Juniper Networks
Steel-Belted Radius

Scripting Guide

Release 6.1
November 2007

Juniper Networks, Inc.
1194 North Mathilda Avenue
Sunnyvale, CA 94089
USA
408-745-2000
www.juniper.net

Part Number: SBR-TD-SCR61 Revision 00

Copyright © 2004–2007 Juniper Networks, Inc. All rights reserved. Printed in USA.

Steel-Belted Radius, Juniper Networks, the Juniper Networks logo are registered trademark of Juniper Networks, Inc. in the United States and other countries. All other trademarks, service marks, registered trademarks, or registered service marks are the property of their respective owners. All specifications are subject to change without notice.

Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

Portions of this software copyright 1989, 1991, 1992 by Carnegie Mellon University Derivative Work - 1996, 1998-2000 Copyright 1996, 1998-2000 The Regents of the University of California All Rights Reserved Permission to use, copy, modify and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of CMU and The Regents of the University of California not be used in advertising or publicity pertaining to distribution of the software without specific written permission.

CMU AND THE REGENTS OF THE UNIVERSITY OF CALIFORNIA DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL CMU OR THE REGENTS OF THE UNIVERSITY OF CALIFORNIA BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM THE LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Portions of this software copyright © 2001-2002, Networks Associates Technology, Inc All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Networks Associates Technology, Inc nor the names of its contributors might be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Portions of this software are copyright © 2001-2002, Cambridge Broadband Ltd. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The name of Cambridge Broadband Ltd. might not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Portions of this software copyright © 1995-2002 Jean-loup Gailly and Mark Adler This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software. Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

- The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
- Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
- This notice might not be removed or altered from any source distribution.

HTTPClient package Copyright © 1996-2001 Ronald Tschalär (ronald@innovation.ch).

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. For a copy of the GNU Lesser General Public License, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

StrutLayout Java AWT layout manager Copyright © 1998 Matthew Phillips (mpp@ozemail.com.au).

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details. For a copy of the GNU Lesser General Public License, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

60-071115:752

Table of Contents

About This Guide vii

Before You Begin ... vii
Audience ... vii
What's In This Manual ... vii
Conventions ... viii
  Syntax ... ix
Related Documentation ... x
  Steel-Belted Radius Documentation ... x
  Requests for Comments (RFCs) ... x
  Third-Party Products ... xi
Contacting Technical Support ... xi

Chapter 1 Introduction to Scripting 1

Scripting Overview ... 1
Script Types ... 2
  About LDAP Authentication Scripts ... 3
  About Realm Selection Scripts ... 3
  About Attribute Filter Scripts ... 4
About JavaScript ... 5

Chapter 2 Creating Scripts 7

Script Development Steps ... 7
JavaScript Initialization Files ... 8
  [Settings] Section ... 8
  [Script] Section ... 9
  [ScriptTrace] Section ... 10
  [Failure] Section ... 10
Writing Steel-Belted Radius Scripts in JavaScript ... 11
  Programming in JavaScript ... 11
  Hidden Wrapper Function ... 11
  Script Return Values ... 11
  Initializing Reusable Data Objects ... 12
  General Recommendations ... 13
Saving the Script File ... 13
Installing the JavaScript Upgrade License ... 13
Sample Script ... 14

Chapter 3 Debugging Scripts 17

Writing Messages to the Server Log ... 17
Script Tracing ... 17
scriptcheck Utility ... 19

Chapter 4 Creating LDAP Scripts 23

LDAP 23
LDAP Request Life Cycle 23
Unscripted LDAP Searches 24
LDAP Script Basics 26
- Working with the Variable Table 26
- Invoking LDAP Queries 27
- Writing to the Steel-Belted Radius Log 27

Choosing the Return Code 27
- Script Return Codes 28

Chapter 5 Creating Realm Selection Scripts 29

Realm Selection Script Functions 30
Enabling Built-In Realm Selection Methods 30
Choosing the Return Code 31
Configuring Realm Selection Scripts 31
- Core Realm Selection Scripts 32
- Tunneled Authentication Plug-in Realm Selection Scripts 33

Chapter 6 Creating Attribute Filter Scripts 35

Attribute Filter Script Functions 36
Choosing the Return Code 36
Configuring Attribute Filter Scripts 37
- Defining Scripted Filters 37

Chapter 7 Working with Data Accessors 41

Data Accessor Overview 41
Variable Containers 42
Internal Variable Table (LDAP Only) 43
Data Accessor Configuration 43
- SQL Data Accessor Configuration 43
- LDAP Data Accessor Configuration 47

Data Conversion Rules 55
- Output Container 56
- Input Container 56
- Examples 57
- Supported Data Types and Conversions 58

Data Accessor Configuration File Examples 59
- Example: LDAP Data Accessor Configuration File 59
- Example: SQL Data Accessor Configuration File 61

Chapter 8 Script Examples 63

LDAP Script Examples 63
- Example 1: Simple Authentication 63
- Example 2: Profile Assignment 64
- Example 3: Received Attribute Normalization 65
- Example 4: Conditional Profile Assignment from User Attribute 66

Realm Selection Script Examples 68
- Example 1: Querying Multiple SQL Databases 68
- Example 2: Using JavaScript to Manipulate Request Attributes 71

Attribute Filter Script Examples 73

Example 1: Using an LDAP Query to Select a Static Filter to Execute 73
Example 2: Adding Values to Multi-Valued Attributes 75

Chapter 9 Script Reference 77

JavaScript Types 77
API Method Support by Script Type 78
Local and Global Variable Declarations 79
Global Object 79
Logging and Diagnostic Methods 79
Ldap Object 80
Ldap Methods 80
LdapVariables Object 81
LdapVariables Methods 81
RealmSelector Object 83
Constructor 83
RealmSelector Methods 84
AttributeFilter Object 86
Constructor 86
AttributeFilter Methods 86
AttributeFilter API 89
DataAccessor Object 90
Properties 90
Constructor 90
Methods 91

Appendix A LDAP Script Return Codes 95

Index 97

About This Guide

The *Steel-Belted Radius Scripting Guide* describes how to use scripts written in the JavaScript programming language to enhance the RADIUS request processing capabilities of the Steel-Belted Radius server.

Before You Begin

Before you use this manual, you should review the *Steel-Belted Radius Administration Guide* to help you understand how the components of Steel-Belted Radius work together. You should also read the Steel-Belted Radius release notes for updates about software features, requirements, and updates to your Steel-Belted Radius documentation.

This manual assumes that you have installed the Steel-Belted Radius server software and the SBR Administrator. For information about how to install the Steel-Belted Radius software, see the *Steel-Belted Radius Getting Started Guide*.

Audience

This guide is intended for experienced system and network specialists working in an Internet access environment. You should be familiar with Lightweight Directory Access Protocol (LDAP) directories and the JavaScript scripting language. You should also be familiar with your network environment and conventions.

What's In This Manual

This manual contains the following chapters and appendices:

- Chapter 1, "Introduction to Scripting," describes the key concepts of scripting and the three scripting types currently available to Steel-Belted Radius users: LDAP, Realm Selection, and Attribute Filter.
- Chapter 2, "Creating Scripts," describes the common elements of writing scripts for Steel-Belted Radius, including the JavaScript initialization file configuration settings.
- Chapter 3, "Debugging Scripts," describes the tools and techniques required to debug scripts and monitor their runtime execution.

- Chapter 4, “Creating LDAP Scripts,” provides an overview of LDAP scripting and describes how to create and configure an LDAP script.
- Chapter 5, “Creating Realm Selection Scripts,” provides an overview of realm selection scripts and describes how to create and configure them.
- Chapter 6, “Creating Attribute Filter Scripts,” provides an overview of attribute filter scripts and describes how to create and configure them.
- Chapter 7, “Working with Data Accessors,” describes the properties, constructor, and functions used in the data accessor, and provides samples of LDAP and SQL data accessor configuration files for scripting.
- Chapter 8, “Script Examples,” provides script examples for the three script types: LDAP, realm selection, and attribute filter.
- Chapter 9, “Script Reference,” lists JavaScript types, API functions, local and global variables, and objects descriptions used in scripting.
- Appendix A, “LDAP Script Return Codes,” provides a table of LDAP script return codes.

Conventions

Table 1 describes the text conventions used throughout this manual.

Table 1: Typographical Conventions

Convention	Description	Examples
Bold typeface	Indicates buttons, field names, dialog names, and other user interface elements.	Use the **Scheduling** and **Appointment** tabs to schedule a meeting.
Plain sans serif typeface	Represents: ■ Code, commands, and keywords ■ URLs, file names, and directories	Examples: ■ Code: certAttr.OU = 'Retail Products Group' ■ URL: Download the JRE application from: http://java.sun.com/j2se/

Table 1: Typographical Conventions (continued)

Convention	Description	Examples
Italics	Identifies: ■ Terms defined in text ■ Variable elements ■ Book names	Examples: ■ Defined term: An *RDP client* is a Windows component that enables a connection between a Windows server and a user's machine. ■ Variable element: Use settings in the **Users** > **Roles** > *Select Role* > **Terminal Services** page to create a terminal emulation session. ■ Book name: See the *Steel-Belted Radius Administration Guide.*

Syntax

- *radiusdir* represents the directory into which Steel-Belted Radius has been installed. By default, this is C:\Program Files\Juniper Networks\Steel-Belted Radius\Service for Windows systems and /opt/JNPRsbr/radius on Linux and Solaris systems.

- Brackets [ ] enclose optional items in format and syntax descriptions. In the following example, the first *Attribute* argument is required; you can include an optional second *Attribute* argument by entering a comma and the second argument (but not the square brackets) on the same line.

  <add | replace> = *Attribute* [,*Attribute*]

  In configuration files, brackets identify section headers:

  ...the [Processing] section of proxy.ini...

  In screen prompts, brackets indicate the default value. For example, if you press **Enter** without entering anything at the following prompt, the system uses the indicated default value (/usr/lib).

- Angle brackets < > enclose a list from which you must choose an item in format and syntax descriptions. Angle brackets < > can also represent a replacement variable consisting of the variable name. Upon execution of an LDAP Search request, the value of the variable replaces the variable name.

  For example, a Search template that uses the User-Name and Service-Type attributes from the RADIUS request might look like this:

  (&(uid = <User-Name>)(type = <Service-Type>))

- A vertical bar (|) separates items in a list of choices. In the following example, you must specify add or replace (but not both):

  <add | replace> = *Attribute* [,*Attribute*]

Related Documentation

The following documents supplement the information in this manual.

Steel-Belted Radius Documentation

The readme.txt file contains the latest information about features, changes, known problems, and resolved problems. If the information differs from the information found in the documentation set, defer to the information in the Release Notes.

In addition to this manual, the Steel-Belted Radius documentation includes the following manuals:

- The *Steel-Belted Radius Reference Guide* describes the configuration files and settings used by Steel-Belted Radius.
- The *Steel-Belted Radius Getting Started Guide* describes how to install, configure, and administer the Steel-Belted Radius software on a server running the Solaris, Linux, or Windows operating system.

Requests for Comments (RFCs)

The Internet Engineering Task Force (IETF) maintains an online repository of Request for Comments (RFCs) online at http://www.ietf.org/rfc.html. Table 2 lists the RFCs that apply to this guide.

Table 2: RFCs

RFC Number	Title
RFC 2548	*Microsoft Vendor-specific RADIUS Attributes.* G. Zorn. March 1999.
RFC 2618	*RADIUS Authentication Client MIB.* B. Aboba, G. Zorn. June 1999.
RFC 2619	*RADIUS Authentication Server MIB.* G. Zorn, B. Aboba. June 1999.
RFC 2620	*RADIUS Accounting Client MIB.* B. Aboba, G. Zorn. June 1999.
RFC 2621	*RADIUS Accounting Server MIB.* G. Zorn, B. Aboba. June 1999.
RFC 2809	*Implementation of L2TP Compulsory Tunneling via RADIUS.* B. Aboba, G. Zorn. April 2000.
RFC 2865	*Remote Authentication Dial In User Service (RADIUS)*. C. Rigney, S. Willens, A. Rubens, W. Simpson. June 2000.
RFC 2866	*RADIUS Accounting*. C. Rigney. June 2000.
RFC 2867	*RADIUS Accounting Modifications for Tunnel Protocol Support*. G. Zorn, B. Aboba, D. Mitton. June 2000.
RFC 2868	*RADIUS Attributes for Tunnel Protocol Support*. G. Zorn, D. Leifer, A. Rubens, J. Shriver, M. Holdrege, I. Goyret. June 2000.
RFC 2869	*RADIUS Extensions.* C. Rigney, W. Willats, P. Calhoun. June 2000.
RFC 2882	*Network Access Servers Requirements: Extended RADIUS Practices*. D. Mitton. July 2000.
RFC 3162	*RADIUS and IPv6*. B. Aboba, G. Zorn, D. Mitton. August 2001.

Third-Party Products

For more information about configuring your access servers and firewalls, consult the manufacturer's documentation provided with each device.

Contacting Technical Support

For technical support, contact Juniper Networks at support@juniper.net, or at 1-888-314-JTAC (in the United States) or 408-745-9500 (outside the United States).

Check our Web site (http://www.juniper.net) for additional information and technical notes. When you are running SBR Administrator, you can select **Web > Steel-Belted Radius User Page** to access a special home page for Steel-Belted Radius users.

When you call technical support, please have the following information at hand:

- Your Steel-Belted Radius product edition and release number (for example, Global Enterprise Edition version 6.0).
- Information about the server configuration and operating system, including any OS patches that have been applied.
- For licensed products under a current maintenance agreement, your license or support contract number.
- Question or description of the problem, with as much detail as possible.
- Any documentation that might help in resolving the problem, such as error messages, memory dumps, compiler listings, and error logs.

Chapter 1

Introduction to Scripting

This chapter introduces the key concepts of Steel-Belted Radius scripting and provides examples of how you can use scripting to extend the capabilities of the Steel-Belted Radius server.

Incorporating scripts into your Steel-Belted Radius configuration enables you to fine-tune the behavior of the Steel-Belted Radius server and implement custom request processing logic. You can use scripts to configure Steel-Belted Radius to evaluate complex decision logic and manipulate RADIUS request data objects in ways that cannot be expressed through settings in the standard Steel-Belted Radius initialization files.

Steel-Belted Radius scripts are written in JavaScript, an easy-to-use, industry standard scripting language with a powerful, object-based syntax.

Scripting Overview

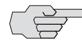
The Steel-Belted Radius server invokes many built-in functional modules while processing RADIUS requests. These modules are configured by initialization files in the Steel-Belted Radius home directory. For example, you configure the realm selection module with settings in the proxy.ini file.

With scripting, you can supplement or override specific functional modules within the Steel-Belted Radius server by implementing custom processing logic written in JavaScript. JavaScript APIs that are exposed by the server enable your scripts to perform the following tasks.

- Manipulate RADIUS request attributes.
- Select the processing realm for a request.
- Query external SQL and LDAP servers.
- Print information and debug messages to the server log.

You compose Steel-Belted Radius scripts in special initialization files that contain both the script text and settings required by Steel-Belted Radius to execute and optionally debug the script. JavaScript initialization (.jsi) files use a parameter syntax similar to that of other Steel-Belted Radius configuration files.

To configure a script to load and run, you refer to it by name using the Script keyword at the appropriate place in a Steel-Belted Radius initialization file. The context in which a script executes, which determines the data objects and JavaScript APIs available to it, depends on where the script reference appears in the Steel-Belted Radius configuration.



NOTE: For the LDAP authentication plugin, script settings are embedded directly in the ldapauth.aut file.

Before a script runs, Steel-Belted Radius must load and compile the script text into JavaScript bytecodes. This occurs at server start time, or in some cases, immediately before the server first invokes the script. If the script fails to load or compile, a diagnostic error message appears in the log and the associated function is either disabled or reverts to its default behavior.

Your script executes each time the flow of control within Steel-Belted Radius enters a functional module that is configured to run that script. The scripting infrastructure automatically sets up the correct environment for the script, depending on its type. The script executes until it returns normally or it encounters a runtime exception. To prevent the script from being caught in an infinite loop, you can configure an optional watchdog counter to terminate the script after it has executed a preset number of operations.

Logging and tracing functions are provided as an aid to script debugging. Scripts can send messages directly to the Steel-Belted Radius server log. Additionally, you can use the script trace feature to write line-by-line debug information to the log. Each script trace frame contains the text, filename, and line number of the next JavaScript statement for the script to execute, and the names and values of user-specified script variables.

Script Types

Steel-Belted Radius supports three types of scripts. The functional module(s) from which the script is invoked by the server determines the script type. The three script types are:

- LDAP authentication—Scripts that control the execution of searches and the processing of attributes by the LDAP authentication plugin. The LDAP authentication scripts are executed only by the LDAP authentication plugin.
- Realm selection—Scripts used to determine the name of a proxy or directed realm to which a RADIUS request is directed for processing. Realm selection scripts are executed during normal request processing by the Steel-Belted Radius server core and during inner authentication by the tunneled authentication plug-ins (FAST, PEAP, and TTLS).
- Attribute filter—Scripts used to manipulate the values of attributes in the RADIUS request or response packets. Attribute filter scripts are executed any time a server core component or plugin module invokes an attribute filter that is configured for scripting.

About LDAP Authentication Scripts

Steel-Belted Radius uses the LDAP authentication plugin to authenticate users and retrieve attributes from external LDAP repositories. LDAP connection parameters and search specifications are defined in the ldapauth.aut file.

You can configure the LDAP authentication plugin to perform scripted or unscripted searches. With unscripted searches, selected attributes can be transferred directly from the RADIUS request into the LDAP search string, and from the LDAP search result into the RADIUS response. You can create a simple search tree to execute a sequence of LDAP searches each time Steel-Belted Radius processes an authentication request.

With LDAP authentication scripts, you have even greater control over the execution of LDAP searches and the processing of attribute values and search results. You can combine, manipulate, and test attribute values, and define conditional logic to select which searches to execute.

Uses for LDAP authentication scripts include:

- Modifying the username and retrying the LDAP search in the case that the initial search returns no result from the repository.
- Selecting a RADIUS response profile for the user based on attributes returned from the LDAP server.
- Reformatting the LDAP result data before assigning values to the RADIUS response.
- Using the results from prior LDAP searches to select subsequent LDAP searches to execute.

For more details, see Chapter 4, “Creating LDAP Scripts” on page 23.

About Realm Selection Scripts

A realm is a collection of authentication methods that Steel-Belted Radius invokes to process a RADIUS request. When an authentication request is received, Steel-Belted Radius uses the username, selected RADIUS attributes, or other properties of the request to determine which realm will handle the request. The selected realm can be a proxy realm, a directed realm, or the default realm (if no explicit realm is selected).

Realm selection is performed both by the Steel-Belted Radius server core and during inner authentication by tunneled authentication plug-ins. Five built-in realm selection methods, plus the scripted method, are supported. Using realm selection scripts, you can define programmed logic to select the realm for processing each RADIUS request. Realm selection scripts may retrieve RADIUS request attributes, query external SQL or LDAP servers, or invoke any of the built-in realm selection methods.

Uses for realm selection scripts include:

- Querying multiple LDAP servers to look up the realm name for a specific user.
- Combining multiple RADIUS request attributes to form a SQL database key for retrieving the realm name.
- Changing the authentication username.
- Setting a profile to be applied to the RADIUS response once the user is authenticated.

For more details, see Chapter 5, “Creating Realm Selection Scripts” on page 29.

