` configuration.

Class `Logger`

Include File `Logger.h`

Syntax

```
void addFilter(
     LogFilter * newFilter
);
```

Parameters

- `newFilter` — A pointer to the filter object.

Return Values None.

Remarks Once the filter has been appended, the `Logger` takes ownership of the filter object. It will delete it when destroyed or reconfigured.

Example The following code sample initializes the empty logger then sets the logging destination.

```
log_dest_flags = destcode;
Logger::init(progName);
int logLevel = (log_dbg_msgs ? ENFORCER_LOG_DEBUG :
     ENFORCER_LOG_INFO);
if ((destcode == 0) || (destcode & ENFORCER_LOGTO_STDERR)) {
   LogDestination *ld = LogStderr::FactoryFunc(progName, NULL);
   LogFilter *lf = new LogFilter();
   lf->setDestination(ld);
   lf->addChannel(NULL, logLevel);
   Logger::addFilter(lf);
}
```

See Also `LogFilter()` on page 263

configure()

Description Configures the `Logger` object based on the given XML input.

Class `Logger`

Include File `Logger.h`

Syntax

```
bool configure(
     const XmlTreeNode * conf
);
```

Parameters

- `conf` — A pointer to an XML node containing file logger configuration information. That is, one of `systemLog` or `file` or `standardErr` or `logServer`. File logger information expects the XML node to be in the form:

  ```
  <file name="/file/name" maxSize="1000000"/>
  ```

  where:

  - `file name` is the name of the file to which messages will be logged.
  - `maxSize` is the maximum size the file can reach before it rolls over. If this attribute is missing or is not greater than zero, the log is not rolled over.

  If `conf` is `NULL`, this function configures an empty `Logger`.

Return Values

- `true` — The `Logger` object was configured successfully.
- `false` — The `Logger` object could not be configured.

Remarks

- Creates all the output filters based on the XML tree.
- If no factory function exists for a filter destination or if the XML is invalid this method sends messages to the built-in system destination (i.e. UNIX syslog or Event Log).
- Attempts to configure as many destinations as possible.

See Also

- `reconfigure()`

fallback()

Description Causes the application to fall back to the system log because something has broken.

Class

```
Logger
```

Include File

```
Logger.h
```

Syntax

```
void fallback(
     const char *     channel,
     int              level,
     const char *     details,
     const char *     toAppend,
     const char *     message,

     ...
     .
     .
     .
);
```

Parameters

- `channel` — 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.
- `level` — The minimum severity level of events to be logged. All events of the given severity level and higher are logged. This parameter can have one of the following values:
  - `ENFORCER_LOG_FATAL` — Fatal errors.
  - `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - `ENFORCER_LOG_WARNING` — Errors worth noting.
  - `ENFORCER_LOG_INFO` — Information only.
  - `ENFORCER_LOG_DEBUG` — Debugging information.
- `details` — The descriptive details of the event/message being logged.
- `toAppend` — Additional data to be appended to the end of the message.
- `message` — A summary of the event/message being logged. This is a printf format sting and is used to display additional parameters.

Return Values None.

Remarks You will typically not use this function. It is typically used as an internal function for Select Access.

init()

Description Initializes a `Logger` by clearing any existing configuration.

Class `Logger`

Include File `Logger.h`

Syntax

```
void init(
     const char * application
);
```

Parameters

- `application` — The application that is sending the message to the log.

Return Values None.

Remarks This method clears all existing `Logger` configuration and registers the built-in logging destinations: one of `standardFile`, `systemLog`, `file`, `standardErr`, and `logServer`.

The normal usage pattern is:

- `init()`
- `registerDest()` — for any non-built-in destinations.
- `configure()`

Example The following code sample initializes the empty logger then sets the logging destination.

```
log_dest_flags = destcode;
Logger::init(progName);
int logLevel = (log_dbg_msgs ? ENFORCER_LOG_DEBUG :
     ENFORCER_LOG_INFO);
if ((destcode == 0) || (destcode & ENFORCER_LOGTO_STDERR)) {
   LogDestination *ld = LogStderr::FactoryFunc(progName, NULL);
   LogFilter *lf = new LogFilter();
   lf->setDestination(ld);
   lf->addChannel(NULL, logLevel);
   Logger::addFilter(lf);
}
```

See Also

- `configure()` on page 254
- `registerDest()` on page 259

log()

Description Logs a message through all possible filters.

Class `Logger`

Include File `Logger.h`

Syntax

```
void log(
     const char *     channel,
     int              level,
     const char *     message
);
```

Parameters

- `channel` — 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.
- `level` — The minimum severity level of events to be logged. All events of the given severity level and higher are logged. This parameter can have one of the following values:
  - `ENFORCER_LOG_FATAL` — Fatal errors.
  - `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - `ENFORCER_LOG_WARNING` — Errors worth noting.
  - `ENFORCER_LOG_INFO` — Information only.
  - `ENFORCER_LOG_DEBUG` — Debugging information.
- `message` — A brief message that describes the event.

Return Values None.

Remarks If this method is used before any filters have been initialized, the default system log is used to record messages with a severity level of `WARNING` or higher.

Example The following code sample logs an information-level message.

```
Logger::log(VAL_CHAN_OP, ENFORCER_LOG_INFO, "Invoking attribute/logic plugin");
```

See Also

- `vlog()`
- `xlog()`

reconfigure()

Description Clears the existing filters then configures the Logger object, based on the XML input.

Class `Logger`

Include File `Logger.h`

Syntax

```
bool reconfigure(
     const XmlTreeNode * config
);
```

Parameters

- `config` — A pointer to an XML node containing file logger configuration information. That is, one of `systemLog` or `file` or `standardErr` or `logServer`. File logger information expects the XML node to be in the form:

  `<file name="/file/name" maxSize="1000000"/>`

  where:

  — `file name` is the name of the file to which messages will be logged.

  — `maxSize` is the maximum size the file can reach before it rolls over. If this attribute is missing or is not greater than zero, the log is not rolled over.

  If `config` is `NULL`, this function configures an empty `Logger`.

Return Values

- `true` — The `Logger` object was reconfigured successfully.
- `false` — The `Logger` object could not be reconfigured.

Remarks This method performs the same action as calling `clear()` then `configure()`.

See Also

- `configure()`

registerDest()

Description Registers a log destination factory. This is an extension point for the logger system. You can create a class that defines your own logging destination.

Class `Logger`

Include File `Logger.h`

Syntax

```
void registerDest(
     const char *     name,
     destFactoryFunc factory
);
```

Parameters

- `name` — The name under which the log destination will be registered.
- `factory` — A pointer to the factory function used to create the log destination.

Return Values None.

See Also `setDestination()` on page 266

vlog()

Description Logs a message to the log destination. This method is a `vprintf` version of `log()`. It includes the extended interface.

Class `Logger`

Include File `Logger.h`

Syntax

```
void vlog(
     const char *     channel,
     int              level,
     const char *     admin,
     const char *     service,
     const char *     resource,
     const char *     user,
     const char *     details,
     const char *     message,
     va_list          ap
);
```

Parameters

- `channel` — 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.
- `level` — The minimum severity level of events to be logged. All events of the given severity level and higher are logged. This parameter can have one of the following values:
  - `ENFORCER_LOG_FATAL` — Fatal errors.
  - `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - `ENFORCER_LOG_WARNING` — Errors worth noting.
  - `ENFORCER_LOG_INFO` — Information only.
  - `ENFORCER_LOG_DEBUG` — Debugging information.
- `admin` — The name of the administrator.
- `service` — The name of the service.
- `resource` — The name of the resource.
- `user` — The name of the identity.
- `details` — The descriptive details of the event/message being logged.
- `message` — The short message that will be logged.
- `ap` — A pointer to an array variable that contains additional arguments to be logged.

Return Values None.

Remarks This method is called by the `EnforcerLog()` function.

See Also

- `log()` on page 257
- `xlog()` on page 261
- `EnforcerLog()` on page 47
- Logging Levels on page 31

xlog()

Description Logs a message to the log destination. This method is an extended interface version of `log()`.

Class `Logger`

Include File `Logger.h`

Syntax

```
void xlog(
     const char *     channel,
     int              level,
     const char *     admin,
     const char *     service,
     const char *     resource,
     const char *     user,
     const char *     details,
     const char *     message
);
```

Parameters

- `channel` — The type of audit event, such as `Logger.ADMIN_CHANNEL` or `Logger.CONFIG_CHANNEL`.
- `level` — The minimum severity level of events to be logged. All events of the given severity level and higher are logged. This parameter can have one of the following values:
  - `ENFORCER_LOG_FATAL` — Fatal errors.
  - `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - `ENFORCER_LOG_WARNING` — Errors worth noting.
  - `ENFORCER_LOG_INFO` — Information only.
  - `ENFORCER_LOG_DEBUG` — Debugging information.
- `admin` — The name of the administrator.
- `service` — The name of the service.
- `resource` — The name of the resource.
- `user` — The name of the identity.
- `details` — The descriptive details of the event/message being logged.
- `message` — The short version of the message that will be logged.

Return Values None.

See Also

- `log()` on page 257
- `vlog()` on page 260
- Logging Levels on page 31

LogFilter

The `LogFilter` class is a singleton class that holds a list of channel/level pairs and a destination.

Class Summary

Table 28 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 28 Class Overview

Method Name	Description	Page
Construction and Destruction		
`LogFilter()`	Constructs the `LogFilter` object.	263
`~LogFilter`	Destructor. Releases the LogFilter object.	NA
Member Methods		
`addChannel()`	Adds a channel/level pair to the list of log filters.	264
`log()`	Logs a message if it matches one of the channel/level pairs held by the `LogFilter` object.	265
`setDestination()`	Sets the log destination.	266

LogFilter()

Description Constructs the `LogFilter` object.

Class `LogFilter`

Include File `LogFilter.h`

Syntax `LogFilter()`

Parameters None.

Return Values None.

Example The following code sample initializes the empty logger then sets the logging destination.

```
log_dest_flags = destcode;
Logger::init(progName);
int logLevel = (log_dbg_msgs ? ENFORCER_LOG_DEBUG :
     ENFORCER_LOG_INFO);
if ((destcode == 0) || (destcode & ENFORCER_LOGTO_STDERR)) {
   LogDestination *ld = LogStderr::FactoryFunc(progName, NULL);
   LogFilter *lf = new LogFilter();
   lf->setDestination(ld);
   lf->addChannel(NULL, logLevel);
   Logger::addFilter(lf);
}
```

addChannel()

Description Adds a channel/level pair to the list of log filters.

Class `LogFilter`

Include File `LogFilter.h`

Syntax

```
void addChannel(
     const char *     channel,
     int              level
);
```

Parameters

- `channel` — The type of audit event that you want to filter on, such as `Logger.ADMIN_CHANNEL`, or `Logger.CONFIG_CHANNEL`. If `channel` is `NULL`, this method matches all channels.
- `level` — The minimum severity level of events to be logged. All events of the given severity level and higher are logged. This parameter can have one of the following values:
  - `ENFORCER_LOG_FATAL` — Fatal errors.
  - `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - `ENFORCER_LOG_WARNING` — Errors worth noting.
  - `ENFORCER_LOG_INFO` — Information only.
  - `ENFORCER_LOG_DEBUG` — Debugging information.

Return Values None.

Remarks The `LogFilter` object copies the channel name and frees it on destruction.

Example The following code sample initializes the empty logger then sets the logging destination.

```
log_dest_flags = destcode;
Logger::init(progName);
int logLevel = (log_dbg_msgs ? ENFORCER_LOG_DEBUG :
     ENFORCER_LOG_INFO);
if ((destcode == 0) || (destcode & ENFORCER_LOGTO_STDERR)) {
  LogDestination *ld = LogStderr::FactoryFunc(progName, NULL);
  LogFilter *lf = new LogFilter();
  lf->setDestination(ld);
  lf->addChannel(NULL, logLevel);
  Logger::addFilter(lf);
}
```

log()

Description Logs a message if it matches one of the channel/level pairs held by the `LogFilter` object.

Class `LogFilter`

Include File `LogFile.h`

Syntax

```
void log(
     const char *     channel,
     int              level,
     const char *     details,
     const char *     toAppend,
     const char *     message,
     va_list          ap
);
```

Parameters

- `channel` — The type of audit event that you want to filter on, such as `Logger.ADMIN_CHANNEL`, or `Logger.CONFIG_CHANNEL`. If `channel` is `NULL`, this method matches all channels.
- `level` — The minimum severity level of events to be logged. All events of the given severity level and higher are logged. This parameter can have one of the following values:
  - `ENFORCER_LOG_FATAL` — Fatal errors.
  - `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - `ENFORCER_LOG_WARNING` — Errors worth noting.
  - `ENFORCER_LOG_INFO` — Information only.
  - `ENFORCER_LOG_DEBUG` — Debugging information.
- `details` — The descriptive details of the event/message being logged.
- `toAppend` — Additional data to be appended to the end of the message.
- `message` — A summary of the event/message being logged. This is a printf format sting and is used to display additional parameters.
- `ap` — A pointer to an array variable that contains additional arguments to be logged.

Return Values None.

setDestination()

Description Sets the log destination.

Class `LogFilter`

Include File `LogFilter.h`

Syntax

```
void setDestination(
     LogDestination *     dest
);
```

Parameters

- `dest` — A pointer to the destination object.

Return Values None.

Remarks The `LogFilter` object takes ownership of the destination and will delete it when destroyed.

Example The following code sample initializes the empty logger then sets the logging destination.

```
log_dest_flags = destcode;
Logger::init(progName);
int logLevel = (log_dbg_msgs ? ENFORCER_LOG_DEBUG :
     ENFORCER_LOG_INFO);
if ((destcode == 0) || (destcode & ENFORCER_LOGTO_STDERR)) {
  LogDestination *ld = LogStderr::FactoryFunc(progName, NULL);
  LogFilter *lf = new LogFilter();
  lf->setDestination(ld);
  lf->addChannel(NULL, logLevel);
  Logger::addFilter(lf);
}
```

LogDestination

The `LogDestination` class is an abstract class that defines the destination for a log message.

Class Summary

Because `LogDestination` is an abstract class, no instance of this class is constructed. However, every concrete `LogDestination` object must define a `FactoryFunc` method, as follows:

```
static LogDestination * FactoryFunc (
     const char *          application,
     const XmlTreeNode *     conf
);
```

When the logger calls a factory, the `application` string is guaranteed to exist for the lifetime of the destination, so the destination can just stash the pointer rather than copying (and later freeing) the string.

The `LogDestination` class is a parent to the child classes listed in Table 29.

Table 29 Class Overview

Class Name	Description	Page
`LogServer`	The `LogServer` class logs messages to a Select Access SOAP audit server.	269
`LogStdio`	The `LogStdio` is an abstract class that logs messages to a C standard I/O file pointer. Used as the parent place holder for `LogFile` and `LogStderr`.	272
`LogFile`	The `LogFile` class logs messages to the given file, rolling over when the file exceeds the given length. Extends the `LogStdio` class.	269
`LogStderr`	The `LogStderr` class logs messages to the standard error output. Extends the `LogStdio` class.	278
`LogSystem`	The `LogSystem` class logs messages to the system log, either syslog on UNIX, or the Event Log on Windows.	281

Table 30 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 30 Method Overview

Method Name	Description	Page
`log()`	Logs a message to the log destination. Overridden by the specific `log()` method of its member classes.	268

log()

Description Logs a message to the log destination. Overridden by the specific `log()` method of its member classes.

Class `LogDestination`

Include File `LogDestination.h`

Syntax

```
void log(
     const char *     channel,
     int              level,
     const char *     details,
     const char *     toAppend,
     const char *     message,
     va_list          ap
);
```

Parameters

- `channel` — The type of audit event that you want to filter on, such as `Logger.ADMIN_CHANNEL`, or `Logger.CONFIG_CHANNEL`.
- `level` — The minimum severity level of events to be logged. All events of the given severity level and higher are logged. This parameter can have one of the following values:
  - `ENFORCER_LOG_FATAL` — Fatal errors.
  - `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - `ENFORCER_LOG_WARNING` — Errors worth noting.
  - `ENFORCER_LOG_INFO` — Information only.
  - `ENFORCER_LOG_DEBUG` — Debugging information.
- `details` — The descriptive details of the event/message being logged.
- `toAppend` — Additional data to be appended to the end of the message.
- `message` — A summary of the event/message being logged. This is a printf format sting and is used to display additional parameters.
- `ap` — A pointer to a an array variable that contains additional arguments to be logged.

Return Values None.

LogServer

The `LogServer` class logs messages to a Select Access SOAP audit server.

Class Summary

Table 31 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 31 Class Overview

Method Name	Description	Page
Construction and Destruction		
`~LogServer`	Destructor.	NA
Member Methods		
`FactoryFunc()`	Creates a SOAP audit server destination and returns an XML pointer.	270
`log()`	Logs a message to the SOAP audit server.	271

FactoryFunc()

Description Creates a SOAP audit server destination and returns an XML pointer.

Class `LogServer`

Include File `LogServer.h`

Syntax

```
LogDestination * FactoryFunc(
     const char *            application,
     const XmlTreeNode *     conf
);
```

Parameters

- `application` — The application that is sending the message to the log.
- `conf` — A pointer to an XML node containing file logger configuration information. That is, one of `systemLog` or `file` or `standardErr` or `logServer`. File logger information expects the XML node to be in the form:

  `<file name="/`*file*`/`*name*`" maxSize="1000000"/>`

  where:

  — `file name` is the name of the file to which messages will be logged.

  — `maxSize` is the maximum size the file can reach before it rolls over. If this attribute is missing or is not greater than zero, the log is not rolled over.

  If `conf` is `NULL`, this function configures an empty `Logger`.

Return Values A pointer to the new log destination, or `NULL`, if unsuccessful.

Example The following code sample clears the Logger object's configuration, then registers the built-in log destinations.

```
Logger::clear();
instance = new Logger(application);

Logger::registerDest("standardErr", LogStderr::FactoryFunc);
Logger::registerDest("file", LogFile::FactoryFunc);
Logger::registerDest("systemLog", LogSystem::FactoryFunc);
Logger::registerDest("logServer", LogServer::FactoryFunc);
```

log()

Description Logs a message to the SOAP audit server.

Class `LogServer`

Include File `LogServer.h`

Syntax

```
void log(
     const char *     channel,
     int              level,
     const char *     details,
     const char *     toAppend,
     const char *     message,
     va_list          ap
) = 0;
```

Parameters

- `channel` — The type of audit event that you want to filter on, such as `Logger.ADMIN_CHANNEL`, or `Logger.CONFIG_CHANNEL`.
- `level` — The minimum severity level of events to be logged. All events of the given severity level and higher are logged. This parameter can have one of the following values:
  - `ENFORCER_LOG_FATAL` — Fatal errors.
  - `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - `ENFORCER_LOG_WARNING` — Errors worth noting.
  - `ENFORCER_LOG_INFO` — Information only.
  - `ENFORCER_LOG_DEBUG` — Debugging information.
- `details` — The descriptive details of the event/message being logged.
- `toAppend` — Additional data to be appended to the end of the message.
- `message` — A summary of the event/message being logged. This is a printf format sting and is used to display additional parameters.
- `ap` — A pointer to an array variable that contains additional arguments to be logged.

Return Values None.

LogStdio

The `LogStdio` is an abstract class that logs messages to a C standard I/O file pointer. Used as the parent place holder for `LogFile` and `LogStderr`.

Class Summary

Table 32 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 32 Class Overview

Method	Description	Page
Construction and Destruction		
`~LogStdio`	Destructor.	NA
Member Methods		
`log()`	Logs a message to a C standard I/O file pointer.	273

log()

Description Logs a message to a C standard I/O file pointer.

Class `LogStdio`

Include File `LogStdio.h`

Syntax

```
void log(
     const char *     channel,
     int              level,
     const char *     details,
     const char *     toAppend,
     const char *     message,
     va_list          ap
);
```

Parameters

- `channel` — The type of audit event that you want to filter on, such as `Logger.ADMIN_CHANNEL`, or `Logger.CONFIG_CHANNEL`.
- `level` — The minimum severity level of events to be logged. All events of the given severity level and higher are logged. This parameter can have one of the following values:
  - `ENFORCER_LOG_FATAL` — Fatal errors.
  - `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - `ENFORCER_LOG_WARNING` — Errors worth noting.
  - `ENFORCER_LOG_INFO` — Information only.
  - `ENFORCER_LOG_DEBUG` — Debugging information.
- `details` — The descriptive details of the event/message being logged.
- `toAppend` — Additional data to be appended to the end of the message.
- `message` — A summary of the event/message being logged. This is a printf format sting and is used to display additional parameters.
- `ap` — A pointer to an array variable that contains additional arguments to be logged.

LogFile

The `LogFile` class logs messages to the given file, rolling over when the file exceeds the given length. Extends the `LogStdio` class.

Class Summary

Table 33 summarizes the member methods of this class.These methods are described in detail in the sections that follow.

Table 33 Class Summary

Method	Description	Page
Construction and Destruction		
`~LogFile`	Destructor. Releases `LogFile` objects.	NA
Member Methods		
`CreateFileLogger()`	Creates a file logger using arguments rather than an XML pointer.	269
`FactoryFunc()`	Creates a file logger using an XML pointer to the destination.	276
`log()`	Logs a message to the standard I/O.	277

CreateFileLogger()

Description Creates a file logger using arguments rather than an XML pointer.

Class `LogFile`

Include File `LogFile.h`

Syntax

```
static LogDestination * CreateFileLogger(
     const char *     application,
     const char *     fileName,
     long             maxSize
);
```

Parameters

- `application` — The application that is sending the message to the log.
- `fileName` — The name of the file to which messages will be logged.
- `maxSize` — The maximum size the file can reach before it rolls over. If `maxSize` is missing or is not greater than zero, the log is not rolled over.

Return Values A new log file, or `NULL`, if unsuccessful.

Remarks Using arguments allows the logger to be used by the `EnforcerLogTo()` function.

Example The following code sample initializes the empty logger then sets the logging destination.

```
log_dest_flags = destcode;
Logger::init(progName);
int logLevel = (log_dbg_msgs ? ENFORCER_LOG_DEBUG :
     ENFORCER_LOG_INFO);
if (destcode & ENFORCER_LOGTO_FILE) {
   LogDestination *ld = LogFile::CreateFileLogger(progName, filename,
     0);
   LogFilter *lf = new LogFilter();
   lf->setDestination(ld);
   lf->addChannel(NULL, logLevel);
   Logger::addFilter(lf);
}
```

See Also

- `EnforcerLogTo()` on page 50

FactoryFunc()

Description Creates a file logger using an XML pointer to the destination.

Class `LogFile`

Include File `LogFile.h`

Syntax

```
LogDestination * FactoryFunc(
     const char *            application,
     const XmlTreeNode *     conf
);
```

Parameters

- `application` — A string identifying the application that is sending the message to the log.
- `conf` — A pointer to an XML node containing file logger configuration information.

Return Values A new log file, or `NULL`, if unsuccessful.

Example The following code sample clears the Logger object's configuration, then registers the built-in log destinations.

```
Logger::clear();
instance = new Logger(application);

Logger::registerDest("standardErr", LogStderr::FactoryFunc);
Logger::registerDest("file", LogFile::FactoryFunc);
Logger::registerDest("systemLog", LogSystem::FactoryFunc);
Logger::registerDest("logServer", LogServer::FactoryFunc);
```

log()

Description Logs a message to the standard I/O.

Class `LogFile`

Include File `LogFile.h`

Syntax

```
void log(
     const char *     channel,
     int              level,
     const char *     details,
     const char *     toAppend,
     const char *     message,
     va_list          ap
);
```

Parameters

- `channel` — The type of audit event that you want to filter on, such as `Logger.ADMIN_CHANNEL`, or `Logger.CONFIG_CHANNEL`. If `channel` is `NULL`, this method matches all channels.
- `level` — The minimum severity level of events to be logged. All events of the given severity level and higher are logged. This parameter can have one of the following values:
  - `ENFORCER_LOG_FATAL` — Fatal errors.
  - `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - `ENFORCER_LOG_WARNING` — Errors worth noting.
  - `ENFORCER_LOG_INFO` — Information only.
  - `ENFORCER_LOG_DEBUG` — Debugging information.
- `details` — The descriptive details of the event/message being logged.
- `toAppend` — Additional data to be appended to the end of the message.
- `message` — A summary of the event/message being logged. This is a printf format sting and is used to display additional parameters.
- `ap` — A pointer to an array variable that contains additional arguments to be logged.

Return Values None.

LogStderr

The `LogStderr` class logs messages to the standard error output. Extends the `LogStdio` class.

Class Summary

Table 34 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 34 Class Overview

Method	Description	Page
Construction and Destruction		
`~LogStderr`	Destructor. Releases `LogStderr` objects.	NA
Member Methods		
`FactoryFunc()`	Creates a file logger using an XML pointer to the destination.	279
`log()`	Logs a message to the `stderr`.	280

FactoryFunc()

Description Creates a file logger using an XML pointer to the destination.

Class `LogStderr`

Include File `LogStderr.h`

Syntax

```
LogDestination * FactoryFunc(
     const char *            application,
     const XmlTreeNode *     conf
);
```

Parameters

- `application` — The application that is sending the message to the log.
- `conf` — A pointer to an XML node containing file logger configuration information.

Example The following code sample initializes the empty `Logger` object then sets the logging destination.

```
log_dest_flags = destcode;
Logger::init(progName);
int logLevel = (log_dbg_msgs ? ENFORCER_LOG_DEBUG :
     ENFORCER_LOG_INFO);
if ((destcode == 0) || (destcode & ENFORCER_LOGTO_STDERR)) {
  LogDestination *ld = LogStderr::FactoryFunc(progName, NULL);
  LogFilter *lf = new LogFilter();
  lf->setDestination(ld);
  lf->addChannel(NULL, logLevel);
  Logger::addFilter(lf);
}
```

log()

Description Logs a message to the `stderr`.

Class `LogStderr`

Include File `LogStderr.h`

Syntax

```
void log(
     const char *     channel,
     int              level,
     const char *     details,
     const char *     toAppend,
     const char *     message,
     va_list          ap
);
```

Parameters

- `channel` — The type of audit event that you want to filter on, such as `Logger.ADMIN_CHANNEL`, or `Logger.CONFIG_CHANNEL`.
- `level` — The minimum severity level of events to be logged. All events of the given severity level and higher are logged. This parameter can have one of the following values:
  - `ENFORCER_LOG_FATAL` — Fatal errors.
  - `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - `ENFORCER_LOG_WARNING` — Errors worth noting.
  - `ENFORCER_LOG_INFO` — Information only.
  - `ENFORCER_LOG_DEBUG` — Debugging information.
- `details` — The descriptive details of the event/message being logged.
- `toAppend` — Additional data to be appended to the end of the message.
- `message` — A summary of the event/message being logged. This is a printf format sting and is used to display additional parameters.
- `ap` — A pointer to an array variable that contains additional arguments to be logged.

Return Values None.

LogSystem

The `LogSystem` class logs messages to the system log, either syslog on UNIX, or the Event Log on Windows.

Class Summary

Table 35 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 35 Class Overview

Method	Description	Page
Construction and Destruction		
`~LogSystem`	Destructor. Releases `LogSystem` objects.	NA
Member Methods		
`FactoryFunc()`	Creates a file logger using an XML pointer.	282
`log()`	Logs a message to the system log, either SYSLOG on UNIX, or the Event Log on Windows.	283

FactoryFunc()

Description Creates a file logger using an XML pointer.

Class `LogSystem`

Include File `LogSystem.h`

Syntax

```
LogDestination *              FactoryFunc(
     const char *             application,
     const XmlTreeNode *      conf
);
```

Parameters

- `application` — A string identifying the application that is sending the message to the log.
- `conf` — A pointer to an XML node containing file logger configuration information. This method expects `XmlTreeNode` to be in the form:

  ```
  <file name="/file/name" maxSize="1000000"/>
  ```

  where:

  `file name` is the name of the file to which messages will be logged.

  `maxSize` is the maximum size the file can reach before it rolls over.

Remarks If the `maxSize` attribute of `*conf` is missing or is not greater than zero, the log is not rolled over.

Example The following code sample clears the `Logger` object's configuration, then registers the built-in log destinations.

```
Logger::clear();
instance = new Logger(application);

Logger::registerDest("standardErr", LogStderr::FactoryFunc);
Logger::registerDest("file", LogFile::FactoryFunc);
Logger::registerDest("systemLog", LogSystem::FactoryFunc);
Logger::registerDest("logServer", LogServer::FactoryFunc);
```

log()

Description Logs a message to the system log, either SYSLOG on UNIX, or the Event Log on Windows.

Class `LogSystem`

Include File `LogSystem.h`

Syntax

```
void log(
     const char *     channel,
     int              level,
     const char *     details,
     const char *     toAppend,
     const char *     message,
     va_list          ap
);
```

Parameters

- `channel` — The type of audit event that you want to filter on, such as `Logger.ADMIN_CHANNEL`, or `Logger.CONFIG_CHANNEL`.
- `level` — The minimum severity level of events to be logged. All events of the given severity level and higher are logged. This parameter can have one of the following values:
  - `ENFORCER_LOG_FATAL` — Fatal errors.
  - `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - `ENFORCER_LOG_WARNING` — Errors worth noting.
  - `ENFORCER_LOG_INFO` — Information only.
  - `ENFORCER_LOG_DEBUG` — Debugging information.
- `details` — The descriptive details of the event/message being logged.
- `toAppend` — Additional data to be appended to the end of the message.
- `message` — A summary of the event/message being logged. This is a printf format sting and is used to display additional parameters.
- `ap` — A pointer to an array variable that contains additional arguments to be logged.

8 Exception Handling API

The Exception Handling API provides one class which is used to handle exceptions. This chapter provides a reference for functions in this single class.

Table 36 Class Summary

Class Name	Description	Page
`EnforcerException`	The `EnforcerException` class manages errors generated by the Enforcer plugin.	285

EnforcerException

The `EnforcerException` class manages errors generated by the Enforcer plugin.

Class Summary

Table 37 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 37 EnforcerException Class Summary

Method Name	Description	Page
Construction and Destruction		
`EnforcerException()`	Constructor. Creates the `EnforcerException` object.	287
`~EnforcerException()`	Destructor. Releases the `EnforcerException` object.	NA
Member Methods		
`getErrorNum()`	Returns the error number of the current exception.	288
`getErrorLoc()`	Returns the location where the error occurred, in string form.	289
`getErrorDetails()`	Returns the specific details of the error, in string form.	290
`getErrorNumString()`	Returns the error string corresponding to this exception's error number.	291
`getErrorString()`	Returns this exception's full error string.	292
`setError()`	Copies the contents of the exception object into thread local storage so that it can be read by C applications.	293

Definitions

The `EnforcerException` class declares an enumerated list of error strings. Table 38 lists the possible error strings that can be returned.

Table 38 EnforcerException Error Strings

Name	Value	Error #
`E_NULL_ARG`	"Null argument error."	`0`
`E_INVALID_ARG`	"Invalid argument error."	`1`
`E_MISSING_ARG`	"Missing argument error."	`2`
`E_OUT_OF_RANGE`	"Argument is out of range."	`3`
`E_NOT_IMPL`	"Method is not implemented."	`4`
`E_COM_FACADE`	"COM Facade error."	`5`
`E_B64_LENGTH`	"Encoded length doesn't match `b64len`."	`6`
`E_DECIDER`	"Decider error occurred."	`7`
`E_LDAP`	"LDAP error."	`8`
`E_FILE`	"Error opening file."	`9`
`E_PARSE`	"Parse error."	`10`
`E_AUTH`	"Authenticator error."	`11`
`E_SIGNATURE`	"Policy data failed signature verification."	`12`
`E_CONFIG`	"Configuration error."	`13`
`E_UNKNOWN`	"Unknown error type."	`14`

EnforcerException()

Description: Constructor. Creates the `EnforcerException` object.

Class: `EnforcerException`

Include File: `EnforcerException.h`

Syntax:

```
EnforcerException(
     unsigned int       errNum,
     const string &     location,
     const string &     details
);
```

Parameters:

- `errNum` — The error number of the exception.
- `&location` — A pointer to the location where the error occurred.
- `&details` — A detailed description of the error that occurred.

Return Values: None.

getErrorNum()

Description: Returns the error number of the current exception.

Class: `EnforcerException`

Include File: `EnforcerException.h`

Syntax:
```
unsigned int getErrorNum();
```

Parameters: None.

Return Values: The error number of the current exception.

getErrorLoc()

Description: Returns the location where the error occurred, in string form.

Class: `EnforcerException`

Include File: `EnforcerException.h`

Syntax:

```
const char * const getErrorLoc();
```

Parameters: None.

Return Values: The location where the error occurred.

getErrorDetails()

Description: Returns the specific details of the error, in string form.

Class: `EnforcerException`

Include File: `EnforcerException.h`

Syntax:

```
const char * const getErrorDetails();
```

Parameters: None.

Return Values: The error details.

getErrorNumString()

Description: Returns the error string corresponding to this exception's error number.

Class: `EnforcerException`

Include File: `EnforcerException.h`

Syntax:

```
const char * const getErrorNumString();
```

Parameters: None.

Return Values: The string corresponding to the error number listed in Table 38.

getErrorString()

Description: Returns this exception's full error string.

Class: `EnforcerException`

Include File: `EnforcerException.h`

Syntax:
```
const char * const getErrorString();
```

Parameters: None.

Return Values: The full error string.

Remarks: The full error string is equivalent to `errString[errorNumber] + errorLocation + errorDetails`.

setError()

Description: Copies the contents of the exception object into thread local storage so that it can be read by C applications.

Class: `EnforcerException`

Include File: `EnforcerException.h`

Syntax:
```
void setError();
```

Parameters: None.

Return Values: None.

Some Addenda on Error Handling

While this chapter exclusively documents the C++ exception handling functions you can use with your C++ Enforcer plugin, you can also use the C equivalents from the Enforcer API. With typical C++ exception handling, you would probably catch Enforcer plugin exception as shown in Code Example: Catching Exceptions below.

Code Example: Catching Exceptions

```
try {
   // ... Call Enforcer C++ functions here
}
catch (EnforcerException &e) {
   cerr << e.getErrorString();
   return;
}
// ... Continue
```

Differences in Error Handling Between Programming Languages

Depending on whether you are creating a C-based Enforcer plugin or one that is written exclusively in C++, the manner in which you handle exceptions will vary.

To handle exceptions in C

Custom Enforcer plugins written in C must use C functions to handle exceptions. C functions usually return `1` for success and `0` for error. Details about the error are usually available with the `EnforcerGetLastError()` function.

The signature of the `EnforcerGetLastError()` function is:

```
const EnforcerErrorData * EnforcerGetLastError(void);
```

This function returns a pointer to a static `EnforcerErrorData` structure that contains the error details. The structure is declared in the `enforcer.h` file as shown in Code Example: EnforcerErrorData Structure.

Code Example: EnforcerErrorData Structure

```
typedef struct {
   unsigned int number; - the error number
   const char *location; - location, usually the function where error
      occured
   const char *details; - an error message explaining what the problem
      was
} EnforcerErrorData;
```

To handle exceptions in C++

Unlike the Enforcer plugins written in C, the C++ equivalents can use either C or C++ functions to handle their exceptions:

- If you would like to use the C functions to handle the exceptions thrown by some of the C++ instances of `EnforcerException`, you must call the C Enforcer API functions available in the API. Code Example: Handling Errors With C Functions on page 295 shows how you could use the C Enforcer API to handle the error in a C++ Enforcer plugin.
- If you would like to use the C++ functions, you need to catch any exceptions that are thrown.

Code Example: Handling Errors With C Functions

```
if ( EnforcerDocParseMem(/*...parameters...*/) != SUCCESS ) {
   const EnforcerErrorData *error = EnforcerGetLastError();
   if ( error == NULL ) {
      /* ... Print error->number, error->location, error->details here */
      return FAILURE;
   }
   return SUCCESS;
}
```

9 C/C++ Constant Strings

The Enforcer plugin uses a large number of constants that are repeatedly copied back and forth. These constants are defined in a separate header file, `EnforcerPredefTable.h`. This chapter provides a reference to C/C++ constant strings.

Instead of defining these constants in the standard manner as in the `enforcer.h` file, the Enforcer C/C++ API treats these constants as lightweight functions. By doing so, plugins using this code are able to use the same memory for all of the constants, resulting in an improvement in performance.

Constants Strings You Can Use

Table 39 lists the `EnforcerPredefTable.h` constants. Only the constants used by the Enforcer plugin have been documented therein. Policy Validator-specific constants have not been added to this table.

Table 39 EnforcerPredef.Table.h Constants

Name	Value	Represents
ENFORCER_ACTION	action	The XML response property element which indicates the action the Enforcer plugin should take. Will be one of: • Allow • Deny • Error • Developer-defined
ENFORCER_ACTION_ALLOW	ALLOW	The Policy Validator value for ENFORCER_ACTION. In this case, the action is allowed.
ENFORCER_ACTION_DENY	DENY	The Policy Validator value for ENFORCER_ACTION. In this case, the action is denied.
ENFORCER_ACTION_ERROR	ERROR	The Policy Validator value for ENFORCER_ACTION. In this case, the action is error.
ENFORCER_ACTION_USER_DEFINED	USER_DEFINED	The Policy Validator value for ENFORCER_ACTION. In this case, the action is developer-defined.

Table 39 EnforcerPredef.Table.h Constants (cont'd)

Name	Value	Represents
ENFORCER_ATTRIBUTE_NAME	NAME	An XML element's NAME attribute.
ENFORCER_AUTH_HINT	authentication_hint	The XML response property element which contains hints for further authentication.
ENFORCER_AUTHENTICATED_DN	authenticated_dn	The XML property in the Policy Validator response, which specifies the authenticated identity's distinguished names.
ENFORCER_AUTHENTICATION_METHOD	authentication_method	The XML property in the Policy Validator response, which specifies the method of authentication used by the client. This property is nested in <PROPERTYLIST NAME="Authentication_Server_Types>.
ENFORCER_AUTHENTICATION_SERVER_TYPES	authentication_server_types	The name of an XML query property list, which contains the types of servers used for authentication.
ENFORCER_CERT	cert	The name of an XML query property, which contains the client's PEM-encoded X.509 certificate.
ENFORCER_CHALLENGE_RESPONSE	chalresp	The name of an XML query property, which contains the challenge response.
ENFORCER_DOMAIN	domain	The name of an XML query property, which contains the authentication domain.
ENFORCER_DSTHOSTNAME	dstHost	The name of an XML query property, which contains the host name of the computer on which the resource is located.
ENFORCER_DSTIP	dstIP	The name of an XML query property, which contains the IP address of the computer on which the resource is located.
ENFORCER_DSTPORT	dstPort	The name of an XML query property, which contains the port number of the computer on which a resource is located.

Table 39 EnforcerPredef.Table.h Constants (cont'd)

Name	Value	Represents
ENFORCER_ENABLE	enable	The value of a boolean query property such as Native Nonce.
ENFORCER_ERROR	error	The name of a response property, which contains an error message from the Policy Validator.
ENFORCER_FORM_ AUTHENTICATOR_TYPE	authenticator	The name of the response property, which contains the name of the authentication form. This constant is sent as the result of an authentication hint from the Policy Validator.
ENFORCER_FORM_ CHALLENGE	form_challenge	The name of the response property, which contains the name of the challenge form. This constant is sent as the result of an authentication hint from the Policy Validator.
ENFORCER_FORM_DATA	form_data	The name of the response property list, which contains the name of the form data that needs to be displayed to the client.
ENFORCER_FORM_ID	form_id	The name of the response property list, which contains the id of the form to be displayed to the client.
ENFORCER_FORM_PASSWORD _MGMT_ID	password_mgmt	The value of the response property, which contains the id of the password form to be displayed to the client.
ENFORCER_FORM_PROFILE _MGMT_ERROR_FORM_ID	profile_error	The value of the response property, which contains the id of the profile management error form to be displayed to the client.
ENFORCER_FORM_PROFILE_ MGMT_ERROR_FORM_TEXT	profile_error	The value of the response property list, which contains the text of the profile management form to be displayed to the client.

Table 39 EnforcerPredef.Table.h Constants (cont'd)

Name	Value	Represents
ENFORCER_FORM_PROFILE_MGMT_ID	profile_mgmt	The value of the response property, which contains the ID of the profile management form to be displayed to the client.
ENFORCER_FORM_PROFILE_MGMT_OK_FORM_ID	profile_ok	The value of the response property, which contains the ID for the OK form to be displayed to the client.
ENFORCER_FORM_PAGE	form_page	The value of the response property, which contains the ID of the form to be displayed to the client.
ENFORCER_FORM_RADIUS_STATE	form_radius_state	The value of the response property, which contains the ID of the RADIUS form to be displayed to the client.
ENFORCER_FORM_REALM	form_realm	The value of the response property, which contains the realm of the form to be displayed to the client.
ENFORCER_FORM_TITLE	form_title	The value of the response property, which contains the title of the form to be displayed to the client.
ENFORCER_FORM_TYPE	form_type	The value of the response property, which contains the type of the form to be displayed to the client.
ENFORCER_FORM_USER	user	The value of the response property, which contains the ID of the identity management form to be displayed to the client.
ENFORCER_GROUPINFO	group_info	The name of the response property list, which contains the identity's group personalization information.
ENFORCER_HTTP_HEADER_LIST	http_header_list	The name of the XML query property list, which contains the HTTP header list sent by the client.
ENFORCER_HTTP_QUERY	http_query	The name of the XML query property, which contains a single CGI query parameter.

Table 39 EnforcerPredef.Table.h Constants (cont'd)

Name	Value	Represents
ENFORCER_HTTP_QUERY_LIST	http_query_list	The name of the XML query property list, which contains properties for each CGI query parameter.
ENFORCER_LOGIN_TIME	login_time	The name of the response property, which contains the time at which identity was authenticated.
ENFORCER_MESSAGE	message	The name of the response property, which contains the text of an information message.
ENFORCER_METHOD	method	The name of the XML query property, which contains the HTTP method in use. The HTTP method can be one of: • Get • Put • Head • Login
ENFORCER_MULTI_RESOURCE_LIST	multiResourceList	The name of the XML query property, which holds a list of resources.
ENFORCER_NATIVE_AUTH	native_auth	The name of the XML query property, which contains the authentication type supported by the Enforcer plugin. The authentication type can be one of: • password • online registration • certificate • nonce
ENFORCER_NATIVE_CERT	native_certificate	There are two options: • For the value of an XML query, which specifies that the Enforcer plugin supports X.509 certificate authentication. • For the value of a response, which specifies that the X.509 certificate authentication failed.

Table 39 EnforcerPredef.Table.h Constants (cont'd)

Name	Value	Represents
ENFORCER_NATIVE_CHAL	native_challenge	For the value of a response, indicates challenge/response authentication is in progress.
ENFORCER_NATIVE_NONCE	native_nonce	The name of the XML query property, which indicates whether nonces have been enabled or not.
ENFORCER_NATIVE_PASSWD	native_password	There are two options: • For the value of an XML query, which specifies that the Enforcer plugin supports password authentication. • For the value of a response, which specifies that password authentication failed.
ENFORCER_NATIVE_ REGISTER	native_register	There are two options: • For the value of an XML query, which specifies that the Enforcer plugin supports registration authentication. • For the value of a response, which specifies that the registration authentication failed.
ENFORCER_NEED_AUTH	need_auth	The name of the XML response property, which indicates whether further authentication is required.
ENFORCER_NONCE	nonce	The name of the XML response property, which contains a nonce whose value is typically an HTTP cookie.
ENFORCER_P13NINFO	personalization	The name of the XML response property list, which contains personalization information for the identity making the request.
ENFORCER_PASSWD	password	The name of the XML query property, which contains the identity's password.

Table 39 EnforcerPredef.Table.h Constants (cont'd)

Name	Value	Represents
ENFORCER_PATH	path	The name of the XML query property, which contains the part of the URL used to locate a resource on a service. For example, "/", "/cgi/finger".
ENFORCER_POST_DATA_LIST	post_data_list	The name of the XML query property, which contains data posted by the client.
ENFORCER_PROTOCOL	protocol	The name of the XML query property, which contains protocol used to request a resource. For example, "http", "ftp", "file", and so on.
ENFORCER_QUERY	PolicyValidatorQuery	The name of the root XML query element for the Policy Validator.
ENFORCER_QUERY_END	</PolicyValidator Query>	The closing tag for the root element of an XML query document.
ENFORCER_QUERY_START	<PolicyValidatorQuery>	The opening tag for the root element of an XML query document.
ENFORCER_QUERYID	queryID	The name of the XML query property, whose value uniquely identifies the query and allows it to be associated with its corresponding response document.
ENFORCER_QUERYID_END	</PROPERTY>	The closing tag for a property containing the query ID.
ENFORCER_QUERYID_START	<PROPERTY="queryID">	The opening tag for a property containing the query ID.
ENFORCER_REDIRECT	redirect_url	The name of the XML response property, which contains the URL to which the browser will be redirected.
ENFORCER_REFERER	referer	The URL from which the request was made.
ENFORCER_REPLY	PolicyValidatorReply	The name of the root XML response element for the Policy Validator.