About Attribute Filter Scripts

Steel-Belted Radius uses attribute filters to allow, exclude, add, or modify attribute values in the RADIUS response and request packets. Attribute filters are also used to transfer attribute values in and out of the inner methods of tunneled authentication plugins. Attribute filters are defined by name using the SBR Administrator and are referred to throughout the server configuration.

Unscripted or static attribute filters use simple, fixed rules for manipulating RADIUS attributes. In contrast, scripted attribute filters enable you to specify detailed algorithms to read, write, modify, and delete request and response attribute values. You can query external SQL or LDAP servers and execute static attribute filters by name from your attribute filter scripts.

Uses for attribute filter scripts include:

- Using an LDAP query to select a static attribute filter to execute.
- Adding or removing selected values from a multi-valued attribute.
- Editing the values of string attributes.
- Accepting or rejecting requests based on mathematical calculations on numeric attribute values.

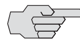
For more details, see Chapter 6, “Creating Attribute Filter Scripts” on page 35.

About JavaScript

Steel-Belted Radius uses the open-source SpiderMonkey JavaScript engine from the Mozilla Foundation to compile and execute scripts. SpiderMonkey is an implementation of JavaScript 1.5, which adheres to the international ECMAScript (ECMA-262) standard. JavaScript's ease of use and powerful syntax have led to its wide adoption as an industry-standard embedded scripting language.

For more information about SpiderMonkey and links to JavaScript references, see the following Website:

http://www.mozilla.org/js/spidermonkey.

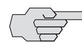


NOTE: The JavaScript compiler and interpreter are components of Steel-Belted Radius and do not depend on the browser or JVM installed on your machine.

Chapter 2

Creating Scripts

This chapter describes the common elements of writing scripts for Steel-Belted Radius, including the JavaScript initialization file configuration settings. A simple example shows the concepts explained in this chapter.



NOTE: Changing passwords through scripting or filters is not supported.

Script Development Steps

To create and deploy a script on the Steel-Belted Radius server, you would typically use the following steps:

1. Create a JavaScript initialization file containing the script statements and runtime settings and save it in the appropriate directory under the Steel-Belted Radius installation root directory.

2. Define SQL or LDAP data accessor .gen files as required by your script.

3. Run the scriptcheck utility to verify your script syntax.

4. Declare your script using the Script keyword in the appropriate Steel-Belted Radius initialization file(s).

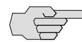
   NOTE: This step is not required for LDAP authentication plugin scripts.

5. Start the Steel-Belted Radius server and check the server log to verify successful loading and compilation of the script.

6. Send a RADIUS request to the server and check the server log for debug messages and trace output from your script.

JavaScript Initialization Files

Steel-Belted Radius scripts are contained in JavaScript initialization (.jsi) files, which are similar in format to other Steel-Belted Radius configuration files. Each .jsi file consists of a number of section headers and associated configuration settings. The JavaScript text itself appears in a separate [Script] section within the file. With minor exceptions, the .jsi file headers and settings are the same for all script types.



NOTE: Script settings for the LDAP authentication plug-in are embedded directly in the ldapauth.aut file. Except where noted, this guide applies equally to both ldapauth.aut and .jsi file types. For information about configuring other LDAP authentication plug-in settings, see the "LDAP Authentication Files" chapter in the *Steel-Belted Radius Reference Guide*.

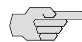
Each .jsi file can contain the following sections:

- [Settings]
- [Script]
- [ScriptTrace] (*optional*)
- [Failure] (*optional*)

[Settings] Section

The [Settings] section contains parameters that control logging and debugging of your script.

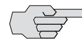
- The LogLevel parameter sets the default level assigned to log messages produced by calls to the SbrWriteToLog() and SbrTrace() API functions. To determine if the message will appear in the log, Steel-Belted Radius compares the message log level to the server log level (configured by the LogLevel parameter in radius.ini). If the server log level is greater than or equal to the message log level, the message will be written to the Steel-Belted Radius log. If server log level is less than the message log level, the message will not be written to the Steel-Belted Radius log.



NOTE: You can override the script file LogLevel parameter when calling SbrWriteToLog() and SbrTrace() using the optional LogLevel function argument. For more details, see "Writing Messages to the Server Log" on page 17 and "Script Tracing" on page 17.

- The ScriptTraceLevel parameter controls the amount of line-by-line debugging information that is produced automatically or under program control by the script. At the lowest level, no tracing is performed, and at the highest level, a trace is written to the log for every JavaScript statement that is executed. See Chapter 3, "Debugging Scripts" on page 17 for more information about script debugging.

- The MaxScriptSteps parameter limits the number of branch callbacks that a script can perform in a single invocation. A branch callback is a backwards branch in the script code (for example, what occurs in a for loop), or a return from a function call. If the limit is reached, the script is automatically terminated with a runtime exception.



NOTE: The implementation of MaxScriptSteps has changed. In the pre-6.0 release of Steel-Belted Radius, the MaxScriptSteps counter applied to the total number of JavaScript statements, not branch callbacks, executed by the script.

Table 3: [Settings] Section Parameters

Parameter	Function
LogLevel	Specifies the default log level at which messages are produced by calls to SbrWriteToLog() and SbrTrace(). The value must be less than or equal to the LogLevel value in the radius.ini file for messages to appear. The parameter can be overridden by supplying a LogLevel argument in the function calls. Default value is 0.
ScriptTraceLevel	Controls the generation of line-by-line script trace information in the log. ■ At Level 0, no traces are logged. ■ At Level 1, traces are logged only when the SbrTrace() function is executed by the script. ■ At Level 2, a trace is generated for every line executed by the script. Default value is 0.
MaxScriptSteps	Limits the number of branch callbacks that can be executed during a single script invocation. If the limit is reached, the script automatically terminates with a runtime exception. Default value is 10000.

[Script] Section

The [Script] section contains the body of your script. Unlike other configuration file sections, where parameters appear on individual lines, the script is entered as multi-line blocks of text. The script is processed until a line is encountered that begins with a left bracket ("[") or the end of the file is reached.

The following example shows a simple [Script] section containing code that writes a message to the server log.

```
[Script]
// Define a function that writes its arguments to the log.
function writeLog(message) {
   SbrWriteToLog("The message is: " + message);
}

// Call the log function.
var msg = "Hello, world";
writeLog(msg);

// Return successfully.
return SCRIPT_RET_SUCCESS;
```

[ScriptTrace] Section

The [ScriptTrace] section is optional. You can use the [ScriptTrace] section to select specific data values to print in the script trace logs. If you enable script tracing but do not specify any parameters in the [ScriptTrace] section, the trace frames will contain statement and line number information but no script data values.

Each line in the [ScriptTrace] section specifies a type string and an argument. The type string selects the type of data value to be traced and the argument specifies its name.

The following types are supported:

- var—The argument is the name of a local or global script variable.
- attr—The argument is the name of an LDAP variable table entry (ldapauth.aut only).

Table 4: [ScriptTrace] Section Parameters

Parameter	Function
var	Declares the name of a local or global JavaScript variable that will appear in script trace logs.
attr	Declares the name of an LDAP variable table entry that will appear in script trace logs (ldapauth.aut only).

```
[ScriptTrace]
var = count
var = userid
attr = User-Name (ldapauth.aut only)
attr = Service-Type (ldapauth.aut only)
```

In this example, the identifiers count and userid refer to the JavaScript variables in the script execution context. The identifiers User-Name and Service-Type refer to entries in the LDAP variable tables and will take effect only when declared in an ldapauth.aut script file.

[Failure] Section

The [Failure] section is optional. It specifies a string value that is ultimately returned by a script if the script first returns SCRIPT_RET_FAILURE.

- For LDAP authentication scripts, the value of the [Failure] section has a more complex interpretation. For details about the [Failure] section of the ldapauth.aut file, see the "LDAP Authentication Files" chapter in the *Steel-Belted Radius Reference Guide.*
- For realm selection scripts, the value of the [Failure] section specifies the name of the realm to be returned if the script execution fails.
- For attribute filter scripts, the value of the [Failure] section specifies the name of a static attribute filter to execute if the script execution fails.

Writing Steel-Belted Radius Scripts in JavaScript

This section provides general information and guidelines for writing Steel-Belted Radius scripts in JavaScript. For descriptions of the Steel-Belted Radius API functions available to LDAP, realm selection, and attribute filter scripts, see Chapter 9, “Script Reference” on page 77.

Programming in JavaScript

The Steel-Belted Radius script engine supports all of the JavaScript 1.5 (ECMA-262) language features. You can use any legal JavaScript syntax in your scripts and invoke the standard global object function and attributes, such as Math, String, and Date.

The JavaScript engine runs entirely within the Steel-Belted Radius server and is not associated with any Web browser. Browser-specific data objects and JavaScript language extensions are not supported by Steel-Belted Radius.

For in-depth information on JavaScript programming, see the official ECMAScript standard documentation at the following URL:

http://www.ecma-international.org/publications/standards/Ecma-262.htm

Hidden Wrapper Function

Before Steel-Belted Radius compiles your script, it wraps the [Script] section statements in a JavaScript function named _sbrScriptMain_. When the server executes the script, the script invokes this function first. There are no arguments to the function call.

The hidden wrapper function enables your scripts to return result values to the server. This is required because the JavaScript language does not support return statements from the global execution context. For convenience, the hidden function call does not appear in script traces, and line numbers in script traces and error messages are adjusted to refer to the exact lines of your JavaScript initialization files.

With few exceptions involving advanced JavaScript programming, the existence of the hidden wrapper function is transparent to your scripts.

Script Return Values

Steel-Belted Radius uses the script return value to determine what action to take on the pending RADIUS request once the script finishes executing. The script type determines how the return value is interpreted by the server. A legal return value must be:

- A pre-defined return code such as SCRIPT_RET_SUCCESS
- A text string
- The JavaScript null object

The SCRIPT_RET_SUCCESS and the SCRIPT_RET_FAILURE codes are defined in the global object for all script types. Returning SCRIPT_RET_SUCCESS indicates to Steel-Belted Radius that script execution completed normally.

Returning SCRIPT_RET_FAILURE indicates that an unexpected error occurred during script execution (for example, a database search returned invalid results). If a script returns SCRIPT_RET_FAILURE and a [Failure] string is defined for that script, that string is returned as the script result. Otherwise, an error message appears in the server log and the pending RADIUS request will be rejected.

Result strings are used by realm selection scripts to return the name of the selected realm. Attribute filter scripts use result strings to return the name of a static filter to execute.

Returning JavaScript null is equivalent to returning SCRIPT_RET_SUCCESS.

The processing of return codes by LDAP authentication scripts is more complicated than for other script types. For information about choosing the LDAP authentication script return code, see "Choosing the Return Code" on page 27.

Initializing Reusable Data Objects

In JavaScript, when you declare a variable using the var keyword, that variable is allocated temporarily on the program stack. When your script returns, the variable goes out of scope and is marked for garbage collection by the script engine. However, variables declared without a var keyword become properties of the script engine's Global object and persist across invocations of the script.

To avoid allocating and deallocating data objects each time a script runs, you can create initialization blocks to allocate reusable global objects the first time a script runs. These objects remain available to use in subsequent executions of the script. The following code example shows this technique.

```
// Initialization block
if (!this.initialized) {
    // Create persistent data as global object properties.
    filter = new AttributeFilter();
    accessor = new DataAccessor();
    someString = "This is a persistent string"

    // Set initialized flag for next time.
    initialized = true;
}
else {
    // /Clear left-over data from prior request.
    accessor.Clear();
}
```

In this example, the variable initialized is a global flag that is tested each time the script runs. If initialized is not set, the persistent data objects are allocated and the flag is set. On subsequent calls to the script, the initialization block is skipped but a call is made to clear the prior contents of the Data Accessor.

General Recommendations

The following lists additional recommendations for developing Steel-Belted Radius scripts.

- JavaScript comments begin with // or /*. Within the [Script] section, lines starting with a semicolon (;) are not treated as comments. In the following example, compilation will fail with a syntax error because the commented-out [Failure] section is passed to the JavaScript compiler.

  ```
  [Script]
  SbrWriteToLog("hello, world");
  return SCRIPT_RET_SUCCESS;

  ;[Failure]
  ;SomeString
  ```

- Thoroughly test all scripts for speed and monitor the performance of Steel-Belted Radius after you deploy a new script. In general, the performance impact of a script on the server is directly related to its complexity.

- After you modify a realm selection or attribute filter script, you can reload it by executing a platform specific HUP command. It is not necessary to restart the server. You cannot use the HUP command to reload LDAP scripts.

Saving the Script File

You must save realm selection and attribute filter script (.jsi) files in the scripts subdirectory of the Steel-Belted Radius service directory. When you refer to a .jsi file using the Script keyword in a Steel-Belted Radius configuration file, the script engine automatically searches the scripts subdirectory for the specified file. If the file is not found, an error message appears in the log and the associated script functionality is disabled.

You must save the ldapauth.aut file in the RADIUS service directory, whether or not LDAP scripting is enabled.

Installing the JavaScript Upgrade License

Before you can use the scripting features of the Global Enterprise Edition (GEE) and Service Provider Edition (SPE) of Steel-Belted Radius, you must obtain a JavaScript Feature Upgrade license and install it on your server. The scripting features include all three script types (LDAP, realm selection, and attribute filter) and the scriptcheck syntax checking utility.

If you are using the Enterprise Edition (EE) of Steel-Belted Radius, there is no JavaScript Feature Upgrade license and you cannot use any of the JavaScript features. For more information about Steel-Belted Radius licenses and license key installation, see the “About Steel-Belted Radius” and “Using SBR Administrator” chapters of the *Steel-Belted Radius Administration Guide*.

Sample Script

The following is an example of a simple realm selection script and the configuration settings to enable it to run. To test this script, copy the following lines to a text file named SampleScript.jsi in the scripts subdirectory of the RADIUS service directory.

```
[Settings]
LogLevel = 2
ScriptTraceLevel = 1

[Script]
// Print a message in the SBR log.
SbrWriteToLog("Executing SampleScript.jsi");

// Allocate a new Realm Selector object.
var selector = new RealmSelector();

// Invoke the built-in Suffix realm selection method to obtain
// the realm for the request.
var realm = selector.Execute("suffix");
SbrWriteToLog("Suffix method returned '" + realm + "'");

// Print a trace frame to the log.
SbrTrace();

//Return the realm name as the script result.
return realm;

[ScriptTrace]
var = realm

[Failure]
DefaultRealm
```

Next, edit the proxy.ini file and add the Script declaration to the [Processing] section as shown:

```
[Processing]
Suffix
Prefix
DNIS
Attribute-Mapping
Script SampleScript
```

NOTE: Use only the script file base name only when configuring the Script setting. If you specify the .jsi extension, Steel-Belted Radius will fail to load the file.

You must also set ExtendedProxy = 1 in the [Configuration] section of radius.ini.

After starting Steel-Belted Radius, you will see log messages indicating that the script successfully loaded and is ready to run.

```
Loading script from file ‘C\radius\Service\scripts\SampleScript’
Extended Proxy: Enabled precedence 4 processing for Script SampleScript
```

When a RADIUS request is received, the script executes, and messages similar to the following appear in the log.

```
Executing ‘SampleScript’
Suffix method returned ‘realm1’
*** Script Trace (C:\radius\Service\scripts\SampleScript)
    (line 41) SbrTrace();
    realm=realm1
CreateRequestEx: using virtual realm realm1 for authentication.
```

NOTE: The actual realm name returned by the script depends on the Steel-Belted Radius configuration and the suffix decoration of the username specified in the RADIUS request.

Chapter 3

Debugging Scripts

This chapter describes the tools and techniques you use to debug your scripts and monitor their runtime execution. Script debugging features include:

- The SbrWriteToLog() function to write messages to the server log.
- The SbrTrace() function and ScriptTraceLevel setting to enable runtime debug tracing.
- The scriptcheck utility to verify script syntax prior to deployment on the server.

Writing Messages to the Server Log

To write text messages to the Steel-Belted Radius log, call the SbrWriteToLog() function. The log message appears if the server log level is greater than or equal to the message log level. The server log level is determined by the LogLevel parameter of the [Configuration] section in radius.ini. The message log level is determined by the LogLevel parameter in the [Settings] section of the script file. For more information, see “[Settings] Section” on page 8.

You can also use the optional LogLevel argument to specify the log level explicitly in the call to SbrWriteToLog():

```
SbrWriteToLog(“This is an INFORMATIONAL level message”, 1);
```

In this example, the message log level is set to the value of 1. The message will appear in the log if the server log level is 1 or greater.

For more information about the SbrWriteToLog() function, see “Logging and Diagnostic Methods” on page 79. For more information about the server LogLevel setting, see the “radius.ini File” chapter in the *Steel-Belted Radius Reference Guide*.

Script Tracing

You can use the script trace feature to set line-by-line debugging of your script as it executes. A script trace is a block of program status information written to the Steel-Belted Radius server log file prior to the execution of a JavaScript statement. Information in each script trace frame includes:

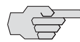
- The name of the .jsi or .aut file in which the script is defined

- The current line number
- The text of the JavaScript statement at that line number
- Listings of selected program variable values
- Listings of selected RADIUS attribute values (for LDAP scripts only)

You define the names of program variables and RADIUS attributes to be displayed in script traces by entering them in the [ScriptTrace] section of the JavaScript initialization file. For more details, see "[ScriptTrace] Section" on page 10.

You have two options to enable tracing of your scripts.

- Manual tracing—You can set the ScriptTraceLevel parameter in the [Settings] section of the script file to 1 and call the SbrTrace() function from within your script. This causes a single script trace frame to appear in the log from the point in your script where the SbrTrace() function was called.
- Automatic tracing—You can set the ScriptTraceLevel parameter in the [Settings] section of the script file to 2 to enable automatic tracing. In this mode, a script trace is performed every time that a JavaScript statement is executed by your script.



NOTE: Enabling script tracing for a single script file has a performance impact on all scripts running on Steel-Belted Radius, whether or not script tracing is enabled for those files. For this reason and because of large volume of log information produced, the use of script tracing is not recommended for production environments.

The following example lists a small script and a portion of the automatic script trace generated from it.

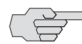
```
[Script]
var a = 1;
var s = "Hello";
return SBR_RET_SUCCESS;

[ScriptTrace]
attr = User-Name
var = a
var = s
    ⋮

*** Script Trace (c:\radius\service\ldapauth.aut)
  (line 1) var a = 1;
  User-Name = testuser
  a = <not found>
  s = <not found>
*** Script Trace (c:\radius\service\ldapauth.aut)
  (line 2) var s = "Hello";
  User-Name = testuser
  a = 1
  s = <not found>
*** Script Trace (c:\radius\service\ldapauth.aut)
```

```
(line 3) return SBR_RET_SUCCESS;
User-Name = testuser
a = 1
s = Hello
   ⋮
```

Note that traces are produced just prior to execution of the JavaScript statement referenced in the trace. For example, the value of variable a is not reflected in the trace on line 1, but appears in the trace on line 2, after the assignment statement has executed. If a variable or attribute has not yet been assigned, or if a variable is out of scope at the time of the trace, the value will be displayed in the log as <not found>.



NOTE: The attr keyword is supported only for LDAP authentication scripts. If this example is configured as a realm selection or attribute filter script, the attr = User-Name entry in the [ScriptTrace] section is ignored.

Script traces appear in the log if the server log level is greater than or equal to the trace log level. The server log level is determined by the LogLevel parameter of the [Configuration] section in radius.ini. The trace log level is determined by the LogLevel parameter in the [Settings] section of the script file. For more information, see "[Settings] Section" on page 8.

You can also use the optional LogLevel argument to specify the log level explicitly in the call to SbrTrace():

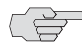
```
SbrTrace(0); //Specify production log level
```

In this example, the argument trace will appear unconditionally in the server log regardless of the script file LogLevel setting.

For more information about the SbrTrace function, see "Logging and Diagnostic Methods" on page 79. For more information about the server LogLevel setting, see the "radius.ini File" chapter in the *Steel-Belted Radius Reference Guide*.

scriptcheck Utility

The scriptcheck utility is a command-line application that enables you to check the Steel-Belted Radius JavaScript configuration files for syntax errors. .



NOTE: The scriptcheck utility verifies that your script is syntactically correct. The scriptcheck utility does not guarantee that your script is free of runtime errors or produces correct results. If your script does not appear to be working properly, review the Steel-Belted Radius log for error messages and enable script tracing to diagnose the problem.

Unpacking the scriptcheck Utility

The scriptcheck utility and its required shared libraries are packaged in three compressed archives, one for each of these supported platforms:

- zip compressed for Windows - scriptcheck.*version*.win.zip
- gzip compressed tar file for Solaris - scriptcheck.*version*.sol.tgz
- gzip compressed tar file for Linux - scriptcheck.*version*.lin.tgz

The archives are located in the Support_Files/scriptcheck directory on the Steel-Belted Radius installation CD-ROM.

You can copy the appropriate scriptcheck executable version to any convenient location and run it there, provided that you also copy the radius.lic file (see "Installing the JavaScript Upgrade License" on page 20) to the same location.

Before you can run the scriptcheck utility, you must unpack the correct version of the archive for your platform into a destination folder or directory:

- Windows—Use a suitable application (such as Winzip) to extract the contents of the archive to the destination folder. The files to extract are:
  - scriptcheck.exe
  - libnspr4.dll
  - js32.dll
  - README.SCRIPTCHECK
- Solaris and Linux—With the archive in the destination directory, enter the following command to extract its contents:

  % gunzip -c scriptcheck.*version*.*platform*.tar.gz | tar xvf -

  The files to extract are:
  - scriptcheck
  - libnspr4.so
  - libjs.so
  - README.SCRIPTCHECK

Installing the JavaScript Upgrade License

Before you can run the scriptcheck utility, you must have the radius.lic file, which contains your JavaScript upgrade license, in the directory where you run the utility. If the radius.lic file is not present in the working directory, the program prints scriptcheck: can't open license file (radius.lic). If the file is present but the license isn't correct, the program prints scriptcheck: not licensed for JavaScript.

- If you install the JavaScript upgrade license using SBR Administrator, the radius.lic file is updated automatically in the *radiusdir* home directory. After that, you can run scriptcheck in that directory with no additional configuration. To run the scriptcheck utility in another location, place a copy of your radius.lic file in the same directory.
- You can use a text editor to create a radius.lic file and enter the JavaScript upgrade license string into the file manually.

Running the scriptcheck Utility

To run the scriptcheck utility and verify scripts, follow the steps for your platform:

- Windows:
  a. Open a command shell (cmd) and change to the scriptcheck directory.
  b. Execute the scriptcheck command by specifying the name of the script as the argument. For example, the following command validates the script contained in the myscript.jsi file:

```
C:\scriptcheck>scriptcheck myscript.jsi
Loading script from file 'myscript.jsi'
Scriptcheck: script file 'myscript.jsi' compiled successfully
```

- Solaris and Linux:
  a. Before running the scriptcheck utility for Solaris and Linux, set the LD_LIBRARY_PATH environment variable to point to the location where the shared object files are installed.
  b. Open a command shell (cmd) and change to the scriptcheck directory.
  c. Execute the scriptcheck command by specifying the name of the script as the argument. For example, the following command validates the script contained in the myscript.jsi file:

```
% cd scriptcheck
% setenv LD_LIBRARY_PATH .
% scriptcheck myscript.jsi
Loading script from file 'myscript.jsi'
Scriptcheck: script file 'myscript.jsi' compiled successfully
```

When the scriptcheck utility runs, it loads the [Script] section in the specified .jsi file and uses the JavaScript interpreter to compile the script text. Any error messages produced during script compilation are printed on the console. You can then correct the errors and rerun scriptcheck to verify that the script compiles correctly.

Chapter 4

Creating LDAP Scripts

LDAP

Many companies use LDAP directory servers to store user authentication and authorization information. Steel-Belted Radius can process authentication requests against records stored in one or more external LDAP databases.

LDAP scripting is used when more sophisticated decision logic or attribute manipulation is required than can be implemented using unscripted searches. Incorporating JavaScript into the Steel-Belted Radius ldapauth.aut file gives you much greater flexibility in the processing of LDAP authentication queries. Scripted authentication enables a level of control comparable to SQL stored procedures.

For example, LDAP scripts can combine data from several LDAP queries and analyze the results to determine which query to invoke next. LDAP scripts can evaluate loops and complicated if-then-else logic, build up RADIUS attribute value strings from scratch, and write status messages to the Steel-Belted Radius log.

LDAP Request Life Cycle

Steel-Belted Radius performs the following steps in response to an LDAP authentication request for both scripted and non-scripted configurations.

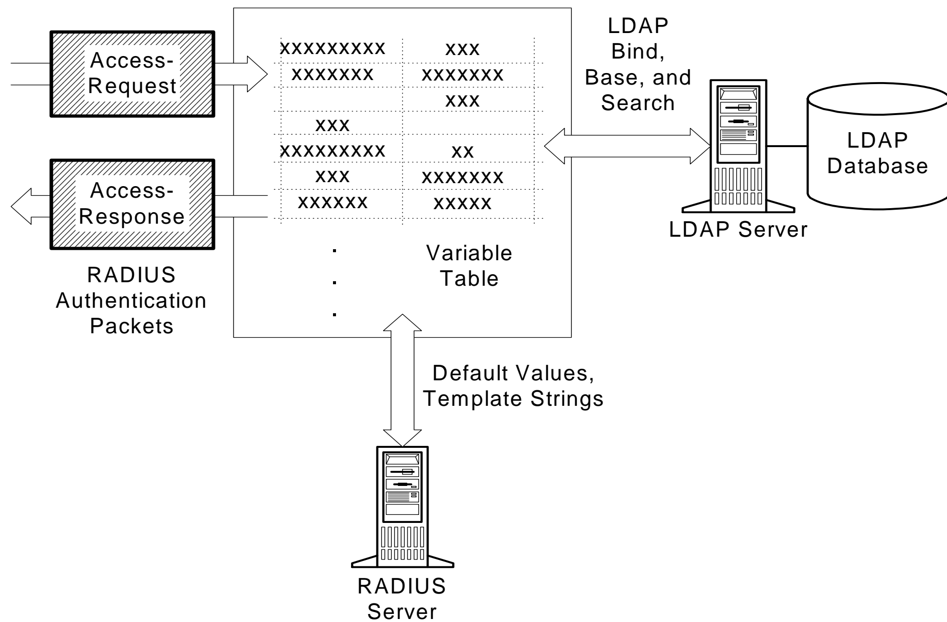
1. At the beginning of each LDAP authentication request, Steel-Belted Radius creates a variable table to map RADIUS access-request attributes to LDAP attributes for use in LDAP Bind, Base, and Search strings. The [Request] section of the LDAP plug-in configuration file is used to select which attributes are extracted from the incoming request and placed in the variable table.

2. Steel-Belted Radius performs one or more LDAP searches. Parameters for each search are given in the Search/name] sections of the configuration file. After a search is performed, selected attributes are copied from the LDAP response and placed in the variable table.

3. Steel-Belted Radius uses the [Response] section to select information from the variable table to be returned to the RADIUS client in the RADIUS response packet.

Figure 1 shows how the LDAP variable table is populated with information coming from a RADIUS access-request message, default values, and the results of LDAP Bind, Base, and Search requests. The information in the variable table is then used to format the access-response packet that is returned to the RADIUS client.

Figure 1: Role of the Variable Table in LDAP Authentication



Unscripted LDAP Searches

Scripting is not required for basic applications of LDAP authentication. In unscripted configurations, search parameters such as base Distinguished Names (DNs), filter strings, and attribute maps are configured in the ldapauth.aut file. Using the OnFound and OnNotFound settings of the [Search/name] sections, you can configure a decision tree in which the result of one LDAP query (Found or Not Found) determines whether another query is executed or the final authentication decision is returned to Steel-Belted Radius. The basic query tree provides sufficient control to meet the needs of many LDAP authentication applications. Figure 2 shows a sample query tree using unscripted branching.

Figure 2: Query Tree with Unscripted Branching

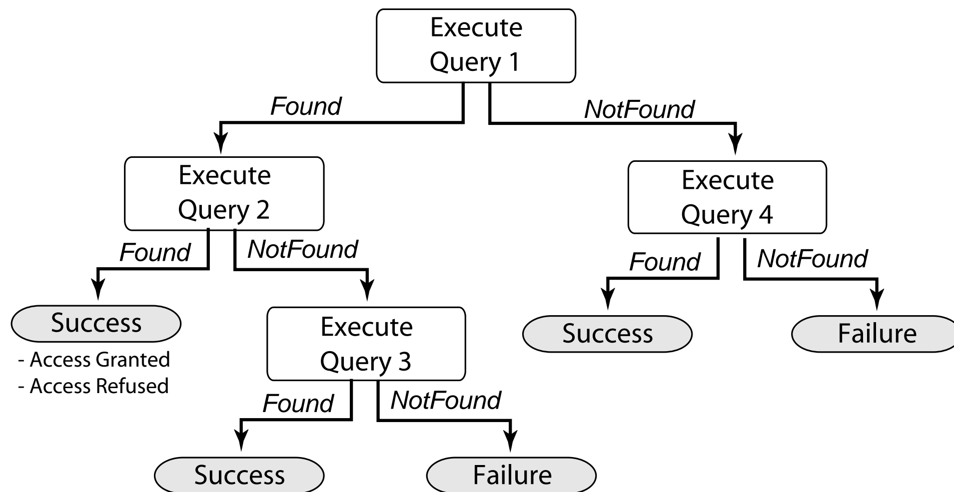
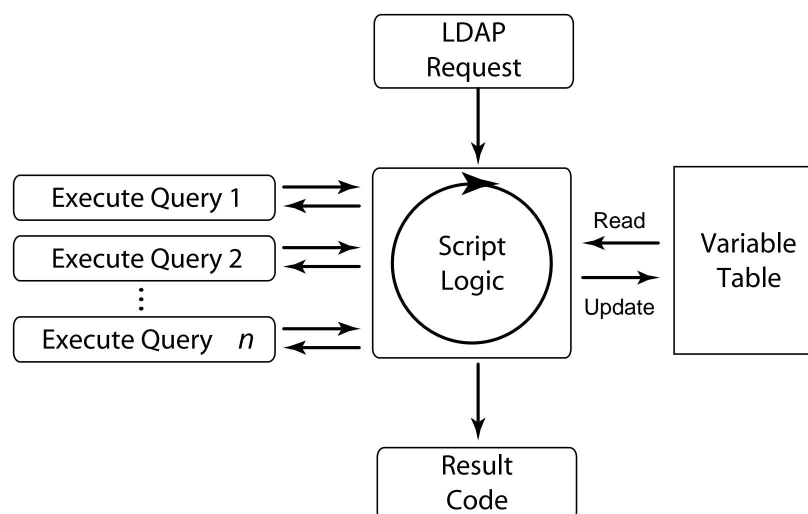


Figure 3 shows the data flow involved in a scripted query. Instead of following a rigid branch structure, the request is processed according to the logic of the LDAP script, which might be arbitrarily complex. The script executes one or more LDAP queries, computes intermediate results from the return values, updates the LDAP variable table, and possibly executes additional queries against the LDAP server. Once the script has completed processing the request and made an authentication decision, it returns a result code to the plug-in.

Figure 3: Scripted Query Data Flow



LDAP Script Basics

To configure LDAP scripting, you add JavaScript instructions to the [Script] section of the ldapauth.aut file. You can perform the following operations in your LDAP scripts:

- Get, set, and reset values of variables stored in the LDAP variable table
- Invoke LDAP queries defined in the [Search/name] sections of the ldapauth.aut file
- Write diagnostic messages and script traces to the Steel-Belted Radius log
- Evaluate arbitrary program logic coded in your script
- Exit the script and return a result code string to the LDAP plugin

When Steel-Belted Radius starts, it reads the text of the section from ldapauth.aut and passes it as a block to the JavaScript interpreter, which compiles it into bytecodes. The bytecodes are stored for execution during subsequent LDAP authentication requests. If syntax errors are detected in the JavaScript text, the script does not compile and the LDAP authentication plugin is disabled. Any error messages generated during script compilation appear in the Steel-Belted Radius log file.

You can use the scriptcheck utility to check your LDAP scripts for syntax errors without having to start Steel-Belted Radius. For more information, see "scriptcheck Utility" on page 19.

Working with the Variable Table

You configure the variable table for scripting the same way you do for unscripted configurations. Input RADIUS attributes that the script manipulates must be identified in the [Request] section of the ldapauth.aut file. Output RADIUS attributes that the script manipulates must be identified in the [Response] section of the ldapauth.aut file.

The LdapVariables object is available to your script for manipulating attributes in the variable table. The LdapVariables object exposes three methods that scripts can call:

- LdapVariables.Get() retrieves the current value or values for a variable stored in the LDAP variable table.
- LdapVariables.Add() creates a new variable or adds a value to an existing variable.
- LdapVariables.Reset() deletes all of the values of the specified variable.

Invoking LDAP Queries

Any query defined in a [Search/name] section of ldapauth.aut can be invoked programmatically by an LDAP script. Use the Ldap.Search() method to invoke the query, giving the name of the query as the argument to the method.

As with unscripted searches, you can identify a set of LDAP attributes to be extracted from the LDAP response and placed in the variable table. You do this by creating an [Attributes/name] section in the ldapauth.aut file and specifying this section with the Attributes parameter in the query definition.

For more information about LDAP attributes, refer to the “LDAP Authentication Files” chapter of the *Steel-Belted Radius Reference Guide*.

Writing to the Steel-Belted Radius Log

Use the SbrWriteToLog() function to insert diagnostic or informational text strings into the Steel-Belted Radius log file. You can use the optional level argument to control the log level visibility of your message.

Use the SbrTrace() function to display trace information about your script in the Steel-Belted Radius log.

For more details about these functions, see “Logging and Diagnostic Methods” on page 79.

Choosing the Return Code

When a script finishes running, it sends a return value back to the LDAP plugin. Depending on the return value and the state of the request, the plugin can do one of several things:

- It can make an authentication decision and send that result directly to Steel-Belted Radius, ending the processing of that request by the plugin.
- It can re-execute the script against a different LDAP server and process the new return value when the script is finished.
- It can perform failure processing and return a result to Steel-Belted Radius based on the [Failure] section in ldapauth.aut.

For information about configuring other LDAP authentication plugin settings, see the “LDAP Authentication Files” chapter in the *Steel-Belted Radius Reference Guide.*

An LDAP script may execute several times while handling a single authentication request but eventually the LDAP plugin must make an authentication decision and send it back to the Steel-Belted Radius server. It is important for the script programmer to understand exactly how the script return code affects the LDAP plugin and the authentication decision.

Script Return Codes

You specify the script return code as an argument to the JavaScript return statement. The return code must be one of the following global constants.

SCRIPT_RET_SUCCESS

The SCRIPT_RET_SUCCESS code indicates to the LDAP plugin that the user has been authenticated and should be accepted. The plugin finishes processing the request and sends an accept decision to the Steel-Belted Radius core.

SCRIPT_RET_DO_NOT_AUTHENTICATE

The SCRIPT_RET_DO_NOT_AUTHENTICATE code indicates to the LDAP plugin that a hard reject should be performed by the server. The plugin finishes processing the request and sends a reject decision to the Steel-Belted Radius core.

SCRIPT_RET_TRY_NEXT_AUTH_METHOD

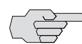
The SCRIPT_RET_TRY_NEXT_AUTH_METHOD code indicates that the LDAP plugin should stop processing the request and ask Steel-Belted Radius to try the next authentication method without immediately rejecting the user. Last resort processing is not performed.

SCRIPT_RET_NOT_AUTHENTICATED

The SCRIPT_RET_NOT_AUTHENTICATED code indicates to the LDAP plugin that the script could not authenticate the user. If a last resort server is defined, the LDAP plugin will re-execute the script against that server. If there is no last resort server, this return code has the same effect as SCRIPT_RET_TRY_NEXT_AUTH_METHOD.

SCRIPT_RET_FAILURE

The SCRIPT_RET_FAILURE code indicates to the LDAP plugin that the a communication failure with the LDAP server occurred. The plugin should re-execute the script against the next LDAP server in the configuration, if defined. If only one server is defined or the last server has already been tried, the LDAP plugin should process the [Failure] section to determine the final result. If there is no [Failure] section, this return code has the same effect as SCRIPT_RET_TRY_NEXT_AUTH_METHOD.



NOTE: The Steel-Belted Radius pre-6.0 release SBR_RET_*xxx* codes have been deprecated and replaced with the new SCRIPT_RET_*xxx* codes. The SBR_RET_*xxx* codes are supported for backward compatibility.

For a list of the LDAP script return codes, see “LDAP Script Return Codes” on page 95.

Chapter 5

Creating Realm Selection Scripts

Steel-Belted Radius executes built-in or scripted realm selection methods to determine the authentication realm for processing a request. For built-in methods, you specify the methods and their order of execution in the [Processing] section of the proxy.ini configuration file. You specify matching rules in the [Realms] and [Directed] sections. For more information about the proxy.ini configuration file, see the "proxi.ini File" section of Chapter 7, "Realm Configuration Files," in the *Steel-Belted Radius Reference Guide*.

For scripted realm selection, use the script setting in proxy.ini to declare the name of a JavaScript initialization (.jsi) file. If the script setting appears anywhere in the [Processing] section, Steel-Belted Radius executes the realm selection script first, before trying any other built-in methods. If the script returns a valid realm name, Steel-Belted Radius sends the current request to that realm for processing. If the script returns the code SCRIPT_RET_SUCCESS instead of a realm name, Steel-Belted Radius invokes the remaining methods in the [Processing] section to try to determine the realm for the request.

You can also specify a realm selection script for the inner authentication setting of tunneled authentication methods using SBR Administrator.

Realm selection scripts are useful when your realm selection strategy is too complex to be implemented using basic matching rules. Realm selection scripts can perform any of the following actions:

- Retrieve RADIUS request attribute and process their values.
- Execute program logic to determine the realm name.
- Execute built-in Steel-Belted Radius realm selection methods.
- Invoke SQL queries and LDAP searches, and process the results.
- Specify a profile to be merged with the response.
- Change the authentication username.

Realm Selection Script Functions

Realm selection scripts can execute any standard JavaScript statements and functions. Additionally, attribute filter scripts use the RealmSelector API to perform operations specific to realm selection. You must instantiate a new RealmSelector object instance and then use it to invoke these methods. For example:

```
var selector = new RealmSelector();
var realm = selector.Execute("suffix");
```

The following four RealmSelector methods are provided:

- new RealmSelector()—Creates a new RealmSelector() object instance.
- Execute()—Executes a built-in realm selection method and returns the resulting realm name.
- SetAuthUserName()—Sets the authentication username for the request.
- SetAuthProfile()—Sets the name of a profile to be merged with the result after the user is accepted.

Realm selection scripts can also execute the AttributeFilter.Get() method and any of the methods of the DataAccessor class.