Table 39 EnforcerPredef.Table.h Constants (cont'd)

Name	Value	Represents
ENFORCER_REPLY_END	</PolicyValidator Reply>	The closing tag for the root element of an XML response document.
ENFORCER_REPLY_START	<PolicyValidatorReply>	The opening tag for the root element of an XML response document.
ENFORCER_RESOURCE	resource	The name of a XML property in a multiple resource query, which contains a resource path.
ENFORCER_ROLEINFO	role_info	The name of a XML response property list, which contains personalization information for an identity based on dynamic group information.
ENFORCER_SEND_REFRESH	selectaccess_send_ refresh	The name of the tag on the Select Access login forms, which tells the Enforcer plugins to send the "Credentials accepted" page upon a successful authentication.
ENFORCER_SERVER	server	The name of the XML query property, which contains the server software name and version.
ENFORCER_SERVICE	service	The name of the XML query property, which contains the name and port number of the service which hosts a resource. For example, "mycompany.com:8099".
ENFORCER_SITE_DATA	site_data	The name of the XML query property, which contains site-specific data for a web server.
ENFORCER_SIZE	size	*For Apache 2 Enforcer plugins only.* The name of the XML query property, which contains the size of the query.
ENFORCER_SRCHOSTNAME	srcHost	The name of the XML query property, which contains the name of the machine on which the request originated.

Table 39 EnforcerPredef.Table.h Constants (cont'd)

Name	Value	Represents
ENFORCER_SRCIP	srcIP	The name of the XML query property, which contains the IP address of the machine of which the request originated.
ENFORCER_SRCPORT	srcPort	The name of the XML query property, which contains the port number being used by the machine on which the request originated.
ENFORCER_SSL_CIPHER	ssl_cipher	The name of the XML query property, which contains the SSL logon cipher used by the client.
ENFORCER_SSL_CLIENT_DN	dn	The name of the XML query property, which contains the DN of the client's certificate.
ENFORCER_SSL_KEYSIZE	ssl_keysize	The name of the XML query property, which contains the SSL cipher's keysize.
ENFORCER_SSL_PROTOCOL	ssl_protocol	The name of the XML query property, which contains the SSL cipher's protocol.
ENFORCER_TAG_PROPERTY	PROPERTY	An XML <PROPERTY> element.
ENFORCER_TAG_ PROPERTYLIST	PROPERTYLIST	An XML <PROPERTYLIST> element.
ENFORCER_TRACE	trace	The name of the XML query property, which contains a verbose rule-evaluation trace from the Policy Validator.
ENFORCER_TRUSTED_AUTH_ DOMAIN	domain_name	The name of the XML query property, which contains the name of a trusted authentication domain.
ENFORCER_TRUSTED_AUTH_ LIST	trusted_authentication	The name of the XML query property, which contains the trusted authentication information.

Table 39 EnforcerPredef.Table.h Constants (cont'd)

Name	Value	Represents
ENFORCER_TRUSTED_AUTH_TYPE	auth_type	The name of the XML query property, which contains the trusted authentication type.
ENFORCER_TRUSTED_AUTH_USER	user_name	The name of the XML query property, which contains the name of a trusted authentication identity.
ENFORCER_USER	user	• For the value of an XML query, which specifies that the Enforcer plugin supports password authentication. • For the value of a response, which specifies that password authentication failed.

Section Two: The Java APIs

This section describes the classes and methods of Select Access' Java API libraries. Table 40 on page 308 provides a brief description of the functions and classes required to build each plugin type.

Select Access libraries can be used to construct:

- Java-based enforcers: An access control component added to a web server, a Java servlet, an EJB, or any other resource. The Enforcer plugin intercepts access requests, then sends queries to the Policy Validator to enforce access control.

  For a detailed, step-by-step tutorial on constructing a Java Enforcer plugin, see Chapter 5, Custom Security Services: the Enforcer API in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.

- Policy Builder plugins: Policy Builder plugins come in two kinds:

  — Authentication plugins: An authentication-method configuration dialog added to the Policy Builder application. An authentication plugin is a Jpanel component which allows you to provide configuration options for a Policy Validator authenticator plugin. Together these plugins allow you to add a new authentication method to SelectAuth.

  — Decision Point plugins: A conditional access control configuration dialog added to the Rule Builder utility of the Policy Builder application. A Decision Point plugin is a Jpanel component which allows you to provide configuration options for a Policy Validator Decider plugin. Together these plugins allow you to add additional methods of controlling conditional access to resources.

  For a detailed, step-by-step tutorial on constructing these plugins, see Chapter 3, Custom Configuration GUIs: the Policy Builder API in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.

The Java API libraries required to create these plugins are found in three packages; one containing Enforcer plugin resources, one containing Policy Builder resources, and one containing resources that will be used by both Enforcers and Policy Builder plugins.

Table 40 Java API Libraries

Class	Description	Used by			Page
		Authenticator	Decider	Enforcer plugin	
Enforcer plugin API (`com.hp.ov.selectaccess.enforcer`)					
`WebTransaction`	Provides a number of abstract methods which must be over-written in application-specific code.			•	310
`Enforcer`	Provides a number of utility methods which support the Enforcer plugin logic, allowing you to, for example, retrieve the Policy Validator handle object, construct XML documents, extract data from the request URL, etc.			•	326
Policy Builder plugin API (`com.hp.ov.selectaccess.policybuilder`)					
`RuleComponent-Panel`	Defines the base class for creating both Authentication and Decision Point plugins.	•	•		372
XML manipulation API (`com.hp.ov.selectaccess.util`)					
`XmlElement`	Defines the methods which handle the XML nodes of a parsed XML document.	•	•	•	391
`PropertyElement`	Defines the methods which create a name/value pair.	•	•	•	424
`PropertyList Element`	Defines the methods the create and hand XML nodes that contains a list of zero or more properties and/or nested property lists.	•	•	•	431
Constant strings (`com.hp.ov.selectaccess`)					
`Variable Data Strings`	A list of commonly used constants.	•	•	•	461

10 Java Enforcer API

The Enforcer API allows you to integrate Select Access security services with most products and custom components. This chapter is the reference you can use with the Java version of the Enforcer API, so you can provide security services for Java objects.

If you intend to build your own servlet Enforcer plugin on a Java version prior to 1.4, 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 Java Enforcer API is comprised of the following classes:

Table 41 Java Enforcer API Classes

Class Name	Description	Page
`WebTransaction`	The `WebTransaction` class is an abstract class which provides a framework for securing network resources.	310
`Enforcer`	The `Enforcer` class provides methods which support Enforcer plugin logic by performing some basic Enforcer tasks, such as: • Retrieving the Validator handle object. • Constructing, parsing and manipulating XML documents. • Extracting data from the request URL in order to construct the query. • Initializing the logging framework and logging messages.	326

For information on creating a sample Java-based Enforcer plugin, see Creating an Enforcer plugin With the Java Enforcer API in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.

WebTransaction

The `WebTransaction` class is an abstract class which provides a framework for securing network resources.

`WebTransaction` contains a number of abstract methods that you must implement in application-specific code. These methods are in turn called by `WebTransaction`'s `isAuthorized()` method at the appropriate times during the life of the transaction object.

The `isAuthorized()` method encapsulates all the Enforcer plugin logic. It receives and analyses the URL and chooses the best course of action. It can:

- automatically deny a suspicious request
- automatically allow a request for an unsecured resource
- construct a query and pass it along to the Policy Validator

In addition to the `isAuthorized()` method, this class also contains a number of public methods which are called by `isAuthorized()`. These methods, including `isAuthorized()`, should never need to be called, and are therefore not documented in this guide.

Class Summary

Table 42 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 42 WebTransaction Class Summary

Method Name	Description	Page
Construction		
`WebTransaction(Enforcer, String, String, int, String, String, String, String)`	Constructor. Creates a `WebTransaction` object.	312
`WebTransaction(Enforcer)`	Constructor. Used for applications that can only initialize things incrementally.	313
Abstract methods		
`addAppDataToQuery()`	Abstract method. Adds application-specific data to the XML query object.	314
`addEncoded(XmlElement, String, String)`	Abstract method. Converts the given `value` parameter to UTF-8 and adds it as a `<Property>` to an existing XML object.	315
`allow()`	Abstract method. Informs the web client that the current request has been allowed.	316
`deny()`	Abstract method. Informs the web client that the current request has been denied.	317

Table 42 WebTransaction Class Summary

Method Name	Description	Page
`disableCaching()`	Abstract method. Prevents the current Policy Validator reply from being cached by the web client.	318
`getPostDataBuffer()`	Abstract method. Retrieves a URL-encoded buffer of POST data from the incoming request.	319
`getPostDataTable()`	Abstract method. Retrieves a hashtable of POST data name/value pairs from the incoming request.	320
`sendBasicAuthPage(String)`	Abstract method. Sends an HTTP basic authentication page to the web client.	321
`sendDynamicForm()`	Abstract method. Sends a dynamic form to the web client.	322
`sendRedirect(String)`	Abstract method. Redirects the web client to a new URL.	323
`setCookie(String, String, String, String, int)`	Abstract method. Formats a cookie in accordance with rfc 2109, and sends a Set-cookie header to the web client.	324
`setPersonalization (String, String)`	Abstract method. 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.	325

WebTransaction(Enforcer, String, String, int, String, String, String, String)

Description Constructor. Creates a `WebTransaction` object.

Syntax

```
public WebTransaction(Enforcer enforcer,
                      String protocol,
                      String server,
                      int port,
                      String path,
                      String httpQuery,
                      String method,
                      String remoteAddr)
```

Parameters

- `enforcer` — A non-`NULL` pointer to a previously-initialized `Enforcer` object.
- `protocol` — The protocol used by the client and server.
- `server` — The virtual hostname of the server. If `NULL`, the function will try to determine the hostname.
- `port` — The port on which the server is listening.
- `path` — The unescaped path to the resource for which access is being requested.
- `httpQuery` — The CGI query parameters. That is, the part of a URL after the `?` character.
- `method` — The action. for example, `GET` or `POST`.
- `remoteAddr` — The address of the client machine where the request originated.

See Also

- `WebTransaction(Enforcer)` on page 313

WebTransaction(Enforcer)

Description Constructor. Used for applications that can only initialize things incrementally.

Syntax

```
public WebTransaction(Enforcer enforcer)
```

Parameters

- `enforcer` — A non-`NULL` pointer to a previously-initialized `Enforcer` object.

See Also

- `WebTransaction(Enforcer, String, String, int, String, String, String, String)` on page 312

addAppDataToQuery()

Description Abstract method. Adds application-specific data to the XML query object.

Class `WebTransaction`

Syntax `void addAppDataToQuery()`

Parameters None.

Return Values None.

Remarks Depending on the application, some examples of application-specific data added to the query using this method include:

- The name and password of an HTTP Basic Authentication header
- An SSL certificate, if the application uses HTTPS
- Site data

Example The following code is a sample implementation of this method:

```
protected void addAppDataToQuery() {
  addBasicAuthData(m_request.getHeader(AUTHORIZATION_HEADER),
       m_request.getCharacterEncoding());
  if (m_request.isSecure()) {
    addCertificateData(ServletHelper.GetKeysize(m_request),
      ServletHelper.GetClientCert(m_request));
  }
  String site_data = (String)m_request.getAttribute
       (DataStrings.ENFORCER_SITE_DATA);
  if (site_data != null) {
    m_query.appendPropertyValue(DataStrings.ENFORCER_SITE_DATA,
      site_data);
  }
  if (m_enforcer.getQueryLevel().equals(DataStrings.QUERY_MAXIMAL)) {
    PropertyListElement headers = m_query.appendNestedPropertyList
      (DataStrings.ENFORCER_HTTP_HEADER_LIST);
    Enumeration e = m_request.getHeaderNames();
    while (e.hasMoreElements()) {
      String key = (String)e.nextElement();
      String value = (String)m_request.getHeader(key);
      headers.appendPropertyValue(key, value);
    }
  }
```

addEncoded(XmlElement, String, String)

Description Abstract method. Converts the given `value` parameter to UTF-8 and adds it as a `<Property>` to an existing XML object.

Class `WebTransaction`

Syntax

```
Property addEncoded(XmlElement parent,
                    String name
                    String value)
```

Parameters

- `parent` — The `<PropertyList>` element that will hold the new `<Property>`.
- `name` — The value of the `NAME` attribute of the `<Property>`.
- `value` — The unencoded value of the `<Property>`.

Return Values The newly-created `<Property>`.

allow()

Description Abstract method. Informs the web client that the current request has been allowed.

Class `WebTransaction`

Syntax

```
boolean allow()
```

Parameters None.

Return Values

- `true` — Access was granted for this request.

deny()

Description Abstract method. Informs the web client that the current request has been denied.

Class `WebTransaction`

Syntax

```
boolean deny()
```

Parameters None.

Return Values

- `false` — Access was denied for this request.

disableCaching()

Description Abstract method. Prevents the current Policy Validator reply from being cached by the web client.

Class `WebTransaction`

Syntax `void disableCaching()`

Parameters None.

Return Values None.

Example The following code is a sample implementation of this method:

```
protected void disableCaching() {
  m_response.addHeader(EXPIRES_HEADER, Rfc822GmtDate(null));
  m_response.addHeader(PRAGMA_HEADER, NO_CACHE);
  m_response.addHeader(CACHE_CONTROL_HEADER, NO_CACHE);
}
```

See Also

- `.isDisableCaching()` on page 349

getPostDataBuffer()

Description Abstract method. Retrieves a URL-encoded buffer of POST data from the incoming request.

Class `WebTransaction`

Syntax

```
String getPostDataBuffer()
```

Parameters None.

Return Values The URL-encoded POST data.

See Also

- `getPostDataTable()` on page 320

getPostDataTable()

Description Abstract method. Retrieves a hashtable of POST data name/value pairs from the incoming request.

Syntax

```
Hashtable getPostDataTable()
```

Parameters None.

Return Values A hashtable of name/value pairs.

Example The following code is a sample implementation of this method:

```
protected Hashtable getPostDataTable() {
   if (m_post_data_table != null) {
      return m_post_data_table;
   }
   m_post_data_table = new Hashtable();
   Enumeration paramNames = m_request.getParameterNames();
   String encoding = m_request.getCharacterEncoding();
   while (paramNames.hasMoreElements()) {
      String tmpStr =(String) paramNames.nextElement();
      String paramVal = m_request.getParameter(tmpStr);
      try {
         if (encoding == null) {
            paramVal = new String(paramVal.getBytes("ISO-8859-1"),
               "UTF8");
         }
         if (! nameValueInQuery(tmpStr, paramVal)) {
            m_post_data_table.put(tmpStr, paramVal);
         }
      }
      catch (java.io.UnsupportedEncodingException e) {
         logError("getPostDataTable failed on " + tmpStr);
      }
   }
   return m_post_data_table;
}
```

See Also

- `getPostDataBuffer()` on page 319

sendBasicAuthPage(String)

Description Abstract method. Sends an HTTP basic authentication page to the web client.

Class `WebTransaction`

Syntax

```
boolean sendBasicAuthPage(String realm)
```

Parameters

- `realm` — The authentication realm.

Return Values

- `false` — Identity must be authenticated before access is granted. If `false` is returned, this method logs the reason for the denial.

Example The following code is a sample implementation of this method:

```
protected boolean sendBasicAuthPage(String realm) {
   String header = "Basic realm=\"" + realm + "\"";
   m_response.addHeader(AUTHENTICATE_HEADER, header);
   sendError(HttpServletResponse.SC_UNAUTHORIZED);
   return false;
}
```

sendDynamicForm()

Description Abstract method. Sends a dynamic form to the web client.

Class `WebTransaction`

Syntax `boolean sendDynamicForm(String url)`

Parameters

- `url` — An optional redirect or refresh URL to send in the form.

Return Values

- `false` — The identity cannot be authorized at this time. The dynamic form is needed for authentication.

Example The following code is a sample implementation of this method.

```
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;
}
```

sendRedirect(String)

Description Abstract method. Redirects the web client to a new URL.

Syntax

```
boolean sendRedirect(String redirectURL)
```

Parameters

- `redirectURL` — The URL to which the client will be redirected.

Return Values

- `false` — The identity must be redirected and is therefore not authorized.

Example The following code is a sample implementation of this method.

```
protected boolean sendRedirect(String url) {
   try {
     m_response.sendRedirect(url);
   }
   catch (java.io.IOException e) {
     logError("sendRedirect failed: " + url);
   }
   return false;
}
```

setCookie(String, String, String, String, int)

Description Abstract method. Formats a cookie in accordance with rfc 2109, and sends a Set-cookie header to the web client.

Syntax

```
void setCookie(String name,
               String value,
               String path,
               String domain,
               int duration)
```

Parameters

- `name` — The cookie name.
- `value` — The cookie value.
- `path` — The cookie path, typically "/".
- `domain` — The cookie domain, null to disable.
- `duration` — The cookie lifetime in seconds. This parameter may also take the following special values:
  - `-1` — The cookie is valid only for the current session.
  - `0` — The cookie will be deleted immediately.

Return Values None.

Example The following code is a sample implementation of this method:

```
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);
}
```

setPersonalization(String, String)

Description Abstract method. 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.

Syntax

```
void setPersonalization(String name,
                        String value)
```

Parameters

- `name` — The `NAME` attribute of the `<Property>`.
- `value` — The value of the `<Property>`.

Return Values None.

Example The following code is a sample implementation of this method:

```
protected void setPersonalization(String name, String value) {
   m_p13nMap.put(name, value);
}
```

Enforcer

The `Enforcer` class provides methods which support Enforcer plugin logic by performing some basic Enforcer tasks, such as:

- Retrieving the `Validator handle` object.
- Constructing, parsing and manipulating XML documents.
- Extracting data from the request URL in order to construct the query.
- Initializing the logging framework and logging messages.

Class Summary

Table 43 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 43 Enforcer Class Summary

Method Name	Description	Page
Construction		
`Enforcer(String, String, String, String, boolean, int)`	Constructor. Creates a new Enforcer object. The Enforcer object is used to parse URLs for dangerous characters, determine if authorization is needed, query the Policy Validator, and obtain the response. It can process multiple authorization requests and it is safe to use in multi-threaded environments.	330
Member methods		
`clearUserDataCharacterSet()`	Clears the character set used by the Enforcer object. Because character set conversion should be handled by the transaction object, this method should always be called when a new Enforcer object is constructed.	331
`debugMsg(String)`	Sends the given debugging message to the log.	332
`debugMsg(int, String)`	Sends the given debugging message to the log describing events of the specified debug level. You can configure logging to record messages of a given level only.	333
`errMsg(String)`	Logs the given message with a severity level of `ERROR`.	334

Table 43 Enforcer Class Summary (cont'd)

Method Name	Description	Page
`getAuthHint(XmlElement, StringBuffer)`	Extracts the authentication hints, if any, out of a Policy Validator reply object. If only one hint is present in the reply, it is returned. If more than one hint is present, this method merges them into a list and returns the list.	335
`getBasenameOfURL(String)`	Extracts the basename of the given URL by terminating it prior to an anchor symbol (`#`) or `jsessionid` tag.	336
`getCookieDomainName()`	Returns the name of the domain where the current cookie originated.	337
`getLastError()`	Returns the details about the last Servlet Enforcer plugin function that failed.	338
`getMultiResourceResult(XmlElement)`	Returns the results of a request for multiple resources.	339
`getName()`	Returns the name of the current object.	340
`getNewQueryID()`	Returns a valid query ID.	341
`getQueryLevel()`	Gets the query level setting for the Enforcer object. The query level specifies how much information will go into the query.	342
`getTransformedForm(string, XmlElement, Hashtable)`	Loads and transforms an HTML form page.	343
`getUserDataCharacterSet()`	Returns the character set that is currently used to encode identity data.	344
`getValidatorHandle()`	Returns a pointer to the Policy Validator.	345
`getVersion()`	Returns the version of the current Enforcer plugin application.	346
`initializeLogging(String, XmlElement)`	Initializes the Enforcer plugin logging system.	347
`isConfigured()`	Determines whether or not the current object has been successfully configured.	348
`isDisableCaching()`	Determines whether the Servlet Enforcer plugin is configured to disable caching of Select Access-protected web pages.	349

Table 43 Enforcer Class Summary (cont'd)

Method Name	Description	Page
`isDomainProtected(String)`	Tests the given domain to determine whether it is in the multi-domain single sign-on (MD-SSO) protected sites list.	350
`isEnableCookies()`	Determines whether the use of nonces (secure cookies) is enabled in the Policy Validator.	351
`isEvilURL(String)`	Determines whether the URL path is safe.	352
`isIgnoredFile(String)`	Tests the given file to determine whether it is in the list of ignored filenames. Files listed in the `ignoredFile` list are unconditionally allowed.	353
`isLoginViaForm()`	Determines whether the Enforcer plugin is configured to use a web form to perform login operations.	354
`isPassthroughDomain(String)`	Tests the given virtual hostname to determine whether it is in the list of pass-though domains. Domains listed in the `passthroughDomain` list are unconditionally allowed.	355
`isProxyMode()`	Determines whether the Enforcer plugin should function as a proxy.	356
`log(int, String) or log(int, String, String) or log(String, int, String)`	Logs events of the given severity level to the `stderr` log. You can use any of the three formats available.	357
`replyNeedsAuth(XmlElement)`	Parses the XML reply document from the Policy Validator to determine whether further authentication is required before an `ENFORCER_ALLOW_ACTION` can be returned.	358
`setUserDataCharacterSet(String)`	Changes the default character set used to encode identity data.	359
`toString()`	Converts the value of the current object into its equivalent string representation.	360
`UrlEncodeBadChars(String)`	URL encode dangerous characters in a string, for protection against cross-site scripting (XSS) attacks.	361

Table 43 Enforcer Class Summary (cont'd)

Method Name	Description	Page
`XmlAppendMultiResource(XmlElement, String)`	Appends the given resource to multiple resource property list in the given XML query document.	362
`XmlAppendMultiResources(XmlElement, String[])`	Append a number of multi-resource resources to the given node.	363
`XmlDocInit(String)`	Creates an XML document, and a root node for that document with the specified tag.	364
`XmlDocParseFile(String)`	Creates an XML document by parsing the contents of the given file.	365
`XmlDocParseMem(byte [])`	Creates an XML document by parsing the contents of a memory buffer.	366
`XmlGetMultiResourceList(XmlElement)`	Returns the first multiple resource list under the given node.	367
`XmlQueryInit(String, String)`	Creates a new XML document named `PolicyValidatorQuery`, adding the given parameters as property values.	368
`XmlQuerySend(XmlElement)`	Converts the given XML node into an XML string and sends the query to the Policy Validator. It then waits for and parses the reply, returning the action that the Enforcer plugin should take.	369

Unimplemented methods

The following methods are currently unimplemented.

- `closeAll()`
- `getLoggingName`

Enforcer(String, String, String, String, boolean, int)

Description

Constructor. Creates a new Enforcer object. The Enforcer object is used to parse URLs for dangerous characters, determine if authorization is needed, query the Policy Validator, and obtain the response. It can process multiple authorization requests and it is safe to use in multi-threaded environments.

Syntax

```
Enforcer(String saConfigFileName,
            String enforcerConfigFileName,
            String loggingName,
            String enforcerTypeName,
            boolean initLogging,
            int debug_level)
```

Parameters

- `saConfigFileName` — The name of the Select Access configuration file to be read.
- `enforcerConfigFileName` — The name of the bootstrap configuration file containing the startup and functional parameters for this Enforcer.
- `loggingName` — The component name for this Enforcer object, which is used to identify those messages it has added to the log.
- `enforcerTypeName` — The name of the Enforcer application.
- `initLogging` — A boolean which indicates whether or not logging for this Enforcer object has been initialized.
- `debug_level` — The debug level.

Return Values

The Enforcer object, or an error, if unable to construct the Enforcer object.

Example

The following sample determines whether a configuration file exists, and, if so, tries to use it to construct the Enforcer handle:

```
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);
}
```

clearUserDataCharacterSet()

Description	Clears the character set used by the Enforcer object. Because character set conversion should be handled by the transaction object, this method should always be called when a new Enforcer object is constructed.
Class	`Enforcer`
Syntax	`void clearUserDataCharacterSet()`
Parameters	None.
Return Values	None.

debugMsg(String)

Description Sends the given debugging message to the log.

Class `Enforcer`

Syntax `void debugMsg(String msg)`

Parameters

- `msg` — A brief message that describes the event.

Return Values None.

Remarks To specify the debug level, use the `debugMsg(int, String)` method.

Example The following code sample checks to see if the Enforcer handle is configured, and if not, it logs a debug message and attempts to configure it.

```
if (!m_enforcer.isConfigured()) {
   debugMsg("isAuthorized: attempting to configure enforcer");
   m_enforcer.configure();
}
```

See Also

- `debugMsg(int, String)`

debugMsg(int, String)

Description Sends the given debugging message to the log describing events of the specified debug level. You can configure logging to record messages of a given level only.

Class `Enforcer`

Syntax

```
void debugMsg(int level,
              String msg)
```

Parameters

- `level` — The debug level.
- `msg` — A brief message that describes the event.

Return Values None.

Example The following code sample checks to see if the Enforcer handle is configured, and if not, it logs a debug message and attempts to configure it.

```
if (!m_enforcer.isConfigured()) {
  debugMsg(DataStrings.DEBUG_LEVEL, "isAuthorized: attempting to
    configure enforcer");
  m_enforcer.configure();
}
```

See Also

- `debugMsg(String)`

errMsg(String)

Description Logs the given message with a severity level of `ERROR`.

Class `Enforcer`

Syntax `void errMsg(String msg)`

Parameters

- `msg` — A brief message that describes the error.

Return Values None.

Example The following code sample tests the response from the Policy Validator. If the Policy Validator returns an error action, an error is sent to the log.

```
ValidatorResponse vr = XmlQuerySend(q);
if (vr.getRetVal() == Enforcer.ENFORCER_ERROR_ACTION) {
  errMsg("configure: error fetching config from validator");
  vh.reset();
}
```

getAuthHint(XmlElement, StringBuffer)

Description Extracts the authentication hints, if any, out of a Policy Validator reply object. If only one hint is present in the reply, it is returned. If more than one hint is present, this method merges them into a list and returns the list.

Class `Enforcer`

Syntax

```
XmlElement getAuthHint(XmlElement reply, StringBuffer passwordHint)
```

Parameters

- `reply` — An XML pointer to the reply node.
- `passwordHint` — The buffer containing a list of the authorization hint(s).

Return Values

- If only one hint is present, a `<Property>` is returned with the hint value.
- If multiple hints are present, a `<PropertyList>` containing the hints is returned.

Example The following code sample gets authentication hints from a reply, and if hints exist, adds them to a form.

```
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);
}
```

getBasenameOfURL(String)

Description Extracts the basename of the given URL by terminating it prior to an anchor symbol (`#`) or `jsessionid` tag.

Class `Enforcer`

Syntax

```
static String GetBasenameOfURL(String url)
```

Parameters

- `url` — The original URL.

Return Values The basename of the given URL.

Remarks This method only removes non-basename components from the provided string. It does not attempt to resolve the URL, nor does it ensure that the URL is valid.

Example The following code sample sets the variable `m_path` to the basename of the `m_path_full` variable.

```
m_path_full = path;
m_path = Enforcer.GetBasenameOfURL(m_path_full);
```

getCookieDomainName()

Description Returns the name of the domain where the current cookie originated.

Class `Enforcer`

Syntax `String getCookieDomainName()`

Parameters None.

Return Values The name of the domain from where the cookie originated.

Example The following sample code test to see whether a request is allowed access and whether it is an authentication URL. If so, the code constructs the originating URL and sets an RFC2109-formatted cookie.

```
boolean isAllowed = allowedByValidator();
if (isAllowed) {
   sso_nonce = new StringBuffer("");
   if (isSSOAuthURL(full_url, sso_nonce)) {
      String r3 = constructSSOOrigURL(m_path, m_http_query);
      logDebug("MD-SSO redirect 3: " + r3);
      setCookie(DataStrings.AUTH_COOKIE_NAME, sso_nonce.toString(), "/
        ", m_enforcer.getCookieDomainName(), -1);
      return sendAcceptedPage(r3);
   }
   return allow();
}
```

getLastError()

Description Returns the details about the last Servlet Enforcer plugin function that failed.

Class `Enforcer`

Syntax

```
String getLastError()
```

Parameters None.

Return Values A description of the most recent Servlet Enforcer plugin error. If no error is found, returns `NULL`.

getMultiResourceResult(XmlElement)

Description Returns the results of a request for multiple resources.

Class `Enforcer`

Syntax

```
static Hashtable getMultiResourceResult(XmlElement response)
```

Parameters

- `response` — An XML pointer to the Policy Validator reply.

Return Values A list of paths to requested resources, and the corresponding action to be taken by the Servlet Enforcer plugin for each resource requested.

Remarks This method is part of the Policy API extension to the Java Enforcer API. For more information on the Policy API, see Chapter 7, Querying for Multiple Resources: the Policy API, in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.

Example The following code sample creates a validator query document, `q`, appends multiple resources to the request, sends the request to the Policy Validator, then gets the access results from the response.

```
  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;
```

See Also

- `XmlAppendMultiResource(XmlElement, String)` on page 362
- `XmlAppendMultiResources(XmlElement, String[])` on page 363
- Chapter 7, Querying for Multiple Resources: the Policy API, in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*

getName()

Description Returns the name of the current object.

Class `Enforcer`

Syntax `String getName()`

Parameters None.

Return Values The name of the current object.

Example The following code sample gets all cookies within a request and stores them in an array. It then tests each cookie to determine if it's name matches the authentication cookie name, and if so, it sets the `m_auth_cookie` variable to the value of the current cookie.

```
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();
      }
   }
}
```

getNewQueryID()

Description Returns a valid query ID.

Class `Enforcer`

Syntax
```
static synchronized String getNewQueryID()
```

Parameters None.

Return Values A new query ID value, returned as a string.

Remarks Using this method will ensure that every query has a unique query ID.

The `Enforcer` class sets a long integer called `m_queryId` to an initial value of `0`. When the `getNewQueryID()` method is called, it stringifies the current value of `m_queryId`, returns the string as a result of the method, then increases `m_queryId` by one.

Example The following example creates an XML document which has a root tag of `<PolicyValidatorQuery>` then populates it with a `<Property>` for query ID, service and path.

```
XmlElement xe = Enforcer.XmlDocInit(DataStrings.ENFORCER_QUERY);
xe.appendPropertyValue(DataStrings.ENFORCER_QUERYID,
  GetNewQueryID());
xe.appendPropertyValue(DataStrings.ENFORCER_SERVICE, service);
xe.appendPropertyValue(DataStrings.ENFORCER_PATH, path);
return xe;
```

getQueryLevel()

Description Gets the query level setting for the Enforcer object. The query level specifies how much information will go into the query.

Class `Enforcer`

Syntax `String getQueryLevel()`

Parameters None.

Return Values The current query level, which may be one of the following values:

- Minimal
- Regular
- Maximal

Example The following code sample text to see whether the query level is set to maximal, and if so, adds HTTP header list data to the query.

```
  if (m_enforcer.getQueryLevel().equals(DataStrings.QUERY_MAXIMAL)) {
    PropertyListElement headers =
m_query.appendNestedPropertyList(DataStrings.ENFORCER_HTTP_HEADER_LIST);
    Enumeration e = m_request.getHeaderNames();
    while (e.hasMoreElements()) {
      String key = (String)e.nextElement();
      String value = (String)m_request.getHeader(key);
      headers.appendPropertyValue(key, value);
    }
  }
```

getTransformedForm(string, XmlElement, Hashtable)

Description Loads and transforms an HTML form page.

Class `Enforcer`

Syntax

```
String getTransformedForm(String fileName,
                          XmlElement formSpec,
                          Hashtable extraValues)
```

Parameters

- `fileName` — The name of the file whose contents will be transformed.
- `formSpec` — An XML document that details a form page.
- `extraValues` — An array that can be used to store extra form substitutions.

Return Values Returns the transformed HTML form page.

Example The following code sample gets authentication hints from a reply, and if hints exist, loads and transforms an HTML form page and adds the authentication hints to it.

```
m_auth_hint = m_enforcer.getAuthHint(m_reply, m_pass_hint);
if (m_auth_hint != null) {
   m_form = m_enforcer.getTransformedForm(vh, m_auth_hint, null);
}
```

getUserDataCharacterSet()

Description Returns the character set that is currently used to encode identity data.

Class `Enforcer`

Syntax

```
String getUserDataCharacterSet()
```

Parameters None.

Return Values The name of the character set used to encode identity data.

getValidatorHandle()

Description Returns a pointer to the Policy Validator.

Class `Enforcer`

Syntax `ValidatorHandle getValidatorHandle()`

Parameters None.

Return Values A pointer to the `ValidatorHandle` object.

Example The following code sample sets the variable `vh` to point to the current Policy Validator, then tests to see if the variable has a value of `NULL`.

```
ValidatorHandle vh = m_enforcer.getValidatorHandle();
if (vh == null) {
   return null;
}
```

getVersion()

Description Returns the version of the current Enforcer plugin application.

Class `Enforcer`

Syntax `static String getVersion()`

Parameters None.

Return Values The version of the Enforcer plugin.

Example The following code sample initializes an XML query document and appends Enforcer configuration data, including the Enforcer plugin version.

```
XmlElement xe = Enforcer.XmlDocInit(DataStrings.ENFORCER_QUERY);
xe.appendPropertyValue(DataStrings.ENFORCER_QUERYID,
  GetNewQueryID());
xe.appendPropertyValue(DataStrings.ENFORCER_CONFIG_REQUEST,
  DataStrings.TRUE);
xe.appendPropertyValue(DataStrings.ENFORCER_CONFIG_ID, enforcerID);
xe.appendPropertyValue(DataStrings.ENFORCER_VERSION, getVersion());
return xe;
```

initializeLogging(String, XmlElement)

Description Initializes the Enforcer plugin logging system.

Class `Enforcer`

Syntax

```
static boolean initializeLogging(String enforcerName,
       XmlElement config)
```

Parameters

- `enforcerName` — Identifies the name of the Enforcer application that is logging the messages. If it is `NULL`, the default logging name, Select Access Java Enforcer plugin, will be used.
- `config` — A pointer to an XML node containing file logger configuration information. This method expects `config` to be in the form:

  ```
  <file name="/file name" maxSize="1000000"/>
  ```

  where:

  — `file name` is the name of the file to which messages will be logged.

  — `maxSize` is the maximum size the file can reach before it rolls over.

Return Values

- `true` — The logger was successfully initiated.
- `false` — Initialization of the logger failed.

Remarks If the `maxSize` attribute of `config` is missing or is not greater than zero, the log is not rolled over.

Messages in the system log (syslog in UNIX; Event Log in Windows) will contain `enforcerName`.

Example The following code sample checks the configuration file to get log configuration information. If log configuration information exists, the Enforcer plugin logging system is initialized.

```
XmlElement logConfig = (XmlElement)r.getChildElement
     (DataStrings.LOG_TAG);
if (logConfig != null) {
  initializeLogging(enforcerName, logConfig);
}
```

isConfigured()

Description Determines whether or not the current object has been successfully configured.

Class `Enforcer`

Syntax `boolean isConfigured()`

Parameters None.

Return Values

- `true` — The object has been configured.
- `false` — The object has not been configured.

Example The following example test to see if the `m_validatorHandle` object has been configured, and if not, it calls the `configure()` method to configure it.

```
if (!m_validatorHandle.isConfigured()) {
     configure();
}
```

See Also

- `configure()`

isDisableCaching()

Description Determines whether the Servlet Enforcer plugin is configured to disable caching of Select Access-protected web pages.

Class `Enforcer`

Syntax

```
boolean isDisableCaching()
```

Parameters None.

Return Values

- `true` — The Servlet Enforcer plugin is configured to disable caching of Select Access-protected web pages.
- `false` — The Servlet Enforcer plugin is not configured to disable caching of Select Access-protected web pages.

Example The following example test to see if the `m_enforcer` object has been configured to disable caching, and if so, it calls the `disableCaching()` method so that the current resource is not cached.

```
if (m_enforcer.isDisableCaching()) {
  disableCaching();
}
```

See Also

- `disableCaching()` on page 318

isDomainProtected(String)

Description Tests the given domain to determine whether it is in the multi-domain single sign-on (MD-SSO) protected sites list.

Class `Enforcer`

Syntax

```
boolean isDomainProtected(String domain)
```

Parameters

- `domain` — The domain to be checked.

Return Values

- `true` — The given domain matches one of the domains listed in the MD-SSO protected sites list.
- `false` — The given domain does not match one of the domains listed in the MD-SSO protected sites list.

Remarks All domains are compared without regard to case, since LDAP DN's are case-insensitive.

Example The following code sample test to see whether the value of `nonce` is non-`NULL` and whether `currentURL` is in the MDSSO protected-sites list, and if so, sets the `value` parameter to the value of `nonce`.

```
String value = NO_NONCE;
if (nonce != null && isDomainProtected(currentURL))
  value = nonce;
```

isEnableCookies()

Description Determines whether the use of nonces (secure cookies) is enabled in the Policy Validator.

Class `Enforcer`

Syntax `boolean isEnableCookies()`

Parameters None.

Return Values

- `true` — The use of cookies is enabled.
- `false` — The use of cookies is not enabled.

Example The following code sample initializes a new query and checks to see if the use of cookies is enabled. If so, a new property is added to the query which indicates that nonces are enabled.

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

isEvilURL(String)

Description Determines whether the URL path is safe.

Class `Enforcer`

Syntax `boolean isEvilURL(String url)`

Parameters

- `url` — The URL to be tested.

Return Values

- `true` — The given URL contains characters that could be unsafe.
- `false` — The given URL is safe.

Remarks The `WebTransaction.UnescapeUrl(String)` method should be executed on the given URL before calling this method.

Example The following code sample tests to see if the given URL is suspicious, and if so, logs the event, disables caching on the web client, and denies access to the resource.

```
m_path = path;
if (isEvilURL(m_path)) {
   log(Logger.LOG_LEVEL_WARNING, "rejecting suspicious url '" + m_path
     + "'");
   disableCaching();
   return deny();
}
```

isIgnoredFile(String)

Description Tests the given file to determine whether it is in the list of ignored filenames. Files listed in the `ignoredFile` list are unconditionally allowed.

Class `Enforcer`

Syntax `boolean isIgnoredFile(String filename)`

Parameters

- `filename` — The filename to be compared.

Return Values

- `true` — The given is on the `ignoredFile` list.
- `false` — The given URL is not on the `ignoredFile` list.

Example The following code sample tests to see if the given filename is to be allowed unconditionally, and if so, logs the event, and allows access to the resource.

```
m_filename = filename;
if (isIgnoredFile(m_filename)) {
   logDebug("Ignore: " + m_filename);
   return allow();
}
```

isLoginViaForm()

Description Determines whether the Enforcer plugin is configured to use a web form to perform login operations.

Class `Enforcer`

Syntax
```
public boolean isLoginViaForm()
```

Parameters None.

Return Values
- `true` — The Enforcer plugin uses web forms for login operations.
- `false` — The Enforcer plugin does not use web forms for login operations.

Example The following code sample determines if authentication hints are used and whether the Enforcer plugin is configured to perform login operations via a web form. If so, it sends an HTTP basic authentication page to the web client.

```
if (m_auth_hint != null) {
   if (m_pass_hint.toString().equals("true") &&
        !m_enforcer.isLoginViaForm()) {
      return sendBasicAuthPage(getAuthRealm());
   }
   return doDynamicForm(null, m_auth_hint, null, false);
}
```

isPassthroughDomain(String)

Description Tests the given virtual hostname to determine whether it is in the list of pass-though domains. Domains listed in the `passthroughDomain` list are unconditionally allowed.

Class `Enforcer`

Syntax
```
boolean isPassthroughDomain(String host)
```

Parameters
- `host` — The name of the domain host to be tested.

Return Values
- `true` — The virtual hostname matches one of the domains listed in the `passthroughDomain` list.
- `false` — The virtual hostname does not match any of the domains listed in the `passthroughDomain` list.

Remarks All domains are compared without regard to case, since LDAP DN's are case-insensitive.

Example The following code sample tests to see if the given hostname is to be allowed unconditionally, and if so, logs the event, and allows access to the resource.

```
m_hostname = hostname;
if (isPassthroughDomain(m_hostname)) {
   logDebug("unsecured_domain: " + m_hostname);
   return allow();
}
```

isProxyMode()

Description Determines whether the Enforcer plugin should function as a proxy.

Class `Enforcer`

Syntax

```
boolean isProxyMode()
```

Parameters None.

Return Values

- `true` — The Enforcer plugin should function as a proxy.
- `false` — The Enforcer plugin should not function as a proxy.

log(int, String) or log(int, String, String) or log(String, int, String)

Description Logs events of the given severity level to the `stderr` log. You can use any of the three formats available.

Class `Enforcer`

Syntax

```
void log(int level,
         String shortMessage)
```

Or

```
void log(int level,
         String shortMessage,
         String longMessage)
```

Or

```
void log(String channel,
         int level,
         String shortMessage)
```

Parameters

- `level` — The severity level of events to be logged. This parameter can have one of the following values:
  - — `ENFORCER_LOG_FATAL` — Fatal errors.
  - — `ENFORCER_LOG_ERROR` — Non-fatal errors.
  - — `ENFORCER_LOG_WARNING` — Errors worth noting.
  - — `ENFORCER_LOG_INFO` — Information only.
  - — `ENFORCER_LOG_DEBUG` — Debugging information.
- `shortMessage` — A brief message that describes the event.
- `longMessage` — The descriptive details of the event/message being logged.
- `channel` — The name of the Select Access stream for which the message is being logged.
- `application` — The name of the application for which the message is being logged.

Return Values None.

replyNeedsAuth(XmlElement)

Description Parses the XML reply document from the Policy Validator to determine whether further authentication is required before an `ENFORCER_ALLOW_ACTION` can be returned.

Class `Enforcer`

Syntax

```
boolean replyNeedsAuth(XmlElement reply)
```

Parameters

- `reply` — An XML pointer to the Policy Validator reply.

Return Values

- `true` — Further authentication is required.
- `false` — No authentication is required.

Example The following code sample tests the given XML reply document to determine whether further authentication is required.

```
if (m_enforcer.replyNeedsAuth(reply)) {
   return true;
}
```

setUserDataCharacterSet(String)

Description Changes the default character set used to encode identity data.

Class `Enforcer`

Syntax

```
void setUserDataCharacterSet(String charSet)
```

Parameters

- `charSet` — Specifies the new character set that will be used to encode identity data.

Return Values None.

toString()

Description Converts the value of the current object into its equivalent string representation.

Class `Enforcer`

Syntax

```
String toString()
```

Parameters None.

Return Values A printable string representation of the current object.

Remarks This method overrides the `toString` method in the `java.lang.Object` class. It is typically used to create debugging output.

UrlEncodeBadChars(String)

Description URL encode dangerous characters in a string, for protection against cross-site scripting (XSS) attacks.

Class `Enforcer`

Syntax `static String UrlEncodeBadChars(String buf)`

Parameters

- `buf` — The string to be encoded.

Return Values An XSS-safe version of the original string.

Example The following code sample unescapes the given property value, then removes dangerous characters before creating a new HTML Input tag.

```
try {
   value = UnescapeUrl(value);
   safe_name = Enforcer.UrlEncodeBadChars(name);
   safe_value = Enforcer.UrlEncodeBadChars(value);
   StringBuffer buf = new StringBuffer("");
   buf.append("<INPUT TYPE=");
   buf.append(type);
   buf.append(" NAME=\"");
   buf.append(nameValue);
   buf.append("\" VALUE=\"");
   buf.append(valueValue);
   buf.append(safe_value);
   buf.append(endTag);
   return buf.toString();
}
```

XmlAppendMultiResource(XmlElement, String)

Description Appends the given resource to multiple resource property list in the given XML query document.

Class `Enforcer`

Syntax

```
static void XmlAppendMultiResource(XmlElement node,
     String resource)
```

Parameters

- `node` — An XML pointer to the `MultiResource` property list element in the XML query under which to append the resource. If the given node does not already exist, it will be created.
- `resource` — The resource URL.

Return Values None.

Remarks This method is part of the Policy API extension to the Java Enforcer API. For more information on the Policy API, see Chapter 7, Querying for Multiple Resources: the Policy API, in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.

Example The following code sample creates a validator query document, `q`, appends multiple resources to the request, sends the request to the Policy Validator, then gets the access results from the response.

```
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;
```

See Also

- `getMultiResourceResult(XmlElement)` on page 339
- `XmlAppendMultiResources(XmlElement, String[])` on page 363
- Chapter 7, Querying for Multiple Resources: the Policy API, in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.

XmlAppendMultiResources(XmlElement, String[])

Description Append a number of multi-resource resources to the given node.

Class `Enforcer`

Syntax

```
static void XmlAppendMultiResources(XmlElement node,
       String[] resources)
```

Parameters

- `node` — An XML pointer to the MultiResource list property element in the XML query under which to append the resources. If the given node does not already exist, it will be created.
- `resources` — An array of resources to which access is being requested.

Return Values None.