For more details about the RealmSelector functions and methods, see "RealmSelector Object" on page 83.

Enabling Built-In Realm Selection Methods

You can call the RealmSelector.Execute() method from realm selection scripts to execute built-in Steel-Belted Radius realm selection methods. The argument to the Execute() method is one of the standard Steel-Belted Radius realm selection method names (suffix, prefix, dnis, attribute-mapping, or undecorated). The return value is a string containing the name of the realm selected by that method, using the matching rules defined in proxy.ini and applied to the username in the current RADIUS request. If no realm is selected, the Execute() method returns null.

The matching realm must be declared in proxy.ini and the corresponding .dir or .pro file must exist, or the Execute() method returns null. This is true even if the current username contains a valid realm name decoration for the specified realm selection method.

If you call the Execute() method with the suffix or prefix argument, you must make sure that the corresponding suffix or prefix realm selection methods are enabled or the result is always null. The suffix and prefix methods are enabled by default when you declare a script in the [Processing] section of proxy.ini. If you declare a realm selection script in the [Inner_Authentication] section of a tunneled authentication plug-in .aut file, the suffix and prefix methods are not enabled by default. To use the suffix and prefix methods with the Execute() function, you must either declare these methods explicitly, or declare a script in the [Processing] section of proxy.ini.

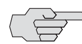
A realm selection script can call built-in realm selection methods at any time during its execution. Depending on its program logic, the script might return the realm name produced by the built-in method as its result or continue processing and return a different result.

Choosing the Return Code

When a realm selection script is finished executing, it returns a result code to the Steel-Belted Radius core. The return code is a string containing the name of the realm selected to process the current request. If the script is unable to select a realm, it might return a success or failure code instead of a realm string.

Valid return codes are:

- <*RealmName*>—Steel-Belted Radius should send the current request to the proxy or directed realm given by *RealmName.*
- SCRIPT_RET_SUCCESS—The realm selection script executed successfully, but did not select a realm. Steel-Belted Radius should try the remaining realm selection methods in the [Processing] section of the proxy.ini file or process the request in the default realm.
- SCRIPT_RET_FAILURE—The realm selection script failed to execute successfully. Steel-Belted Radius should terminate request processing and reject the user.

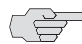


NOTE: Do not return the name of an undefined realm from a realm selection script, otherwise a runtime error will occur.

Configuring Realm Selection Scripts

You can configure realm selection scripts using either of the following two methods:

- For core realm selection—Core realm selection occurs first for all RADIUS requests. Add the script keyword to the [Processing] section of the proxy.ini file and specify the base filename of the realm selection script file as its argument.
- For tunneled authentication methods (PEAP, FAST, and TTLS)—Using the SBR Administrator tool, specify a realm selection script from the **Inner Authentication** tab of the Edit panel for the authentication method.



NOTE: For both realm selection script configuration methods, do not include the .jsi extension when you enter/specify the name of the script file.

Core Realm Selection Scripts

To configure core realm selection, you configure realm selection scripts in the [Processing] section of proxy.ini. All authentication requests go through this phase even if a second realm selection script is run from a tunneled authentication method.

When scripted realm selection is configured in proxy.ini from the [Processing] section, it runs before (and possibly replaces) all other realm selection methods.

[Processing] Section

If no [Processing] section is present in the proxy.ini file, then the standard methods are applied following this specific default order: Suffix, Prefix, DNIS, Attribute-mapping, and Undecorated.

If a [Processing] section is present in the proxy.ini file, it enables you to specify which realm selection rules are applied and the order in which they are applied.

```
[Processing]
RealmSelector
.
.
.
```

Table 5: proxy.ini [Processing] Syntax

Parameter	Description
RealmSelector	This can be one of six methods: attribute-mapping, DNIS, prefix, suffix, undecorated, or script *scriptname*. These are case-insensitive; except for Solaris/Linux, the script file rootname is case-sensitive. If a [Processing] section is present in the proxy.ini file, then the following special rules apply: ■ If no scripts are declared in the [Processing] section, then all methods are applied using the order in which they appear in the list. ■ If a script is declared anywhere within the [Processing] section, then the script *scriptname* method runs first. ■ If a script cannot determine the realm, it might return the null keyword, an empty string, or the SCRIPT_RET_SUCCESS return code. In this case, the remaining declared methods are applied using the order in which they appear in the list.

The following example shows a [Processing] section with a declared script:

```
[Processing]
Script scriptname
Suffix
Prefix
DNIS
Attribute-mapping
Undecorated
```

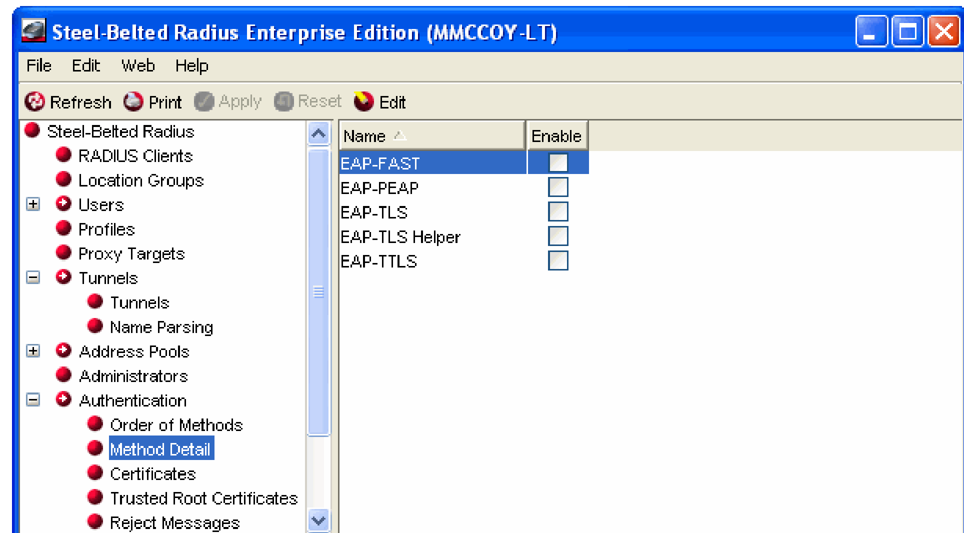
Matching rules for the methods are as defined in the [Realms] and [Directed] sections of proxy.ini.

Tunneled Authentication Plug-in Realm Selection Scripts

To specify a realm selection script for the inner authentication method of a tunneled authentication method, you must use the SBR Administrator tool.

1. From the SBR Administrator main window, select **Authentication > Method Detail**, as shown in Figure 4.

Figure 4: Authentication Methods Configuration

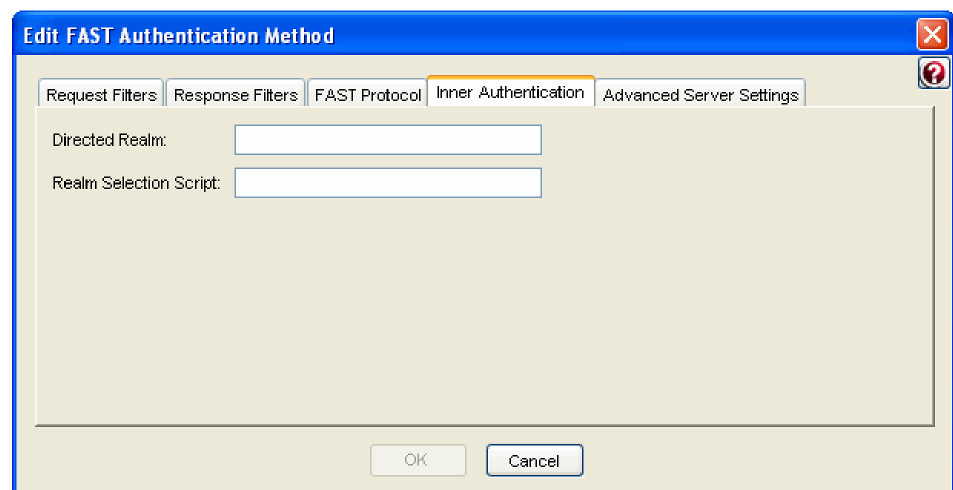


2. Select the check box next to the authentication method you want; for example EAP-FAST.

3. Double-click the selected authentication method to specify its settings. The dialog associated with your selection appears.

4. To specify a realm selection script with an authentication method (such as EAP-FAST), select the **Inner Authentication** tab as shown in Figure 5.

Figure 5: Inner Authentication Tab



5. To specify a realm for the authentication method, enter the name of the realm in the **Directed Realm** text box.

   To specify a realm selection script for the authentication method, enter the name of the script in the **Realm Selection Script** text box.

6. Select **OK**.

For more details about these dialogs and their options, see the *Steel-Belted Radius Administration Guide.*

Chapter 6

Creating Attribute Filter Scripts

You use attribute filter scripts to manipulate the values of attributes in the RADIUS request or response packets. Attribute filter scripts are executed any time a server core component or plug-in module invokes an attribute filter that is configured for scripting.

Steel-Belted Radius supports the following filters:

- Static attribute filters
- Scripted attribute filters

You declare both filters by name using the **Filters** panel of SBR Administrator.

- Static attribute filters—Specify fixed rules using keywords such as ALLOW, ADD, and EXCLUDE. The filter rules and the action of the filter are the same each time Steel-Belted Radius executes the filter.
- Scripted attribute filters—Declare the name of a JavaScript initialization (.jsi) file containing the script code for the filter. When the Script setting is present, Steel-Belted Radius ignores any other filter rules for that filter.

Because a scripted filter is defined with JavaScript, its behavior can change dynamically from one execution to the next. Scripted filters can perform any of the following actions:

- Get, add, delete, or replace attributes in the RADIUS request or response, depending on the filter context.
- Manipulate attribute values under program control.
- Execute static attribute filters by name.
- Invoke SQL queries and LDAP searches, and process the results.

Static and scripted attribute filters are referred to by name throughout the rest of the Steel-Belted Radius configuration. Externally, the two types of filters are equivalent and interchangeable, making it easy to switch between the two. Initially, you can configure Steel-Belted Radius with static filters and then change to scripted filters after you test the basic configuration.

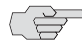
Attribute Filter Script Functions

Attribute filter scripts can execute any standard JavaScript statements and functions. Additionally, attribute filter scripts use the AttributeFilter API to perform filter-specific operations. You must instantiate a new AttributeFilter object instance and then use it to invoke these methods, such as:

```
var filter = new AttributeFilter();
var csid = filter.Get("Calling-Station-ID");
```

The following six AttributeFilter methods are provided:

- new AttributeFilter()—Creates a new AttributeFilter object instance.
- Get()—Gets the value of an attribute from the request or response packet.
- Add()—Adds a new attribute value to the request or response packet.
- Reset()—Deletes one or all values of an attribute.
- Replace()—Deletes the value of an attribute and replaces it with a new value.
- Execute()—Executes a static attribute filter.



NOTE: Do not use the Execute() method to invoke a scripted attribute filter, otherwise a runtime error will occur.

Attribute filter scripts can also execute any of the methods of the DataAccessor class.

For more details about the AttributeFilter functions and methods, see "AttributeFilter Object" on page 86.

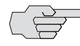
Choosing the Return Code

When an attribute filter script is finished executing, it returns a result code to the Steel-Belted Radius core. The code indicates whether or not the script executed successfully. The script may also return a string containing the name of a static attribute filter for Steel-Belted Radius to execute after the script is finished.

Valid return codes are:

- SCRIPT_RET_SUCCESS—The attribute filter script executed successfully. Steel-Belted Radius should continue processing the current request.
- SCRIPT_RET_FAILURE—The attribute filter script failed to execute successfully. Steel-Belted Radius should terminate request processing and reject the user.

- *<FilterName>*—Steel-Belted Radius should execute the static filter given by *FilterName* after the script has finished executing and then continue processing the current request.



NOTE: Do not return the name of a scripted attribute filter from your script, otherwise a runtime error will occur.

Configuring Attribute Filter Scripts

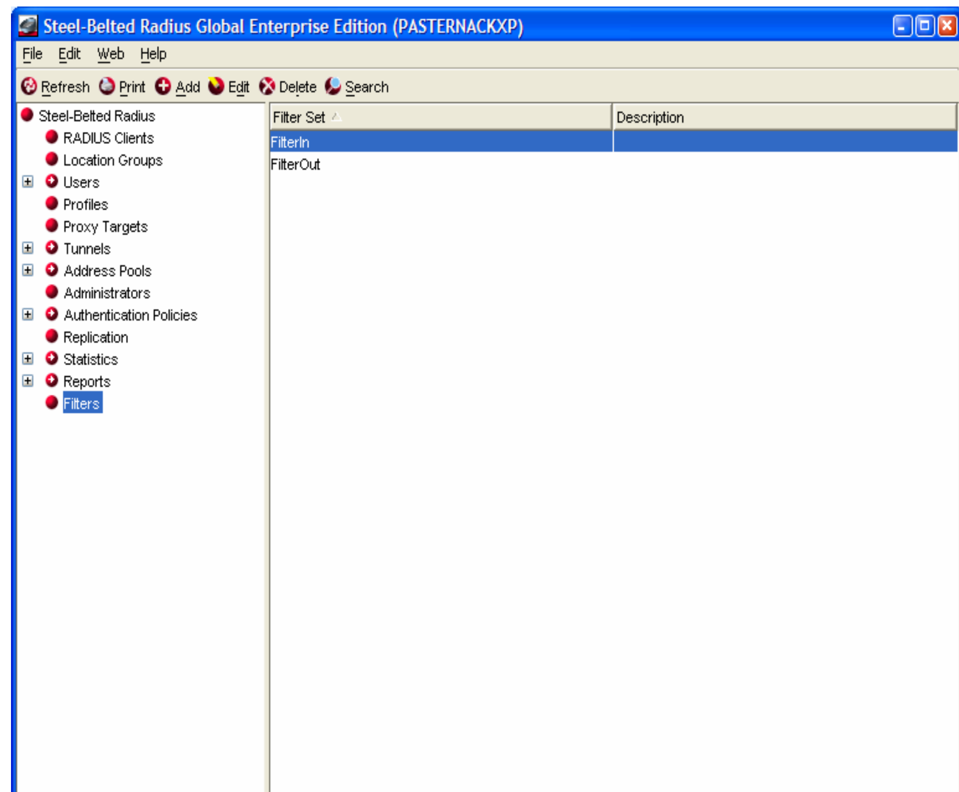
Use the SBR Administrator tool to configure scripts for attribute filters. The attribute filter data are stored in the filter.ini file.

Defining Scripted Filters

To define a scripted filter:

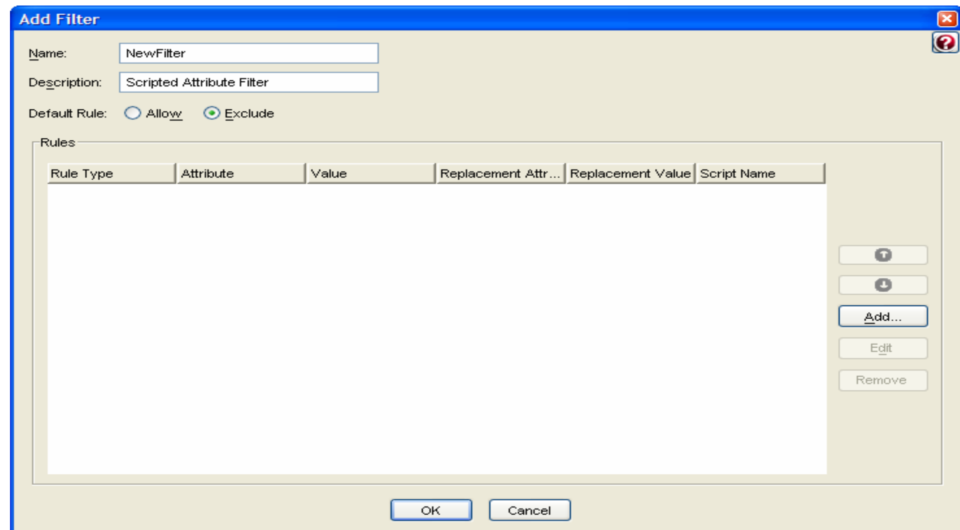
1. From the SBR Administrator's main window, select the **Filters** panel, as shown in Figure 6.

Figure 6: Filters panel



2. Click to open the Add Filter dialog, shown in Figure 7.

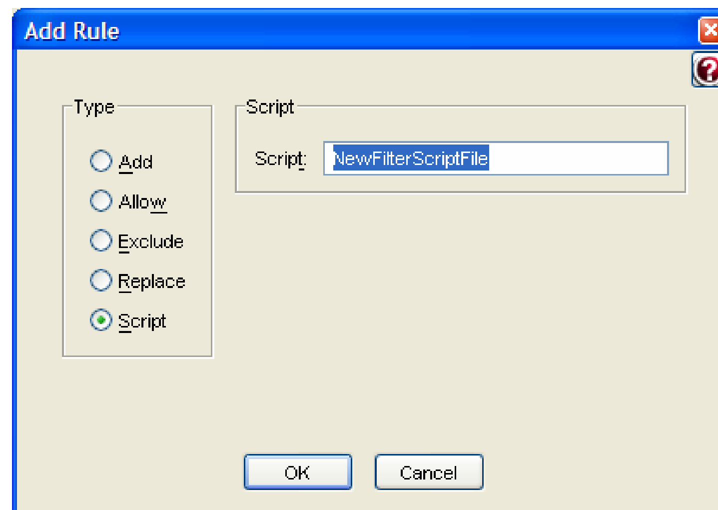
Figure 7: Add Filter dialog



3. Enter the name and description of the new filter in the **Name** and **Description** text boxes, respectively. The **Description** text box is optional.

4. Click **Add** to open the Add Rule dialog, shown in Figure 8.

Figure 8: Add Rule dialog



5. From the **Type** list of radio buttons, select **Script** to add a script rule type to the filter. Enter the name the script file for the filter in the **Script** text box. If you add a script rule type to a filter, all other rule types are ignored.

NOTE: Do not enter the .jsi extension for the script name.

6. Click **OK** to return to the Add Filter dialog, then click **OK** again to return to the Filters panel. The new script that you added now appears in the list of **Filter Sets**.

After you define the attribute filter script, it will function exactly the same as any other regular attribute filter. You can define multiple instances of standard and scripted filters. For details about referencing the attribute filter script, see the "Referencing Attribute Filters" section of Chapter 4, "Attribute Processing Files," in the *Steel-Belted Radius Reference Guide*.

Chapter 7
Working with Data Accessors

This chapter explains how to configure Steel-Belted Radius data accessors and how to use the DataAccessor API to connect your scripts to external SQL databases and LDAP repositories.

You can use data accessors to search for and retrieve credentials and user identity attributes, policy settings, accounting data, and other information from SQL and LDAP data sources. You can insert new information into SQL databases, execute SQL stored procedures, and process the result data values under program control.

Data Accessor Overview

Data accessors are plug-in modules, similar to the LDAP and SQL authentication plug-ins. Because they do not implement authentication methods, data accessors are called *generic* plugins. You configure data accessors with initialization files having the .gen filename extension.

When Steel-Belted Radius starts, it processes all the .gen files in its service directory. Each .gen file defines a separate instance of a data accessor plug-in. Settings in the initialization file specify the name of the shared object file (.so or .dll) to load and a name string to identify the plug-in instance. Many .gen files may refer to a single shared object file, but the name string must be unique for each plug-in instance.