Remarks This method is part of the Policy API extension to the Java Enforcer API. For more information on the Policy API, see Chapter 7, Querying for Multiple Resources: the Policy API, in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*.

Example The following code sample creates a validator query document, `q`, appends multiple resources to the request, sends the request to the Policy Validator, then gets the access results from the response.

```
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;
```

See Also

- `getMultiResourceResult(XmlElement)` on page 339
- `XmlAppendMultiResource(XmlElement, String)` on page 362
- Chapter 7, Querying for Multiple Resources: the Policy API, in the *HP OpenView Select Access 6.2 Developer's Tutorial Guide*

XmlDocInit(String)

Description Creates an XML document, and a root node for that document with the specified tag.

Class `Enforcer`

Syntax `static XmlElement XmlDocInit(String rootTag)`

Parameters

- `rootTag` — The tag of the root node of the newly-created XML document. If `NULL`, the root node of the document has the empty string as its tag.

Return Values The new XML document.

Remarks In practice, most programs will use the `XmlQueryInit(String, String)` method to create new XML documents and immediately populate them with some basic query data.

This method is called by `XmlQueryInit(String, String)`.

Example The following example creates an XML document which has a root tag of `<PolicyValidatorQuery>` then populates it with a `<Property>` for query ID, service and path.

```
XmlElement xe = Enforcer.XmlDocInit(DataStrings.ENFORCER_QUERY);
xe.appendPropertyValue(DataStrings.ENFORCER_QUERYID,
     GetNewQueryID());
xe.appendPropertyValue(DataStrings.ENFORCER_SERVICE, service);
xe.appendPropertyValue(DataStrings.ENFORCER_PATH, path);
return xe;
```

See Also `<Code word in text>.XmlQueryInit(String, String)`

XmlDocParseFile(String)

Description Creates an XML document by parsing the contents of the given file.

Class `Enforcer`

Syntax

```
static XmlElement XmlDocParseFile(String fname)
```

Parameters

- `fname` — The file to be parsed. Must be non-`NULL`.

Return Values The root document node, or NULL if the function was unable to find or parse the file.

Remarks If the file being parsed contains improperly structured XML, this method will parse as much of the file as possible and return a properly formatted XML document.

XmlDocParseMem(byte [])

Description Creates an XML document by parsing the contents of a memory buffer.

Class `Enforcer`

Syntax

```
static XmlElement XmlDocParseMem(byte [] buf)
```

Parameters

- `buf` — A pre-formatted string and its length (not including the trailing `NULL` byte) must be non-`NULL`.

Return Values The root document node, or `NULL` if the function was unable to parse the block of memory.

Remarks If the memory buffer being parsed contains improperly-structured XML, this method will parse as much of the memory as possible, and return a properly formatted document.

XmlGetMultiResourceList(XmlElement)

Description Returns the first multiple resource list under the given node.

Class `Enforcer`

Syntax `static PropertyListElement XmlGetMultiResourceList(XmlElement node)`

Parameters

- `node` — An `XmlElement` that represents the parent node of the desired `multiResourceList`.

Return Values The first `multiResourceList` that is a child of the given node, or `NULL` if `node` contains no `multiResourceList`.

Example The following code sample tries to find the multiple resource list in the Policy Validator reply and, if one exists, counts the number of child nodes belonging to the list. It then cycles through each node testing to see if it is a resource property list. If so, the method extracts the resource path and the corresponding action to be taken by the Enforcer plugin and adds it to a hashtable.

```
  PropertyListElement mlist =
Enforcer.XmlGetMultiResourceList(response);
  Hashtable ht = new Hashtable();
  int c = mlist.getChildNodeCount();
  Node n = null;
  for (int i=0;i<c;i++) {
    n = mlist.getChildNodeAt(i);
    Property xe = (Property)n;
    if (xe.isPropertyList() && xe.getName().compareToIgnoreCase
        ("resource") == 0){
      String path = ((PropertyListElement)xe).
        getChildStringValue("path");
      String action = ((PropertyListElement)xe).
        getChildStringValue("action");
      if (path != null && action != null) {
        ht.put(path,action);
      }
    }
  }
```

XmlQueryInit(String, String)

Description Creates a new XML document named `PolicyValidatorQuery`, adding the given parameters as property values.

A unique query ID value is also automatically generated and added.

Class `Enforcer`

Syntax

```
static XmlElement XmlQueryInit(String service,
                               String path)
```

Parameters

- `service` — The name of the service for which access is being requested. For example, `http://mymenu.foodcompany.com:8070`.
- `path` — The path to the resource for which access is being requested. For example, `/index.html`.

Return Values The newly-created XML document. If any of the arguments are `NULL`, this function returns `NULL`.

Example The following code sample initializes an XML query and appends several additional properties to it.

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

See Also

- `XmlDocInit(String)` on page 364
- `.appendPropertyValue(String, String)` on page 398

XmlQuerySend(XmlElement)

Description Converts the given XML node into an XML string and sends the query to the Policy Validator. It then waits for and parses the reply, returning the action that the Enforcer plugin should take.

Class `Enforcer`

Syntax `ValidatorResponse XmlQuerySend(XmlElement query)`

Parameters

- `query` — The XML query document that has been constructed by the identity.

Return Values An element of the `EnforcerAction` enumeration indicating the Enforcer action to be taken. The returned value can be one of:

- `ENFORCER_ALLOW_ACTION` — Access is granted.
- `ENFORCER_DENY_ACTION` — Access is denied.
- `ENFORCER_ERROR_ACTION` — Access could not be granted due to an error during processing. If any of the arguments are `NULL`, this value is returned.
- `ENFORCER_USER_DEFINED` — Signals a site-specific request for further information. Typically used by custom decision criteria and deciders to signal that special processing is required. If this value is returned, it is the caller's responsibility to do whatever further work is necessary.

Example The following code sample sends the XML query document `q` to the Policy Validator for assessment, and if the response is an error, it logs an error message.

```
ValidatorResponse vr = XmlQuerySend(q);
if (vr.getRetVal() == Enforcer.ENFORCER_ERROR_ACTION) {
  errMsg("configure: error fetching config from validator");
  vh.reset();
}
```

11 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.

Two Policy builder plugins can be created using this API:

- Policy Builder authentication plugin: The authentication plugin allows you to create a Java-based property dialog box that is added to the Policy Builder and used to configure a new authentication service.

  Every authentication plugin must have a corresponding Authenticator plugin on the Policy Validator.

- Rule Builder decision-point plugin: Decision Point plugins are added to the Rule Builder utility in the Policy Builder. They define certain criteria which is evaluated by the decider plugin. Once the Decider plugin determines whether or not the conditions in the Decision Point plugin have been met, the Policy Validator can proceed to evaluate the next node in the rule tree, eventually either allowing or denying access.

  Every Decision Point plugin must have a corresponding Decider plugin on the Policy Validator.

The Policy Builder API consists of a single class:

Table 44 Class Overview

Class Name	Description	Page
`RuleComponentPanel`	The core of the Policy Builder API is the `RuleComponentPanel` class. This class is essentially a standard Java `JPanel` class that also encapsulates the Policy Store communication.	372

RuleComponentPanel

The core of the Policy Builder API is the `RuleComponentPanel` class. This class is essentially a standard Java `JPanel` class that also encapsulates the Policy Store communication.

The `RuleComponentPanel` class extracts the appropriate XML property document for your plugin and stores it in the inherited variable, `m_properties`. When your property editor exits normally, the Policy Builder API stores the `m_properties` variable back in the Policy Store.

Class Summary

Table 45 summarizes the member methods of this class. These methods are described in detail in the sections that follow.

Table 45 Enforcer Class Summary

Method Name	Description	Page
Construction		
`RuleComponentPanel()`	Constructor. Subclasses typically construct the GUI components used in a property editor dialog here.	374
Member methods		
`displayErrMsg(String)`	Displays a message box with the given message describing an error for the identity.	375
`displayErrMsg(String, int, int)`	Displays a message box with the given button predefined button configuration and icon image, and containing the given message describing an error for the identity.	376
`endConfigurator()`	Closes the dialog box.	377
`getMainFrame()`	Returns a handle to the Policy Builder JFrame.	378
`getPlainTextDataDescription()`	Supplies the Rule Builder with a plain-text description of the Decision Point plugin. This description is displayed in the bottom pane of the Rule Builder whenever its decision point is added to a rule.	379
`getProperties()`	Returns the XML properties for the given node.	380
`getRevisionInfo()`	Retrieves the name and number of the current Policy Builder plugin version.	381
`initialize()`	Initializes the graphical interface based on the `component.xml` configuration information.	382

Table 45 Enforcer Class Summary (cont'd)

Method Name	Description	Page
`resetProperties()`	Returns the property settings to their default settings.	383
`resetProperties(Property)`	Returns the given property to its default setting.	384
`setProperties(Property)`	Sets the properties for a decision point. Used as a callback by the Policy Builder framework to set the XML properties for a decision point.	385
`setProperties(Property, boolean)`	Used as a callback by the Policy Builder framework to set the XML properties for a decision point.	386
`showConfigurator(JFrame, String, boolean)`	Loads and displays the property editor dialog for a decision point. It should be called only if the decider has more than one panel (that is, if it is a tabbed dialog box).	387

RuleComponentPanel()

Description	Constructor. Subclasses typically construct the GUI components used in a property editor dialog here.
Class	`RuleComponentPanel`
Syntax	`RuleComponentPanel()`
Parameters	None.
Return Values	None.

displayErrMsg(String)

Description Displays a message box with the given message describing an error for the identity.

Class `RuleComponentPanel`

Syntax

```
void displayErrMsg(String msg)
```

Parameters

- `msg` — A brief message that describes the error.

Return Values None.

Example The following code sample displays the error message `ERR_MSG1`.

```
displayErrMsg(ERR_MSG1);
```

See Also

- `displayErrMsg(String, int, int)` on page 376

displayErrMsg(String, int, int)

Description Displays a message box with the given button predefined button configuration and icon image, and containing the given message describing an error for the identity.

Class `RuleComponentPanel`

Syntax

```
void displayErrMsg(String msg,
                   int style,
                   int image)
```

Parameters

- `msg` — A brief message that describes the error.
- `style` — A predefined layout of push buttons. Possible values, defined in the `com.hp.ov.selectaccess.widgets.MessageBox` package, include:
  - `MB_ABORTRETRYIGNORE`
  - `MB_OK`
  - `MB_OKCANCEL`
  - `MB_RETRYCANCEL`
  - `MB_YESNO`
  - `MB_YESNOCANCEL`
  - `MB_CANCEL`
  - `MB_CLOSE`
- `image` — An icon identifier. Possible values, defined in the `com.hp.ov.selectaccess.widgets.MessageBox` package, include:
  - `MB_ICONINFORMATION`
  - `MB_ICONEXCLAMATION`
  - `MB_ICONQUESTION`

Return Values None.

Example The following code sample displays the error message `ERR_MSG1`.

```
displayErrMsg("string to display", MessageBox.MB_OK,
  MessageBox.MB_ICONINFORMATION);
```

See Also

- `displayErrMsg(String)` on page 375

endConfigurator()

Description Closes the dialog box.

Class `RuleComponentPanel`

Syntax `void endConfigurator()`

Parameters None.

Return Values None.

Remarks

- Declared in the `RuleComponentConfigurator` interface.
- Any special cleanup should happen here. In a typical case, a subclass would do nothing here.

getMainFrame()

Description Returns a handle to the Policy Builder JFrame.

Class `RuleComponentPanel`

Syntax

```
MainFrame getMainFrame()
```

Parameters None.

Return Values A handle to the Policy Builder JFrame.

Example The following code sample initializes an identity source combo box.

```
Vector sourceNames = getMainFrame().getMainService().
  getSubjectSourceNames();
```

getPlainTextDataDescription()

Description Supplies the Rule Builder with a plain-text description of the Decision Point plugin. This description is displayed in the bottom pane of the Rule Builder whenever its decision point is added to a rule.

Class `RuleComponentPanel`

Syntax `String getPlainTextDataDescription()`

Parameters None.

Return Values The description string.

Remarks

- Declared in the `RuleComponentConfigurator` interface.
- Subclasses must override this to provide a meaningful string.

Example The following code is a sample subclass implementation of this method:

```
public String getPlainTextDataDescription() {
   StringBuffer outBuf = new StringBuffer(DataStrings.DASH_SPACE);
   if (!parseFilterExpression(m_properties, 0, outBuf))
      return(DataStrings.INVALID_DATA);
   else {
      return outBuf.toString();
   }
}
```

getProperties()

Description	Returns the XML properties for the given node.
Class	`RuleComponentPanel`
Syntax	`Property getProperties()`
Parameters	None.
Return Values	The XML properties for the given node.

getRevisionInfo()

Description	Retrieves the name and number of the current Policy Builder plugin version.
Class	`RuleComponentPanel`
Syntax	`Vector getRevisionInfo()`
Parameters	None.
Return Values	A vector which contains the version name and number.

initialize()

Description Initializes the graphical interface based on the `component.xml` configuration information.

Class `RuleComponentPanel`

Syntax `boolean initialize()`

Parameters None.

Return Values

- `true`

Remarks You should override this method to process the XML decision point data and set the state of the graphical interface to reflect the configuration.

Example The following code is a sample implementation of the `initialize()` method.

```
protected boolean initialize(){
    try {
        m_filterRowCount = 0;
        for (int i = 0; i < m_filterRows.length; i++) {
            FilterRow row = m_filterRows[i];
            row.setVisible(false);
        }
       buildLoadableAttrs();
       // Initialize the comboboxes with available attribute types.
       if (!parseFilterExpression(m_properties, 0, null)) {
           m_filterRowCount = 0;
           displayErrMsg("><Invalid logical or comparison expression.");
       }
       // Display the filter rows.
       m_firstRowLabel.setVisible(m_filterRowCount > 0);
       for (int i = 0; i < m_filterRows.length; i++) {
           FilterRow row = m_filterRows[i];
           row.setVisible(i < m_filterRowCount);
       }
       // Enable/disable the buttons.
       m_buttonPanel.setEnabled(ADD_BUTTON_TEXT, m_filterRowCount <
           m_filterRows.length);
       m_buttonPanel.setEnabled(REMOVE_BUTTON_TEXT, false);
   } catch (Throwable ex) {
       getMainFrame().handleException(getParentDialog(),
getClass().getName(), ex);
   }
}
```

resetProperties()

Description	Returns the property settings to their default settings.
Class	`RuleComponentPanel`
Syntax	`void resetProperties()`
Parameters	None.
Return Values	None.
See Also	• `resetProperties(Property)` on page 384

resetProperties(Property)

Description Returns the given property to its default setting.

Class `RuleComponentPanel`

Syntax `void resetProperties(Property properties)`

Parameters

- `properties` — A pointer to an XML node containing the property settings.

Return Values None.

See Also

- `resetProperties()` on page 383

setProperties(Property)

Description Sets the properties for a decision point. Used as a callback by the Policy Builder framework to set the XML properties for a decision point.

Class `RuleComponentPanel`

Syntax `void setProperties(Property newProperties)`

Parameters

- `newProperties` — A pointer to the property list which contains the new properties for the decision point.

Return Values None.

Remarks Declared in the `RuleComponentConfigurator` interface.

Example The following code is a sample subclass implementation of this method:

```
public void setProperties(Property properties) {
     m_properties = properties;
     m_originalProperties = (Property)properties.cloneNode(true);
     m_p13nDisabled = Boolean.valueOf(m_properties.getAttribute
        (RuleComponentData.PERSONALIZATION_OFF)).booleanValue();
     m_authPanel = new AuthPanel();
     m_p13nPanel = new PersonalizationPanel();
     m_authPanel.setProperties(m_properties, false);
     if (!m_p13nDisabled)
       m_p13nPanel.setProperties(m_properties, false);
   }
```

See Also

- `setProperties(Property, boolean)` on page 386

setProperties(Property, boolean)

Description Used as a callback by the Policy Builder framework to set the XML properties for a decision point.

Class `RuleComponentPanel`

Syntax

```
void setProperties(Property properties,
                   boolean setOriginal)
```

Parameters

- `properties` — A pointer to an XML node containing the new property settings.
- `setOrigninal` — Indicates whether or not the original properties for this decision point will be over-written by `properties`.
  - `true` or `NULL` — The original properties will be over-written.
  - `false` — The original properties will not be over-written.

Return Values None.

Example The following code sample checks to see if personalization is disabled, and, if not, configures the dialog to display the personalization properties.

```
if (!m_p13nDisabled)
  m_p13nPanel.setProperties(m_properties, false);
}
```

See Also

- `setProperties(Property)` on page 385

showConfigurator(JFrame, String, boolean)

Description Loads and displays the property editor dialog for a decision point. It should be called only if the decider has more than one panel (that is, if it is a tabbed dialog box).

Class `RuleComponentPanel`

Syntax

```
int showConfigurator(JFrame frame,
                     String title,
                     boolean modal)
```

Parameters

- `frame` — A pointer to the frame in which the properties are to be displayed.
- `title` — The text to display in the dialog's title bar.
- `modal` — Indicates whether or not this dialog is modal:
  - `true` — Indicates that the dialog is modal.
  - `false` — Indicates that the dialog allows other windows to be active at the same time.

Return Values This method is called automatically by the plugin framework for subclasses of `RuleComponentPanel`.

Remarks

- Declared in the `RuleComponentConfigurator` interface.
- In order to use this method, a property editor screen should be derived from the `RuleComponentPanel` class.

Example The following code is a sample subclass implementation of this method:

```
public int showConfigurator(JFrame frame, String title, boolean modal) {
   m_frame = frame;
      if(frame instanceof MainFrame)
        m_mainFrame = (MainFrame)frame;
      else if(frame instanceof AppFrame)
        m_mainFrame = ((AppFrame)frame).getMainFrame();
   m_dlg = new BaseDialog(frame, "><Authentication Properties", modal);
   m_authPanel.initialize(m_mainFrame);
   if (m_p13nDisabled || m_mainFrame.getClientConfigData().
        getIsDelegatedAdministrator()) {
      m_dlg.addPanel(m_authPanel);
   }
   else {
      m_p13nPanel.initialize(m_mainFrame);
      BasePanel[] panels = {
        m_authPanel,
        m_p13nPanel
      };
      m_dlg.addPanels(panels);
   }
   int nRetVal = m_dlg.showDialog();
   if (nRetVal != BaseDialog.IDOK)
      resetProperties();
   return(nRetVal);
}
```

12 Java XML Manipulation API

The Enforcer plugin communicates with the Policy Validator using XML documents. The Enforcer API includes three classes that allow you to manipulate XML. This chapter provides a reference for methods in these classes in Java.

Understanding XML Elements

The XML used by Select Access is extremely simple. An XML tree in Select Access can be composed of only two type of elements:

- Property: A property is a name/value pair. That is, 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>
  ```

- Property list: A property list is an element which contains a list of zero or more properties, as well as zero or more nested property lists. An example of a `PropertyListElement` is:

  ```
  <PROPERTYLIST>
     <PROPERTY NAME="color">red</PROPERTY>
  </PROPERTYLIST>
  ```

Classes in this API

The XML manipulation API contains a class for each of the element types (`PropertyElement` and `PropertyListElement`) as well as a parent class, `XmlElement`, which contains methods that can be used to handle nodes of either kind.

Table 46 Parent Class Overview

Class	Description	Page
`XmlElement`	This class extends the class `ElementNode`. The `XmlElement` class contains methods for creating and manipulating XML. Instances of this class represent the XML tags in the XML documents used by Select Access. The XML tags are mapped into instances of this class by the Java XML element factory.	page 391
`PropertyElement`	Extends the class `XmlElement` and implements the `Property` interface. The `PropertyElement` class contains methods for creating and manipulating XML elements with the `<Property>` tag.	page 424
`PropertyListElement`	Extends the class `XmlElement` and implements the `Property` interface. The `PropertyListElement` class contains methods for creating and manipulating XML elements with the `<PropertyList>` tag.	page 431

While several third-party APIs allow you to manipulate XML, they are overly complex for the needs of Select Access, and therefore possess unnecessary overhead. Because it has been designed to meet the specific needs of Select Access, the XML manipulation API dramatically improves performance over these third-party libraries. We therefore strongly recommend using these libraries when building Select Access plugins.

XmlElement

This class extends the class `ElementNode`. The `XmlElement` class contains methods for creating and manipulating XML. Instances of this class represent the XML tags in the XML documents used by Select Access. The XML tags are mapped into instances of this class by the Java XML element factory.

`XmlElement` is extended by the following classes:

Table 47 Class Summary

Class Name	Description	Page
`PropertyElement`	Extends the class `XmlElement` and implements the `Property` interface. The `PropertyElement` class contains methods for creating and manipulating XML elements with the `<Property>` tag.	page 424
`PropertyListElement`	Extends the class `XmlElement` and implements the `Property` interface. The `PropertyListElement` class contains methods for creating and manipulating XML elements with the `<PropertyList>` tag.	page 431

Methods

`XmlElement` contains the following methods. These methods are described in detail in the sections that follow.

Table 48 Method Summary

Method Name	Description	Page
`appendClone(Node)`	Clones the given node and appends it to the current property list in an XML document.	page 394
`appendElement(String)`	Appends an empty element with the given tag to the current property list in an XML document.	page 395
`appendElement(XmlElement)`	Appends the given XML node to the current property list in an XML document.	page 396
`appendNestedPropertyList (String)`	Creates a new `<PropertyList>` element with the given name and appends it to the current property list.	page 397
`appendPropertyValue(String, String)`	Creates a new `<Property>` element with the given name and value and makes it a child of the current property list.	page 398
`getChild(String)`	Gets the first child node with the given `NAME` attribute from the current property list.	page 399
`getChildElement(String)`	Gets the first child with the given element tag from the current property list.	page 400

Table 48 Method Summary (cont'd)

Method Name	Description	Page
`getChildElement(String, String, String)`	Gets the first child with the given element tag and attribute name and value from the current property list.	page 401
`getChildElements()`	Gets all children of the current property list.	page 402
`getChildElements(String)`	Gets all children with the given element tag from the current property list.	page 403
`getChildNodeAt(int)`	Gets the child node found at the given index within the current property list.	page 404
`getChildNodeCount()`	Counts the number of children a node has.	page 405
`getChildStringValue(String)`	Gets the string value associated with the first child node in a property list whose `NAME` attribute has the given value.	page 406
`getChildText(Element, boolean)`	Gets a text node that is a child of the given node.	page 407
`getName()`	Gets the value associated with a node's `NAME` attribute. If no name has ever been specified for the node, an empty string is returned.	page 408
`getNodeXml()`	Merges a node with all its children into a single node then returns the result as a text string.	page 409
`getText()`	Gets text that is associated with a node.	page 410
`getText(String)`	Gets text that is a child of a node with the given tag.	page 411
`newProperty(String)`	Creates a new empty `<Property>` element with the given `NAME` attribute within the current XML document, but does not make it a child of the current property list.	page 412
`newProperty(String, String)`	Creates a new `<Property>` element with the given `NAME` attribute and with the given value within the current XML document, but does not make it a child of the current property list.	page 413
`newPropertyList(String)`	Creates a new empty `<PropertyList>` element with the given `NAME` attribute within the current XML document, but does not make it a child of the current property list.	page 414
`removeNodefromParent()`	Removes the current child node from its parent node.	page 415

Table 48 Method Summary (cont'd)

Method Name	Description	Page
`setChildElement(String, String, String, Element)`	Sets the value of the child with the given element tag. If no child with the given element tag exists within the given parent node, one is created.	page 416
`setChildStringValue(String, String)`	Sets the string value of the child node with the given `NAME` attribute. If no such `<Property>` element exists, one is created.	page 417
`setName(String)`	Sets the `NAME` attribute for a node. If no `NAME` attribute is present, one is created.	page 418
`setText(String)`	Sets the given text string as a child of the current node.	page 419
`setText(String, String)`	Sets the given text string to an element of the given type and appends it as a child of the current node.	page 420
`toByteArray()`	Merges the children of the given node into a single node and fills a byte array with the resultant data.	page 421
`toString()`	Converts the value of the current object into its equivalent string representation.	page 422
`writeDocument(OutputStream)`	Writes the current XML document to an output stream.	page 423

appendClone(Node)

Description: Clones the given node and appends it to the current property list in an XML document.

Class: `XmlElement`

Syntax: `Node appendClone(Node node)`

Parameters:

- `node` — A pointer to the node to be cloned.

Return Values: The newly-created node.

appendElement(String)

Description Appends an empty element with the given tag to the current property list in an XML document.

Class `XmlElement`

Syntax
```
XmlElement appendElement(String tag)
```

Parameter:
- `tag` — The element type of the node to be appended.

Return Values The newly-created node.

See Also
- `appendElement(XmlElement)` on page 396

appendElement(XmlElement)

Description: Appends the given XML node to the current property list in an XML document.

Class: `XmlElement`

Syntax: `XmlElement appendElement(XmlElement node)`

Parameters:
- `node` — A pointer to the node to be appended.

Return Values: The newly-created node.

See Also:
- `appendElement(String)` on page 395

appendNestedPropertyList(String)

Description:
- Creates a new `<PropertyList>` element with the given name and appends it to the current property list.

Class: `XmlElement`

Syntax: `PropertyListElement appendNestedPropertyList(String name)`

Parameters:
- `name` — The value of the `NAME` attribute of the property list to be created.

Return Values: The newly-created element.

Example: The following code sample creates a new nested property list called "`http_header_list`", which it appends to the XML query document. It then appends each element of a header list as a property to the newly-created property list:

```
if (m_enforcer.getQueryLevel().equals(DataStrings.QUERY_MAXIMAL)) {
   PropertyListElement headers = m_query.appendNestedPropertyList
      (DataStrings.ENFORCER_HTTP_HEADER_LIST);
   Enumeration e = m_request.getHeaderNames();
   while (e.hasMoreElements()) {
      String key = (String)e.nextElement();
      String value = (String)m_request.getHeader(key);
      headers.appendPropertyValue(key, value);
   }
}
```

See Also:
- `appendElement(String)` on page 395
- `appendElement(XmlElement)` on page 396
- `.appendNestedProperty(String)` on page 434
- `.appendPropertyList(Document)` on page 435
- `.appendPropertyList(Element)` on page 437

appendPropertyValue(String, String)

Description: Creates a new `<Property>` element with the given name and value and makes it a child of the current property list.

Class: `XmlElement`

Syntax:
```
Property appendPropertyValue(String name,
                             String value)
```

Parameters:
- `name` — The value of the `NAME` attribute of the new `<Property>` element.
- `value` — The value of the new `<Property>` element.

Return Values: The newly-created element.

Example: The following code sample creates an XML document which has a root tag of `<PolicyValidatorQuery>` then populates it with a property for query ID, service and path.

```
XmlElement xe = Enforcer.XmlDocInit(DataStrings.ENFORCER_QUERY);
  xe.appendPropertyValue(DataStrings.ENFORCER_QUERYID,
    GetNewQueryID());
  xe.appendPropertyValue(DataStrings.ENFORCER_SERVICE, service);
  xe.appendPropertyValue(DataStrings.ENFORCER_PATH, path);
return xe;
```

See Also:
- `appendElement(String)` on page 395
- `appendElement(XmlElement)` on page 396

getChild(String)

Description: Gets the first child node with the given `NAME` attribute from the current property list.

Class: `XmlElement`

Syntax: `XmlElement getChild(String name)`

Parameters:

- `name` — The value of the `NAME` attribute of the child element being searched for. This parameter is case-sensitive.

Return Values: The child node, or `NULL` if the given child name is not found.

Example: The following code sample gets the first child node:

```
PropertyElement chal = (PropertyElement)xe.getChild(DataStrings.
  ENFORCER_NATIVE_CHAL);
if (chal != null) {
  return enforcer.getTransformedForm(xe, null);
}
```

See Also:

- `getChildElement(String)` on page 400
- `.getNestedProperties(String)` on page 441
- `.getPropertyValues(String)` on page 450

getChildElement(String)

Description: Gets the first child with the given element tag from the current property list.

Class: `XmlElement`

Syntax: `Element getChildElement(String tag)`

Parameters:

- `tag` — The element type of the child.

Return Values: The child node, or `NULL` if a child with the given element tag is not found.

Example: The following code sample checks the configuration file to get log configuration information. If log configuration information exists, the Enforcer plugin logging system is initialized.

```
XmlElement logConfig = (XmlElement)r.getChildElement
   (DataStrings.LOG_TAG);
if (logConfig != null) {
   initializeLogging(enforcerName, logConfig);
}
```

See Also:

- `getChildElement(String, String, String)` on page 401

getChildElement(String, String, String)

Description: Gets the first child with the given element tag and attribute name and value from the current property list.

Class: `XmlElement`

Syntax: `Element getChildElement(String tag, String attrName, String attrValue)`

Parameters:

- `tag` — The element type of the child. This can be either `<Property>` or `<PropertyList>`. If `tag` is `NULL`, this method will search for `attrName` and `attrValue` for either element type.
- `attrName` — The name of the attribute whose value will be changed.
- `attrValue` — The new or modified value of `attrName`.

Return Values: The child node, or `NULL` if the specified child is not found.

Example: The following code sample finds child element that matches the given parameters and if it exists, removes it.

```
Element child = getChildElement(tag, attrName, attrValue);
if (child != null)
   removeChild(child);
```

See Also:

- `getChildElements(String)` on page 403
- `.getNestedProperties(String)` on page 441
- `.getPropertyValues(String)` on page 450

getChildElements()

Description: Gets all children of the current property list.

Class: `XmlElement`

Syntax: `Vector getChildElements()`

Parameters: None.

Return Values: A list of the child elements of the current property list.

See Also:

- `getChildElements(String)` on page 403
- `.getNestedProperties()` on page 440
- `.getProperties()` on page 444)
- `.getPropertyValues()` on page 449

getChildElements(String)

Description: Gets all children with the given element tag from the current property list.

Class: `XmlElement`

Syntax:
```
Vector getChildElements(String tag)
```

Parameters:
- `tag` — The element type of the children to be returned.

Return Values: A list of the child elements of the current property list with an element type of `tag`.

See Also:
- `getChildElements()` on page 402
- `.getNestedProperties()` on page 440
- `.getProperties()` on page 444)
- `.getPropertyValues()` on page 449

getChildNodeAt(int)

Description: Gets the child node found at the given index within the current property list.

Class: `XmlElement`

Syntax: `Node getChildNodeAt(int index)`

Parameters:
- `index` — The position of the child in a property list. Must be a non-negative integer. The range for this value is `0` to (the *`number of children`*`-1`)

Return Values: The child node lodated at the specified position.

Example: The following code sample counts the number of child nodes belonging to the current element, then cycles through each node. It retrieves the `NAME` attribute of the node to see if the list is a multiple resource list.

```
int c = node.getChildNodeCount();
Node n = null;
for (int i=0;i<c;i++) {
   n = node.getChildNodeAt(i);
   if ((n instanceof XmlElement) == false)
      continue;
   Property xe = (Property) n;
   if (xe.isPropertyList() && xe.getName().compareToIgnoreCase
        (DataStrings.MULTI_RESOURCE_LIST) == 0)
      return (PropertyListElement)xe;
   }
return null;
```

See Also:
- `.getNestedProperty(int)` on page 442
- `.getPropertyValue(int)` on page 447

getChildNodeCount()

Description: Counts the number of children a node has.

Class: `XmlElement`

Syntax: `int getChildNodeCount()`

Parameters: None.

Return Values: The number of child nodes.

Remarks: If the current node is a `<Property>` element, this method will always return `0`, since by definition a property cannot contain any children.

Example: The following code sample counts the number of child nodes belonging to the current element, then cycles through each node. It retrieves the `NAME` attribute of the node to see if the list is a multiple resource list.

```
int c = node.getChildNodeCount();
Node n = null;
for (int i=0;i<c;i++) {
   n = node.getChildNodeAt(i);
   if ((n instanceof XmlElement) == false)
      continue;
   Property xe = (Property) n;
   if (xe.isPropertyList() && xe.getName().compareToIgnoreCase
      (DataStrings.MULTI_RESOURCE_LIST) == 0)
      return (PropertyListElement)xe;
   }
return null;
```

See Also:

- `.getNestedPropertyCount()` on page 425
- `.getPropertyValueCount()` on page 426
- `.getNestedPropertyCount()` on page 443
- `.getPropertyCount()` on page 445
- `.getPropertyValueCount()` on page 448

getChildStringValue(String)

Description: Gets the string value associated with the first child node in a property list whose `NAME` attribute has the given value.

Class: `XmlElement`

Syntax: `String getChildStringValue(String name)`

Parameters:
- `name` — The value of the `NAME` attribute of the child. This parameter is case sensitive.

Return Values: The value of the specified node returned as a string, or `NULL`, if the node has no string value or `name` is `NULL`.

Example: The following code sample tries to find the multiple resource list in the Policy Validator reply and, if one exists, counts the number of child nodes belonging to the list. It then cycles through each node testing to see if it is a resource property list. If so, the method extracts the resource path and the corresponding action to be taken by the Enforcer plugin and adds it to the hashtable.

```
  PropertyListElement mlist =
Enforcer.XmlGetMultiResourceList(response);
  Hashtable ht = new Hashtable();
  int c = mlist.getChildNodeCount();
  Node n = null;
  for (int i=0;i<c;i++) {
    n = mlist.getChildNodeAt(i);
    Property xe = (Property)n;
    if (xe.isPropertyList() && xe.getName().compareToIgnoreCase
        ("resource") == 0)
      String path = ((PropertyListElement)xe).
        getChildStringValue("path");
      String action = ((PropertyListElement)xe).getChildStringValue
        ("action");
      if (path != null && action != null) {
        ht.put(path,action);
      }
    }
  }
```

See Also:
- `.getStringValue()` on page 427

getChildText(Element, boolean)

Description: Gets a text node that is a child of the given node.

Class: `XmlElement`

Syntax:

```
Text getChildText(Element node,
                  boolean normalize)
```

Parameters:

- `node` — The parent node containing the nodes which will be returned.
- `normalize` — A boolean which indicates whether or not the children of the given node should be merged into a single node. If `NULL`, `false` is assumed.

Return Values:

- If `normalize` is set to `true`, the normalized text of all child nodes.
- If `normalize` is set to `false`, the text of a single child.
- `NULL`, if no text or no child is found.

See Also:

- `getText()` on page 410
- `getText(String)` on page 411

getName()

Description: Gets the value associated with a node's NAME attribute. If no name has ever been specified for the node, an empty string is returned.

Class: XmlElement

Syntax: String getName()

Parameters: None.

Return Values: The NAME attribute for the current node, or NULL, if no name has been specified for the node.

Example: The following code sample counts the number of child nodes belonging to the current element, then cycles through each node. It retrieves the NAME attribute of the node to see if the list is a multiple resource list.

```
int c = node.getChildNodeCount();
Node n = null;
for (int i=0;i<c;i++) {
   n = node.getChildNodeAt(i);
   if ((n instanceof XmlElement) == false)
      continue;
   Property xe = (Property) n;
   if (xe.isPropertyList() && xe.getName().compareToIgnoreCase
         (DataStrings.MULTI_RESOURCE_LIST) == 0)
      return (PropertyListElement)xe;
   }
return null;
```

getNodeXml()

Description: Merges a node with all its children into a single node then returns the result as a text string.

Class: `XmlElement`

Syntax: `String getNodeXml()`

Parameters: None.

Return Values: The XML node and its children as a single text string.

Example: The following code sample sends the XML query document to the Policy Validator and receives the reply. If the reply is not NULL and a certain debug level has been set, the reply is merged into a single node and sends the text to the log.

```
ValidatorResponse reply = null;
reply = m_enforcer.XmlQuerySend(query);
if (reply != null && debug > 1) {
   dbgmsg("reply is:\n" + reply.getXmlElement().getNodeXml());
}
```

See Also:

- `getText(String)` on page 411

getText()

Description: Gets text that is associated with a node.

Class: `XmlElement`

Syntax: `String getText()`

Parameters: None.

Return Values: The string value of the child node, or `NULL`, if no text is found.

Example: The following example steps through each property in a Personalization property list, decodes any URL encoded value, then sets the name/value pair for each personalization parameter.

```
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);
}
```

See Also:

- `getText()` on page 410

getText(String)

Description: Gets text that is a child of a node with the given tag.

Class: `XmlElement`

Syntax: `String getText(String tag)`

Parameters:

- `tag` — The element type of the child containing the text to be returned.

Return Values: The string value of the child node, or `NULL`, if no text is found.

Example: The following example steps through each property in a personalization property list, decodes any URL encoded value, then sets the name/value pair for each personalization parameter.

```
Enumeration e = p13nList.getPropertyValues();
while (e.hasMoreElements()) {
   XmlElement child = (XmlElement)e.nextElement();
   String name  = child.getName();
   String value = child.getText(DataStrings.PROPERTY_TAG);
   value = URLDecoder.decode(value, "UTF8");
   setPersonalization(name, value);
}
```

newProperty(String)

Description: Creates a new empty `<Property>` element with the given `NAME` attribute within the current XML document, but does not make it a child of the current property list.

Class: `XmlElement`

Syntax:
```
PropertyElement newProperty(String name)
```

Parameters:

- `name` — The value of the `NAME` attribute of the `<Property>` element to be created.

Return Values: The newly-created `<Property>` element.

See Also:

- `newProperty(String, String)` on page 413

newProperty(String, String)

Description: Creates a new `<Property>` element with the given `NAME` attribute and with the given value within the current XML document, but does not make it a child of the current property list.

Class: `XmlElement`

Syntax:

```
PropertyElement newProperty(String name, String value)
```

Parameters:

- `name` — The value of the `NAME` attribute of the `<Property>` element to be created.
- `value` — The value of the `<Property>` element to be created.

Return Values: The newly-created `<Property>` element.

See Also:

- `newProperty(String)` on page 412

newPropertyList(String)

Description: Creates a new empty `<PropertyList>` element with the given `NAME` attribute within the current XML document, but does not make it a child of the current property list.

Class: `XmlElement`

Syntax: `PropertyListElement newPropertyList(String name)`

Parameters:

- `name` — The value of the `NAME` attribute of the `<PropertyList>` element to be created.

Return Values: The newly-created `<PropertyList>` element.

Example: The following example creates a new property list with a `NAME` attribute of authentication_hint.

```
multiHint = reply.newPropertyList (DataStrings.
     ENFORCER_AUTHENTICATION_HINT);
multiHint.setChildStringValue(pageTag, DataStrings.
     ENFORCER_MULTIAUTH_FORM);
return enforcer.getTransformedForm(multiHint, null);
```

removeNodeFromParent()

Description: Removes the current child node from its parent node.

Class: `XmlElement`

Syntax: `void removeNodeFromParent()`

Parameters: None.

Return Values: None.

See Also:

- `PropertyListElement.removeNestedProperties(String)`
- `PropertyListElement.removeProperty(Property)`
- `PropertyListElement.removePropertyValues(String)`

setChildElement(String, String, String, Element)

Description: Sets the value of the child with the given element tag. If no child with the given element tag exists within the given parent node, one is created.

Class: `XmlElement`

Syntax:
```
Element setChildElement(String tag,
                        String attrName,
                        String attrValue,
                        Element node)
```

Parameters:

- `tag` — The element type of the child whose value is being set.
- `attrName` — The name of the attribute whose value will be changed.
- `attrValue` — The new or modified value of `attrName`.
- `node` — A pointer to the parent node to which the new child will be added, if it doesn't already exist.

Return Values: The newly-created `<Property>` or `<PropertyList>` element.

setChildStringValue(String, String)

Description: Sets the string value of the child node with the given `NAME` attribute. If no such `<Property>` element exists, one is created.

Class: `XmlElement`

Syntax:

```
void setChildStringValue(String name,
                         String val)
```

Parameters:

- `name` — The value of the `NAME` attribute of the child node.
- `val` — The new or modified value of the child `<Property>` element. If `NULL`, the child's value is set to the empty string.

Return Values: None.

Example: The following code sample creates a new property list then adds POST data to it.

```
PropertyListElement postDataList = query.newPropertyList
   (DataStrings.ENFORCER_POST_DATA_LIST);
query.appendChild(postDataList);
postDataList.setChildStringValue(DataStrings.ENFORCER_USER, user);
postDataList.setChildStringValue(DataStrings.ENFORCER_PASSWD,
   password);
```

setName(String)

Description: Sets the `NAME` attribute for a node. If no `NAME` attribute is present, one is created.

Class: `XmlElement`

Syntax: `void setName(String value)`

Parameters:

- `value` — The new value of the `NAME` attribute of the current node.

Return Values: None.

setText(String)

Description: Sets the given text string as a child of the current node.

Class: `XmlElement`

Syntax: `void setText(String value)`

Parameters:
- `value` — The new value of the text string.

Return Values: None.

setText(String, String)

Description: Sets the given text string to an element of the given type and appends it as a child of the current node.

Class: `XmlElement`

Syntax:
```
void setText(String tag,
             String value)
```

Parameters:
- `tag` — The element type of the node to which the given text will be set.
- `value` — The new value of the text string.

Return Values: None.

toByteArray()

Description: Merges the children of the given node into a single node and fills a byte array with the resultant data.

Class: `XmlElement`

Syntax: `byte[] toByteArray()`

Parameters: None.

Return Values: The new byte array.

toString()

Description: Converts the value of the current object into its equivalent string representation.

Class: `XmlElement`

Syntax: `String toString()`

Parameters: None.

Return Values: The string representation of the current object.

Remarks: Overrides the `toString` method in the `java.lang.Object` class.

writeDocument(OutputStream)

Description: Writes the current XML document to an output stream.

Class: `XmlElement`

Syntax: `void writeDocument(OutputStream out)`

Parameters:
- `out` — The stream into which the document will be written.

Return Values: None.

PropertyElement

Extends the class `XmlElement` and implements the `Property` interface. The `PropertyElement` class contains methods for creating and manipulating XML elements with the `<Property>` tag.

Instances of the `PropertyElement` class represent the `<PROPERTY>` tag in XML documents. The XML tags are mapped into instances of this class by the Java XML element factory.

Methods

This class contains the following methods:

Table 49 Method Summary

Method Name	Description	Page
`getNestedPropertyCount()`	Counts the number of nested `<PropertyList>` tags the current property or property list contains.	425
`getPropertyValueCount()`	Counts the number of nested `<Property>` tags the current property or property list contains.	426
`getStringValue()`	Gets a node's contained text. This value may be `NULL`.	427
`isPropertyList()`	Determines whether this node is actually a list of other properties, as opposed to a name/value pair.	428
`isPropertyValue()`	Determines whether this node is actually a name/value pair, as opposed to a list of other properties.	429
`setStringValue(String)`	Sets or changes the given node's contained text. `NULL` will clear the string value.	430

Unsupported methods

The following methods are unsupported implementations of methods from the `Property` interface:

- `appendNestedProperty(String)`
- `appendPropertyValue(String, String)`
- `getChild(String)`
- `getNestedProperties()`
- `getNestedProperties(String)`
- `getNestedProperty(int)`
- `getProperties()`
- `getPropertyIndex(Property)`
- `getPropertyCount()`
- `getPropertyValue(int)`
- `getPropertyValues()`
- `getPropertyValues(String)`
- `removeProperty(Property)`
- `removeAllProperty()`
- `removePropertyValues()`
- `removePropertyValues(String)`
- `removeNestedProperties()`
- `removeNestedProperties(String)`

getNestedPropertyCount()

Description: Counts the number of nested `<PropertyList>` tags the current property or property list contains.

Class: `PropertyElement`

Syntax: `int getNestPropertyCount()`

Parameters: None.

Return Values:

- `0` — `PropertyElement` objects can have no nested properties.

Remarks: Implements `Property.getNestedPropertyCount()`.

Example: The following code sample cycles through each nested property in a property list and removes it.

```
for (Enumeration i = getNestedPropertyCount())
   removeProperty(getNestedProperty(i));
```

See Also:

- `.getNestedPropertyCount()` on page 443

getPropertyValueCount()

Description: Counts the number of nested `<Property>` tags the current property or property list contains.

Class: `PropertyElement`

Syntax: `int getPropertyValueCount()`

Parameters: None.

Return Values:

- `0` — Property elements can contain no children.

Remarks: Implements `Property.getPropertyValueCount()`.

Example: The following code sample counts the number of nested `<Property>` tags found in a property list. It then iterates through them extracting names and values, and creates a string of attributes whoes properties have a value of "`true`".

```
int nPropSize = m_properties.getPropertyValueCount();
for (int i=0; i<nPropSize; i++) {
   Property property = m_properties.getPropertyValue(i);
   String strName = property.getName();
   String strVal = property.getStringValue();
   if (strVal.equals("true")) {
      strEngText.append(strName);
      if (i != nPropSize-1) {
         strEngText.append(", ");
      }
   }
}
```

See Also:

- `.getPropertyValueCount()` on page 448

getStringValue()

Description: Gets a node's contained text. This value may be `NULL`.

Class: `PropertyElement`

Syntax: `String getStringValue()`

Parameters: None.

Return Values:
- Returns the string value.
- If there is no string value, returns `NULL`.

Remarks: Implements `Property.getStringValue()`.

Example: The following code sample counts the number of nested `<Property>` tags found in a property list. It then iterates through them extracting names and values, and creates a string of attributes whoes properties have a value of "`true`".