The .gen file specifies the connection settings and query strings for the SQL database or LDAP repository. These settings are similar or identical to those in the corresponding SQL and LDAP.aut files. The .gen file also specifies type declarations used when passing data between scripts and the plug-ins.

Scripts use the DataAccessor API to communication with data accessors. The DataAccessor API exposes five methods:

- new DataAccessor()—Creates a new DataAccessor object bound to a single data accessor plugin instance.
- SetInputVariable()—Sets a request data value in the data accessor’s input variable container.
- GetOutputVariable()—Gets a response data value from the data accessor’s output variable container.

- Clear()—Deletes all data values in the data accessor's input and output variable containers.
- Execute()—Performs a single operation on the external data source.

For more information about the DataAccessor API, see "DataAccessor Object" on page 90.

The .gen file hides the details of the external data connection from your scripts. The DataAccessor API appears the same way to your scripts whether you are connecting to a SQL database or an LDAP repository.

Variable Containers

You use variable containers to pass data values back and forth between scripts and data accessors. Each data accessor instance has two variable containers: an input variable container and an output variable container. The input container receives data values from the script, and the output container returns data values to the script.

Variable containers are data arrays. Rows in the array have three columns:

- A unique string identifying one variable in the container.
- The variable data type (string, binary, integer, ipaddress, ipv6address, ipv6prefix, ipv6interface, or date).
- The actual data value.

Settings in the .gen file specify the number of rows in each variable container, the variable names, and their data types. These settings are fixed when Steel-Belted Radius starts up and cannot be changed by your scripts. Only the data values change when you call the DataAccessor script API.

Call the SetInputVariable() and GetOutputVariable() API calls to transfer data values into and out of the variable containers. Each function call transfers a single data value. The name argument selects the container row to read or modify. The name must match one of the variable declarations in the .gen file or a runtime error will occur.

To clear all values in both the input and output variable containers, call the data accessor object's Clear() method.

When you call the Execute() method, values from the input container are bound into the SQL query or LDAP search specified in the .gen file. When the operation completes, selected values are copied from the result to the output container where they may be read by your script.

Internal Variable Table (LDAP Only)

Like the LDAP authentication plugin, the LDAP data accessor has an internal variable table used for temporary data storage during the processing of LDAP search requests. The data accessor variable table works in almost the same way as the authentication plugin variable table, except that it maps to entries in the input and output variable containers instead of attributes in the RADIUS request and response packets. The data flow in and out of the internal variable table is:

1. At the beginning of each Execute() call to an LDAP data accessor, Steel-Belted Radius uses settings in the [Request] section of the .gen file to select which entries in the input variable container are copied into the internal variable table.

2. Steel-Belted Radius performs one or more LDAP searches. After each search is performed, selected attributes are copied by name from the LDAP result and placed in the internal variable table. Settings in the [Attributes/*name*] sections of the .gen file determine which LDAP result attributes are copied.

3. Steel-Belted Radius uses the [Response] section to select information from the variable table to be copied to the LDAP data accessor output variable container.

For more information about the variable table, see "Working with the Variable Table" on page 26, and for more information about the LDAP request life cycle, see "LDAP Request Life Cycle" on page 23.

Data Accessor Configuration

SQL Data Accessor Configuration

[Bootstrap] Section

The [Bootstrap] section of the SQL data accessor configuration file specifies information that Steel-Belted Radius uses to load and start the SQL data accessor plugin.

You can configure more than one SQL data accessor plug-in instance. Each requires its own .gen file in the Steel-Belted Radius service directory. The [Bootstrap] section of each .gen file must provide a LibraryName for the following:

- Windows—sqlaccessor.dll
- Linux—radsql_accessor_jdbc.so
- Solaris—radsql_accessor_ora8.so, radql_accessor_ora9.so, radql_accessor_ora10.so, or radsql_accessor_jdbc.so

```
[Bootstrap]
LibraryName=sqlaccessor.dll
Enable=0
```

Table 6: [Bootstrap] Syntax

Parameter	Function
LibraryName	Specifies the name of the SQL data accessor plugin. ■ Windows—sqlaccessor.dll ■ Linux—radsql_accessor_jdbc.so ■ Solaris—radsql_accessor_ora8.so, radql_accessor_ora9.so, radql_accessor_ora10.so, or radsql_accessor_jdbc.so
Enable	Specifies whether the SQL data accessor instance is enabled. ■ If set to 0, the SQL data accessor instance is disabled. ■ If set to 1, the SQL data accessor instance is enabled. Default value is 0.

[Results] Section

The purpose of the [Results] section is to declare data accessor output container variables and map them to columns in the SQL query result set.

Consider the following SELECT statement:

```
SELECT user_pwd, attribs, fullname FROM rasusers WHERE user_id =@User-Name
```

where user_pwd, attribs, fullname, and user_id are the names of columns in the SQL table, and rasusers is the name of the SQL table itself. The [Results] section maps the output variables to the columns retrieved from the SQL database; for example:

```
[Results]
Password=1
Profile=2
FullName=3
```

Columns in the SQL query are identified in the [Results] section by number; 1 represents the first column in the SELECT query (from left to right), and if other columns are also references, 2 represents the second, and 3 the third, and so on.

[Settings] Section

The [Settings] section of the SQL data accessor configuration file defines parameters that control the database connection.

```
[Settings]
Connect=DSN=<dsn_name_here>;UID=<username_for_dB>;PWD=<password_for_dB>
SQL=SELECT password, profile FROM userlist WHERE name = @User-Name
ParameterMarker=?
MaxConcurrent=1
ConcurrentTimeout=30
ConnectTimeout=25
QueryTimeout=25
WaitReconnect=2
MaxWaitReconnect=360
```

Table 7: *.gen [Settings] Syntax

Parameter	Function
ConcurrentTimeout	Specifies the number of seconds a request may wait for execution before it is discarded. Since there may be only up to MaxConcurrent SQL statements executing at one time, new requests must be queued as they arrive until other statements are processed.
Connect	Specifies the string that must be passed to the database client engine to establish a connection to the database. This string has (or refers to) information about the name of the database, its location on the network, the password required to access it, and so forth. The format of the connect string depends on the type of database you use: Oracle: Connect=<dB_username>/<dB_password> JDBC: Connect=DSN=<dsn_name_here>;UID=<username_for_dB>;PWD=<password_for_dB
ConnectDelimiter	(JDBC only) Specifies the character used to separate fields (DSN, UID, PWD) in the connect string. Default value is ; (semicolon). If the JDBC connect string requires use of semicolons as part of a field value, you can use this parameter to specify a different delimiter, such as : (colon).
ConnectTimeout	Specifies the number of seconds to wait when attempting to establish the connection to the database before timing out. This value is ignored if the client database engine does not support this feature.
Driver	(JDBC only) Specifies the third-party JDBC driver to load. For example: Driver=com/provider/jdbc/sqlserver/SQLServerDriver **NOTE:** Third-party JDBC drivers must be installed in /radius/jre/lib/ext. Refer to the JDBC driver documentation for information on how to install the JDBC driver and supporting files.
LogLevel	Activates logging for the Data Accessor and sets the rate at which it writes entries to the server log file (.LOG). The LogLevel may be the number 0, 1, or 2, where 0 is the lowest logging level, 1 is intermediate, and 2 is the most verbose. If the LogLevel that you set in the .gen file is different than the LogLevel in radius.ini, the radius.ini setting determines the rate of logging.
MaxConcurrent	Specifies the maximum number of instances of a single SQL statement that may be executing at one time.
MaxWaitReconnect	Specifies the maximum number of seconds to wait after successive failures to reconnect after a failure of the database connection. WaitReconnect specifies the time to wait after failure of the database connection. This value is doubled on each failed attempt to reconnect, up to a maximum of MaxWaitReconnect.
ParameterMarker	Specifies the character or sequence of characters used as the parameter marker in a parameterized SQL query. Normally, this is the question mark (?), but this could vary among database vendors.

Table 7: *.gen [Settings] Syntax (continued)

Parameter	Function
QueryTimeout	Specifies the number of seconds to wait for a response to a query before timing out. This value is passed to the client database engine, which may or may not implement the feature.
SQL	Specifies the SQL statement used to access and insert information in the database and indicates the names of the variables to create in the input variable container when a SQL statement variable is preceded by an @ sign. The SQL statement may be broken over several lines by ending each line with a backslash. The backslash must be preceded by a space character, and followed by a newline character. The subsequent lines may be indented for better readability. Example: SQL=SELECT password, profile, fullname \FROM usertable \WHERE username = @User-Name
WaitReconnect	Specifies the number of seconds to wait after a failure of the database connection before trying to connect again.

NOTE: Declare input variables within the SQL query string (SQL parameter) by putting an @ sign before the variable name. This tells Steel-Belted Radius to create an entry for the variable in the data accessor's input variable container. This is different from the LDAP data accessor, which uses a separate [Request] section to declare input container variables.

For an example of a SQL accessor configuration, see "Example: SQL Data Accessor Configuration File" on page 61.

[VariableTypes] Section

The [VariableTypes] section of the SQL data accessor configuration file specifies the storage data type for each entry in the input and output variable containers.

```
[VariableTypes]
<variable>=<type>
<variable>=<type>
.
.
.
```

where *variable* is the name of an input or output container variable and *type* is the data type specifier, one of string, binary, integer, ipaddress, ipv6address, ipv6prefix, ipv6interface, or date.

For more information about selecting the variable type specifier, see "Data Conversion Rules" on page 55.

LDAP Data Accessor Configuration

[Bootstrap] Section

The [Bootstrap] section of the LDAP data accessor configuration file specifies information that Steel-Belted Radius uses to load and start the LDAP data accessor plugin.

You can configure more than one LDAP data accessor plug-in instance. Each requires its own .gen file in the RADIUS service directory. The [Bootstrap] section of each .gen file must provide a LibraryName of ldapaccessor.so (for Solaris/Linux) or ldapauth.dll (for Windows).

Table 8: *.gen [Bootstrap] Syntax

Parameter	Function
LibraryName	Specifies the name of the LDAP data accessor plugin. Solaris/Linux: Enter ldapaccessor.so Windows: Enter ldapauth.dll
Enable	Specifies whether the LDAP data accessor instance is enabled. ■ If set to 0, the LDAP data accessor instance is disabled. ■ If set to 1, the LDAP data accessor instance is enabled. Default value is 0.

[Attributes/*name*] Sections

An LDAP search returns all of the attributes associated with an LDAP entry. Many of these attributes may not be relevant to your script. When specifying an LDAP Search for the data accessor, you can provide a list of specific LDAP result attributes to retain by name in the internal variable table. The other attributes are discarded.

You configure [Attributes/*name*] sections in the LDAP data accessor .gen file to create named lists of LDAP attributes. The syntax is as follows:

```
[Attributes/name]
attribute
attribute
    ⋮
```

where *attribute* is the name of an LDAP attribute and *name* is an arbitrary name for the section. You must type the attribute names exactly as they appear in your LDAP database schema. Use one line per attribute. For example:

```
[Attributes/InterestingAttributes]
User-Secret
RADIUS-Profile
Inactivity-Timeout
    ⋮
```

An [Attributes/*name*] section is associated with a [Search/*name*] section using the Attributes parameter. For example:

```
[Search/DoLdapSearch]
```

```
Attributes = InterestingAttributes
```

When the search executes, the selected result attributes are stored by name in the LDAP data accessor internal variable table. If the Attributes parameter is omitted from a [Search/*name*] section, all of the attributes returned by the LDAP search are stored. Of these attributes, only those referred to in the [Response] section of the .gen file are copied into the output variable container; the rest are discarded once all LDAP searches have completed.

[Response] Section

The [Response] section tells the LDAP data accessor the names of variables to create in the output variable container. It also tells the data accessor which values from the internal variable table to copy to entries in the output variable container once all LDAP repository searches have completed.

The [Response] section syntax is as follows:

```
[Response]
outvar = tablevar
outvar = tablevar
    ⋮
```

where *outvar* is the name of a variable in the output variable container and *tablevar* is the name of an entry in the internal variable table.

[Search/*name*] Sections

Each [Search/*name*] section in the LDAP data accessor configuration file specifies the complete details of one LDAP Search request. You can use the same search request on various LDAP repositories because the details of the LDAP server connection are specified separately.

By default, when you execute the search request specified in a [Search/*name*] section, all the attributes associated with the resulting LDAP record are retained. Use the Attributes parameter to specify a list of specific attributes you want to store in the internal variable table.

```
[Search/DoLdapSearch]
Base = o=bigco.com
Scope = 2
Filter = item1=<value1>
Attributes = LdapAttrs
Timeout = 20
%DN = dn

[Attributes/LdapAttrs]
item2

[Response]
Output-Var = item2
```

Table 9: *.gen [Search/*name*] Syntax

Parameter	Function
%DN	Specifies an entry in the internal variable into which the distinguished name that results from the Search should be placed.
Attributes	Specifies the LDAP attributes relevant to Steel-Belted Radius, by referencing an [Attributes/*name*] section elsewhere in the same .gen file.
Base	Specifies the distinguished name (DN) of the entry that serves as the starting point for the search. This filter is a template for an LDAP distinguished name string. The filter follows conventional LDAP syntax and may be as simple or as complex as LDAP syntax permits. It may also include replacement variables from the Variable Table. Each replacement variable consists of the variable name enclosed in angle brackets (< >). Upon execution of the LDAP Search request, the value of the variable replaces the variable name.
OnFound	Specifies the next request section when data is found. The value of this parameter is a string, *name*. The *name* specifies an LDAP Search request by referencing a [Search/*name*] section elsewhere in the same .gen file. If there is no next request section, the overall operation succeeds.
OnNotFound	Specifies the next request section when data is not found. The value of this parameter is a string, *name*. The *name* specifies an LDAP Search request by referencing a [Search/*name*] section elsewhere in the same .gen file. If there is no next request section, the overall operation fails.
Filter	Specifies the filter to apply to the search. This filter is a template for an LDAP Search string. The filter follows conventional LDAP syntax and may be as simple or as complex as LDAP syntax permits, with multiple attribute/value assertions in boolean combination. It may also include replacement variables from the Variable Table. Each replacement variable consists of the variable name enclosed in angle brackets (< >). Upon execution of the LDAP Search request, the value of the variable replaces the variable name. For example, a Search template that uses the User-Name and Service-Type attributes from the RADIUS request might look like this: (&(uid = <User-Name>)(type = <Service-Type>))
Scope	Specifies the scope of the search; 0 (search the base), 1 (search all entries one level beneath the base), or 2 (search the base and all entries beneath the base at any level).

The OnFound and OnNotFound parameters can be used in [Search/*name*] sections to create serial chains of search requests. The OnFound parameter specifies the name of a search to try if the current search returns a non-empty result from the LDAP repository. The OnNotFound parameter specifies the name of a search to try if the current search fails to return data. Arbitrarily complex search trees may be created.

The following example shows a simple LDAP search tree:

```
[Search/DoSearch2]
Base = o=xyz.com
Scope = 2
Filter = uid=<User-Name>
```

```
Attributes = AttrList
Timeout = 20
%DN = dn
OnFound = DoSearch8
OnNotFound = DoSearch9

[Search/DoSearch8]
Base = o=xyz.com
Scope = 2
Filter = uid=<User-Name>
Attributes = AttrList
Timeout = 20
%DN = dn
OnFound = DoSearch9
OnNotFound = DoSearch9

[Search/DoSearch9]
Base = o=xyz.com
Scope = 2
Filter = uid=<User-Name>
Attributes = AttrList
Timeout = 20
%DN = dn
```

[Request] Section

The [Request] section tells the LDAP data accessor the names of variables to create in the input variable container. It also tells the data accessor which input variable container entries to copy to the internal variable table prior to execution of the LDAP search request tree.

```
[Request]
invar = tablevar
invar = tablevar
.
.
.
```

where *invar* is the name of a variable in the input variable container and *tablevar* is the name of an entry in the internal variable table.

tablevar may be omitted from any [Request] entry. If so, the variable in the input variable container is copied to an internal variable table entry named *invar*.

```
[Request]
invar =
```

[Defaults] Section

The [Defaults] section of the LDAP data accessor configuration file enables you to add named entries to the internal variable table before the LDAP search tree is executed. You can reference these variables in your queries, even if they are not initialized from the [Request] section.

The format of each [Defaults] entry is:

tablevar = value

where *tablevar* is the name of a variable in the internal variable table and *value* is the value you want to assign to it. For example:

```
[Defaults]
Filter = campus_only
SessionLimit = 600
```

[Server/*name*] Sections

Several sections of the LDAP data accessor file work together to configure the connection between the Steel-Belted Radius server and the LDAP database server(s). The sections are: [Server], [Server/*name*], and [Settings].

Each [Server/*name*] section of the LDAP data accessor file contains configuration information about a single LDAP server. You must provide a [Server/*name*] section for each server you've named in the [Server] section. For example:

```
[Server]
s1=
s2=
[Server/s1]
Host = ldap_1
Port = 389
.
.
.
[Server/s2]
Host = 130.4.67.1
LastResort = 1
.
.
.
```

Table 10: *.gen [Server/*name*] Syntax

Parameter	Function
BindName	The BindName parameter specifies the distinguished name (DN) to be used to connect to the LDAP server. The [Server/*name*] section lets you specify a unique BindName for a specific server. Use the [Settings] section to specify a default BindName to use for all servers.
BindPassword	The BindPassword specifies the password to be used to connect to the LDAP server. The [Server/*name*] section lets you specify a unique BindPassword for a specific server. Use the [Settings] section to specify a default BindPassword to use for all servers.
Certificates	Specifies the path of the certificate database for use with SSL. This path must not end in a filename. The certificate database must be the cert7.db and key3.db files used by Netscape Communicator 4.x or later.
ConnectTimeout	Specifies the number of seconds to wait when attempting to establish the connection to the database before timing out. This value is passed to the client database engine, which may or may not implement the feature.

Table 10: *.gen [Server/*name*] Syntax (continued)

Parameter	Function
FlashReconnect	If the server is down when performing a Search, setting this parameter to 1 triggers a reconnection attempt before rejecting the request. Therefore, requests are not rejected due to inactivity timeouts. This setting applies to a particular server. To apply it for all servers, place it in the [Settings] section.
Host	The host name or IP address of the LDAP server.
LastResort	You may identify a "last resort" LDAP server by providing a LastResort parameter in one of these [Server/*name*] sections, and setting its value to 1. If the search returns no result, the execution of the search tree completes (unless OnNotFound is configured).
LdapVersion	Specifies the version of LDAP protocol, if needed to override the default given in the [Settings] section.
MaxConcurrent	Specifies the maximum number of instances of a single LDAP request that may be executing at one time.
MaxWaitReconnect	Specifies the maximum number of seconds to wait after successive failures to reconnect after a failure of the database connection. WaitReconnect specifies the time to wait after failure of the database connection. This value is doubled on each failed attempt to reconnect, up to a maximum of MaxWaitReconnect.
Port	The TCP port of the LDAP server, or 0 to use the standard port. Default value is 0.
QueryTimeout	Specifies the number of seconds to wait for the execution of an LDAP request to complete before timing out. This value is passed to the database engine, which may or may not implement the feature.
Search	The value of this parameter is a string, *name*. The *name* specifies an LDAP Search request by referencing a [Search/*name*] section elsewhere in the same .gen file.
SSL	■ If set to 0, SSL is not used over the LDAP connection. ■ If set to 1, SSL is used over the LDAP connection. Default value is 0.
WaitReconnect	Specifies the number of seconds to wait after a failure of the database connection before trying to connect again.