```
int nPropSize = m_properties.getPropertyValueCount();
for (int i=0; i<nPropSize; i++) {
   Property property = m_properties.getPropertyValue(i);
   String strName = property.getName();
   String strVal = property.getStringValue();
   if (strVal.equals("true")) {
      strEngText.append(strName);
      if (i != nPropSize-1) {
         strEngText.append(", ");
      }
   }
}
```

isPropertyList()

Description: Determines whether this node is actually a list of other properties, as opposed to a name/value pair.

Class: `PropertyElement`

Syntax: `Boolean isPropertyList()`

Parameters: None.

Return Values: `False`.

Remarks: Implements `Property.isPropertyList()`.

See Also:

- `isPropertyValue()` on page 429
- `.isPropertyList()` on page 452
- `.isPropertyValue()` on page 453

isPropertyValue()

Description: Determines whether this node is actually a name/value pair, as opposed to a list of other properties.

Class: `PropertyElement`

Syntax: `boolean isPropertyValue()`

Parameters: None.

Return Values: `true`

Remarks: Implements `Property.isPropertyValue()`.

See Also:

- `isPropertyList()` on page 428
- `.isPropertyList()` on page 452
- `.isPropertyValue()` on page 453

setStringValue(String)

Description: Sets or changes the given node's contained text. `NULL` will clear the string value.

Class: `PropertyElement`

Syntax: `void setStringValue(String value)`

Parameters:

- `value` — The text that will be copied and appended to an existing string.

Return Values: None.

Remarks: Implements `Property.setStringValue(String)`.

PropertyListElement

Extends the class `XmlElement` and implements the `Property` interface. The `PropertyListElement` class contains methods for creating and manipulating XML elements with the `<PropertyList>` tag.

Instances of the `PropertyListElement` class represent the `<PROPERTY>` tag in XML documents. The XML tags are mapped into instances of this class by the Java XML element factory.

Methods

This class contains the following methods:

Table 50 Method Summary

Method Name	Description	Page
`appendNestedProperty(String)`	Creates a new `<PropertyList>` element with the given name and makes it a child of the current property list.	510
`appendPropertyList(Document)`	Creates a new empty and unnamed `<PropertyList>` element and appends it to an XML document.	511
`appendPropertyList(Document, String)`	Creates a new empty `<PropertyList>` element with the given `NAME` attribute and makes it a child of the given parent property list.	512
`appendPropertyList(Element)`	Creates a new empty and unnamed `<PropertyList>` element and makes it a child of the given parent property list.	513
`appendPropertyList(Element, String)`	Creates a new empty `<PropertyList>` element with the given `NAME` attribute and makes it a child of the given parent property list.	514
`getChild(String)`	Gets the first child node with the given `NAME` attribute from the current `<PropertyList>`.	515
`getNestedProperties()`	Gets all the `<PropertyList>` elements contained within the current property list.	517
`getNestedProperties(String)`	Gets the property list with the given `NAME` attribute contained within the current property list.	518
`getNestedProperty(int)`	Gets the property list found at the given index within the current property list.	519
`getNestedPropertyCount()`	Counts the number of nested property lists the current property list contains.	520

Table 50 Method Summary (cont'd)

Method Name	Description	Page
`getProperties()`	Gets all elements with either a `<Property>` tag or a `<PropertyList>` tag contained within the current property list.	521
`getPropertyCount()`	Counts the number of both `<Property>` and `<PropertyList>` tags contained within the current property list.	522
`getPropertyIndex(Property)`	Determines the position of the given property or property list within the current property list.	523
`getPropertyValue(int)`	Gets the property element found at the given index in the current property list.	524
`getPropertyValueCount()`	Counts the number of nested `<Property>` tags the current property list contains.	525
`getPropertyValues()`	Gets all elements with a `<Property>` tag contained within the current property list.	526
`getPropertyValues(String)`	Retrieves the `<Property>` tag with the given `NAME` attribute contained in the current property list.	527
`isPropertyList()`	Determines whether this node is actually a list of other properties, as opposed to a name/value pair.	529
`isPropertyValue()`	Determines whether this node is actually a name/value pair, as opposed to a list of other properties.	530
`removeAllProperty()`	Removes all properties from the current property list.	532
`removeNestedProperties()`	Removes all nested elements with a `<PropertyList>` tag from the current property list.	533
`removeNestedProperties (String)`	Removes the element with a `<PropertyList>` tag and the given `NAME` attribute from the current property list.	534
`removeProperty(Property)`	Removes the given element from the current property list.	535

Table 50 Method Summary (cont'd)

Method Name	Description	Page
`removePropertyValues()`	Removes all nested elements with a `<Property>` tag from the current property list.	536
`removePropertyValues(String)`	Removes the element with a `<Property>` tag and the given `NAME` attribute from the current property list.	537
`toString()`	Converts the value of the current object into its equivalent string representation.	538

The following are unsupported implementations of methods from the `Property` interface:

- `getStringValue`
- `setStringValue`

appendNestedProperty(String)

Description: Creates a new `<PropertyList>` element with the given name and makes it a child of the current property list.

Class: `PropertyListElement`

Syntax: `Property appendNestedProperty(String name)`

Parameters:

- `name` — The `NAME` attribute of the property list to be created. If `NULL`, an unnamed property list is added.

Return Values: The newly-created `<PropertyList>` element.

Remarks: Implements `Property.appendNestedProperty(String)`.

Example: The following code sample adds a new multi-resource property list and then appemnds resources to it.

```
mrList = node.appendNestedProperty(DataStrings.MULTI_RESOURCE_ LIST);

for (int i=0; i < resources.length;i++) {
   XmlElement res = mrList.appendNestedProperty(ENFORCER_RESOURCE);
   res.appendPropertyValue(DataStrings.ENFORCER_PATH, resources[i]);
}
```

See Also:

- `appendPropertyList(Document)` on page 435
- `appendPropertyList(Document, String)` on page 436
- `appendPropertyList(Element)` on page 437
- `appendPropertyList(Element, String)` on page 438

appendPropertyList(Document)

Description: Creates a new empty and unnamed `<PropertyList>` element and appends it to an XML document.

Class: `PropertyListElement`

Syntax: `PropertyListElement appendPropertyList(Document xmlDocument)`

Parameters:

- `xmlDocument` — A pointer to the XML document to which the new `<PropertyList>` element will be appended.

Return Values: The newly-created `<PropertyList>` element.

See Also:

- `appendNestedProperty(String)` on page 434
- `appendPropertyList(Document, String)` on page 436
- `appendPropertyList(Element)` on page 437
- `appendPropertyList(Element, String)` on page 438

appendPropertyList(Document, String)

Description: Creates a new empty `<PropertyList>` element with the given `NAME` attribute and makes it a child of the given parent property list.

Class: `PropertyListElement`

Syntax:
```
PropertyListElement appendPropertyList(Document xmlDocument,
                                       String name)
```

Parameters:
- `xmlDocument` — A pointer to the XML document to which the new `<PropertyList>` element will be appended.
- `name` — The `NAME` attribute of the property list to be created. If `NULL`, an unnamed property list is added.

Return Values: The newly-created `<PropertyList>` element.

See Also:
- `appendNestedProperty(String)` on page 434
- `appendPropertyList(Document)` on page 435
- `appendPropertyList(Element)` on page 437
- `appendPropertyList(Element, String)` on page 438

appendPropertyList(Element)

Description: Creates a new empty and unnamed `<PropertyList>` element and makes it a child of the given parent property list.

Class: `PropertyListElement`

Syntax:
```
PropertyListElement appendPropertyList(Element parent)
```

Parameters:
- `parent` — A pointer to a previously created property list to which the new `<PropertyList>` element will be appended.

Return Values: The newly-created `<PropertyList>` element.

See Also:
- `appendNestedProperty(String)` on page 434
- `appendPropertyList(Document)` on page 435
- `appendPropertyList(Document, String)` on page 436
- `appendPropertyList(Element, String)` on page 438

appendPropertyList(Element, String)

Description: Creates a new empty `<PropertyList>` element with the given `NAME` attribute and makes it a child of the given parent property list.

Class: `PropertyListElement`

Syntax: `PropertyListElement appendPropertyList(Element parent, String name)`

Parameters:

- `parent` — A pointer to a previously created property list to which the new `<PropertyList>` element will be appended.
- `name` — The `NAME` attribute of the property list to be created. If `NULL`, an unnamed property list is added.

Return Values: The newly-created `<PropertyList>` element.

See Also:

- `appendNestedProperty(String)` on page 434
- `appendPropertyList(Document)` on page 435
- `appendPropertyList(Document, String)` on page 436
- `appendPropertyList(Element)` on page 437

getChild(String)

Description: Gets the first child node with the given `NAME` attribute from the current `<PropertyList>`.

Class: `PropertyListElement`

Syntax: `XmlElement getChild(String name)`

Parameters:

- `name` — The value of the `NAME` attribute of the child being searched for. This parameter is case-sensitive.

Return Values: The child node, or `NULL` if the given child name is not found.

Example: The following code sample counts the number of children in a `<PropertyList>`, determines if one of these child nodes is a property list containing POST information, and if so, finds the child within that list which contains preserved port data.

```
num_children = xml.getChildNodeCount();
for (int i=0; i<num_children; i++) {
   cur_node = xml.getChildNodeAt(i);
   if (! (cur_node instanceof PropertyListElement))
      continue;
   XmlElement xe = (XmlElement) cur_node;
   cur_name = xe.getName();
   if (!cur_name.equals(DataStrings.ENFORCER_POST_DATA_LIST))
      continue;
   PropertyElement pe = (PropertyElement)xe.getChild
        (DataStrings.SA_PRESERVED_POST);
   if (pe == null)
      return null;
   return pe.getStringValue();
}
return null;
```

getNestedProperties()

Description: Gets all the `<PropertyList>` elements contained within the current property list.

Class: `PropertyListElement`

Syntax: `Property getNestedProperties()`

Parameters: None.

Return Values: All the elements in within the current property list with an element type of `<PropertyList>`.

Remarks: Implements `Property.getNestedProperties()`.

Example: The following code sample creates a list of all the nested properties, then removes each of them from the parent property list.

```
Vector toRemove = new Vector();
for (Enumeration ep = getNestedProperties(); ep.hasMoreElements(); )
  toRemove.addElement(ep.nextElement());
for (Enumeration ep = toRemove.elements(); ep.hasMoreElements(); )
  removeProperty((Property)ep.nextElement());
```

See Also:

- `getNestedProperties(String)` on page 441
- `getNestedProperty(int)` on page 442
- `getProperties()` on page 444

getNestedProperties(String)

Description: Gets the property list with the given `NAME` attribute contained within the current property list.

Class: `PropertyListElement`

Syntax: `Property getNestedProperties(String name)`

Parameters:

- `name` — The `NAME` attribute of the property list being searched for. This parameter is case-sensitive.

Return Values: The property list with a `NAME` attribute of `name`, if it exists.

Remarks: Implements `Property.getNestedProperties(String)`.

Example: The following code sample creates a list of all the nested properties with the `NAME` attribute of `resource`, then removes each of them from the parent property list.

```
  Vector toRemove = new Vector();
  for (Enumeration ep = getNestedProperties(resource);
ep.hasMoreElements(); )
    toRemove.addElement(ep.nextElement());
  for (Enumeration ep = toRemove.elements(); ep.hasMoreElements(); )
    removeProperty((Property)ep.nextElement());
```

See Also:

- `getNestedProperties()` on page 440
- `getNestedProperty(int)` on page 442
- `getProperties()` on page 444

getNestedProperty(int)

Description: Gets the property list found at the given index within the current property list.

Class: `PropertyLIstElement`

Syntax: `Property getNestedProperty(int position)`

Parameters:

- `position` — The position of the nested property list within the current property list. Must be a non-negative integer. The range for this value is `0` to (the *`number of children`*`-1`).

Return Values: The `<Property>` element located at the specified position.

Remarks: Implements `Property.getNestedProperty(int)`.

Example: The following code sample cycles through each nested property in a property list and removes it.

```
for (Enumeration i = getNestedPropertyCount())
   removeProperty(getNestedProperty(i));
```

See Also:

- `getNestedProperties()` on page 440
- `getNestedProperties(String)` on page 441
- `getProperties()` on page 444

getNestedPropertyCount()

Description: Counts the number of nested property lists the current property list contains.

Class: `PropertyListElement`

Syntax: `int getNestedPropertyCount()`

Parameters: None.

Return Values: Returns the number of nested property lists, or `NULL`, if unsuccessful.

Remarks: Implements `Property.getNestedPropertyCount()`.

Example: The following code sample cycles through each nested property in a property list and removes it.

```
for (Enumeration i = getNestedPropertyCount())
  removeProperty(getNestedProperty(i));
```

See Also:

- `getPropertyCount()` on page 445
- `.getNestedPropertyCount()` on page 425

getProperties()

Description: Gets all elements with either a `<Property>` tag or a `<PropertyList>` tag contained within the current property list.

Class: `PropertyListElement`

Syntax: `Enumeration getProperties()`

Parameters: None.

Return Values: An enumerated list of all the elements contained within the current property list.

Remarks: Implements `Property.getProperties()`.

Example: The following code sample gets all the properties and property list elements contained within the given property list and adds them to a vector called `childElements`.

```
Vector childElements = new Vector();
for (Enumeration ep = getProperties(); ep.hasMoreElements(); )
   childElements.addElement(ep.nextElement());
```

See Also:

- `getNestedProperties()` on page 440
- `getNestedProperties(String)` on page 441
- `getNestedProperty(int)` on page 442

getPropertyCount()

Description: Counts the number of both `<Property>` and `<PropertyList>` tags contained within the current property list.

Class: `PropertyListElement`

Syntax: `int getPropertyCount()`

Parameters: None.

Return Values: The number of `<Property>` and `<PropertyList>` tags contained within the current property list.

Remarks: Implements `Property.getNestedPropertyCount()`.

Example: The following code sample cycles through all the properties and property list elements contained within the given property list and adds them to a vector called `childElements`.

```
Vector childElements = new Vector();
for (Enumeration ep = getPropertyCount(); )
   childElements.addElement(ep.nextElement());
```

See Also:

- `getNestedPropertyCount()` on page 443
- `getPropertyValueCount()` on page 448
- `.getChildNodeCount()` on page 405

getPropertyIndex(Property)

Description: Determines the position of the given property or property list within the current property list.

Class: `PropertyListElement`

Syntax: `Property getPropertyIndex(Property property)`

Parameters:

- `property` — A pointer to the property or property list whose position you are trying to determine.

Return Values: The index value of the given property's position within the current property list. The range for this value is `0` to (the *`number of children`*`-1`)

If the given property does not exist in the current property list, a value of `-1` is returned.

Remarks: Implements `Property.getPropertyIndex(Property)`.

getPropertyValue(int)

Description: Gets the property element found at the given index in the current property list.

Class: `PropertyListElement`

Syntax: `Property getPropertyValue(int position)`

Parameters:

- `position` — The position in a property list of the nested property. Must be a non-negative integer. The range for this value is `0` to (the *`number of children`*`-1`).

Return Values: The `<Property>` element located at the specified position.

Remarks: Implements `Property.getPropertyValue(int)`.

Example: The following code sample counts the number of nested `<Property>` tags found in a property list. It then iterates through them extracting names and values, and creates a string of attributes whoes properties have a value of "`true`".

```
int nPropSize = m_properties.getPropertyValueCount();
for (int i=0; i<nPropSize; i++) {
   Property property = m_properties.getPropertyValue(i);
   String strName = property.getName();
   String strVal = property.getStringValue();
   if (strVal.equals("true")) {
      strEngText.append(strName);
      if (i != nPropSize-1) {
         strEngText.append(", ");
      }
   }
}
```

See Also:

- `getNestedProperty(int)` on page 442

getPropertyValueCount()

Description: Counts the number of nested `<Property>` tags the current property list contains.

Class: `PropertyListElement`

Syntax: `int getPropertyValueCount()`

Parameters: None.

Return Values: Returns the number of properties, or `NULL`, if unsuccessful.

Remarks: Implements `Property.getPropertyValueCount()`.

Example: The following code sample counts the number of nested `<Property>` tags found in a property list. It then iterates through them extracting names and values, and creates a string of attributes whoes properties have a value of "`true`".

```
int nPropSize = m_properties.getPropertyValueCount();
for (int i=0; i<nPropSize; i++) {
   Property property = m_properties.getPropertyValue(i);
   String strName = property.getName();
   String strVal = property.getStringValue();
   if (strVal.equals("true")) {
      strEngText.append(strName);
      if (i != nPropSize-1) {
         strEngText.append(", ");
      }
   }
}
```

See Also:

- `getNestedPropertyCount()` on page 443
- `getPropertyCount()` on page 445
- `.getPropertyValueCount()` on page 426
- `.getChildNodeCount()` on page 405

getPropertyValues()

Description: Gets all elements with a `<Property>` tag contained within the current property list.

Class: `PropertyListElement`

Syntax: `Enumeration getPropertyValues()`

Parameters: None.

Return Values: An enumerated list of all the elements with a `<Property>` tag contained within the current property list.

Remarks: Implements `Property.getPropertyValues()`.

See Also:

- `getProperties()` on page 444
- `getPropertyValue(int)` on page 447
- `getPropertyValues(String)` on page 450

getPropertyValues(String)

Description: Retrieves the `<Property>` tag with the given `NAME` attribute contained in the current property list.

Class: `PropertyListElement`

Syntax: `Property getPropertyValues(String name)`

Parameters:

- `name` — The value of the `NAME` attribute of the property being searched for. This parameter is case-sensitive.

Return Values: The `<Property>` with a `NAME` attribute of `name`, if it exists.

Remarks: Implements `Property.getPropertyValues(String)`.

Example: The following code sample retrieves the XML tree node that stores the current state of the component. If this node cannot be found, this method creates it.

```
if (m_properties.getPropertyValueCount() == 0)
{
  m_properties.appendPropertyValue("state", "true");
}
Enumeration allValues = m_properties.getPropertyValues("state");
return (Property)allValues.nextElement();
```

See Also:

- `getProperties()` on page 444
- `getPropertyValue(int)` on page 447
- `getPropertyValues()` on page 449

hasAttribute()

Description: Checks the current node to see if it has any attributes.

Class: `PropertyListElement`

Syntax: `boolean hasAttribute()`

Parameters: None.

Return Values:
- `false` — The property list has no attributes.

See Also:
- `.newProperty(String)` on page 412

isPropertyList()

Description: Determines whether this node is actually a list of other properties, as opposed to a name/value pair.

Class: `PropertyListElement`

Syntax: `boolean isPropertyList()`

Parameters: None.

Return Values: `true`

Remarks: Implements `Property.isPropertyList()`.

Example: The following code sample determines whether the given property in a reply is a property list and if it has a `NAME` attribute of `resource`. If so, it extracts the resource path and the action return by the Policy Validator.

```
Property xe = (Property)n;
if (xe.isPropertyList() && xe.getName().compareToIgnoreCase
     ("resource") == 0){
   String path = ((PropertyListElement)xe).getChildStringValue
     ("path");
   String action = ((PropertyListElement)xe).getChildStringValue
     ("action");
}
```

See Also:

- `isPropertyValue()` on page 453
- `.isPropertyList()` on page 428
- `.isPropertyValue()` on page 429

isPropertyValue()

Description: Determines whether this node is actually a name/value pair, as opposed to a list of other properties.

Class: `PropertyListElement`

Syntax: `boolean isPropertyValue()`

Parameters: None.

Return Values: `false`

Remarks: Implements `Property.isPropertyList()`.

See Also:

- `isPropertyList()` on page 452
- `.isPropertyList()` on page 428
- `.isPropertyValue()` on page 429

removeAllProperty()

Description: Removes all properties from the current property list.

Class: `PropertyListElement`

Syntax: `void removeAllProperty()`

Parameters: None.

Return Values: None.

Remarks: Implements `Property.removeAllProperty()`.

Example: The following code sample removes old properties to reflect the current status of the GUI.

```
m_properties.removeAllProperty();
```

See Also:

- `removeNestedProperties()` on page 455
- `removeNestedProperties(String)` on page 456
- `removePropertyValues()` on page 458

removeNestedProperties()

Description: Removes all nested elements with a `<PropertyList>` tag from the current property list.

Class: `PropertyListElement`

Syntax: `void removeNestedProperties()`

Parameters: None.

Return Values: None.

Remarks: Implements `Property.removeNestedProperties()`.

See Also:

- `removeAllProperty()` on page 454
- `removeNestedProperties(String)` on page 456
- `.removeNodeFromParent()` on page 415

removeNestedProperties(String)

Description: Removes the element with a `<PropertyList>` tag and the given `NAME` attribute from the current property list.

Class: `PropertyListElement`

Syntax: `void removeNestedProperties(String name)`

Parameters:
- `name` — The value of the `NAME` attribute of the property list you want to remove. This parameter is case-sensitive.

Return Values: None.

Remarks: Implements `Property.removeNestedProperties(String)`.

See Also:
- `removeNestedProperties()` on page 455
- `removePropertyValues()` on page 458
- `.removeNodeFromParent()` on page 415

removeProperty(Property)

Description: Removes the given element from the current property list.

Class: `PropertyListElement`

Syntax: `void removeProperty(Property property)`

Parameters:
- `property` — A pointer to the property or property list whose position you are trying to determine.

Return Values: None.

Remarks: Implements `Property.removeProperty(Property)`.

See Also:
- `removeAllProperty()` on page 454
- `removeNestedProperties()` on page 455
- `removeNestedProperties(String)` on page 456
- `removePropertyValues()` on page 458
- `.removeNodeFromParent()` on page 415

removePropertyValues()

Description: Removes all nested elements with a `<Property>` tag from the current property list.

Class: `PropertyListElement`

Syntax: `void removePropertyValues()`

Parameters: None.

Return Values: None.

Remarks: Implements `Property.removePropertyValues()`.

See Also:

- `removeAllProperty()` on page 454
- `removePropertyValues(String)` on page 459

removePropertyValues(String)

Description: Removes the element with a `<Property>` tag and the given `NAME` attribute from the current property list.

Class: `PropertyListElement`

Syntax: `void removePropertyValues(String name)`

Parameters:

- `name` — The value of the `NAME` attribute of the property you want to remove. This parameter is case-sensitive.

Return Values: None.

Remarks: Implements `Property.removePropertyValues(String)`.

See Also:

- `removeProperty(Property)` on page 457
- `removePropertyValues()` on page 458
- `.removeNodeFromParent()` on page 415

toString()

Description: Converts the value of the current object into its equivalent string representation.

Class: `PropertyListElement`

Syntax: `String toString()`

Parameters: None.

Return Values: The string representation of the current object.

Remarks: Overrides the `toString` method in the `java.lang.Object` class.

13 Variable Data Strings

A number of variables are used repeatedly throughout the Select Access code. These common variables are predefined in the `DataStrings.java` file. This chapter provides a reference for these string names and their values in Java.

Data Strings You Can Use

Table 51Java Data Strings

String Name	Value	Represents
ACCESSRULE_CLASS	com.hp.ov.selectaccess.rulebuilder.AccessRule	The package containing the AccessRule class.
AUTH_COOKIE_NAME	PolicyUser	The name of the authorization cookie.
AUTH_COOKIE_LOGOUT_VALUE	Logout	Used internally by Select Access. You are not expected to use this data string.
BOOT_CONF_TAG	enforcerBootConfig	Used internally by Select Access for configuration. You are not expected to use this data string.
SA_PRESERVED_POST	SA_PRESERVED_POST	An HTML hidden form field that contains all the name/value pairs that were in the original POSTed data.
SA_PRESERVED_POST_EMPTY	SA_PRESERVED_POST + "=&"	An HTML hidden form field that does not contain the name/value pairs that is typically sent in the original POSTed data.
CA_CERT_TAG	serverCertCA	Used internally by Select Access. You are not expected to use this data string.
CLIENT_CERT_TAG	clientSSLCert	Used internally by Select Access for configuration. You are not expected to use this data string.
CLIENT_KEY_TAG	clientSSLKey	Used internally by Select Access for configuration. You are not expected to use this data string.
COLON	:	The colon symbol.
COMPONENT	COMPONENT	Used internally by Select Access. You are not expected to use this data string.
COMPONENT_NAME	HP OpenView Select Access Java Enforcer Plugin	Used internally by Select Access for configuration. You are not expected to use this data string.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
CONF_TAG	enforcerConfig	Used internally by Select Access for configuration. You are not expected to use this data string.
CONNECT_TIMEOUT	connectTimeout	Used internally by Select Access for configuration. You are not expected to use this data string.
COOKIE_DOMAIN	cookieDomain	Used internally by Select Access for configuration. You are not expected to use this data string.
DEBUG_LEVEL	debugLevel	Used internally by Select Access for configuration. You are not expected to use this data string.
DEFER_OPEN_VAR	deferOpen	Used internally by Select Access. You are not expected to use this data string.
DISABLE_CACHING	disableCaching	Used internally by Select Access for configuration. You are not expected to use this data string.
DOMAIN_NAME	domain_name	Used internally by Select Access for configuration. You are not expected to use this data string.
ENABLE_COOKIES	enableCookies	Used internally by Select Access for configuration. You are not expected to use this data string.
ENFORCER_ACCEPTED_PAGE	accepted.html	The confirmation page displayed when an identity's credentials have been accepted.
ENFORCER_ACTION	action	The XML response property which indicates the action the Enforcer plugin should take. Will be one of: • Allow • Deny • Error • Developer-defined

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
ENFORCER_ACTION_ALLOW	ALLOW	A Policy Validator allow action.
ENFORCER_ACTION_DENY	DENY	A Policy Validator deny action.
ENFORCER_ACTION_ERROR	ERROR	A Policy Validator error.
ENFORCER_ACTION_USER_DEFINED	USER_DEFINED	A Policy Validator developer-defined action.
ENFORCER_ATTRIBUTE_NAME	NAME	The value of an XML element's NAME attribute.
ENFORCER_ATTRIBUTES	attributes	An XML property list which holds attribute name/value properties.
ENFORCER_ATTRINFO	attribute_info	The name of the XML property list element in the Policy Validator response. Property lists contain identity attribute name/value pairs in the Policy Validator response.
ENFORCER_AUTH_URL	ENFORCER_AUTH_URL	Used internally by Select Access for configuration. You are not expected to use this data string.
ENFORCER_AUTHENTICATED_DN	authenticated_dn	The XML response property which specifies the authenticated identity DN.
ENFORCER_AUTHENTICATION_ASSERTION	authentication_assertion	The XML response property element which contains the authentication assertion.
ENFORCER_AUTHENTICATION_HINT	authentication_hint	The XML response property element which contains hints for further authentication.
ENFORCER_AUTHENTICATION_INFORMATION	authentication_information	An XML property which holds authentication assertion information.
ENFORCER_AUTHENTICATION_INSTANT	authentication_instant	Used internally by Select Access for configuration. You are not expected to use this data string.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
ENFORCER_ AUTHENTICATION_METHOD	authentication_method	The XML response property which specifies the method of authentication used by the client. This property is nested in <PROPERTYLIST NAME= "Authentication_ Server_ Types>.
ENFORCER_ AUTHENTICATION_SERVER_ TYPES	authentication_server_ types	An XML query property list which contains the types of servers used for authentication.
ENFORCER_CERT	cert	The name of an XML query property, which contains the client's PEM-encoded X.509 certificate.
ENFORCER_CERTIFICATE	certificate	An authentication_method type returned in Policy Validator reply.
ENFORCER_CHALLENGE_ RESPONSE	chalresp	A challenge response property.
ENFORCER_CLIENT	client	The client browser name and version.
ENFORCER_COMPONENT	COMPONENT	The top-level element in the XML representation of a rule.
ENFORCER_CONDITION	CONDITION	Used internally by Select Access. You are not expected to use this data string.
ENFORCER_CONFIG_ID	enforcerID	Used internally by Select Access for configuration. You are not expected to use this data string.
ENFORCER_CONFIG_ REQUEST	nxEnforcerConfig Request	Used internally by Select Access for configuration. You are not expected to use this data string.
ENFORCER_DENY_PAGE	deny.html	*Used by IIS Enforcer plugins* only. The name of the form used by this plugin, because the IIS web server does not have its own default deny page.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
ENFORCER_DN	dn	The name of an XML query property, which contains the authentication domain.
ENFORCER_DSTHOSTNAME	dstHost	The name of the machine on which the resource is located.
ENFORCER_DSTIP	dstIP	The IP address of the machine on which the resource is located.
ENFORCER_DSTPORT	dstPort	The port number being used by the service on the destination machine.
ENFORCER_EMPTY_NAME_ VALUE		Used internally by Select Access. You are not expected to use this data string.
ENFORCER_ENABLE	enable	An enable action used to enable a nonces, etc.
ENFORCER_ERROR	error	The name of a response property, which contains an error message from the Policy Validator.
ENFORCER_EVALUATOR	EVALUATOR	Used internally by Select Access. You are not expected to use this data string.
ENFORCER_FORM_ AUTHENTICATOR_TYPE	authenticator	The name of the response property, which contains the name of the authentication form. This constant is sent as the result of an authentication hint from the Policy Validator.
ENFORCER_FORM_ CHALLENGE	form_challenge	The name of the response property, which contains the name of the challenge form. This constant is sent as the result of an authentication hint from the Policy Validator.
ENFORCER_FORM_DATA	form_data	The name of the response property list, which contains the name of the form data that needs to be displayed to the client.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
ENFORCER_FORM_DECIDER_TYPE	decider	The name of the response property list, which contains the type of decider used to authenticate the client.
ENFORCER_FORM_ID	form_id	The name of the response property list, which contains the ID of the form to be displayed to the client.
ENFORCER_FORM_PAGE	form_page	The value of the response property, which contains the ID of the profile management form to be displayed to the client.
ENFORCER_FORM_PASSWORD_CHANGED	password_changed_form.html	The name of the response property list, which contains the name of the form sent when a password change request is made by an identity.
ENFORCER_FORM_POST_PASSWORD_CHANGED	post_password_changed_form.html	The name of the response property list, which contains the name of the form that is sent to the client when the password change was successful.
ENFORCER_FORM_RADIUS_STATE	form_radius_state	The value of the response property, which contains the ID of the RADIUS form to be displayed to the client.
ENFORCER_FORM_REALM	form_realm	The value of the response property, which contains the realm of the form to be displayed to the client.
ENFORCER_FORM_TITLE	form_title	The value of the response property, which contains the title of the form to be displayed to the client.
ENFORCER_FORM_TYPE	form_type	The value of the response property, which contains the type of the form to be displayed to the client.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
ENFORCER_FORM_USER	user	The value of the response property, which contains the ID of the identity management form to be displayed to the client.
ENFORCER_GROUPINFO	group_info	The name of the response property list, which contains the identity's group personalization information.
ENFORCER_HTTP_HEADER_LIST	http_header_list	The XML query property list which contains the header list parameters.
ENFORCER_HTTP_QUERY	http_query	The CGI query parameters, that is, the part of a URL after the ? character.
ENFORCER_HTTP_QUERY_LIST	http_query_list	The XML query property list which contains properties for each CGI query parameter.
ENFORCER_LOGIN_TIME	login_time	The time at which identity was authenticated.
ENFORCER_MESSAGE	message	The text of an error message.
ENFORCER_METHOD	method	The HTTP request method being used. Can be one of: • Get • Put • Head • Login
ENFORCER_MULTIAUTH_FORM	multiauth_form.html	The form sent when a multiple resource request is made.
ENFORCER_NATIVE_AUTH	native_auth	The type of authentication used by the client: • password • online registration • certificate • nonce
ENFORCER_NATIVE_CERT	native_certificate	Authentication by an X.509 certificate.
ENFORCER_NATIVE_CHAL	native_challenge	Not yet supported.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
ENFORCER_NATIVE_NONCE	native_nonce	The XML query property which indicates whether nonces have been enabled or not.
ENFORCER_NATIVE_PASSWD	native_password	Authentication by password.
ENFORCER_NATIVE_REGISTER	native_register	Authentication by www online registration.
ENFORCER_NEED_AUTH	need_auth	The XML response property which indicates whether further authentication is required.
ENFORCER_NONCE	nonce	The nonce, typically an HTTP cookie.
ENFORCER_OWNER	owner	The owner of an object.
ENFORCER_P13NINFO	personalization	An XML response property list which contains personalization information for the identity making the request.
ENFORCER_PARTNER_NAME	partner_name	The value of an XML element with the an attr tag whose value will be the actual partner name.
ENFORCER_PASSWD	password	A static password for a username (ENFORCER_USER).
ENFORCER_PATH	path	The part of the URL used to locate a resource on a service. For example, "/", "/cgi/finger", etc.
ENFORCER_POST_ACCEPTED_FORM	post_accepted.html	The name of the XML query property, which contains the name of the form that tells a client that their credentials have been accepted after a POST.
ENFORCER_POST_DATA_CONSUMED	post_data_consumed	Used internally by Select Access. You are not expected to use this data string.
ENFORCER_POST_DATA_LIST	post_data_list	The name of the XML query property, which contains the identity credentials POSTed by the client.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
ENFORCER_POST_REDIRECT_FORM	post_redirect.html	The name of the XML query property, which contains the name of the form that redirects the client.
ENFORCER_PROTOCOL	protocol	The protocol used by a requested resource. For example, 'http", "ftp", "file", etc.
ENFORCER_QUERY	PolicyValidatorQuery	The name of the root XML query element for the Policy Validator.
ENFORCER_QUERY_END	</" + ENFORCER_QUERY + ">	The closing tag for the root element of an XML query document.
ENFORCER_QUERY_START	<" + ENFORCER_QUERY + ">	The opening tag for the root element of an XML query document.
ENFORCER_QUERYID	queryID	The name of the XML query property, whose value uniquely identifies the query and allows it to be associated with its corresponding response document.
ENFORCER_QUERYID_END	</" + ENFORCER_TAG_PROPERTY + ">	The closing tag for a property containing the query ID.
ENFORCER_QUERYID_START	<" + ENFORCER_TAG_PROPERTY + " " + ENFORCER_ATTRIBUTE_NAME + "=\"" + ENFORCER_QUERYID + "\">	The opening tag for a property containing the query ID.
ENFORCER_REDIRECT	redirect_url	The name of the XML response property, which contains the URL to which the browser will be redirected.
ENFORCER_REGISTER_LIFE	register_cookie_lifetime	The URL from which the request was made.
ENFORCER_REGISTER_NONCE	register_cookie	The name of the root XML response element for the Policy Validator.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
ENFORCER_REPLY	PolicyValidatorReply	The closing tag for the root element of an XML response document.
ENFORCER_REPLY_END	</" + ENFORCER_REPLY + ">	The opening tag for the root element of an XML response document.
ENFORCER_REPLY_START	<" +ENFORCER_REPLY +">	The name of the XML response property, which contains the URL to which the browser will be redirected.
ENFORCER_ROLEINFO	role_info	The name of a XML response property list, which contains personalization information for an identity based on dynamic group information.
ENFORCER_SA_CONF_ RESISTRY_ROOT	HKEY_LOCAL_MACHINE	Used internally by Select Access for configuration. You are not expected to use this data string.
ENFORCER_SA_CONF_ RESISTRY_SUBKEY	SOFTWARE\\HP Openview\\SelectAccess	Used internally by Select Access for configuration. You are not expected to use this data string.
ENFORCER_SA_CONF_ RESISTRY_VAL	ConfigFile	Used internally by Select Access for configuration. You are not expected to use this data string.
ENFORCER_SEND_REFRESH	selectaccess_send_ refresh	The name of the tag on the Select Access login forms, which tell the Enforcer plugins to send the "Credentials accepted" page upon a successful authentication.
ENFORCER_SERVER	server	The name of the XML query property, which contains the server software name and version.
ENFORCER_SERVICE	service	The name of the XML query property, which contains the name and port number of the service which hosts a resource. For example, "mycompany.com:8099".

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
ENFORCER_SITE_DATA	site_data	The name of the XML query property, which contains site-specific data for a web server.
ENFORCER_SIZE	size	*For Apache 2 Enforcer plugins only.* The name of the XML query property, which contains the size of the query.
ENFORCER_SRCHOSTNAME	srcHost	The name of the XML query property, which contains the name of the machine on which the request originated.
ENFORCER_SRCIP	srcIP	The name of the XML query property, which contains the IP address of the machine of which the request originated.
ENFORCER_SRCPORT	srcPort	The name of the XML query property, which contains the port number being used by the machine on which the request originated.
ENFORCER_SSL_CIPHER	ssl_cipher	The name of the XML query property, which contains the SSL logon cipher used by the client.
ENFORCER_SSL_CLIENT_DN	dn	The name of the XML query property, which contains the DN of the client's certificate.
ENFORCER_SSL_KEYSIZE	ssl_keysize	The name of the XML query property, which contains the SSL cipher's keysize.
ENFORCER_SSL_PROTOCOL	ssl_protocol	The name of the XML query property, which contains the SSL cipher's protocol.
ENFORCER_SUBJECT_NAME	subject_name	The name for which SAML assertions were received.
ENFORCER_TAG_PROPERTY	PROPERTY	An XML <PROPERTY> element.
ENFORCER_TAG_ PROPERTYLIST	PROPERTYLIST	An XML <PROPERTYLIST> element.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
ENFORCER_TIMESTAMP	timestamp	Used internally by Select Access for configuration. You are not expected to use this data string.
ENFORCER_TIMESTAMP_FORMAT	%Y:%m:%d:%w:%H:%M:%S	Used internally by Select Access for configuration. You are not expected to use this data string.
ENFORCER_TRACE	trace	The name of the XML query property, which contains a verbose rule-evaluation trace from the Policy Validator.
ENFORCER_UID	uid	Used internally by Select Access. You are not expected to use this data string.
ENFORCER_USER	user	• For the value of an XML query, which specifies that the Enforcer plugin supports password authentication. • For the value of a response, which specifies that password authentication failed.
ENFORCER_USERINFO	user_info	The name of the XML response property list, which contains the identity information (part of the personalization data).
ENFORCER_VERSION	enforcerVersion	The name of the XML response property, which contains the Enforcer plugin's version.
EQUALS	=	The equal sign.
FALSE	false	False.
IGNORED_FILENAME	ignoredFilename	A file designated as safe, and therefore not requiring Select Access protection.
LOG_TAG	logClientConfig	Used internally by Select Access for configuration. You are not expected to use this data string.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
LOGIN_VIA_FORM	loginViaForm	The name of the response property, which contains the name of the login form. This constant is sent as the result of an authentication hint from the Policy Validator.
MULTI_RESOURCE_LIST	multiResourceList	The name of a response property list, which contains the names of multiple resources.
NULL	""	A NULL or empty string.
NULL_STR	""	A NULL or empty string.
OS_NAME	os.name	Used internally by Select Access. You are not expected to use this data string.
P13N_DATA_TAG	Personalization	Used internally by Select Access for personalization. You are not expected to use this data string.
P13N_USER_DN_TAG	UserDN	Used internally by Select Access for personalization. You are not expected to use this data string.
P13N_USER_NAME_TAG	UserName	Used internally by Select Access for personalization. You are not expected to use this data string.
P13N_GROUP_NAMES_TAG	GroupNames	Used internally by Select Access for personalization. You are not expected to use this data string.
P13N_ROLE_NAMES_TAG	RoleNames	Used internally by Select Access for personalization. You are not expected to use this data string.
P13N_USER_ATTRS_TAG	UserAttributes	Used internally by Select Access for personalization. You are not expected to use this data string.
P13N_GROUP_ATTRS_TAG	GroupAttributes	Used internally by Select Access for personalization. You are not expected to use this data string.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
P13N_ROLE_ATTRS_TAG	RoleAttributes	Used internally by Select Access for personalization. You are not expected to use this data string.
PASSTHROUGH_DOMAIN	unsecuredDomain	The name of a property which indicates that the domain is designated as safe, and therefore not requiring Select Access protection.
PER_VAL_POOL_SIZE	perValidatorPoolSize	Used internally by Select Access for configuration. You are not expected to use this data string.
PRESERVED_POST_HTML	ENFORCER_PRESERVED_ POST_HTML	The name of the XML query property, which contains the name of the place holder tag used on post data forms.
PROTECTED_DOMAIN	protectedDomain	A multi-domain single sign-on (MD-SSO) protected site.
PROXY_MODE	proxyMode	Used internally by Select Access for configuration. You are not expected to use this data string.
QUERY_LEVEL	queryLevel	Used internally by Select Access for configuration. You are not expected to use this data string.
QUERY_MAXIMAL	Maximal	Used internally by Select Access for configuration. You are not expected to use this data string.
QUERY_MINIMAL	Minimal	Used internally by Select Access for configuration. You are not expected to use this data string.
QUERY_REGULAR	Regular	Used internally by Select Access for configuration. You are not expected to use this data string.
RANDOM_START	random_start	Used internally by Select Access for configuration. You are not expected to use this data string.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
RANDOM_START_VAR	randomStart	Used internally by Select Access for configuration. You are not expected to use this data string.
REGISTER_COOKIE_NAME	PolicyRegisteredUser	Used internally by Select Access for configuration. You are not expected to use this data string.
REPLY_TIMEOUT	replyTimeout	Used internally by Select Access for configuration. You are not expected to use this data string.
RESULT	result	Result.
RETRY_TIME	retry_time	Used internally by Select Access for configuration. You are not expected to use this data string.
RETRY_TIME_VAR	retryTime	Used internally by Select Access for configuration. You are not expected to use this data string.
ROUND_ROBIN	round_robin	Used internally by Select Access for configuration. You are not expected to use this data string.
ROUND_ROBIN_VAR	roundRobin	Used internally by Select Access for configuration. You are not expected to use this data string.
RTIMEOUT	rtimeout	Used internally by Select Access for configuration. You are not expected to use this data string.
SERVER_HOST	serverHost	Used internally by Select Access for configuration. You are not expected to use this data string.
SERVER_POOL	server_pool	Used internally by Select Access for configuration. You are not expected to use this data string.

Table 51Java Data Strings (cont'd)

String Name	Value	Represents
SERVER_PORT	serverPort	Used internally by Select Access for configuration. You are not expected to use this data string.
SERVER_USE_ALL	serverUseAll	Used internally by Select Access for configuration. You are not expected to use this data string.
SERVER_USE_SSL	server_use_ssl	Used internally by Select Access for configuration. You are not expected to use this data string.
SERVER_USE_SSL_VAR	serverUseSSL	Used internally by Select Access for configuration. You are not expected to use this data string.
SPACE	" "	An empty space.
SUBMIT_BUTTON_NAME	.submit	The name.
TRUE	true	True.
USER_DATA_CHARACTER_SET_TAG	userDataCharacterSet	Used internally by Select Access for configuration. You are not expected to use this data string.
UTF8	UTF8	Used internally by Select Access for configuration. You are not expected to use this data string.
VERSION_STR	com.hp.ov.selectaccess.common.AppVersion.getVersionNumber()	Used internally by Select Access for configuration. You are not expected to use this data string.
WINDOWS	Windows	Used internally by Select Access for configuration. You are not expected to use this data string.
WRAP_STR	%%	Used internally by Select Access for configuration. You are not expected to use this data string.

A Authentication Failure Reasons

Policy Validator reply can contain reasons for authentication failure. These reasons are included in an `auth_fail_reason` parameter.

The following sections list all know authentication failure reasons.

Password Failures

```
const char AuthPlugin::AUTH_FAILED[] = "Authentication failed";

const char AuthPlugin::AUTH_DISABLED[] = "Account disabled";

const char AuthPlugin::AUTH_DISABLED_INVALID[] = "Account disabled due
to too many invalid passwords";

const char AuthPlugin::AUTH_DISABLED_INACTIVE[] = "Account disabled due
to inactivity";

const char AuthPlugin::AUTH_UNKNOWN_USER[] = "Unknown user name";

const char AuthPlugin::AUTH_INCORRECT_PASSWORD[] = "Incorrect password";

const char AuthPlugin::AUTH_LOGGED_OUT[] = "Logged out from session";

const char AuthPlugin::AUTH_SESSION_TIMEOUT[] = "Session timed out";

// for registration

const char AuthPlugin::AUTH_USER_EXISTS[] = "User name already exists";

const char AuthPlugin::AUTH_MISSING_INFO[] = "Missing information";

// for certificates

const char AuthPlugin::AUTH_CERT_REVOKED[] = "Certificate revoked";

const char AuthPlugin::AUTH_CERT_EXPIRED[] = "Certificate expired";

const char AuthPlugin::AUTH_CERT_STATUS_UNAVAILABLE[] = "Certificate
status information unavailable";

const char AuthPlugin::AUTH_CERT_AUTHORITY[] = "Certificate issued by
untrusted authority";
```

Password Management Failures

```
const char AuthPlugin::AUTH_MUST_CHANGE[] = "Password must be changed";

const char AuthPlugin::AUTH_EXPIRED[] = "Password expired";
```

```
const char AuthPlugin::AUTH_WILL_EXPIRE[] = "Password will expire soon";

const char AuthPlugin::AUTH_CHANGE_FAILED[] = "Password change attempt
failed";

const char AuthPlugin::AUTH_PASSWORD_STRENGTH[] = "New password does not
meet rules";
```

Index

D

E

H

J

L

O

P

R

S

T

U

W

X