[Server] Section

The [Server] section of the LDAP data accessor configuration file lists the LDAP servers. You can specify more than one server in the [Server] section for load-balancing or backup. When more than one server is specified, Steel-Belted Radius authenticates against these databases in a round-robin fashion.

The syntax is as follows:

[Server]
ServerName=TargetNumber
ServerName=TargetNumber
⋮

where *ServerName* is the name of a header file section that contains configuration information for that server, and *TargetNumber* is an *activation target number*, a number that controls when this server is activated for backup purposes. TargetNumber is optional and may be left blank. For example:

```
[Server]
s1 =
s2 =
[Server/s1]
.
.    ;Connection details for server s1
.
[Server/s2]
.
.     ;Connection details for server s2
⋮
```

A Steel-Belted Radius server maintains connectivity with its LDAP servers according to the following rules:

- The priority of the server by order. The first entry in the [Server] section has the highest priority.
- By activation target number. The rule for the activation target is that if the number of LDAP servers that Steel-Belted Radius is connected to is less than the activation target, Steel-Belted Radius connects to the server and includes it in the round-robin list. While the number of active servers is equal to or greater that the activation target, Steel-Belted Radius does not use that server in the round-robin list. An activation target of 0 indicates that, in the current configuration, this machine is never used.

[Settings] Section

The [Settings] section of the LDAP data accessor configuration file forms a basis for all Search requests to the LDAP database server(s).

The values set in [Settings] for some parameters, such as ConnectTimeout, MaxConcurrent, or WaitReconnect, provide defaults that apply to all servers. These default values can be overridden for a particular server by entering the same parameter with a different value in a [Server/*name*] section.

Table 11: *.gen [Settings] Syntax

Parameter	Function
BindName	In the [Settings] section, BindName and BindPassword specify a default LDAP template to use for all servers. You can also use BindName and BindPassword in [Server/*name*] sections to override this default for an individual server
ConnectTimeout	Specifies the number of seconds to wait when attempting to establish the connection to the database before timing out. This value is passed to the client database engine, which may or may not implement the feature. Default value is 25 seconds.

Table 11: *.gen [Settings] Syntax (continued)

Parameter	Function
FilterSpecial CharacterHandling	■ If set to 1, specifies that non-alphanumeric characters, such as (or), should be converted to an ASCII hex value preceded by a backslash when they are encountered in a user name. ■ If set to 0, non-alphanumeric characters are not converted. Default value is 0.
FlashReconnect	If the server is down when performing a Search, setting this parameter to 1 triggers a reconnection attempt before rejecting the request. Therefore, requests are not rejected due to inactivity timeouts. **NOTE:** The value specified in this parameter can be overridden in individual [Server/*name*] sections of this file.
LdapVersion	Specifies the version of LDAP protocol. Default value is 2.
LogLevel	Activates logging for the LDAP data accessor and sets the rate at which it writes entries to the Steel-Belted Radius server log file (.LOG). This value may be the number 0, 1, or 2, where 0 is the lowest logging level, 1 is intermediate, and 2 is the most verbose. If the LogLevel that you set in the .gen file is different than the LogLevel in radius.ini, the radius.ini setting determines the rate of logging. The LogLevel is re-read whenever the server receives a HUP signal.
MaxConcurrent	Specifies the maximum number of instances of a single LDAP request that may be executing at one time. **NOTE:** The value specified in this parameter can be overridden in individual [Server/*name*] sections of this file.
MaxWaitReconnect	Specifies the maximum number of seconds to wait after successive failures to reconnect after a failure of the database connection. WaitReconnect specifies the time to wait after failure of the database connection. This value is doubled on each failed attempt to reconnect, up to a maximum of MaxWaitReconnect. **NOTE:** The value specified in this parameter can be overridden in individual [Server/*name*] sections of this file.
OnFound	Specifies the next request section when data is found. The value of this parameter is a string, *name*. The *name* specifies an LDAP Search request by referencing a [Search/*name*] section elsewhere in the same .gen file. If there is no next request section, the overall operation succeeds.
OnNotFound	Specifies the next request section when data is not found. The value of this parameter is a string, *name*. The *name* specifies an LDAP Search request by referencing a [Search/*name*] section elsewhere in the same .gen file. If there is no next request section, the overall operation fails.
QueryTimeout	Specifies the timeout value in seconds for an individual search performed against the LDAP server. Default value is 10 seconds.
Search	The value of this parameter is a string, *name*. The *name* specifies an LDAP Search request by referencing a [Search/*name*] section elsewhere in the same .gen file.

Table 11: *.gen [Settings] Syntax (continued)

Parameter	Function
SSL	■ If set to 0, SSL is not used over the LDAP connection. ■ If set to 1, SSL is used over the LDAP connection. Default value is 0. **NOTE:** The value specified in this parameter can be overridden in individual [Server/*name*] sections of this file.
WaitReconnect	Specifies the number of seconds to wait after a failure of the database connection before trying to connect again.
Timeout	Specifies the maximum number of seconds for the overall timeout for each request, which includes the delay in acquiring resources, attempts against multiple LDAP servers, and so forth. Default value is 20 seconds.
UTC	■ If set to 0, time values are displayed using the local time. ■ If set to 1, time values are displayed using universal time coordinates (UTC).
WaitReconnect	Specifies the number of seconds to wait after a failure of the database connection before trying to connect again. **NOTE:** The value specified in this parameter can be overridden in individual [Server/*name*] sections of this file.

[VariableTypes] Section

The [VariableTypes] section of the LDAP data accessor configuration file specifies the storage data type for each entry in the input and output variable containers.

```
[VariableTypes]
<variable>=<type>
<variable>=<type>
⋮
```

where *variable* is the name of an input or output container variable and *type* is the data type specifier, one of string, binary, integer, ipaddress, ipv6address, ipv6prefix, ipv6interface, or date.

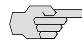
For more information about selecting the variable type specifier, see “Data Conversion Rules” on page 55.

Data Conversion Rules

This section describes the data conversions that occur when you use the SQL and LDAP data accessors. It provides the rules you need to specify variable types in your scripts, external databases, and data conversion files.

A single data variable can be represented differently in a script, a data accessor, and an external database. You must configure all three components correctly so that Steel-Belted Radius can perform the correct data conversions. Improper configuration can cause data corruption or runtime errors.

The SQL or LDAP schema determines how data are stored in external databases. Use the [VariableTypes] section of the LDAP or SQL data accessor .gen file to specify the corresponding data types for input and output container variables. You must also use the appropriate JavaScript syntax in your scripts.

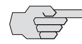


NOTE: The examples in this section refer to SQL database queries, tables, and columns, but can apply to LDAP searches, records, and tables.

Output Container

SQL and LDAP data accessor plugins require that output container variables map to database columns of type string. When the DataAccessor.Execute() function is called, the result strings returned from the database are parsed according to the respective output container variable types declared in the data accessor configuration (.gen) file. The parsed data values are stored in the output container.

Subsequently, when the DataAccessor.GetOutputVariable() function is called, a second conversion is performed. Output container variables of type integer are returned to the script as JavaScript integer data. All other output container variable types are returned as strings. To avoid unnecessary data conversions, output container variables that refer to integer data should be declared as integer in the [VariableTypes] section of the .gen file. All other output container variables should be declared as string.



NOTE: It is legal, but inefficient, to declare an output container variable to be a type other than integer or string. For example, if an output container variable is declared to be type ipaddress, when the data accessor executes, the string representation of the address in the database must first be converted to a binary IP address structure, then when the script retrieves the value from the output container, the binary IP address must be converted back to a string representation. This conversion process wastes a significant amount of computing resources.

Attempting to map an output container variable to a non-string database column will cause a runtime error when the data accessor is executed regardless of the variable type declaration in the .gen file.

Input Container

Data conversions are not performed when data are passed from the script to the input container, and from the input container to the database. Input container variables might refer to string, numeric, or binary column data in the database. The data types must all match across the script, the .gen file, and the database schema.

In most cases, the only input container variable types used with the JavaScript API (and the corresponding database column types) will be integer and string. Other data types can be stored in the input container by coding the binary data directly as unformatted JavaScript strings using the standard String.fromCharCode() function. You are responsible for programming the binary coding correctly.

Examples

Example 1

When DataAccessor.SetInputVariable() is called for an integer variable, the JavaScript API can determine how to convert the supplied argument. For example, the following .gen file configuration declares an input container variable Count of type integer:

```
Select FooVar FROM bar WHERE index = @Count

[VariableTypes]
FooVar=string
Count=integer

[Results]
FooVar=1/64
```

Both of the following statements will result in the correct integer data type being passed to the Count variable in the SQL query:

```
DataAccessor.SetInputVariable("Count", 1234);
DataAccessor.SetInputVariable("Count", "1234");
```

In this example, the index column in the database must also be of integer type to match the Count input container variable. If not, a type mismatch error will occur when the plug-in attempts to execute the SQL statement.

Example 2

Types other than integer should be represented as strings. In the following configuration file, the ipaddress column in database table bar contains IP address values represented as formatted strings. The input container variable IP-Address is declared as a string in the .gen file; and in the call to DataAccessor.SetInputVariable(), the argument value is also passed as a string:

```
Select FooVar FROM bar WHERE address = @ip-address

[VariableTypes]
FooVar=string
IP-Address=string

[Results]
FooVar=1/64

DataAccessor.SetInputVariable("IP-Address", "128.18.40.1");
```

The .gen file could be changed to make IP-Address a binary (ipaddress) input container variable.

```
[VariableTypes]
FooVar=string
IP-Address=ipaddress
```

Then the script might have to pass the data to SetInputVariable() as an integer or unformatted binary string.

```
var intData = 47276480;
DataAccessor.SetInputVariable("IP-Address", intData);
```

or:

```
var binData = String.fromCharCode(0x80, 0x12, 0x28, 0x01);
DataAccessor.SetInputVariable("IP-Address", binData);
```

Because the container variable ip-address is declared as type ipaddress, the data accessor expects the input container data to be in a binary format appropriate to the type and passes it along unchanged to the database. The definition of the database address column must be changed to an appropriate binary type.

Example 3

This example shows how to retrieve an IP address value from the database. Because column IP-Address is mapped to an output container variable, the database schema must declare IP-Address to be of string type.

```
Select IP-Address FROM foobar WHERE username = @User-Name

[VariableTypes]
IP-Address=string
User-Name=string

[Results]
IP-Address=1/64

DataAccessor.SetInputVariable("User-Name", "testuser");
DataAccessor.Execute();
var addrStr = DataAccessor.GetOutputVariable("IP-Address");
```

Similar results can be generated by declaring output container variable IP-Address to be of type ipaddress. As previously noted, this will cause the data value to be converted to a binary form internal to the data accessor, but will waste computing resources.

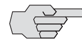
Supported Data Types and Conversions

Table 12 lists supported data types and conversions for *input* container variables.

Table 12: Data Types and Conversions for Input Container Variables

Input Container Variable Type	JavaScript Type	Database Type
Integer	Integer, String	Integer
String	String	String
■ IPv4 Address ■ IPv6 Address ■ IPv6 Prefix ■ IPv6 Interface ■ Time	Unformatted Binary String (*not recommended*)	<Binary Type>

The "JavaScript Type" column refers to the JavaScript variable types to be passed to the DataAccessor.SetInputVariable() function for each input container variable type.



NOTE: Configuring the .gen file with input container variables of type IPv4 Address, IPv6 Address, IPv6 Prefix, IPv6 Interface, or Time is not recommended. You should represent these data types as formatted strings within the script, the data accessor, and the database.

Table 13 lists supported data types and conversions for *output* container variables.

Table 13: Data Types and Conversions for Output Container Variables

Output Container Variable Type	JavaScript Type	Database Type
Integer	Integer	String
String	String	String
■ IPv4 Address ■ IPv6 Address ■ IPv6 Prefix ■ IPv6 Interface	Type-Specific Formatted String	String
Time	Time-String (*yyyy/mm/dd [hh[:mm[:ss]]]*)	String

All columns/fields referred to by output container variables must contain string data. Integer output container variables are returned to the script as JavaScript integers. All other types are converted to formatted JavaScript strings.

Data Accessor Configuration File Examples

The following shows sample LDAP and SQL data accessor configuration files. The examples in Chapter 8, "Script Examples" on page 63 also refer to these data accessor configurations.

Example: LDAP Data Accessor Configuration File

```
[Bootstrap]
LibraryName=ldapaccessor.dll
Enable=0

;
;Define data accessor name, search name, and connection settings
;
[Settings]
MethodName=LdapAccessor
Timeout=20
ConnectTimeout=25
QueryTimeout=10
WaitReconnect=2
MaxWaitReconnect=360
UpperCaseName = 0
SSL = 0
```

```
Search = DoLdapSearch

;
;Define the server address and bind credentials
;
[Server]
s1=

[Server/s1]
Host=<LDAPServerNameOrIPAddress>
Port = 389
BindName=uid=admin, ou=sales, o=bigco.com
BindPassword=secret

;
;Map the input variables
;
[Request]
Input-Var = value1

;
;Map the output variables
;
[Response]
Output-Var = item2

;
;Specify attributes to be copied from search response
;
[Attributes/LdapAttrs]
item2

;
; Define the search filter
;
[Search/DoLdapSearch]
Base = o=bigco.com
Scope = 2
Filter = item1=<value1>
Attributes = LdapAttrs
Timeout = 20
%DN = dn

;
;Define the input and output variable types; default is string
;
[VariableTypes]
Input-Var = string
Output-Var = string
```

Example: SQL Data Accessor Configuration File

```
[Bootstrap]
LibraryName=sqlaccessor.dll
Enable=0

;
;Define data accessor name, SQL query, and connection settings
;
[Settings]
MethodName=SQLAccessor
SQL=SELECT item1, item2 FROM tablename WHERE item3 = @Input-Var1
Connect=DSN=<dsn_name_here>;UID=<username_for_dB>;PWD=<password_for_dB>
ParameterMarker=?
MaxConcurrent=1
ConcurrentTimeout=30
ConnectTimeout=25
QueryTimeout=25
WaitReconnect=2
MaxWaitReconnect=360

;
;Map output variables to SQL query result string columns
;
[Results]
Output-Var1 = 1/32
Output-Var2 = 2/64

;
;Define the input and output variable types; default is string
;
[VariableTypes]
Output-Var1 = string
Output-Var2 = string
Input-Var1 = integer
```

Chapter 8

Script Examples

This chapter provides examples for the following script types:

- LDAP—Use to search and modify the LDAP variable tables in Steel-Belted Radius.
- Realm Selection—Use scripts to control realm selection during RADIUS request processing in Steel-Belted Radius.
- Attribute Filter—Use scripts to control attribute filtering during RADIUS request and response processing in Steel-Belted Radius.

LDAP Script Examples

Example 1: Simple Authentication

The following script executes the search criteria specified in the [Search/LdapSearch1] section of the ldapauth.aut file. If the search is unsuccessful, the script prepends myco. to the user name and executes the search criteria specified in the [Search/LdapSearch2] section.

```
[Script]
// Try the initial query.
Status = Ldap.Search('LdapSearch1');
if (status == Ldap.NOTFOUND) {
// Add "myco." to the user name and run new query.
userName = LdapVariables.Get('User-Name');
LdapVariables.Reset('User-Name');
LdapVariables.Add('User-Name', 'myco.' + userName);
status = Ldap.Search('LdapSearch2');
}

// Return value depends on final search status.
switch (status) {
case Ldap.FOUND:
    return SBR_RET_SUCCESS;
case Ldap.NOTFOUND:
    return SBR_RET_NOT_AUTHENTICATED;
default:
    return SBR_RET_FAILURE;
}
```

Example 2: Profile Assignment

Scripts can use authentication information to determine the profile that should be assigned to a user. In the following example, the script executes the query specified in the [Search/Radius] section. This query looks up an object named ProfileData that contains multiple instances of the radiusattrs attribute. The script iterates through the returned values of radiusattrs, looking for the first instance that begins with the prefix sbr-. If a matching attribute is found, the prefix is stripped from the attribute and returned as the name of the user profile.

The following is the LDIF representation of the ProfileData object, showing the values of the radiusattrs attributes:

```
dn: name=ProfileData, ou=radius, dc=funk,dc=com
name: ProfileData
objectClass: top
objectClass: radiusobject
radiusattrs: attr1
radiusattrs: attr2
radiusattrs: sbr-defaultprofile
radiusattrs: attr3
```

The relevant sections of the ldapauth.aut file are shown below.

```
[Attributes/RadiusAttrs]
radiusattrs

[Response]
%Profile = Return-Profile

[Search/Radius]
Base = ou=radius,dc=funk,dc=com
Scope = 2
Filter = name=ProfileData
Attributes = RadiusAttrs
Timeout = 20
%DN = dn

[Script]
// Look up "ProfileData" object using the "Radius" query.
if (Ldap.Search("Radius") == Ldap.FOUND) {
  var attr = "";
  var profile = "default";

  // Loop through all "radiusattrs" attributes.
  for(i = 0; attr != null; i++) {
    attr = LdapVariables.Get("radiusattrs", i);
    // If prefix matches "sbr-" extract profile name.
    if ((attr !=null) && (att.substr(0, 4) == "sbr-")) {
      profile = attr.substr(4);
      break;
    }
  }

  // Add profile name to the variable table and return.
  LdapVariables.Add("Return-Profile", profile);
  return SBR_RET_SUCCESS;
```

```
}

// Object wasn't found, so signal a failure.
return SBR_RET_FAILURE;
```

Example 3: Received Attribute Normalization

Users frequently need to normalize incoming RADIUS attributes to a common format before performing an LDAP search. The following example checks the length of the telephone number string in the Calling-Station-ID attribute, preserving only the final seven digits, if necessary. The truncated telephone number is saved as a new entry (Stripped-CSID) in the variable table. The value of Stripped-CSID is specified as part of the Filter parameter in the [Search/Query1] query definition. This query is executed by the script, and the resulting status code determines the script return code.

```
[Request]
%UserName = User-Name
Calling-Station-Id = Received-CSID

[Search/Query1]
Base=ou=people,dc=funk,dc=com
Scope = 2
Filter = (&(uid=<User-Name>)(callingStationId=<Stripped-CSID>))
Timeout = 20
%DN = dn

[Script]
// Get the received Calling-Station-ID attribute.
var csid = LdapVariables.Get("Received-CSID");

// Check length and retain last seven digits of CSID.
var length = csid.length;
if (length > 7) {
    csid = csid.substr(length - 7);
    SbrWriteToLog("Shortened CSID to: " + csid);
}

// Save result to variable table so we can search on it.
LdapVariables.Add("Stripped-CSID", csid);

// Perform the search with normalized CSID.
var status = Ldap.Search("Query1");

// Generate return code based on search result.
if (status == Ldap.FOUND) {
    return SBR_RET_SUCCESS;
}
return SBR_RET_NOT_AUTHENTICATED;
```

Example 4: Conditional Profile Assignment from User Attribute

The following example illustrates how you can use LDAP scripts to implement multiple queries and complex decision logic. The script starts by invoking the FindUser query to look up the specified user in the LDAP repository. Depending on the employeetype attribute returned from the first query, a second query is selected and invoked to retrieve attributes specific to the user's employee type. Finally, the Radius-Profile attribute of the employee type record is returned as the profile name for the authentication response.

The LDIF data for a sample user is as follows:

```
dn: uid=SStudent, ou=People, dc=funk,dc=com
employeeType: Student
uid: SStudent
userPassword:: e1NTSEF9cTZvdFF0YXArcFowaG5rOWJQU3dZYlExbkFIL1doMXBnMIR4
 ==
objectClass: Sam
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetorgperson
sn: Student
cn: Sam Student
```

The following LDIF entries present the data objects holding the “Radius-Profile” attributes associated with each employee type:

```
dn: ou=radius, dc=funk,dc=com
ou: radius
objectClass: top
objectClass: organizationalunit

dn: name=VendorType, ou=radius, dc=funk,dc=com
Radius-Profile: Vendor-Profile
name: VendorType
objectClass: top
objectClass: radius

dn: name=FacultyType, ou=radius, dc=funk,dc=com
Radius-Profile: Faculty-Profile
name: FacultyType
objectClass: top
objectClass: radius

dn: name=StudentType, ou=radius, dc=funk,dc=com
Radius-Profile: Student-Profile
name: StudentType
objectClass: top
objectClass: radius
```

Finally, here are the configuration settings and the LDAP search script:

```
[Request]
%UserName = User-Name

[Response]
```

```
%Profile = Radius-Profile
%Password = userpassword

[Attributes/UserAttributes]
employeetype
userpassword

[Attributes/TypeAttributes]
radius-profile

[Search/FindUser]
Base=ou=people,dc=funk,dc=com
Scope = 2
Filter = uid=<User-Name>
Attributes = UserAttributes
Timeout = 20
%DN = dn

[Search/Student]
Base=ou=radius,dc=funk,dc=com
Scope = 2
Filter = name=StudentType
Attributes = TypeAttributes
Timeout = 20

[Search/Faculty]
Base=ou=radius,dc=funk,dc=com
Scope = 2
Filter = name=FacultyType
Attributes = TypeAttributes
Timeout = 20

[Search/Vendor]
Base=ou=radius,dc=funk,dc=com
Scope = 2
Filter = name=VendorType
Attributes = TypeAttributes
Timeout = 20

[Script]
// Look up the specified user in the LDAP repository.
var status = Ldap.Search("FindUser");
if (status != Ldap.FOUND) {
   return SBR_RET_NOT_AUTHENTICATED;
}

// Get the employeetype attribute from the query result.
var type = LdapVariables.Get("employeetype");

// Execute query to look up employee type object.
switch (type) {
case "Student":
   status = Ldap.Search("Student");
   break;
case "Faculty":
   status = Ldap.Search("Faculty");
   break;
```

```
case "Vendor":
   status = Ldap.Search("Vendor");
   break;
default:
   SbrWriteToLog("Invalid employee type: " + type);
   return SBR_RET_DO_NOT_AUTHENTICATE;
}

// This error should never happen.
if (status != Ldap.FOUND) {
   SbrWriteToLog("No record for employee type: " + type);
   return SBR_RET_DO_NOT_AUTHENTICATE;
}

// Get the profile name for this employee type.
var profile = LdapVariables.Get("profilename");
if (profile == null) {
   profile = "Default-Profile";
}

// Save profile name to variable table and return.
LdapVariables.Add("Radius-Profile", profile);
return SBR_RET_SUCCESS;
```

Realm Selection Script Examples

Example 1: Querying Multiple SQL Databases

A common application of realm selection scripts is to query a database for information used to determine the realm name. If the database query fails, you can program the script to query other databases for the required information. In the following example, the script allocates two DataAccessor objects, each pointing to a different SQL database. It also allocates AttributeFilter and RealmSelector objects.

When the script executes, it uses the AttributeFilter object to obtain the value of the Called-Station-ID attribute from the request. It uses this value as the key to look up a realm selection method name in the first SQL database. If the database record is not found, it retries the query on the second database.

The result of the database query is the name of a built-in Steel-Belted Radius realm selection method. The script then calls the RealmSelector.Execute() method to determine the realm name from the current request, which it returns as the script result.

Table 14 and Table 15 show example data for the two databases:

Table 14: Database #1

CallStationId	RealmMethod
1111111	Suffix
2222222	Prefix
3333333	DNIS
4444444	Attribute-Mapping

Table 15: Database #2

CallStationId	RealmMethod
5555555	Suffix
6666666	Prefix
7777777	DNIS
8888888	Attribute-Mapping

Two data accessor .gen files are required. They are identical except for the MethodName and Connect settings.

```
[Bootstrap]
LibraryName=sqlaccessor.dll
Enable=1

[Settings]
Connect=DSN=<dsn_name_here>;UID=<username_for_dB>;PWD=<password_for_dB>
ParameterMarker=?
SQL=SELECT RealmMethod FROM RealmExample1 WHERE CallStationId =
@Called-Station-Id
MethodName=<RealmExample1-1 | RealmExample1-2>
MaxConcurrent=1
ConcurrentTimeout=30
ConnectTimeout=25
QueryTimeout=25
WaitReconnect=2
MaxWaitReconnect=360
PasswordFormat = 0
DefaultResults = 0

[Results]
RealmMethod= 1/16

[VariableTypes]
Called-Station-ID = string
RealmMethod = string
```

Finally, here is the script configuration file (.jsi) for this example. For script efficiency, you can use an initialization block to allocate persistent API objects.

```
[Settings]
LogLevel = 2
ScriptTraceLevel = 2

[Script]
// Initialization block
if (!this.initialized) {
   filter = new AttributeFilter();
   accessor1 = new DataAccessor("RealmExample1-1");
   accessor2 = new DataAccessor("RealmExample1-2");
   selector = new RealmSelector();
   initialized = true;
}
else {
   accessor1.Clear();
   accessor2.Clear();
}

// Get the Called-Station-ID attribute from the request.
var csid = filter.Get("Called-Station-ID");
if (csid == null) {
   SbrWriteToLog("RealmExample1: Called-Station-ID attribute is null");
   return SCRIPT_RET_FAILURE;
}

accessor1.SetInputVariable("Called-Station-ID", csid);
accessor2.SetInputVariable("Called-Station-ID", csid);

// Try one database, then the other if the first search fails.
if (accessor1.Execute() == DataAccessor.FOUND) {
   var method = accessor1.GetOutputVariable("RealmMethod");
}
else if (accessor2.Execute() == DataAccessor.FOUND) {
  var method = accessor2.GetOutputVariable("RealmMethod");
}
else {
  SbrWriteToLog("RealmExample1: SQL search failed for Called-Station-ID = '" + csid +
"'");
   return SCRIPT_RET_FAILURE;
}

// Execute the specified realm selection method and return the result.
SbrWriteToLog("RealmExample1: Executing method '" + method + "'");
var realm = selector.Execute(method);
return realm;

[ScriptTrace]
var = csid
var = method
var = realm
```

Example 2: Using JavaScript to Manipulate Request Attributes

In this example, an AttributeFilter object is used to obtain the User-Name attribute from the request. JavaScript string functions are used to extract the realm suffix decoration from the username. The suffix string is concatenated with the value of the Called-Station-Id attribute and is used to look up the realm name in a SQL database. The database query also returns the name of a profile to be merged with the result list upon successful authentication. The SetAuthProfile() function is called by the script to register the profile name.

The following is an example database table for use with this script.

Table 16: Database Information

KeyString	RealmName	Profile
spacely1111111	spacely-realm1	SPACELY-PROFILE
spacely2222222	spacely-realm2	SPACELY-PROFILE
spacely3333333	spacely-realm3	SPACELY-PROFILE
spacely4444444	spacely-realm4	SPACELY-PROFILE
cogswell1111111	cogswell-realm1	COGSWELL-PROFILE
cogswell2222222	cogswell-realm2	COGSWELL-PROFILE
cogswell3333333	cogswell-realm3	COGSWELL-PROFILE
cogswell4444444	cogswell-realm4	COGSWELL-PROFILE

The corresponding data accessor .gen file is:

```
[Bootstrap]
LibraryName=sqlaccessor.dll
Enable=1
[Settings]
Connect=DSN=<dsn_name_here>;UID=<username_for_dB>;PWD=<password_for_dB>
ParameterMarker=?
SQL=SELECT RealmName, Profile FROM RealmExample2 WHERE KeyString =
@Key-String
MethodName=RealmExample2
MaxConcurrent=1
ConcurrentTimeout=30
ConnectTimeout=25
QueryTimeout=25
WaitReconnect=2
MaxWaitReconnect=360
PasswordFormat = 0
DefaultResults = 0

[Results]
RealmName = 1/32
Profile = 2/32

[VariableTypes]
RealmName = string
Profile = string
Key-String = string
```

The script configuration file is as follows.

```
[Settings]
LogLevel = 2
ScriptTraceLevel = 0

[Script]
// Allocate API objects at first execution.
if (!this.initialized) {
   filter = new AttributeFilter();
   selector = new RealmSelector();
   accessor = new DataAccessor("RealmExample2");
}
else {
   accessor.Clear();
}

// Get the realm suffix from the username attribute.
var suffix = "default";
var username = filter.Get("User-Name");
var index = username.lastIndexOf("@");
if (index >= 0) {
   suffix = username.substring(index + 1);
}

// Get the Called-Station-ID attribute from the request.
var csid = filter.Get("Called-Station-ID");
if (csid == null) {
   SbrWriteToLog("RealmExample2: Called-Station-ID attribute is null");
   return SCRIPT_RET_FAILURE;
}

// Concatenate suffix and Calling-Station-ID string to from SQL search key.
accessor.SetInputVariable("Key-String", suffix + csid);

// Execute the data accessor and get realm name if search succeeds.
var realm;
if (accessor.Execute() == DataAccessor.FOUND) {
   realm = accessor.GetOutputVariable("RealmName");
}
else {
   SbrWriteToLog("RealmExample2: SQL search failed for key string '" + suffix + csid +
"'");
   return SCRIPT_RET_FAILURE;
}

// Set a profile to be merged after request is authenticated.
var profile = accessor.GetOutputVariable("Profile");
selector.SetAuthProfile(profile);

// Return the realm name for processing request.
return realm;

[ScriptTrace]
var = suffix
var = csid
var = realm
var = profile
```

Attribute Filter Script Examples

Example 1: Using an LDAP Query to Select a Static Filter to Execute

This example uses the same LDAP repository as "Example 4: Conditional Profile Assignment from User Attribute" on page 66. In this case, the value of the Employee-Type attribute is used to select the name of a statically-defined filter that is returned as the result from the script. If an unknown Employee-Type is returned, no static filter is executed, but the script modifies the Service-Type attribute programmatically.

Here is the ldap accessor .gen file:

```
[Bootstrap]
LibraryName=ldapaccessor.dll
Enable=1

[Settings]
MethodName=FilterExample1
Timeout=20
ConnectTimeout=25
QueryTimeout=10
WaitReconnect=2
MaxWaitReconnect=360
UpperCaseName = 0
SSL = 0
Search = FindUser

[Server]
s1=

[Server/s1]
Host=<server-name>
Port = 389
BindName=uid=admin, ou=Administrators, ou=TopologyManagement, o=NetscapeRoot
BindPassword=<password>

[Request]
User-Name=my-user-name

[Response]
Employee-Type=employeetype

[Attributes/UserAttributes]
employeetype

[Search/FindUser]
Base=ou=people,dc=bigco,dc=com
Scope = 2
Filter = uid=<my-user-name>
Attributes = UserAttributes
Timeout = 20
%DN = dn
```

```
[VariableTypes]
User-Name = string
Employee-Type = string
```

Here is the .jsi file for an outbound filter script:

```
[Settings]
LogLevel = 2
ScriptTraceLevel = 0

[Script]
// Initialization block
if (!this.initialized) {
   filter = new AttributeFilter();
   accessor = new DataAccessor("FilterExample1");
   initialized = true;
}
else {
   accessor.Clear();
}

// Get the employee type from the username and set accessor input variable.
var username = filter.Get("User-Name");
accessor.SetInputVariable("User-Name", username);

// Execute the accessor.
if (accessor.Execute() == DataAccessor.FOUND) {
   // Get the employee type from the query result.
   var type = accessor.GetOutputVariable("Employee-Type");
   if (type == null) {
      SbrWriteToLog("FilterExample1: accessor returned null employee type");
      return SCRIPT_RET_FAILURE;
}

   // For known employee types, return name of built-in filter to execute.
   switch (type) {
   case "Student":
      return "StudentFilter";
   case "Faculty":
      return "FacultyFilter";
   case "Vendor":
      return "VendorFilter";
   default:
      SbrWriteToLog("FilterExample1: unknown employee type '" + type + "'");
   }
}

// Unknown employee type – change Service-Type to Authenticate-Only.
filter.Replace("Service-Type", 8);

// Return successfully, but do not execute a built-in filter.
return SCRIPT_RET_SUCCESS;

[ScriptTrace]
var = username
var = type
```

Example 2: Adding Values to Multi-Valued Attributes

This example demonstrates the difference between adding values to orderable and non-orderable multi-valued attributes. The script is configured as an inbound filter. A profile is created to assign three unique Reply-Message and three unique Filter-Id attribute value strings to the response attribute list. Subsequently, the script adds four new values to each attribute, two of them unique and two of them duplicates. All attribute values are printed to the server log before and after the new values are added. Because the Reply-Message attribute is orderable, the new values are concatenated to the value list and the duplicates are preserved. Because the Filter-Id attribute is non-orderable, the new attributes are merged to the list and the duplicates do not appear in the final list.

```
[Settings]
LogLevel = 2
ScriptTraceLevel = 0

[Script]
// Initialization block
if (!this.initialized) {
   filter = new AttributeFilter();
   initialized = true;
}

// Print out all Reply-Message attribute values.
var attr = "";
for(var i = 0; attr != null; i++) {
   attr = filter.Get("Reply-Message", i);
   if (attr != null) {
      SbrWriteToLog("Reply-Message[" + i + "] = '" + attr + "'");
}
}

// Add Reply-Message attribute values.
filter.Add("Reply-Message", "Reply-Message 1");
filter.Add("Reply-Message", "Reply-Message 2");
filter.Add("Reply-Message", "Reply-Message 4");
filter.Add("Reply-Message", "Reply-Message 5");

// Print out the new set of Reply-Message attribute values. Since Reply-Message
// is orderable, the values are concatenated, not merged.
attr = "";
for(var i = 0; attr != null; i++) {
   attr = filter.Get("Reply-Message", i);
   if (attr != null) {
      SbrWriteToLog("Reply-Message[" + i + "] = '" + attr + "'");
   }
} while (attr != null);

// Print out all Filter-Id attribute values.
attr = "";
for(var i = 0; attr != null; i++) {
   attr = filter.Get("Filter-Id", i);
   if (attr != null) {
      SbrWriteToLog("Filter-Id[" + i + "] = '" + attr + "'");
   }
} while (attr != null);
```

```
// Add Filter-Id attribute values.
filter.Add("Filter-Id", "Filter-Id 1");
filter.Add("Filter-Id", "Filter-Id 2");
filter.Add("Filter-Id", "Filter-Id 4");
filter.Add("Filter-Id", "Filter-Id 5");

// Print out the new set of Filter-Id attribute values. Since Filter-Id
// is not orderable, the values are merged, not concatenated.
attr = "";
for(var i = 0; attr != null; i++) {
   attr = filter.Get("Filter-Id", i);
   if (attr != null) {
      SbrWriteToLog("Filter-Id[" + i + "] = '" + attr + "'");
   }
} while (attr != null);

// All done.
return SCRIPT_RET_SUCCESS;
```

Chapter 9

Script Reference

JavaScript Types

JavaScript types are not defined explicitly. Variable data can be any of the following types:

- Integer
- Floating point
- Boolean
- String
- Array of one of the previous types
- Object

Script developers can use the full capabilities of the JavaScript language, such as defining new object types and storing data in arrays. However, the AttributeFilter, RealmSelector, and DataAccessor APIs use simple JavaScript types, and there is no support in the scripting API for structured IP address or time value types.

Steel-Belted Radius data types are represented in JavaScript variables as follows:

- Signed and unsigned integers are stored as JavaScript integers.
- Binary objects (HEXSTRINGs) are stored as hexadecimal coded JavaScript strings.
- Other data types (IP address types and times) are converted to formatted JavaScript strings.

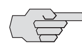
NOTE: The Steel-Belted Radius scripting API does not support binary conversion of attributes and/or data accessor variable values from or to unformatted binary strings.

API Method Support by Script Type

Some API methods are supported for all scripts, while others are only supported for one or two type(s) of scripts. For details about what methods are supported by the scripts, see Table 17.

Table 17: API Method Support by Script Type

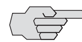
Object	Method	LDAP Script	Realm Selection Script	Attribute Filter Script
Global Object	SbrWriteToLog ()	X	X	X
	SbrTrace ()	X	X	X
LDAP	Ldap.Search	X		
LDAPVariables	LdapVariables.Get	X		
	LdapVariables.Add	X		
	LdapVariables.Reset	X		
RealmSelector	Execute ()		X	
	SetAuthUserName ()		X	
	SetAuthProfile ()		X	
AttributeFilter	Get ()		X	X
	Add ()			X
	Reset ()			X
	Replace ()			X
	Execute ()			X
DataAccessor	SetInputVariable ()		X	X
	GetOutputVariable ()		X	X
	Execute ()		X	X
	Clear ()		X	X



NOTE: Attempting to call an unsupported method from a script causes the script to return SCRIPT_RET_FAILURE and an error message to appear in the log.

Local and Global Variable Declarations

Variables declared using the var keyword are local variables, which persist only for the lifetime of the scope in which they are declared. Top-level local variables in your script are deleted when the script returns. However, variables declared without the var keyword become properties of the global object. Global variables and their data persist between calls to your script.



NOTE: To avoid unintended consequences, such as memory leaks or security breaches, you should use local variables by declaring them using the var keyword:

```
var count = 0; //Local variable declaration
filter = new AttributeFilter(); //Persistent global property
```

Global Object

The global object is available to all scripts at all times and does not need to be instantiated explicitly.

Logging and Diagnostic Methods

You can use the SbrWriteToLog() and SbrTrace() methods to insert messages and trace information into the Steel-Belted Radius log file.

SbrWriteToLog()

Purpose

The SbrWriteToLog() method writes text strings to the Steel-Belted Radius log file.

Syntax

SbrWriteToLog(*msg*, [*logLevel*])

Parameters

logLevel	Specifies the log level of the message. The log level configured in Steel-Belted Radius must be greater than or equal to this value for the message to appear in the log. Optional log level value in range of 0-2.
msg	Specifies the message text to be logged.

Returns

Nothing.

Example

```
SbrWriteToLog("Hello, world.", 0);
```

SbrTrace()

Purpose

The SbrTrace() method writes a script trace to the log from the point in the script where the statement appears.

Syntax

SbrTrace([*logLevel*])

Parameters

logLevel	Specifies the log level of the message. The log level configured in Steel-Belted Radius must be greater than or equal to this value for the message to appear in the log. Optional log level value in range of 0-2.

Returns

Nothing.

Example

SbrTrace(2);

Ldap Object

The Ldap object exposes methods for invoking LDAP queries from scripts.

Ldap Methods

Ldap.Search()

Purpose

The Ldap.Search() method is used to invoke an LDAP query defined in a [Search/*name*] section of the ldapauth.aut file.

Syntax

Ldap.Search(*SearchSection*)

Parameters

SearchSection	Specifies the name of a [Search/name] section of the ldapauth.aut file.

Returns

Ldap.FOUND	The search found a matching LDAP entry.
Ldap.NOTFOUND	The search did not find a matching LDAP entry.
Ldap.FAILURE	The search encountered an unexpected failure.
Ldap.NOSUCHSEARCH	The specified section was not found in ldapauth.aut.

Example

Given the following query definition:

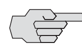
```
[Search/vpn]
Base = ou=vpn dc=jcn dc=com
Scope = 2
Filter = uid=<Vpn-User-Name>
Attributes = VpnAttrList
Timeout = 20
%DN = dn
```

You would use the following JavaScript command to invoke the query:

```
Ldap.Search("vpn");
```

LdapVariables Object

The LdapVariables object exposes methods for manipulating attributes in the LDAP plug-in variable table.



NOTE: If you are trying to access a binary attribute value, you must use the "= bin" syntax for the attribute in the [Attributes] section of ldapauth.aut file. The result will be returned to JavaScript as a raw binary (not hexidecimal-coded) string.

LdapVariables Methods

LdapVariables.Get()

Purpose

The LdapVariables.Get() method retrieves the current value or values for a variable stored in the LDAP variable table. LdapVariables.Get() can be used to retrieve binary (raw) data or text strings.

Syntax

LdapVariables.Get(*variableName*[, *nItem*])

Parameters

VariableName	Specifies the name of the variable in the variable table.
nItem	Specifies the index of the value for multi-valued attributes. You can specify the value of *nItem* as part of the command, or you can use a separate variable to set the value of *nItem* (as shown in the following example).

Returns

The value of the specified attribute, or a null value if the attribute doesn't exist or if the index is out of bounds.

Examples

The following example illustrates a single-value attribute lookup:

```
var name = LdapVariables.Get("User-Name");
```

The following example illustrates a multi-value attribute lookup:

```
var index = 2;
var alias = LdapVariables.Get("User-Alias", index);
```

LdapVariables.Add()

Purpose

The LdapVariables.Add() method creates a new variable or adds a value to an existing variable.

Syntax

LdapVariables.Add(*variableName*, *value*[, *raw*])

Parameters

variableName	Specifies the name of the variable to be created or updated.
value	Specifies the value of the variable, which might be text or binary data.
raw	■ If set to True, specifies that the value is binary data. ■ If set to False, specifies that the value is text. Default value is False.

Returns

Nothing.

Example

```
LdapVariables.Add("Vpn-User-Name", "Fred");
```

LdapVariables.Reset()

Purpose

The LdapVariables.Reset() method deletes all of the values of the specified variable from the variable table. If the specified variable does not exist, the method call is ignored.

Syntax

LdapVariables.Reset(*variableName*)

Parameters

VariableName	Specifies the name of the variable to be removed from the table.

Returns

Nothing.

Example

LdapVariables.Reset("Vpn-User-Name");

RealmSelector Object

Objects of the RealmSelector class are used to execute built-in Steel-Belted Radius realm selection methods from realm selection scripts. To use the RealmSelector class, scripts must first create an object instance by calling the new RealmSelector() constructor.

Constructor

new RealmSelector()

Purpose

The new RealmSelector() constructor is used to create a new object instance of the RealmSelector class.

Syntax

new RealmSelector()

Parameters

Nothing.

Returns

The new RealmSelector() object reference.

Example

filter = new RealmSelector();

RealmSelector Methods

These methods are provided to execute built-in realm selection methods, set the authentication user name passed to the selected directed or proxy realm, and set the user profile upon return from the realm.

Execute()

Purpose

The Execute() method is used to execute built-in Steel-Belted Radius realm selection methods.

Syntax

selector.Execute(method)

Parameters

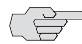
method	Specifies the name of the realm selection method: prefix, suffix, dnis, attribute-mapping, or undecordated.

Returns

The selected realm name or JavaScript null if no realm is matched.

Example

selector.Execute(“suffix”);



NOTE: Before you can execute the prefix or suffix built-in methods, you must enable them in the proxy.ini file. If a script has been declared in the [Processing] section of the proxy.ini file, then the prefix and suffix methods are enabled automatically. However, if a script has been declared in an [Inner_Authentication] section of a tunneled .aut file of an authentication method (FAST, TTLS, or PEAP), then you must explicitly declare the prefix and suffix methods in the [Processing] section of the proxy.ini file.

SetAuthUserName()

Purpose

The SetAuthUserName() method is used to set the authentication username for the request.

Syntax

selector.SetAuthUserName(username)

Parameters

username	Specifies the desired authentication username.

Returns

Nothing.

Example

selector.SetAuthUserName(“RButler”);

SetAuthProfile()

Purpose

The SetAuthProfile() method is used to set the profile to be returned upon successful authentication.

Syntax

selector.SetAuthProfile(profile)

Parameters

profile	Specifies the profile name.

Returns

Nothing.

Example

selector.SetAuthProfile(“Profile1”);

AttributeFilter Object

Objects of the AttributeFilter class are used to read, write, modify, and reset attributes in the RADIUS request or response packets. To use the AttributeFilter class, scripts must first create an object instance by calling the new AttributeFilter() constructor.

Constructor

new AttributeFilter()

Purpose

The new AttributeFilter() constructor is used to create a new object instance of the AttributeFilter class.

Syntax

new AttributeFilter()

Parameters

Nothing.

Returns

The new AttributeFilter() object reference.

Example

filter = new AttributeFilter();

AttributeFilter Methods

Methods of the AttributeFilter class operate on either the request or the response attribute list, depending on the context in which they are called. All of the methods of the AttributeFilter class are available to attribute filter scripts. The AttributeFilter.Get() method is also available to realm selection scripts.

Get()

Purpose

The Get() method is used to get a named attribute from the request or response attribute list.

Syntax

Get(*name,* [*index*])

Parameters

name	Specifies the name of the attribute dictionary.
index	Specifies the optional index for multi-value attributes; default is 0.

Returns

The attribute value or JavaScript null if not found.

Example

filter.Get("Calling -Station-ID");

Add()

Purpose

The Add() method is used to add a named attribute to the request or response attribute list. If an attribute of the same name already exists, the new data value will be appended to the end of the attribute list.

Syntax

Add(*name,* value)

Parameters

name	Specifies the name of the attribute dictionary.
value	Specifies the string representation of the attribute value.

Returns

Nothing.

Example

filter.Add("NAS-IP-Address", "1.2.3.4");

Reset()

Purpose

The Reset() method is used to remove one or all instances of a named attribute from the request or response attribute list. If the index is given, the specified value will be removed from the attribute list and higher-indexed values will drop down to fill in the vacancy.

Syntax

Reset(*name, [index]*)

Parameters

name	Specifies the name of the attribute dictionary.
index	Specifies the optional index for multi-value attributes; default is remove all.

Returns

Nothing.

Example

filter.Reset(“NAS-IP-Address”, 3);

Replace()

Purpose

The Replace() method is used to delete a named attribute and add a new attribute of the same name and given value. This is equivalent to Reset() followed by Add(). When an index is given, the specified attribute value is removed, the remaining higher-indexed values drop down to fill in the vacancy, and the new value is appended to the end of the attribute list. List order is not preserved.

Syntax

Replace(*name, value, [index]*)

Parameters

name	Specifies the name of the attribute dictionary.
value	Specifies the string representation of the new attribute value.
index	Specifies the optional index for multi-value attributes; default is remove all.

Returns

Nothing.

Example

filter.Replace(“Calling-Station-ID”, “5551212”, 2);

Execute()

Purpose

The Execute() method is used to execute a statically-defined attribute filter.

Syntax

Execute(name)

Parameters

name	Specifies the name of the filter defined in filter.ini.

Returns

Nothing.

Example

```
filter.Execute("MyStaticFilter");
```

AttributeFilter API

Table 18 provides a list of the RADIUS attribute data types supported by the AttributeFilter API.

Table 18: AttributeFilter API Data Types

RADIUS Attribute Type	JavaScript Type
■ 8-bit integer ■ 32-bit integer ■ 32-bit hex (HEX4) (Get only) ■ IPX address pool	integer
■ Fixed-length string ■ Zero-terminated string	string
Arbitrary length hex (HEXSTRING)	hexadecimal-coded string
■ IPv4 address ■ IPv4 address pool ■ IPv6 address ■ IPv6 prefix ■ IPv6 interface	type-specific formatted string
Time	time string (*yyyy/mm/dd [hh[:mm[:ss]]]*)

Script developers do not control attribute types; a given attribute's type is determined from its name and dictionary entry. Many bit-length variations of integer and hex data types are possible, but not all are supported or used by Steel-Belted Radius.

When AttributeFilter.Get() is called, the binary data in the attribute are converted into a JavaScript variable. When either AttributeFilter.Add() or AttributeFilter.Replace() is called, the data in a JavaScript variable are converted into the appropriate RADIUS attribute binary type.

NOTE: Due to an internal limitation in Steel-Belted Radius, the HEX4 attribute type is supported for Get() only. Attempting to Add() or Replace() a variable of type HEX4 will cause a script error.

DataAccessor Object

Objects of the DataAccessor class enable scripts to perform SQL queries and LDAP searches using the Steel-Belted Radius SQL and LDAP data accessor plug-ins.

All DataAccessor methods are available to both realm selection and attribute filter scripts.

Properties

FOUND

Purpose

The FOUND property is returned by the Execute() method when the requested data object is found.

Example

```
found = (da.Execute() == DataAccessor.FOUND);
```

NOTFOUND

Purpose

The NOTFOUND property is returned by the Execute() method when the requested data object is not found.

Example

```
notfound = (da.Execute() == DataAccessor.NOTFOUND);
```

FAILED

Purpose

The FAILED property is returned by the Execute() method when the data request fails.

Example

```
failed = (da.Execute() == DataAccessor.FAILED);
```

Constructor

The data accessor name is given as an argument to the object constructor.

new DataAccessor()

Purpose

The new DataAccessor() object instance is required to use the DataAccessor methods.

Syntax

new DataAccessor(name)

Parameters

name	Specifies the name of the data accessor, as configured in the .gen file.

Returns

The new DataAccessor() object reference.

Example

da = new DataAccessor("LDAP-Accessor");

Methods

SetInputVariable()

Purpose

The SetInputVariable() method is used to set a named variable in the data accessor input container.

Syntax

accessor.SetInputVariable(*name,* value)

Parameters

name	Specifies the name of the input container variable from the data accessor .gen file.
value	Specifies the string representation of the input variable value.

Returns

Nothing.

Example

SetInputVariable ("Phone-Number", "5551212");

GetOutputVariable()

Purpose

The GetOutputVariable() method is used to get a named variable from the data accessor output container.

Syntax

accessor.GetOuputVariable(*name*)

Parameters

name	Specifies the name of the output container variable from the data accessor .gen file.

Returns

The string representation of the output variable value.

Example

address = GetOutputVariable (“Street-Address”);

Execute()

Purpose

The Execute() method is used to execute the query or search using the current input container data values.

Syntax

accessor.Execute()

Parameters

Nothing.

Returns

FOUND, NOTFOUND, OR FAILED properties.

Example

found = (da.Execute() == DataAccessor.FOUND);

Clear()

Purpose

The Clear() method is used to clear accessor input and output container values.

Syntax

accessor.Clear()

Parameters

Nothing.

Returns

Nothing.

Example

```
accessor.Clear();
```

Appendix A
LDAP Script Return Codes

Table 19 provides a list of the LDAP script return codes.

Table 19: LDAP Script Return Codes

Script Return Code	Action	Plug-in Return Code
SBR_RET_SUCCESS	Accept the user.	SBR_RET_SUCCESS
SBR_RET_DO_NOT_AUTHENTICATE	Hard reject. Do not invoke another authentication method.	SBR_RET_DO_NOT_AUTHENTICATE
SBR_RET_TRY_NEXT_AUTH_METHOD	Return from the LDAP plug-in and invoke the next authentication method. Do not process [Failure] section or try last resort server.	SBR_RET_NOT_AUTHENTICATED
SBR_RET_FAILURE	A communication error occurred. Retry script with next server in list, or go to [Failure] section if no server is available.	If another server is available, plug-in return code depends on script return code when script is re-executed. If no server is available, process [Failure] section and return SBR_RET_SUCCESS or SBR_RET_NOT_AUTHENTICATED depending on configuration.
SBR_RET_NOT_AUTHENTICATED	Retry script with last resort server, if defined. Otherwise, go to the next authentication method.	If LastResort is defined, plug-in return code depends on script return code when script is re-executed. If LastResort is not defined, return SBR_RET_NOT_AUTHENTICATED.

Index

Symbols

.gen file 41, 43, 47, 56

A

access-request attributes 23
Add() method 36, 87
adding rule types 38
adding values to attributes 75
API
  attribute filter 89
  data accessor 55
  methods by script type 78
  realm selector 30
attr parameter 10
attribute filter
  configuring scripts 37
  creating scripts 35
  methods 86
  overview 4
  script examples 73
  script functions 36
  scripted 35
  static 35
attribute maps 24
attribute normalization 65
AttributeFilter API 89
AttributeFilter object 86
Attributes parameter 49
Attributes/name section 47
automatic tracing 18

B

Base parameter 49
Base strings 23
binary objects 77
Bind strings 23
BindName parameter 51, 53
BindPassword parameter 51
boolean 77
Bootstrap section 43, 47
built-in realm selection methods 30
bytecodes 26

C

Calling-Station-ID attribute 65
Certificates parameter 51
choosing return code 27, 31, 36
Clear() method 42, 92
ConcurrentTimeout parameter 45
Connect parameter 45
ConnectDelimiter parameter 45
ConnectTimeout parameter 45, 51, 53
core realm selection 31
creating scripts 7

D

data accessor
  API 55
  configuration file examples 59
  constructor 90
  conversion rules 55
  LDAP configuration 47
  LDAP example 59
  methods 91
  overview 41
  properties 90
  SQL configuration 43
  SQL example 61
  variable containers 42
data arrays 42
data conversion
  examples 57
  rules 55
  supported data types 58
data objects
  initializing reusable 12
database types 58
debugging scripts 17
Defaults section 50
defining scripted filters 37
developing scripts 7
diagnostic methods 79
Distinguished Names (DNs) 24, 49
Driver parameter 45

E

employeetype attribute 66
Enable parameter 44, 47
Execute() method 30, 36, 42, 84, 88, 92

F

FAILED property 90
Failure section 10
Filter parameter 49
filter strings 24
FilterName 37
FilterSpecial CharacterHandling parameter 54
FlashReconnect parameter 52, 54
floating point 77
FOUND property 90

G

generic plugin 41
Get() method 36, 86
GetOutputVariable() method 41, 92
global object 79
global variables 79

H

HEXSTRING 77
hidden wrapper function 11
Host parameter 52

I

index parameter 87, 88
initialization file 1
input container 56
installing upgrade license 13
integer 77
internal variable table 43
interpreter 26

J

JavaScript 5
  engine 5
  initialization file (.jsi) 8
  interpreter 26
  overview 5
  programming 11
  types 58, 77
  upgrade license 20

L

LastResort parameter 52
LDAP
  attributes 23
  authentication scripts 3
  database schema 47
  internal variable table 43
  methods 80
  object 80
  overview 3, 23
  repository 41
  request life cycle 23
  script basics 26
  script examples 63
  script return codes 95
  scripting 23
  search tree example 49
  searches 35
  unscripted searches 24
LDAP data accessor configuration 47
Ldap.FAILURE parameter 81
Ldap.FOUND parameter 81
Ldap.NOSUCHSEARCH parameter 81
Ldap.NOTFOUND parameter 81
Ldap.Search() method 27, 80
ldapaccessor.so file 47
ldapauth.aut file 23, 26, 64
ldapauth.dll file 47
LdapVariables methods 81
LdapVariables object 26, 81
LdapVariables.Add() method 26, 82
LdapVariables.Get() method 26, 81
LdapVariables.Reset() method 26, 83
LdapVersion parameter 52, 54
LibraryName parameter 44, 47
license upgrade installation 13, 20
Lightweight Directory Access Protocol, see LDAP
Linux platform 20, 21, 43
local variables 79
log file 17, 26, 27
logging methods 79
logLevel 8, 9, 45, 54, 79

M

manipulating request attributes 71
manual tracing 18
MaxConcurrent parameter 45, 52, 54
MaxScriptSteps parameter 9
MaxWaitReconnect parameter 45, 52, 54
method parameter 84
msg parameter 79

N

name parameter 87, 88
new AttributeFilter() object instance 36, 86
new DataAccessor() object instance 41, 91
new RealmSelector() object instance 30, 83
nItem parameter 82
normalization 65
NOTFOUND property 90

O

object 77
OnFound 24, 49, 54
OnNotFound 24, 49, 54
output container 56

P

ParameterMarker parameter....45
plugin
- data accessor....41, 56
- generic....41
- LDAP....3
- tunneled authentication realm selection scripts..33

Port parameter....52
Processing section....32
profile assignment....64
profile parameter....85
programming in JavaScript....11
proxy.ini....32

Q

query tree....24
querying multiple SQL databases....68
QueryTimeout parameter....46, 52, 54

R

RADIUS attribute data types....89
radius.lic file....20
radiusattrs attribute....64
Radius-Profile attribute....66
raw parameter....82
realm selection
- configuring scripts....31
- creating scripts....29
- enabling built-in methods....30
- overview....3
- script examples....68
- script functions....30
- strategy....29

RealmName return code....31
RealmSelector API....30
realmselector methods....32, 84
realmselector object....83
Replace() method....36
Request section....23, 50
Reset() method....36, 87, 88
Response section....48
Results section....44
rule types....38

S

SBR Administrator tool....31, 33, 37
SBR_RET_DO_NOT_AUTHENTICATE....95
SBR_RET_FAILURE....95
SBR_RET_NOT_AUTHENTICATED....95
SBR_RET_SUCCESS....95
SBR_RET_TRY_NEXT_AUTH_METHOD....95
SbrTrace()....27, 79, 80
SbrWriteToLog()....17, 27, 79
Scope parameter....49
script return values....11, 27
Script section....9
script tracing....17
- appear in logs....19
- example....18

SCRIPT_RET_DO_NOT_AUTHENTICATE....28
SCRIPT_RET_FAILURE....12, 28, 31, 36
SCRIPT_RET_NOT_AUTHENTICATED....28
SCRIPT_RET_SUCCESS....28, 31, 36
SCRIPT_RET_TRY_NEXT_AUTH_METHOD....28
scriptcheck utility....26
- overview....19
- running....21
- unpacking....20

scripted attribute filters....35
scripted data flow....25
scripted queries....25
scripting types
- attribute filter....35
- LDAP....23
- realm selection....29

scripts
- basics....7
- creating....7
- debugging....17
- examples....63
- overview....1
- recommendations....13
- reference....78
- sample....14
- saving....13
- sections....8
- types....2
- writing Steel-Belted Radius scripts....11

ScriptTrace section....10
ScriptTraceLevel....8, 9
Search parameter....52, 54
Search strings....23
Search/name section....27, 48
SearchSection parameter....80
selecting a static filter....73
Server section....52
Server/name section....51
SetAuthProfile() method....30, 85
SetAuthUserName() method....30, 84
SetInputVariable() method....41, 91
Settings section....8, 44, 53
Solaris platform....20, 21, 43
SpiderMonkey JavaScript engine....5
SQL data accessor configuration....43
SQL databases....41
SQL parameter....46
SQL queries....1, 35
SSL parameter....52, 54

static attribute filters 35
Steel-Belted Radius scripts 11
string 77
supported conversions 58
syntax errors 26

T

Timeout parameter 55
tunneled authentication methods 29, 31

U

unscripted searches 23, 24
upgrade license 20
username parameter 85
UTC parameter 55

V

value parameter 82, 87, 88
var parameter 10
variable containers 42
variable table 23, 26
VariableName parameter 82, 83
variableName parameter 82
VariableTypes section 46, 55

W

WaitReconnect parameter 46, 52, 55
Windows platform 20, 21, 43
writing messages to server log 17